Lua scripting and viewer automation feature

Table of Contents

Chapter 1 - Generalities....3
- Automation script....3
- Other Lua programs/commands input methods....3
- Lua programs watchdog and timeout....4
- Lua excluded features....4
- Lua source files pre-processing....4
- Trusted scripts....5
- Miscellaneous....5
- Syntax description conventions....5

Chapter 2 – Lua functions....6
- Printing text....6
- Utility functions....6
- Objects related functions....8
- Agent and avatars related functions....9
- Group related functions....11
- Chatting and instant messaging related functions....12
- Muting and blocking related functions....13
- Radar related functions....14
- Camera related functions....15
- Grid, simulator, parcel and position related functions....16
- Agent movements controls related functions....17
- Agent puppetry related functions....19
- Agent inventory related functions....19
- Agent outfit related functions....20
- Windlight and extended environment related functions....21
- SLURL, URL and RestrainedLove commands dispatching functions....22
- Networking functions....23
- User interface related functions....24
- Debug and persistent settings related functions....33
- Lua threads related functions....35

Chapter 3 – Lua callbacks....36
- Utility callbacks....36
- Objects related callback....36
- Agent position and movement related callbacks....37
- Agent and avatar related callbacks....38
- Camera related callback....38
- Windlight and extended environment related callback....38
- Radar related callbacks....39
- RestrainedLove related callbacks....39
- Chat and instant messaging related callbacks....40
- Networking related callback....40
- User interface related callbacks....41

Chapter 4 – Lua automation threads....44

Appendix A – Automation script example....47

Appendix B – Custom Lua floater example....58
Appendix C – LSL to Lua scripting example....61
Appendix D – Constants....64

Chapter 1 - Generalities

The Cool VL Viewer can execute Lua v5.4 scripts (that, in their turn, may trigger some viewer actions) on events (e.g. chat, IMs, radar detections, etc), and from various input methods. This manual documents this client-side scripting feature.

Automation script

On startup, the viewer attempts to read a Lua script ("automation.lua", by default; this file name is configurable via the "LuaAutomationScript" debug setting), from the user_settings/ directory, and executes it if found. The automation script is the only one that may contain callbacks for viewer events (i.e. the said event callbacks won't work if provided in a command line coming from the chat bar or LSL scripts).

- You may, at any time, load a new automation script (which would then replace any already loaded script) with "Advanced" -> "Lua scripting" -> "Load new automation script...".
- You may, at any time, re-load the currently running automation script (if any) with "Advanced" -> "Lua scripting" -> "Re-load current automation script".
- You may, at any time, stop (i.e. disable) any currently loaded automation script with "Advanced" -> "Lua scripting" -> "Stop current automation script".

See Appendix A for an example automation script.

Other Lua programs/commands input methods

- You may, at any time, execute a Lua script file (which does not replace the automation script, but may interact with it) via "Advanced" -> "Lua scripting" -> "Execute a Lua script file...".

- You may issue Lua command lines from the chat bar, by prefixing them with "/lua "; this prefix can be configured via the "LuaCommandPrefix" debug setting (it is recommended that you keep a "/" as the first character for your custom prefix, so that the viewer behaves like when you type a gesture name in the chat bar and does not send an "agent typing" event to the server which would then be relayed to other residents around you).

- Starting with the Cool VL Viewer v1.32.0.11, you also got a new Lua console floater which you may open via "Advanced" -> "Lua scripting" -> "Lua console" or the CTRL ALT L keyboard shortcut. The console got its own Lua messages printing window and a window for entering commands or a small Lua program (you may even load the text contained in a Lua script file into this window); it is handier than the chat line since it does not need any "/lua " prefix and got its own commands history.

- When enabled (via "Advanced" -> "Lua scripting" -> "Accept Lua from LSL scripts", which toggles the "LuaAcceptScriptCommands" debug setting), your **own** scripts may send Lua command lines via llOwnerSay("/lua ...") and/or llInstantMessage(llGetOwner(), "/lua ..."). Here again, the "/lua " prefix is configurable, independently from the chat bar prefix, via the "LuaScriptCommandPrefix" debug setting (you can see this as a sort of "password", and scripters that would like to distribute scripted items making use of the Cool VL Viewer Lua scripting feature shall provide a mean to configure that prefix in their scripted items). See Appendix C for an example.

- Under Linux only, when enabled (via "Advanced" -> "Lua scripting" -> "Accept Lua commands from D-Bus", which toggles the "LuaAcceptDbusCommands" debug setting), the viewer accepts Lua commands sent via the D-Bus session (i.e. D-Bus messages sent from the same Linux user account). An example 'dbuslua.tk' Tcl/Tk script is provided in the viewer installation directory. You may also use 'dbus-send' via a terminal or a shell script, like so:

```
dbus-send --session --type=method_call --print-reply \
--dest=com.secondlife.ViewerAppAPIService /com/secondlife/ViewerAppAPI \
com.secondlife.ViewerAppAPI.LuaExec string:'print("Hello world !")'
```

  When the viewer receives such a D-Bus command, it interprets it (just like if you sent it via the chat input line, but without the need for a /lua prefix), redirecting all print() and error messages to an internal buffer (instead of printing them to the viewer console/chat history) and returns the contents of this buffer via D-Bus to the sender. If an error is encountered, all former print() messages are discarded from the buffer and only the error message is returned via D-Bus.

Lua programs watchdog and timeout

Since the current implementation of the script engine runs in the main viewer loop, it must not block ("freeze") the latter, so there is a timeout Watchdog mechanism implemented. You may adjust the timeout of Lua scripts/commands execution from 0.01s to 2s via the "LuaTimeout" debug setting (defaults to 0.2s which allows dozens of thousands of Lua instructions to be executed before the script would time out), and, for one-shot scripts loaded from files, via the "LuaTimeoutForScriptFile" to 0.01s to 30s (defaults to 3s).

Lua excluded features

For security or compatibility reasons, the following Lua libraries (AKA "packages" in Lua jargon) are **NOT** available from the Cool VL Viewer Lua engine:

- "package" (no "require" Lua command available and no third party Lua library loading allowed, for obvious security reasons)
- "os" (because of the dangerous/blocking actions it would allow, such as "os.execute()", "os.exit()", etc).
- "io" (because of the dangerous actions that could allow Lua scripts to overwrite important files, or read files to gather and re-emit/relay private info).
- "debug" (because it would allow to bypass some of the above restrictions).
- "coroutine" (because Lua callbacks are executed from the main viewer thread and coroutines in them would block the viewer execution). But there is now support for threaded Lua scripts, which largely makes for it !

For the same security reasons, the load(), loadfile() and dofile() Lua base functions are also disabled.

Lua source files pre-processing

When loading a Lua script file (be it the automation script, a Lua thread script, or any script you would load via the "Advanced" -> "Lua scripting" menu), the viewer first attempts to load the file as a genuine Lua script file and, on failure (which happens if it encounters preprocessor #directives elsewhere than on the first line of the script (*)), it hands over the file to the its own built-in preprocessor, which returns a (long) string containing the preprocessed sources, the Lua interpreter then being fed again with it in a second (and last) attempt to execute the script.

(*) As per Lua specifications, the first line of any script that would start with a '#' is ignored. This is to cope with shebang lines.

Trusted scripts

In the rest of this document, I will refer to "trusted scripts" or "trusted non-threading scripts" for some functions availability restrictions, meaning scripts loaded locally, from files present on your computer (including the automation script itself, and excluding Lua thread scripts for "trusted non-threading scripts") or command lines (entered in the chat input line, the Lua console, or coming from commands sent by Lua UI elements callbacks), i.e. scripts flagged as pertaining to the agent (bearing your avatar Id), which excludes commands sent via LSL or D-Bus, which are deemed "unsafe".
Note that before the Cool VL Viewer versions 1.30.0.22 and 1.30.1.5 (in which this concept got relaxed to include Lua scripts manually loaded from local files, Lua commands entered from the chat input line and Lua UI callbacks), the only "trusted script" was the automation script itself.

Miscellaneous

Three global variables are made available to all Lua scripts executed by the viewer:

- `VIEWER_NAME` contains a Lua string set to the viewer name ("Cool VL Viewer").
- `VIEWER_VERSION` contains a Lua string set to the viewer version (e.g. "1.30.2.2").
- `VIEWER_VERNUM` contains a Lua number set to the viewer version in the form Mmmmbbbrrr (with M for the major number, mmm for the minor on three digits, bbb for the branch, rrr for the release). This variable was introduced in v1.30.2.2, for which `VIEWER_VERNUM` would then contain 1030002002.

For RestrainedLove users, note that the associated restrictions still apply (i.e. all received or sent chat/IM text, or names, will be properly censored when needed, and all floaters will properly fail to open if the corresponding restriction is in force, etc); that is, Lua scripting won't buy you a mean of cheating RestrainedLove !

Syntax description conventions

In the syntax description used in this manual for the Lua functions, square brackets denote an optional parameter, and a pipe character between two (or more) names denotes the possible choices for that parameter; the square brackets or the pipe character themselves shall not be typed. A "[, ...]" denotes a possible repetition of an arbitrary number of additional parameters.

E.g. a function syntax described in this document as:
SomeFunction(number1[, string|boolean[, number2]])
May be used (typed) as either:

- *SomeFunction(number1)*
- *SomeFunction(number1, string)*
- *SomeFunction(number1, boolean)*
- *SomeFunction(number1, string, number2)*
- *SomeFunction(number1, boolean, number2)*

Chapter 2 – Lua functions

Printing text

- The print() Lua function is replaced with a viewer-specific one that prints into the viewer chat history and chat console, or in the Lua console output window for commands executed via it. When printed in chat, the printed messages are all prefixed with "Lua: " and colored with the "system" messages color.

- The warn() Lua function (new in Lua v5.4) is supported and, by default, causes a "WARNING: ..." message to be printed into the viewer chat history and chat console, or in the Lua console output window for commands executed via it. When printed in chat, the printed messages are all prefixed with "Lua: " and colored with the "system" messages color.
  The following control messages are also supported:
  - "@off" and "@on": to respectively pause and resume warning messages printing. As long as the Lua warning system is turned "@off", warnings are concatenated in an internal text buffer which is printed as a single warning as soon as the system is turned back to "@on".
  - "@prefix:<custom_prefix>" and "@prefix": allow to respectively change the "WARNING" prefix and reset it in the issued messages. "<custom_prefix>" is trimmed from leading and trailing spaces and if it results in an empty string, the default "WARNING" prefix is restored.
  - "@tochat:0" or "@tochat:off" or "@tochat:false", and "@tochat" or "@tochat:<anything>": allow to respectively reset to the default output, or force warning messages to be printed to the viewer chat; this control is useful when using the Lua D-Bus execution feature under Linux, since in this case all outputs are normally redirected to an internal print buffer which contents is returned to the D-Bus call on return.

Utility functions

- GetSourceFileName() returns a Lua string containing the name of the Lua sources file for the currently running script, or *nil* if the script was not loaded from a file (e.g. for command line).

- GetWatchdogState() returns two Lua numbers, the first being the remaining time before the watchdog would kick-in and interrupt the currently running script, and the second being the elapsed time since the said script execution started. Both values are counted in seconds.

- CallbackAfter(delay, SomeFunction[, func_param1[, ...]]) is only available to the automation script, and instructs the viewer to call back SomeFunction() after "delay" seconds (clamped to 1.0 second minimum, to avoid abuses), passing the optional parameters (func_param1 ... func_paramN) listed after the function name. SomeFunction() must be a function defined in the automation script. The number of parameters is limited only by the stack size (20 per default in Lua v5.4), minus three slots.

- AutomationMessage(message) is only available from Lua command lines sent by LSL scripts or Lua dialog/overlay bar/status bar/side bar buttons/pie menu slices, or via the chat lines, or the Lua console, or D-Bus (i.e. this function is not defined for the automation script). It sends 'message' (string or string convertible Lua type) to the OnAutomationMessage() callback of the automation script, thus allowing the former command lines to send data/feedback to the latter script. It is especially handy for use in conjunction with OverlayBarLuaButton(), StatusBarLuaIcon(), SideBarButton() and LuaPieMenuSlice().

- AutomationRequest(request) is only available from Lua command lines sent by LSL scripts or Lua dialog/overlay bar/status bar/side bar buttons/pie menu slices, or via the chat lines, or the Lua console, or D-Bus (i.e. this function is not defined for the automation script). It sends 'request' (string or string convertible Lua type) to the OnAutomationRequest() callback of the automation script, thus allowing the former command lines to request data from the latter script. This function returns the string which was itself returned by the OnAutomationRequest() callback, or an empty string when no such callback exists.

- GetFrameTimeSeconds() returns a float representing the number of seconds elapsed since the beginning of the current viewer session (it's the viewer's gFrameTimeSeconds global variable value).

- GetTimeStamp([tz, [format]]) returns a time stamp string containing the date and time for the 'tz' time zone (integer type, UTC time zone by default) formatted according to 'format' (string type), or to the format defined in the viewer preferences ("Cool features", "User interface") by default. E.g.: *GetTimeStamp(-8,"%Y-%m-%d %H:%M:%S")* returns the time in the PST (SL) time zone in the year-month-date and 24 hours clock format.

- GetTimeSinceEpoch() returns a float representing the number of seconds elapsed since the UNIX Epoch (1970-01-01 00:00:00 UTC).

- GetClipBoardString([clipboard_type]) returns the current string contained in one of the three available paste buffers: when 'clipboard_type' is omitted or equal to 0, the returned string comes from the private paste buffer of the viewer (this is the last string that got copied into the clipboard from a viewer input line or text editor, via CTRL C or a menu “Copy” action); when 'clipboard_type' is equal to 1, the OS paste buffer is used (the buffer where text gets copied into with CTRL C or a menu “Copy” action, from any running program window); when 'clipboard_type' is equal to 2, the “primary paste buffer” is used (this is a super cool X Window System feature: left-click-drag to select, middle-click to paste), and under Linux, it corresponds to the text you simply selected by using the mouse, from any program (X11) window (but without using CTRL C or any menu copy action), while under macOS and Windows which do not have this facility, it is an emulated primary buffer private to the Cool VL Viewer (and AFAIK no other viewer got it) that only works for text you selected with the mouse in one of its UI input line or text editor. An empty string is returned by this function when the corresponding clipboard is empty or does not contain an object of a string type.

- SetClipBoardString([text]) sets the clipboard (paste buffer only) text to 'text' (or to an empty string when 'text' is omitted).

- IsUUID(value) returns *true* if 'value' evaluates properly to an UUID, *false* otherwise. 'value' can be of any Lua type, making this function an easy way to check for valid UUIDs in table keys or values returned by other functions below.

- MinimizeWindow() is only available from the automation script and allows to minimize the viewer window. While minimized, the "BackgroundYieldTime" debug setting is used to sleep the viewer by the configured amount of milliseconds at each frame, and the rendering itself is (more or less) paused, causing the processing power consumption to be reduced to a minimum (great for unattended/robot/scripted sessions). Note: this only works under Linux and Windows; macOS will only see the mouse cursor released by the viewer (which still should cause the yielding to happen, but would not pause the rendering).

- ForceQuit([exit_code]) is only available from the automation script and allows to close the viewer session unconditionally. The 'exit_code' (0 by default, when omitted) parameter is returned to the OS by the viewer process and, beside 0 (which is used to indicate a normal exit, without error), can be any positive number between 5 and 125 inclusive; other exit codes are reserved by either the viewer (1 = viewer initialization failed, 3 = login failed, 4 = logged out after a connection loss with the server or by a server force-logout command) or by the OS or its command shell (e.g. 2, 126 and 127 are reserved by the bash shell, and exit codes above 127 may be turned into negative exit codes, which may be related to exceptions and events handling, depending on the OS or shell).

- EncodeJSON(lua_table) takes a Lua table and attempts to encode it as a JSON string. Returns *nil* on failure.

- DecodeJSON(json_string) takes a Lua string to attempt and decode it as JSON data into a Lua table. Returns *nil* on failure.

- EncodeBase64(text) takes a 'text' Lua string and returns its base64-encoded equivalent.

- DecodeBase64(data) takes a base-64 encoded 'data' Lua string and returns its plain text equivalent.

Objects related functions

- IsObject(id) returns *true* if 'id' indeed corresponds to an object currently present in the viewer's objects list (i.e. an object present in the current sim or nearby simulators), or *false* otherwise.

- GetObjectInfo(object_id) sends a request for info about a rezzed object which Id is object_id. This function is only available from trusted scripts and returns *false* when the object is unknown to the viewer (not rezzed) or *true* otherwise. When the info request succeeds and the details are received by the viewer (which happens asynchronously), the OnObjectInfoReply() callback (see the callbacks section) is called, when it exists in the automation script.

- DerenderObject([object_id[, true|false]]) derenders an in-world object (already rezzed/around or not) which UUID is 'object_id' (when *true* or no boolean is passed) or removes it from the derendered objects list when *false* is passed. If no parameter is passed at all to this function, it clears the whole derendered objects list. This function returns *true* on success or *false* on failure.
  NOTE 1: even once cleared from the derendered objects list, a formerly derendered object will render again only after the server sends an object update data packet, which may not occur at all till you leave the region and come back later into it (only scripted and moving objects may cause regular server update packets to be emitted).
  NOTE 2: objects attached to your avatar cannot be derendered (the function aborts and returns *false* if you try).
  NOTE 3: objects your avatar is sitting onto while under RestrainedLove sit restrictions would fail to derender (the function aborts and returns *false* if you try).

- GetDerenderedObjects() returns a Lua array containing the list of the UUIDs of the derendered objects (empty array if none).

Agent and avatars related functions

- GetAgentInfo() returns data about your own avatar (the "agent" in SL viewers jargon) in a Lua table with the following "key"/value pairs: "name" is the agent (legacy) name (e.g. "John Doe" for an old resident, or "JohnDoe Resident" for a family-name-less one), "id" is the agent UUID (as a Lua string), "display_name" is the agent display name, "maturity" is the agent maturity level (a Lua string containing either "adult", "mature", "teen" or "unknown"), "active_group_id" is the UUID (as a Lua string) of the current active group (null UUID if no active group set), "camera_mode" is a Lua integer reflecting the current camera mode (see Appendix D, Camera modes), "control_flags" is a Lua integer reflecting the current control flags (see Appendix D, Agent control flags), "occupation" is a Lua number (integer) reflecting the current AFK/Busy/Auto-reply state/mode for your agent (0=none, 1=AFK, 2=Busy, 3=Auto-reply), "flying" is a Lua boolean reflecting whether the agent is flying (*true*) or not (*false*), "sitting" is a Lua boolean reflecting whether the agent is sitting (*true*) or not (*false*), "sitting_on_ground" is a Lua boolean reflecting whether the agent is sitting on ground instead of on an object (*true*) or not (*false*), "baked" is a Lua boolean reflecting whether the agent is fully baked (*true*) or not (*false*), "can_rebake_region" is a Lua boolean reflecting whether the agent has permission to re-bake the current region path-finding navmesh (*true*) or not (*false*), "rlv" is a Lua boolean reflecting whether the RestrainedLove features are enabled (*true*) or not (*false*), "restrictions" (present only when "rlv" is *true*) is a Lua string containing a comma-separated list of the restrictions the agent is currently the subject of (note that these are not detailed restrictions, but just the type; e.g. 'detach' is listed (once only) for any number of locked attachments, without info about what attachment is locked). The following "agent benefits" data is also present (each key being associated with the corresponding number as a Lua integer): "animation_upload_cost", "sound_upload_cost", "texture_upload_cost", "max_upload_cost" (not really part of the benefits, but set to the maximum of the three former listed costs), "create_group_cost", "picks_limit", "group_membership_limit", "attachment_limit" and "animated_object_limit".
  NOTE: when called prior to login, this function only returns the "name" entry in the table, and its associated value corresponds to the agent name for the last session.

- SetAgentOccupation(mode) allows to set your agent occupation, with mode=0 to reset to "none" (present/available), 1 to set "AFK", 2 to set "Busy" and 3 to enable "Auto-reply".

- GetAgentPushes() returns a Lua string containing the statistics for all the bumps, pushes and hits the agent was the victim of (empty string if none).

- IsAvatar(id) returns *true* if 'id' indeed corresponds to an avatar currently present in the viewer's avatars list (i.e. an avatar present in the current sim or nearby simulators), or *false* otherwise.

- IsAgentFriend(id|name) returns two booleans, the first being *true* if 'id' (or 'name') corresponds to an avatar part of the users' friends list, or *false* otherwise, the second boolean being *true* if the avatar is indeed a friend and currently online, or *false* otherwise. For 'name' both the legacy name (with 'Resident' last name made optional) and the "Display Name [Legacy Name]" name formats are accepted.

- GetAgentFriends([name_type]) returns the list of the agent’s friends as a Lua table with the friends UUIDs (Lua string type) as keys and their name as values; the returned name may be either the legacy name (when 'name_type' is 0 or omitted), the legacy name followed with the display name (if any) between square brackets (when 'name_type' is 1), the display name alone (replaced with the legacy name when no display name is set for a friend, when 'name_type' is 2), or the name as displayed in the Friends floater (when 'name_type' is -1), i.e. obeying the user settings for such names. This function is only available from trusted scripts only.

- GetAvatarName(avatar_id[, name_type]) returns the name of the avatar corresponding to the 'avatar_id' UUID (Lua string type), if currently cached in the viewer, and when not yet known, triggers a request on the sim server for that name (by calling that function again a little later, such as via the CallbackAfter() function, you will then get the name returned). When the avatar name is unknown (or when the passed UUID is not an avatar id), an empty string is returned. The optional 'name_type' is a Lua integer, defaulting to 0: 0 causes the function to return the avatar legacy name, 1 to return the display name, 2 to return the both names in the form "Legacy Name [Display Name]" (or just "Legacy Name" is no display name is set for that avatar).

- IsAdmin(avatar) returns *true* whenever 'avatar' (Lua string type) corresponds to the (legacy) name of a grid administrator (i.e. any avatar with a "Linden" second name in SL, and any avatar name flagged as administrator in OpenSim) or the UUID of a grid admin, or *false* otherwise. Whenever an UUID is passed in 'avatar' and either the UUID is not an avatar's one, or the avatar name cache does not yet contains that avatar UUID, *nil* is returned instead (in the latter case, a name cache request gets queued for that UUID, and you may retry IsAdmin() later to get the result).

- SetAvatarMinimapColor(avatar_id[, color]) displays the mini-map avatar dot using 'color' (which must be a Lua string containing either the name of a known predefined color, or as a "R,G,B" triplet, with R, G and B as floats in the [0.0, 1.0] range for Red, Green, Blue) for 'avatar_id'. If 'color' is an empty string or omitted, the default color (yellow for friends, green for all other avatars) is used for that avatar. This function returns a Lua boolean which is *true* whenever the operation succeeded (avatar rezzed and color set for it) or *false* otherwise (avatar not found/not rezzed, or avatar_id is self).

- SetAvatarNameTagColor((avatar_id[, color]) displays the avatar tag text using 'color' (which must be a Lua string containing either the name of a known predefined color, or as a "R,G,B" triplet, with R, G and B as floats in the [0.0, 1.0] range for Red, Green, Blue) for 'avatar_id'. If 'color' is an empty string or omitted, the default color is used for that avatar. This function returns a Lua boolean which is *true* whenever the operation succeeded (avatar rezzed and color set for it) or *false* otherwise (avatar not found/not rezzed).

Group related functions

- IsAgentGroup(id) returns two booleans, the first being *true* if 'id' corresponds to a group to which the user belongs, or *false* otherwise, the second boolean being *true* if it is the user's current active group (i.e. one of its group titles is set on the agent), or *false* otherwise.

- GetAgentGroups() returns the list of the agent's groups as a Lua table, with group UUIDs (Lua string type) as keys and the group names as values. This function is only available from trusted scripts only.

- GetGroupName(group_id) returns the name of the group corresponding to the 'group_id' UUID (Lua string type), if currently cached in the viewer, and when not yet known, triggers a request on the sim server for that name (by calling that function again a little later, such as via the CallbackAfter() function, you will then get the name returned). When the group name is unknown (or when the passed UUID is not a group id), an empty string is returned.

- GetAgentGroupData([group]) returns a Lua table containing information about a 'group' (Lua string evaluating to either a group UUID or a (case sensitive) group name) to which the agent pertains as a member. if "group" is omitted, then the agent's current active group is used. The Lua table contains the following "key"/value pairs: "group_id" contains the group Id (as a Lua string) or a null UUID if the function could not find the group, if the group is "none" (agent current active group not set), or if the group data was not yet available when it was invoked (a CallbackAfter() could then be used to retry it), "name" contains the name of the group ("none" if a query was made for the agent current active group and the latter was not set), "active" is a Lua boolean which is *true* when the group is the agent's current active group or *false* otherwise, "powers" is a Lua integer reflecting the agent powers in that group (see Appendix D, Group powers).
  When the group was actually found and its data is available the table also contains: "founder_id" is the avatar UUID (as a Lua string) of the group founder, "insignia_id" contains the UUID (as a Lua string) for the group insignia (image), "contribution" is a Lua integer reflecting the agent's contribution to the group land (in square meters), "in_profile" is a Lua boolean which is *true* when the group is listed in the agent's profile or *false* otherwise, "accept_notices" is a Lua boolean which is *true* when the agent accepts notices from the group or *false* otherwise, "chat_muted" is a Lua boolean which is *true* when the agent muted the group chat or *false* otherwise, "charter" is a Lua string containing the group charter, "open_enrollment" is a Lua boolean which is *true* when anyone can freely join the group or *false* for an invite-only group, "fee" is a Lua integer reflecting the membership fee, "member_count" is a Lua integer reflecting the number of member in the group, "mature" is a Lua boolean which is *true* when the group is rated (at least) mature or *false* for a PG group, "members_list_ok" is a Lua boolean which is *true* when the group members list has been received, "roles_list_ok" is a Lua boolean which is *true* when the group roles list has been received, "properties_ok" is a Lua boolean which is *true* when the group properties have been received, "group_titles_ok" is a Lua boolean which is *true* when the group titles have been received.
  Finally, for each title (AKA role) in the group (still when found and with its data available), one table entry appears which key is the title/role UUID (as a Lua string) and which value is the title name (Lua string). If the group is active, then the active title also has its name and UUID added to the table under (respectively) the "current_title_name" and "current_title_id" keys.

- SetAgentGroup([group, [title]]) attempts to set the agent's active group to 'group' (Lua string, group UUID or case sensitive group name, "none" when omitted), optionally with 'title' role (Lua string, role UUID or case sensitive group title, no change to group role/title when omitted). Returns *true* on success, *false* on failure (invalid group, invalid title, RestrainedLove restriction on group change...), or *nil* when the roles data for this group is not yet available and a title change was requested (in this case, a data request is sent to the server and an asynchronous attempt at changing the group title will be made after the data is received).

- AgentGroupInvite(avatar_id|avatars_id_table, group_id[, role_id]) attempts to invite avatars to a group which Id is 'group_id' with the role corresponding to 'role_id' (or to the default "Member" role when omitted or a null UUID is passed). To invite just one avatar, pass their UUID as 'avatar_id'; to invite several (up to 100 at once; if you try to invite more, the list will get truncated), pass their UUIDs in a 'avatars_id_table' Lua table. All UUIDs are to be passed as Lua strings. The function returns the actual number of invitations sent (or 0 on failure, e.g. if the group Id is wrong, or when you cannot invite members to that group for that role). The function is "clever enough" to skip sending the invitation whenever the invited avatar already pertains to the group with the same role (but will still report a success in that case). Finally, note that this function may fail whenever the group data has not yet been received: it is therefore recommended to try and fetch info at least once in the session for that group (e.g. using GetAgentGroupData()) before you use this function to invite people to it.

Chatting and instant messaging related functions

- SetAgentTyping([true|false]) sends an "agent typing" event to the server when no parameter or *true* is used as an argument, or an "agent stopped typing" event when *false* is used as an argument.

- SendChat(text[, type]) sends 'text' (string or string convertible type) in chat, as a "say" (normal, chat range), without animating your avatar with the corresponding chat anim (head nodding for normal chat range). With the string variable/parameter 'type', you may specify different chat type. E.g. "whisper animate" will cause your avatar to whisper and play the whispering anim. Recognized keywords for "type" are: "whisper", "normal", "shout" and "animate" (case sensitive !).
  NOTE: SendChat() may be used to send text to scripts on non-zero channels (i.e. not on main chat), simply by prefixing the text with "/channel_number", just like what you would do in the chat bar (e.g. "/-1 This is sent to channel -1 and any scripted object listening to your avatar on that channel").

- GetIMSession(target_id) returns the session id for the IM session opened for 'target_id' (string type), which may be an avatar or group id, and if such a session does not exist, it is created/opened. You should let some time (a few seconds) before using the session id in a SendIM() (see below), so that the viewer gets a chance to open and connect to the session (especially for group session). You may achieve such a delay via the CallbackAfter() function (see below).

- CloseIMSession(session_id[, snooze_time]) is only available from the trusted scripts (for security reasons) and, when invoked, attempts to close the IM session corresponding to 'session_id' (UUID pased as a Lua string type), if any such session exists and is still active. For group IM sessions, an optional 'snooze_time' parameter (Lua integer, time in minutes) may be passed that, when greater than zero, will cause the session to be temporarilly closed for the corresponding duration instead of having a "leave session" event sent to the server (which would close it until next login).

- SendIM(session_id, text) sends 'text' (string or string convertible type) to the IM session which id is held in 'session_id' (string type).

Muting and blocking related functions

- AddMute(name_or_id[, type[, mute_flags]]) attempts to add a mute (AKA "block" in v2+ viewers), using 'name_or_id' to designate/find the object (for either per-id or named mutes), avatar or group (for per-id mutes only). 'type' (0 if not passed) can be used to specify what type of asset is to be muted (0 to mute objects by name, 1 to mute avatars by Id, 2 to mute objects by Id, 3 to mute groups by Id), and 'mute_flags' to specify what is to be muted (valid only for avatars and groups, objects being always fully muted): 1 for text chat, 2 for voice chat, 4 for particles, 8 for sounds (the default, when no 'mute_flags' value is passed, being to perform a "full mute", which includes a rendering mute for avatars); the 'mute flags' may be cumulated (added) to mute, for example, sounds and particles with 12 as the value. This function returns *false* in case of error (invalid parameters or unknown object Id) and *true* otherwise (which does not mean the mute will succeed, since it is an asynchronous operation and, for example, an invalid avatar or group Id would result in a mute failure.
  NOTE: unlike avatars (which may not be around neither online when the mute request is issued) and groups, objects can only be muted by Id if they are currently loaded in the objects list of the viewer (i.e. if they are rezzed).

- RemoveMute(name_or_id[, type[, mute_flags]]) attempts to remove a mute from the mute list. The parameters and return value are the same as for AddMute() above.

- IsMuted(name_or_id[, type[, mute_flags]]) takes the same parameters as for AddMute() above and returns *true* if the mute is in force, or *false* otherwise.

- SetVisualMute(avatar_id, level) sets the visual muting level (AKA "dollification") of the avatar corresponding to 'avatar_id' (UUID, passed as a Lua string type). The 'level' (Lua number) is either 0 to switch or revert to automatic dollification (based on avatar render complexity settings), 1 to always dollify the avatar, or 2 to never dollify it (whatever is render complexity level).

- BlockSound(sound_id[, true|false]) adds 'sound_id' to the list of the blocked sounds (i.e. to the list of sounds that are prevented from playing), or remove it when *false* is passed as the second parameter. This function does not return anything.
  NOTE: the blocked sounds list is not persistent. It is forgotten at the end of the viewer session.

- IsBlockedSound(sound_id) returns *true* if sound_id is currently blocked, or *false* otherwise.

- GetBlockedSounds() returns a Lua array containing the list of the UUIDs of the blocked sounds (empty array if none).

Radar related functions

- GetRadarList([name_type]) returns the list of the avatars listed in the Radar floater when either opened or configured to update in the background, or *nil* when the floater is closed and the radar inactive. The list is returned as a Lua table with the avatars UUIDs (Lua string type) as keys and their name as values; the returned name may be either the legacy name (when 'name_type' is 0 or omitted), the legacy name followed with the display name (if any) between square brackets (when 'name_type' is 1), the display name alone (replaced with the legacy name when no display name is set for a friend, when 'name_type' is 2), or the name as displayed in the Radar floater (when 'name_type' is -1), i.e. obeying the user settings for such names.

- GetRadarData(avatar_id) accepts an avatar UUID (Lua string type) as a parameter and returns either *nil* (Radar not active, or no such avatar around, or the entry for that avatar is flagged as dead and about to be removed) or a Lua table containing the following keys and associated values: "id" key with the avatar UUID as a string, "name" key, with the avatar legacy name as a string, "display_name" key, with the name of the avatar as listed in the Radar (the format of the name depending on the Radar settings) as a string, "name_color" key with a string containing the "R, G, B" triplet of floats (each from 0.0 to 1.0) for the name color, "global_x" and "global_y" keys, each associated with a number reflecting the avatar position on the grid, "altitude" key associated with the number reflecting the avatar altitude, "friend" key with a boolean which is *true* if the avatar is one of your friends, "muted" key with a boolean which is *true* if for a muted avatar, "derendered" key with a boolean which is *true* if for a derendered avatar, "marked" key with a boolean which is *true* if the avatar is flagged as marked in the Radar, or *false* otherwise, "mark_char" key with a string containing the marker character(s) in use, "mark_color" key with a string containing the "R, G, B" triplet of floats (each from 0.0 to 1.0) for the mark color, "focused" key with a boolean which is *true* if the camera is focused on that avatar via the Radar, or *false* otherwise, "drawn" key with a boolean which is *true* if the avatar is within draw distance, or *false* otherwise, "in_sim" key which is *true* if the avatar is present in the same sim as yours, or *false* otherwise, and "entry_age" key with a number reflecting the age of the entry as a number of seconds since last refresh by the Radar.

- SetRadarNameColor(avatar_id[, color]) accepts an avatar UUID (Lua string type) and displays the avatar name in the radar list using 'color' (which must be a Lua string containing either the name of a known predefined color, or as a "R,G,B" triplet, with R, G and B as floats in the [0.0, 1.0] range for Red, Green, Blue) for 'avatar_id'. If 'color' is an empty string or omitted, the black color is used for that avatar. Please note that the "green3", "red2" and “orange4” colors are already natively used by the radar code (respectively for friends, muted and derendered avatars); starting with v1.30.0.17 these colors are however defined in colors_base.xml instead of being hard-coded, meaning you may change them with SetDebugSetting(). This function returns a Lua boolean which is *true* whenever the operation succeeded (avatar rezzed and color set for it) or *false* otherwise (avatar not found/not rezzed).

- SetRadarToolTip(avatar_id[, text]) adds a tool tip 'text' (Lua string type) to the radar list entry corresponding to the 'avatar_id' UUID (Lua string type). When text is omitted or empty, the standard tool tips are restored for this avatar. This function returns a boolean which is *true* whenever the operation succeeded or *false* otherwise (avatar not found/not rezzed).

- SetRadarMarkChar(avatar_id[, character]) accepts an avatar UUID (Lua string type) and displays in the corresponding marker column for that avatar the specified 'character' (Lua string type and up to three bytes long, that is up to three ASCII characters or one UTF-8 special character encoded in the form "\u{XXYY}" with XX and YY the hexadecimal numbers for that UTF-8 character). Note that the mark/character is displayed, whether the avatar is already Marked or not (i.e. this function overrides the marked status for the display in the Radar floater, but that not actually change the marked status for that avatar). When 'character' is empty or omitted, this special mark is cleared (and the standard “X” mark is then used again, when the avatar is actually marked). This function returns a boolean which is *true* whenever the mark is successfully (re)set or *false* otherwise (avatar not found/not rezzed).

- SetRadarMarkColor(avatar_id[, color]) accepts an avatar UUID (Lua string type) and displays the mark in the radar list using 'color' (which must be a Lua string containing either the name of a known predefined color, or as a "R,G,B" triplet, with R, G and B as floats in the [0.0, 1.0] range for Red, Green, Blue) for 'avatar_id'. If 'color' is an empty string or omitted, the default mark color (blue) is used for that avatar. This function returns a boolean which is *true* whenever the color is successfully (re)set or *false* otherwise (avatar not found/not rezzed).

- SetRadarTracking(avatar_id[, true|false]) accepts an avatar UUID (Lua string type) and an optional boolean (defaulting to *false* when omitted) and attempts to setup the tracking of the corresponding avatar via the Radar. Returns a *true* boolean if successful or *false* otherwise (Radar not active or avatar not around). Passing a *true* boolean to the function causes it to force the activation of the Radar (with background refresh and tracking) if it was not active.

Camera related functions

- GetCameraMode() returns the current camera mode (see Appendix D, Camera modes).

- SetCameraMode(mode[, animate]) attempts to set the camera 'mode' (1 for mouse-look, 0 for third person, -1 for default, -2 for "reset camera view" action which is equivalent to pressing SHIFT ESC) and returns *false* if changing the mode failed (invalid/forbidden mode or because of RestrainedLove restrictions), *true* otherwise, which does not mean the camera mode actually changed either; e.g. restoring to the "default" mode could fail if the former mode was, for example a follow-cam on an object that has since vanished. Note that invoking this function does not trigger the OnCameraModeChange() callback (to avoid potential infinite recursions). When 'animate' is passed and *false*, the camera does not animate while the mode changes (it does animate by default).

- SetCameraFocus([object_id]) attempts to set the camera focus on an object or avatar corresponding to the 'object_id' UUID (Lua string), or on the agent's avatar when that parameter is omitted or empty. It does not return any value. Note that invoking this function does not trigger the OnCameraModeChange() callback (to avoid potential infinite recursions), even if the camera mode does change as a result of focusing on the new object.

Grid, simulator, parcel and position related functions

- GetGridSimAndPos() returns a Lua table with the following "key"/value pairs: the "grid" name (string value), the "region" name (string value), the simulator server "version" for that region (Lua string value), the region "width" (Lua number value) in meters (always 256 in SL, maybe more in OpenSim, for "var regions"), the "water_height" (Lua number value) in meters, the region "flags" (Lua integer value, see Appendix D, Region flags), the amount of "neighbors" regions (Lua integer value from 0 to 8), the avatar global position as "global_x" and "global_y" (Lua number values) in meters, the avatar local position in the region as "local_x" and "local_y" (Lua number values) in meters, the avatar "altitude" (Lua number value) in meters, "navmesh" (string value) which is either "none" (no path-finding feature available), "dirty" (when the navmesh is dirty and needs a rebake), "re-baking" (when the navmesh is currently being re-baked), "enabled" (when the path-finding feature is enabled and the navmesh up to date) or "disabled" (when the path-finding feature is available but disabled).
  NOTE: the "neighbors" number may take some time to settle after a TP, since connections to neighboring sims are only performed after the TP is finished. You might therefore want to use GetGridSimAndPos() via CallbackAfter() if this number is to be checked after TPs. See the example script below for such a usage.

- GetParcelInfo() returns a Lua table with the following "key"/value pairs: the "name" (string value) of the parcel the agent is currently in, its "description" (string value), its "flags" (integer value, see Appendix D, parcel flags), a "build" Lua boolean which is *true* when the agent can build on the parcel, a "damage" Lua boolean which is *true* when the parcel got agent damage enabled, a "fly" Lua boolean which is *true* when flying over the parcel is permitted, a "push" Lua boolean which is *true* when scripted pushing is permitted, a "see" Lua boolean which is *true* when agents inside the parcel can see and chat with agents outside it, a "voice" Lua boolean which is *true* when voice chat is available.

Agent movements controls related functions

- AgentSit([object_id]) attempts to sit your avatar down on object which UUID is 'object_id' (string type) or, when object_id is omitted, on the ground. Returns a boolean which is true on success or false on failure (object not found, sit restriction by RestrainedLove...).

- AgentStand() causes your avatar to stand up when sat. Returns a boolean which is true on success or false on failure (sit restriction by RestrainedLove...).

- AgentRotate(heading) rotates the agent's avatar to face 'heading' degrees (Lua number), with 0 for North, 90 for East, etc...

- GetAgentRotation() returns the 'heading' the agent's avatar is facing, in degrees (Lua number), with 0 for North, 90 for East, etc...

- TeleportAgentHome() teleports the agent (your avatar) home.

- TeleportAgentToPos(x, y[, z[, preserve_look_at]]|vec_pos_str[, preserve_look_at]) takes global grid coordinates as either three Lua numbers (the altitude, 'z', may be omitted) or as a Lua string containing the coordinates in the form of three floating point numbers separated from each other with a space (such strings are returned as table elements by GetAgentPosHistory()), and attempts to teleport the agent (your avatar) over there. If a boolean is passed as the 'preserve_look_at' last optional parameter (in the case of number coordinates usage, 'z' must then be passed too, as 0 if unknown), then the teleport will preserve the departure "look at" direction if that boolean is *true*. This function does not return anything.

- GetAgentPosHistory() returns the position history for the agent as a Lua table of strings containing the agent global coordinates in the form of three floating point numbers separated from each other with a space. The table size is limited by the "LuaMaxAgentPosHistorySize" debug setting value (12 by default, which should cover at least the past 2 minutes of the agent’s movements) and the last stored agent position appears first (i.e. at index 1) in the Lua table. Should the "LuaMaxAgentPosHistorySize" debug setting be reduced to 0, no position would ever be stored and *nil* is returned by the function instead of a Lua table. The positions are added to the history at most once every 10 seconds and on the condition that the new position is at least 3m away from the last stored position (i.e. the same constraints as for the OnPositionChange() Lua callback). Note that the last position of the agent is stored in the history after the OnPositionChange() Lua callback is invoked since that callback already receives the last position: this way you have one more usable history entry and the former valid position is at index 1 in the table (useful when you just got ejected from a parcel, for example).

- AgentAutoPilotToPos(global_pos_x, global_pos_y[, altitude[, allow_flying[, stop_distance]]]) triggers an auto-pilot action to the target global coordinates <global_pos_x, global_pos_y, altitude>. When 'altitude' is omitted, the agent's current altitude is used. 'allow_flying' is an optional boolean (defaulting to *false*) to permit flying in order to reach the target. 'stop_distance' is a Lua number (defaulting to 0.0) which specifies how close the agent must get to the target to consider it did reach it. The triggered auto-pilot action will have a "Lua auto-pilot N" 'reason' (see OnAutoPilotFinished() below) attached (with N an integer incremented at each such new auto-pilot action) and this function returns that reason as a Lua string. Note that this function automatically cancels and overrides any already ongoing auto-pilot action.

- AgentAutoPilotFollow(target_object_id[, allow_flying[, stop_distance]]) triggers an auto-pilot action to follow (and if possible reach) the target object which UUID is 'target_object_id' (Lua string); that object may also be an avatar. 'allow_flying' is an optional boolean (defaulting to *false*) to permit flying in order to reach the target. 'stop_distance' is a Lua number (defaulting to 0.0) which specifies how close the agent must get to the target to consider it did reach it. The triggered auto-pilot action will get the UUID string attached as a 'reason' (see OnAutoPilotFinished() below). This function returns *true* when successful and *false* when it failed (when the target object was not found, or when another auto-pilot action was already in progress).

- AgentAutoPilotStop() stops any ongoing auto-pilot action, if any. Does not return anything.

- AgentAutoPilotLoad([filename]) attempts to load an auto-pilot recorder path file for future replay. If the 'filename' Lua string is omitted, the name specified by the "AutoPilotFile" debug setting is used. When not present in 'filename', the ".plt" extension is automatically added. The file is always searched for in the per-account user settings directory. This function returns *true* when successful and *false* when it failed (not yet logged in, file not found, recording or replay in progress).

- AgentAutoPilotSave([filename]) attempts to save the current auto-pilot recorder path data into a file. If the 'filename' Lua string is omitted, the name specified by the "AutoPilotFile" debug setting is used. When not present in 'filename', the ".plt" extension is automatically added. The file is always created inside the per-account user settings directory. This function returns *true* when successful and *false* when it failed (not yet logged in, no data recorded, file not writable, recording or replay in progress).

- AgentAutoPilotRemove([filename]) attempts to delete an auto-pilot recorder path file. If the 'filename' Lua string is omitted, the name specified by the "AutoPilotFile" debug setting is used. When not present in 'filename', the ".plt" extension is automatically added. The file is always searched for in the per-account user settings directory. This function does not return anything.

- AgentAutoPilotRecord(record) attempts to begin (when the 'record' Lua boolean is *true*) a new path recording, or ends (when the 'record' is *false*) any ongoing record. This function returns *true* when successful and *false* when it failed (not yet logged in, or recording or replay already in progress when attempting to begin a new record, or no recording in progress when attempting to end one).

- AgentAutoPilotReplay(replay[, runs[, allow_flying]]) attempts to replay a record (when the 'replay' Lua boolean is *true*), or to stop an ongoing replay (when the 'replay' is *false*). When starting a replay, the function accepts one or two more parameters, namely the number of 'runs' (defaulting to -1, i.e. continuous looping) and an 'allow_flying' boolean (defaulting to *false*). This function returns *true* when successful and *false* when it failed (not yet logged in, no recorded data, or recording or replay already in progress when attempting to start a replay, or no replay in progress when attempting to stop one).

Agent puppetry related functions

Note 1: these functions were made available in Cool VL Viewer v1.30.1.1 and newer versions, and only from the trusted non-threading scripts.
Note 2: these functions have been removed from released binaries (but may still be enabled in custom builds) starting with the Cool VL Viewer v1.32.0.24 (stable branch) and v1.32.1.7 (experimental branch) since Linden Lab announced that the Puppetry project has been withdrawn.

- AgentPuppetryStart([command|[filename[, pass_camera_number]]]) attempts to start an agent puppetry plugin. It returns *true* when successful or *false* otherwise (no puppetry available in sim, or disabled by the user, bad command or file name, failed plugin launch, or a plugin was already running). When not passed any parameter, it attempts to relaunch the last successfully launched puppetry plugin command. When passed a 'command' (Lua string), it attempts to launch that command (a command consists in a filename with the full path to the script or executable file and any needed options); when passed a 'filename' (full path only), it attempts to use the corresponding file as a plugin, passing it, when the 'pass_camera_number' boolean (defaulting to *false*) is *true*, the camera option (as defined by the "PuppetryCameraOption" debug setting) and the current camera number (as defined by the "PuppetryCamera"debug setting).

- AgentPuppetryStop() stops any running puppetry plugin. Does not return anything.

Agent inventory related functions

- FindInventoryObject(item) accepts an 'item' string parameter containing the full path (case sensitive,with "|" used as folders separator) of an item in the inventory, or an item UUID, and returns a Lua table containing the following keys and associated values: "id" key with the item UUID as a string value (a null UUID if the item was not found), "name" key, with the item name as a string value (the "item" string parameter contents is returned here if the item was not found), "type" key with the item asset type (see Appendix D, Asset types), as a value (-1 = AT_NONE if the item was not found), such as 8 (AT_CATEGORY) for a folder ("category" in LL's viewer code jargon), and three (or four in OpenSim grids with export permission support) boolean values with keys "copy_ok", "mod_ok" and "xfer_ok" (and "export_ok" for OpenSim grids with export permission support) to reflect the current owner (your avatar) permissions.

- GiveInventory(avatar_id, inventory_item_path_and_name) attempts to give an inventory item to the resident corresponding to 'avatar_id'. The inventory item may be a single item (object, note-card, texture, etc) or an inventory folder. The 'inventory_item_path_and_name' parameter must represent the full path (case sensitive, with "|" used as folders separator) of the item in the inventory. The usual restrictions (item permissions with confirmation dialog for no-copy items, number of item in given folder, RestrainedLove restrictions...) apply. Returns *true* on "success" (this is only the success to find the corresponding inventory item and send the give order to the server: what happens after the order is given, such as a failure to deliver the item or a decline from the recipient, are not known when this function returns to the Lua script), *false* otherwise.

- MakeInventoryLink(item_to_link, new_link_parent_folder) takes the full path of the item to link and of the folder where to create the corresponding link (case sensitive, with "|" used as folders separator), and creates the said link. It returns a boolean: *false* on failure (bad paths, bad item type since not every item type may be linked) and *true* on "success" (this is only the success to send the link creation request to the server after validating every requirement, but the asset server may still fail to create the link in case of lag or usual SL inventory issues...).

- DeleteInventoryLink(link_item) takes the full path of an existing link item and deletes it if found. Returns a boolean: *false* on failure (link item not found) and *true* on "success" (see MakeInventoryLink() above for the actual meaning of "success").

- NewInventoryFolder(parent_folder_path, new_folder_name) takes the full path where to create the new folder plus a name for the new folder and creates the latter. Returns a boolean: *false* on failure (parent folder not found) and *true* on "success" (see MakeInventoryLink() above for the actual meaning of "success").

- ListInventoryFolder(path) takes a full 'path' for the folder to list the contents from, and returns either *nil* (folder not found) or a Lua table containing key/value pairs with inventory items UUIDs as the keys and their name as the values; the names of folders are post-fixed with a trailing "|" (path separator) so to easily distinguish them from non-folder items and make it easy to concatenate their name with item names.

- MoveToInventoryFolder(dest_folder, inv_object|inv_table) takes a full 'dest_folder' path or UUID for the destination folder (which may be any existing folder with the exceptions of the “Trash”, the “Current Outfit” and “Marketplace Listings” ones), as well as either the 'inv_object' full path or UUID for the inventory object (item or folder) to move, or an 'inv_table' Lua table which may be either a list of the inventory objects full paths or UUIDs, or an associative array with the inventory objects full paths or UUIDs used as keys (such a an associative array is returned by ListInventoryFolder()). The function returns *true* on success (request to move the inventory objects sent to the server), or *false* otherwise (request not sent, due to missing or forbidden destination folder, or a missing inventory objects).

Agent outfit related functions

- GetAgentAttachments([string_match]) takes an optional 'string_match' string (case insensitive) and scans the agent avatar's attachments, returning a table containing all the (matching by UUID, (part) name or joint name) attachments with their UUID as the key and a "inventory item name|joint name" string as the value for each attachment. Note that the joint names are the same as the ones appearing in the viewer UI (they are the lower-cased "translated" names, as listed in skins/default/xui/en-us/strings.xml): e.g. "Right Ring Finger" is returned as "r ring finger". Temporary attachments appear in the list with a special "temp_attachment:<uuid>" name where *<uuid>* is the UUID of that attachment (which corresponds to the UUID of the attachment object for temp attachments, instead of the UUID of the inventory item for normal attachments).

- GetAgentWearables([string_match]) takes an optional 'string_match' string (case insensitive) and scans the agent avatar's body parts (shape, eyes, skin, hair) and clothing layers, returning a table containing all the (matching by UUID, (part) name or wearable type name) wearables with their UUID as the key and a "inventory item name|type name" string as the value for each wearable. Note that the wearable type names are the same as the ones appearing in the viewer UI (they are the lower-cased "translated" names, as listed in skins/default/xui/en-us/strings.xml).

Windlight and extended environment related functions

- SetDayTime(percent) sets the environment to use a fixed time or the region/parcel time (depending on what renderer is in use); when 'percent" (Lua number) is comprised between 0.0 and 1.0, a fixed time is used (with 0.0=00:00 and 1.0=24:00) and when it is negative (or greater than 1.0), the region time (in WL rendering mode) or the parcel environment and time (in EE rendering mode or WL overriding) are used. This function does not return anything.
  NOTE: if called before login happens, this function does nothing (since environment is not yet enabled at this point).
  NOTE 2: this function replaces UseRegionTime() which was removed from the Cool VL Viewer v1.28.0.1.

- ApplyDaySettings(settings_name) attempts to load and apply Windlight day settings 'settings_name' (string value, case sensitive); the search is first made for EE settings in the "Settings" inventory folder, then in Windlight settings. This function returns *true* when successful and *false* when it failed to find and load the settings. The following settings names got a special meaning (they are equivalent to using the entries for the presets in the "World" -> "Environment settings" sub-menu): "sunrise", "midday" (or "noon"), “noon_pbr” (as of v1.32.0.17), "sunset" and "midnight" set the region time to the corresponding fixed time, "animate" causes the time of day to get animated again (e.g. after "midday" was applied), "region" only works in Windlight rendering mode (would cause the function to return *false* in EE rendering mode) and sets the time of day to the estate/region one. The "parcel" pseudo-setting name turns on the EE parcel environment to Windlight translation on, with parcel-time. The "windlight" pseudo-setting name turns off parcel and local environment settings selection.
  NOTE: if called before login happens, this function does nothing (since environment is not yet enabled at this point) and returns *false*.

- ApplySkySettings(settings_name) attempts to load and apply sky settings 'settings_name' (string value, case sensitive); the search is first made for EE settings in the "Settings" inventory folder, then in Windlight settings. This function returns *true* when successful and *false* when it failed to find and load the settings.
  NOTE 1: if called before login happens, this function does nothing (since Windlight is not yet enabled at this point) and returns *false*.
  NOTE 2: this function implicitly freezes the region/parcel time.
  NOTE 3: before v1.28.0.0, this function accepted the special "sunrise", "noon", "sunset", "midnight", "region" and "parcel" settings names: they are now to be used with ApplyDaySettings().

- ApplyWaterSettings(settings_name) attempts to load and apply Windlight water settings 'settings_name' (string value, case sensitive); the search is first made for EE settings in the "Settings" inventory folder, then in Windlight settings. This function returns *true* when successful and *false* when it failed to find and load the settings.
  NOTE: if called before login happens, this function does nothing (since environment is not yet enabled at this point) and returns *false*.

- GetEESettingsList([type]) returns a Lua array containing the list of the environment settings of 'type' (Lua integer equal to 0 for skies, 1 for waters, 2 for days, anything else or no 'type' parameter for everything) present in the "Settings" folder or the inventory, or *nil* if no matching setting is found. Each key in the array is the name of the setting, and each value is a Lua string containing "sky", "water", "day", or any comma-separated list of those values (for settings with the same name but different types).

- GetWLSettingsList([type]) returns a Lua array containing the list of the Windlight settings of 'type' (Lua integer equal to 0 for skies, 1 for waters, 2 for days, anything else or no 'type' parameter for everything) present in the OS per-account user_settings/windlight/ directory or in the viewer installation app_settings/windlight/ sub-directory, or *nil* if no matching setting is found. Each key in the array is the name of the setting, and each value is a Lua string containing "sky", "water", "day", or any comma-separated list of those values (for settings with the same name but different types).

- GetEnvironmentStatus() returns a Lua array containing the following keys associated with Lua boolean values and reflecting the current environment configuration of the viewer: "enhanced rendering" (*true* when the EE renderer is active, *false* when the Windlight renderer is active), "windlight override" (*true* when the Windlight settings are overridden with EE settings for rendering), "local environment" (*true* when local EE settings are active), "parcel environment" (*true* when parcel EE settings are active), "region time" (*true* when the region (for WL settings) or parcel (for EE settings) time is active), and "rlv locked" (*true* when under @setenv restriction).

SLURL, URL and RestrainedLove commands dispatching functions

- ExecuteRLV(rlv_command_line) takes a 'rlv_command_line' string and attempts to execute it on behalf of the Agent's avatar object (RestrainedLove associates commands to their emitter object, which is normally not an avatar but a scripted object) when it is emitted from the automation script, a Lua thread, a Lua callback, a Lua command line entered from the chat input line, or from a D-Bus Lua message, or on behalf of the scripted object when ExecuteRLV() was sent via llOwnerSay() or llInstantMessage(llGetOwner(),...). The command line shall not have any "@" prefix and may contain several, comma-separated RLV commands (e.g.: "version=-1,remoutfit:shape=n"). Note that the executed commands can only add/complement the existing RLV restrictions (i.e. you cannot cancel RLV restrictions imposed by other objects with this feature; this is not a way to cheat RLV). The RLV commands are (with a few exceptions) queued and executed asynchronously, so they are normally executed after this call returns and the Lua script returns control to the viewer main loop. When the "RestrainedLoveLuaNoBlacklist" debug setting is TRUE (toggle in "Advanced" -> "RestrainedLove" menu) and ExecuteRLV() is executed on behalf of the Agent (i.e. when the function call is not part of a command line sent by a script), it bypasses the RestrainedLove commands blacklist; this allows to execute commands via RestrainedLove that you would otherwise refuse when emitted directly by scripted attachments or via a relay.

- BrowseToURL(url[, browser_type]) loads 'url' in the Web browser. When 'browser_type' is "0" or omitted, the built-in or external (system) browser is used according to the viewer preferences (as set in "Preferences" -> "Network & Web"), if it is "1" then the built-in browser is used, and if it is 2 the external (system) browser is used.

- DispatchSLURL(slurl[, trusted]) causes the viewer to "dispatch" (i.e. decode and act upon) the SLURL passed as the 'slurl' parameter (string type), as a "non-trusted" one (default or when 'trusted' evaluates to *false*), or as a trusted one when 'trusted' evaluates to a *true* boolean. This is a very powerful command allowing to display all sorts of floaters (landmark info, avatar info, group info, experiences info, inspect object/avatars info, search, pay floater prompt, web pages displaying, etc, etc), and to trigger actions (start IM, TP, wear inventory folder, etc). Note that this function does not trigger a call to the OnSLURLDispatch() callback.

Networking functions

- GetHTTP(uri[, timeout[, accept]]) sends a GET HTTP request to the 'uri' (Lua string) with optional 'timeout' (Lua integer, in seconds) and optional 'accept' (Lua string) content type specification for the server reply (for example "application/json"). By default, the timeout is 60s and any content reply type ("*/*") is accepted. This function returns a handle in the form of a Lua integer. Whenever the HTTP request is replied (or times out), the OnHTTPReply() callback is called with the request handle, a 'success' boolean and the body of the server HTTP reply. This function is available only from the automation scripts and from Lua threads.

- PostHTTP(uri, data[, timeout[, accept[, content_type]]]) sends a POST HTTP request to the 'uri' (Lua string), sending 'data' (Lua string) to the server in the request body, with optional 'timeout' (Lua integer, in seconds), optional 'accept' (Lua string) content type specification for the server reply (for example "application/json"), and optional 'content_type' to tell the server what is the type of the posted 'data' (for example "application/json"). By default, the timeout is 60s, any content reply type ("*/*") is accepted, and the posted data type is "application/x-www-form-urlencoded". This function returns a handle in the form of a Lua integer. Whenever the HTTP request is replied (or times out), the OnHTTPReply() callback is called with the request handle, a 'success' boolean and the body of the server HTTP reply. This function is available only from the automation scripts and from Lua threads.

User interface related functions

- PlayUISound(sound_name[, force]) plays an UI sound (valid, case-sensitive, Lua strings for 'sound_name' are: Alert, BadKeystroke, Click, ClickRelease, HealthReductionF, HealthReductionM, InvalidOp, MoneyChangeDown, MoneyChangeUp, NewIncomingIMSession, ObjectCreate, ObjectDelete, ObjectRezIn, ObjectRezOut, PieMenuAppear, PieMenuHide, PieMenuSliceHighlight, Snapshot, StartIM, TeleportOut, Typing, WindowClose, WindowOpen). When 'force' (Lua boolean) is passed and *true*, the UI sound is played regardless of user preferences (as set in "Preferences" -> "Audio & media" -> "Audio") about allowed UI sounds.

- AlertDialogResponse(dialog_id, button_number) allows to reply to the alert dialog 'dialog_id' (UUID as a Lua string type), like if the button 'button_number' (the number of the button, from 1 onward, corresponding to the key value for this button entry in the array returned by the OnAlertDialog() callback). It returns a *true* boolean if successful (dialog still valid and button present in it) or *false* otherwise.

- NotificationResponse(dialog_id, btn_name) or ScriptDialogResponse(dialog_id, button_name) are synonymous functions to reply to the dialog 'dialog_id' (UUID as a Lua string type), like if the button 'button_name' got pressed. They return a *true* boolean if successful (dialog still valid and button present in it) or *false* otherwise. Note: NotificationResponse was added as a synonymous to ScriptDialogResponse in the Cool VL Viewer v1.30.2.32.

- GroupNotificationAcceptOffer(dialog_id) accepts any inventory offered via the group notification corresponding to 'dialog_id' (UUID as a Lua string type). It returns a *true* boolean if successful (group notification dialog still open and the notification does have an unsaved inventory item attached) or *false* otherwise. Note that when the inventory offer corresponds to a “viewable” item (for example a texture or a note card), a preview floater for that item is also automatically opened.

- OpenNotification(type, text) opens a notification to display 'text' (string or string convertible type). A 'type' (integer type or integer convertible type) of 0 opens a notify tip, 1 opens a pull-down notification, 2 opens a (non-modal) alert box. In order to prevent fake "official" notification attempts, pull-down notifications and notification tips got a green background and the alert boxes bear a "Lua alert" title.

- CancelNotification(notification_id) allows to cancel a notification which UUID (Lua string type) is notification_id. Returns a *true* boolean if successful (notification found and closed) or *false* otherwise (notification not found or already closed).

- OpenFloater(name[, target]) opens a specific floater with optional 'target' (an UUID or a number/index passed as a string) for floaters needing or supporting it. Currently recognized names are "active speakers", "area search", "avatar info" (needs an avatar id), "beacons", "camera controls", "chat", "debug settings", "experiences", "friends", "gestures", "group info" (needs a group id), "groups", "inspect" (needs an object or avatar id), "instant messages", "inventory", "land", "land holdings", "map", "media filter", "mini map", "mute list" (with optional avatar or object id), "nearby media", "notifications", path-finding "characters" and "linksets", "preferences" (takes an optional preferences tab index, from 0 to 11), "pushes", "radar", "region", "search", "snapshot", "sounds list", "stats" and "teleport history".
  Note that some floaters (e.g. "land", "region") may refuse to open whenever a RestrainedLove restriction forbids to display some info they would give up (in our example, when under @showloc restriction).

- CloseFloater(name) closes a specific floater. The names are the same as for OpenFloater().

- MakeDialog(title, text, suggestion, button1label, button2label, button3label, button1command, button2command, button3command) opens a dialog floater with 'title' used as its title, 'text' displayed in a text box (actually, an non-editable text editor with word-wrapping and URL parsing enabled), 'suggestion' stuffed in an input-line (when 'suggestion' is a single space, the input line is hidden, and when it is a single "*" the input line is setup as a password input), 'buttonNlabel' are text labels for each of the three buttons (when a label is empty, the button is hidden), and 'buttonNcommand' are Lua script commands associated with each button, with "GetDialogInput()", "SetDialogInput(text)" and "DialogClose()" commands added/recognized ("GetDialogInput()" returns the text contained in the input line, "SetDialogInput(text)" sets the dialog input text, and "DialogClose()" causes the dialog to be closed after the rest of the Lua command line is executed). Also, closing a Lua dialog opened with this command causes the OnLuaDialogClose() callback (see below) to be called when it exists in the automation script.
  NOTE: the buttonNcommand commands are executed in their own Lua context (AKA "state", in Lua jargon).

- OpenLuaFloater(floater_name[, parameter, [position [, show [, ctrl_value]]]]) tries and creates a custom floater which XUI definition should be held in an XML file placed in the cvlv-skins/default/xui/en-us/ sub-directory of the Second Life data folder ("~/.secondlife/" under Linux and "%appdata%\SecondLife\" under Windows).
  The 'floater_name' (Lua string) is used to form the XML file name of the floater definition ("floater_lua_<floater_name>.xml"); 'floater_name' can be anything but "dialog" and "console" which are reserved names (used by the MakeDialog() function and the Lua console). Starting with the Cool VL Viewer v1.32.2.28, the name may only contain alphanumeric (non accented) characters, dashes and underlines, as well as at most one pipe ("|" character) which is also used as a delimiter between the part of the name used to determine the XML file name (this part being at the left of the pipe character), while allowing to form several unique floater names: this allows to use the same floater XML definition to open simultaneously several different Lua floaters (for example, Lua "inventory" floaters rooted at different inventory folders: see note 4 below), since OpenLuaFloater() would refuse to open simultaneously two floaters with the same name.
  The optional 'parameter' (Lua string) is associated with the floater and passed to the Lua OnLuaFloaterOpen() and OnLuaFloaterClose() automation script callbacks (see below).
  The optional 'position' (Lua string) is used to place the floater on the screen and may be "center" (or an empty string), "top" (or "top-center"), "bottom" (or "bottom-center"), "left" (or "center-left"), "right" (or "center-right"), "top-left", "top-right", "bottom-left" or "bottom-right", and starting with the Cool VL Viewer v1.32.2.28, this parameter may also be coordinates (in "x,y" form, e.g. "100,50", "-50,50", "-10,-1", etc) with the first number being the x delta between either the left border of the screen and the left side of the floater when positive, or the right border of the screen and the right side of the floater when negative, and the second number being the y delta between either the top border of the 3D view (i.e. menu bar excluded) and the top of the floater when positive, or the bottom of the 3D view (i.e. tool bar and overlay buttons excluded) and the bottom side of the floater when negative. Finally, and still starting with v1.32.2.28, this parameter may be the name of a debug setting of the Rect type (any existing setting, such as "FloaterInventoryRect" for example, or one of the new Lua "LuaFloaterRect0", "LuaFloaterRect1", ... "LuaFloaterRect9" Rect settings added for this very purpose).
  The optional 'show' Lua boolean defaults to *true*; when *false*, it prevents the floater to be shown on creation (you will then have to use explicitly ShowLuaFloater() to actually open it), which may be used to setup all the UI elements via Lua before showing it.
  The optional parameter 'ctrl_value' (new in Cool VL Viewer v1.32.2.27) may be a Lua string or a Lua array of strings: each string must be in the form "ctrl_name=value" and would cause the UI control named "ctrl_name" to be set to "value", just like if you had used SetLuaFloaterValue(floater_name, ctrl_name, value); note that for UI controls such as lists, you may

use the same control name with several values in a ctrl_value array of strings (this way, you can initialize a full list when opening the floater), for example *{"list1=first row", "list1=second row", "list1=third row"}*.
This function returns a Lua boolean which is *true* on success and *false* on failure (missing XML file or invalid XUI definition in it).
NOTE 1: OpenLuaFloater() automatically sets the callbacks to the OnLuaFloaterAction() automation script function for buttons (named "buttonN" where N is a positive number), flyout buttons (named "flyoutN"), check boxes (named "checkN"), radio groups buttons (named "radioN"), combo boxes (named "comboN"), sliders (named "sliderN"), spinners (named "spinN"), line editors (named "lineeditN"), search editors (named "searcheditN"), text editors (named "texteditN"), lists (named "listN"), name lists (named "namelistN") and inventory panels (named "inventoryN"), as long as the first control of each type in the XUI definition is numbered "1" (N=1) and that all subsequent controls of the same type have consecutive numbers (N=2, 3, etc); that is, if you only have a "button2" control in your floater XUI definition, it will not get its callback automatically set, and if you have "button1", "button2" and "button4", then the latter will not have its callback set, unlike the former two.
NOTE 2: when this function succeeds and 'show' was omitted or *true*, the OnLuaFloaterOpen() automation script callback is called (when present).
NOTE 3: in v1.32.0.7, the support for a new "lua_command" XUI attribute has been added, which can be used to set an on-commit Lua command on the various controls cited in note 1, but regardless of their name format (i.e. such a command would also work if no OnLuaFloaterAction() callback is set for the corresponding UI control). Using "lua_command" in a control basically saves you from calling SetLuaFloaterCommand() for that control.
NOTE 4: in v1.32.2.27 an “inventory” Lua floater has been added to the distributed XUI definitions. It can therefore be used “as is” by people not knowing anything about XUI file format. For example:
OpenLuaFloater("inventory|objects", "", "", true, "inventory1=|Objects")
opens an inventory floater pointing at the root of the Objects folder in your inventory.
NOTE 5: in v1.32.2.27 the user’s skins override folder was renamed from “skins” to “cvlv-skins” to avoid collisions with other viewers. If you stored custom Lua floater XUI files in “skins”, make sure to copy or rename “skins” into “cvlv-skins”.

- ShowLuaFloater(floater_name[, show]) tries and shows (or hides when the optional 'show' Lua boolean is specified and *false*) the custom floater 'floater_name'. Returns a Lua boolean which is *true* on success and *false* on failure (no such open floater). When this function succeeds and 'show' was omitted or *true*, and the floater was not yet shown, the OnLuaFloaterOpen() automation script callback is called (when present).

- CloseLuaFloater(floater_name) tries and closes the custom floater 'floater_name'. Returns a Lua boolean which is *true* on success and *false* on failure (no such open floater).
  NOTE: when this function succeeds, the OnLuaFloaterClose() automation script callback is called (when present) at the end of the floater destructor method, passing it the 'floater_name' and the 'parameter' strings associated with this floater when OpenLuaFloater() was invoked.

- SetLuaFloaterEnabled(floater_name, control_name[, true|false]) attempts to enable (when *true* or nothing is passed as a third parameter) or disable (when *false* is passed) a control named 'control_name' in a floater named 'floater_name'. It returns a Lua boolean which is *true* on success and *false* on failure (no such open floater or no such control in the floater).

- SetLuaFloaterVisible(floater_name, control_name[, true|false]) attempts to show (when *true* or nothing is passed as a third parameter) or hide (when *false* is passed) a control named 'control_name' in a floater named 'floater_name'. It returns a Lua boolean which is *true* on success and *false* on failure (no such open floater or no such control in the floater).

- SetLuaFloaterCommand(floater_name, control_name, lua_command) tries and sets the 'lua_command' commit callback command for floater 'floater_name' and XUI control 'control_name'. This function returns a Lua boolean which is *true* on success and *false* on failure (no such open floater, or no such control in the floater, or an invalid value such as an invalid folder path or UUID for an inventory panel).
  The 'lua_command' may use special (local) functions: 'GetFloaterName()' which returns the floater name (i.e. the 'floater_name' parameter as passed to 'OpenLuaFloater()'), 'GetFloaterParam()' which returns the floater parameter (i.e. 'parameter' as passed to 'OpenLuaFloater()'), 'GetValue()' which returns the value for that control (equivalent to a call to 'GetLuaFloaterValue()' but without the need to pass the floater and control names), and if (and only if) the control is a button, 'FloaterClose()' that will cause the floater to self-destruct after the 'lua_command' is completed.
  NOTE 1: the 'lua_command' is executed on "commit" event (which depends on the control: for a button, it's when it is pressed, for a check box, it's when it is checked, for a text or line input, it's on focus lost, for a combo or a list, it's on selection change, etc). This comes in excess (and is executed before) of the call to the OnLuaFloaterAction() automation script callback (see below), which is also setup for 'control_name' if it was not already (i.e. if the control name is not one expected by the OpenLuaFloater() command, or if the control is not one of the default, supported types).
  NOTE 2: the 'lua_command' commit callback command is executed in its own Lua context (AKA "state", in Lua jargon).

- GetLuaFloaterValue(floater_name, control_name) returns the value (Lua string) for an UI control named 'control_name' in floater 'floater_name', or *nil* when no such control or floater exist.
  NOTE: this function always returns an empty string for some control types (buttons, for example), and only the "first selected item" (which may have different meanings, like in "first in scroll list current sort order", or "actually selected first" for inventory panels) for controls with multiple selections support.

- GetLuaFloaterValues(floater_name, control_name) returns the values (as a Lua table containing strings, or nothing for no selected value) for an UI control named 'control_name' in floater 'floater_name', or *nil* when no such control or floater exist. This function is useful for scroll list and inventory panel controls, when they are configured to accept multiple selections.

- GetLuaFloaterListLine(floater_name, list_name[, line_index]) returns a Lua table containing key/value pairs of columns name as the key (a Lua string, usually "colN", with N=0 to the number of columns minus 1) and cell contents as the value (as a Lua string) for the line at index 'line_index' in the list 'list_name' of Lua floater 'floater_name', or *nil* when no such list or floater exist. When 'line_index' is omitted or negative, the currently selected list line index is used. Please note that 'line_index' starts at 0 for the first list line.

- SetLuaFloaterValue(floater_name, control_name, value) attempts to set the the 'value' (Lua string) for an UI control named 'control_name' in floater 'floater_name'. It returns a Lua boolean which is *true* on success and *false* on failure (no such open floater or no such control in the floater). For check box controls, a 'value' of "1" or "true" causes the box to be checked and anything else causes it to be un-checked. For scroll lists this function will add a new line entry at the bottom of the list and the 'value' parameter must contain pipe ('|') separated strings that will be used to fill-up the various columns of the scroll list (may be a single string, of course, if the scroll list only got one column); an empty string passed as 'value' causes the scroll list to be cleared. For name lists, there is only support for simple lists (one column, only holding the name, and which must use a 'name="name"" XML tag): passing an UUID in 'value' to this function will cause a new entry to be added for the corresponding avatar (and the name of the said avatar is automatically resolved and displayed in the list); you may also pass a group UUID instead, by prefixing it with the special "<GROUP>" tag; an empty string passed as 'value' causes the name list to be cleared. For inventory panels, 'value' represents the full path or the UUID (same conventions as for 'FindInventoryObject()' parameter) of

the root folder (only the child items of that folder are listed in the inventory panel); passing an empty string as 'value' causes the inventory panel root folder to be set to 'My Inventory'.
NOTE 1: this function got no effect on some control types (buttons, for example).
NOTE 2: in the case of scroll list controls, each column string may be prefixed with special '<BOLD>', '<ITALIC>', '<color_name>' or '<R,G,B>' (with 'color_name' and 'R,G,B' following the same conventions as for 'OnChatTextColoring()' colors) tags, in this order (i.e. '<BOLD><ITALIC><green3>' is valid, but not '<ITALIC><BOLD>', for example) so to affect the font and color of each column entry.
NOTE 3: still for scroll lists, when you specify column names ('name="'") in the XML file of your custom menus, be sure to use 'colN' (with N= 0 to the number of columns in the list minus one). The 'label' itself (which is used in the list header, when shown) can be anything fancying you.

- SetLuaFloaterInvFilter(floater_name, control_name, search_string) attempts to set the the 'search_string' (Lua string) for an inventory panel named 'control_name' in floater 'floater_name'. It returns a Lua boolean which is *true* on success and *false* on failure (no such open floater or no such inventory panel in the floater). The search string is case-insensitive, and when set to a non-empty string, it causes the listed inventory items to be filtered on their name, with only those containing the search string being listed. Passing an empty search string to this function resets the filter and allows to list all inventory items again.

- GetFloaterInstances([floater_name]) is only available to trusted scripts and tries and finds all the existing floater instances (*any* floater, not just Lua ones) with a name corresponding exactly to the passed 'floater_name' string parameter, or when the latter is empty or omitted, all the floaters, whatever their name. This function returns a Lua table with a handle (a global ordinal counter is incremented each time a new floater is created and the resulting number assigned to the latter as this handle) as the key and a string as the value which is either the floater name alone, or the floater name followed with an equal sign and the floater title when the said title has been changed after the floater creation (this allows to distinguish various instances of the same floater, such as, for example, various (detached) IM session floaters). If no matching floater is found, the function returns an empty table.

- ShowFloater(floater_handle) is only available to trusted scripts and attempts to show the floater bearing the 'floater_handle' (a Lua integer corresponding to a key found in the the table returned by GetFloaterInstances()). It returns *true* on success (floater does exist and became visible), or *false* on failure (no such floater, or could not show it).

- GetFloaterControls(floater_handle[, list_disabled[, list_hidden[, control_tag[, control_match]]]]) is only available to trusted scripts and attempts to get a list of the various UI controls found in the floater corresponding to 'floater_handle' (a Lua integer corresponding to a key found in the the table returned by GetFloaterInstances()). If 'list_disabled' is passed and evaluating to a *true* Lua boolean, then disabled UI controls will be listed as well as enabled ones, and if 'list_hidden' is passed and evaluating to a *true* Lua boolean, hidden UI controls are listed too. 'control_tag' (Lua string, case sensitive) allows to list only specific UI controls (e.g. "input_line", "text", "button", etc), and 'control_match' (Lua string) to further restrict the list to controls with a name partly matching that string (case sensitive). The found controls are returned as a Lua table with the control name as the key and the control tag as the value. If no matching floater is found, *nil* is returned by this function, while if no matching control is found, an empty table is returned.

- GetFloaterCtrlState(floater_handle, control_name) is only available to trusted scripts and attempts to get the status of a specific control (named 'control_name', which must evaluate to a Lua string, case sensitive) for the floater bearing the 'floater_handle' (a Lua integer corresponding to a key found in the the table returned by GetFloaterInstances()). It returns a table with key string and value pairs as follow: "type" key with the control tag as a string, "children" key with the number of children for that control as the value (Lua integer), "visible" key with a Lua boolean value indicating whether this control is visible or not, "enabled" key with a Lua boolean value indicating whether this control is enabled or not, "chrome" key with a Lua boolean value indicating whether this control is part of a “chrome” floater or not (“chrome” floaters cannot be focused), "control" key with the debug setting name associated with that control as a string (if any: empty string when not associated with a debug setting value), "value" key with the current value for that control as a string (when any: empty string when no value), "dirty" key with a Lua boolean value indicating whether this control is dirty (needs an update) or not, "focusable" key with a Lua boolean value indicating whether this control can be focused or not, "focused" key with a Lua boolean value indicating whether this control is currently focused or not, "mouse_capture" key with a Lua boolean value indicating whether this control currently has mouse capture or not, and "text_input" key with a Lua boolean value indicating whether this control accepts text input or not. If no matching floater or control is found, *nil* is returned by this function.

- GetFloaterList(floater_handle, list_name) is only available to trusted scripts and attempts to get the contents of a specific scroll list (named 'list_name', which must evaluate to a Lua string, case sensitive) for the floater bearing the 'floater_handle' (a Lua integer corresponding to a key found in the the table returned by GetFloaterInstances()). This function returns a Lua table with the "columns" key associated with a Lua integer corresponding to the number of columns in this table, "headers" key associated with a Lua string containing the columns header names separated from each other with a pipe ("|") character, and for each line in the list, a table entry (numbered/keyed from 1 to the number of lines) containing a string made up of the cell values separated from each other with a pipe character. If no matching floater or list is found, *nil* is returned by this function.

- OverlayBarLuaButton(label, command[, tooltip]) sets up an overlay bar button (one such button is dedicated for Lua scripts) with 'label' as its label and 'command' as the Lua command line to execute when the button is clicked, and an optional 'tooltip' string. Passing an empty command string removes the button from the overlay bar.
  NOTE 1: the command line associated with the button is executed in its own Lua context (AKA "state", in Lua jargon).
  NOTE 2: whenever the automation script is stopped or reloaded (via the corresponding entries in the Advanced menu), any existing overlay bar Lua button is removed.

- StatusBarLuaIcon(command[, tooltip]) sets up the status bar Lua icon, with 'command' as the Lua command line to execute when the icon is clicked, and an optional 'tooltip' string. Passing an empty command string removes the Lua icon from the status bar.
  NOTE: whenever the automation script is stopped or reloaded (via the corresponding entries in the Advanced menu), any existing status bar Lua icon is removed.

- SideBarButton(number[, icon_or_label, command[, tooltip]]]) sets up the Lua side bar button referenced via its 'number' (1 to 12), with 'icon_or_label' as the icon/label (file name corresponding to a skin texture name suitable for use as an icon, or up to two characters, including any supported UTF-8 special symbols or emojis, to use as a button label), 'command' as the Lua command line to execute when the button is clicked (with "nop" recognized as a special no-operation command, mainly useful for toggle control buttons not needing a callback action), and an optional 'tooltip' string. If only the button number is specified, the button is removed from the side bar while if "command" is specified as an empty string, and provided the button was already defined, the icon_or_label and optional tool-tip are modified without changing the existing command (which allows to change a side bar button aspect without modifying its action). This sidebar function is also the only one to accept 0 as a button number: in this case, the first empty slot will be used for the button, if any. The function returns the button number on success, or 0 on failure.
  NOTE 1: icons are searched for in the skin folder and skin overlay folders, including the cvlv-skins/default/textures/ folder that you can add in the Second Life data folder ("~/.secondlife/" under Linux, and "%appdata%\SecondLife\" under Windows: see the SL Wiki for the location of that folder with regards to your specific OS and OS "flavor"). Also, by default, icons are centered in the button. You may however instruct this function to justify the icon left or right, by prefixing its file name with either "left|" or "right|".
  NOTE 2: by default, the side bar appears on the right side of the viewer window, but if you prefer, you may place it on the left side via the "Advanced" -> "User interface" -> "Lua side-bar on left" toggle.
  NOTE 3: because the viewer tool bars and root view are initialized only after login, any SideBarButton() invocation performed before your avatar is logged in (e.g. from the main body of automation.lua) would be ignored; you should therefore use the OnLogin() callback to initialize side bar buttons from the automation script (see the example scripts below).
  NOTE 4: whenever the automation script is stopped or reloaded (via the corresponding entries in the Advanced menu), all existing side bar Lua buttons are removed.

- SideBarButtonToggle(number[, toggle]) returns the toggle state of the already defined side bar button 'number' and may be used to change its toggle state or control. When 'toggle' is omitted, this function simply returns the toggled state as a boolean or *nil* when the button is not a toggle one. When 'toggle' is a Lua boolean (i.e. *true* or *false*), the button is changed into a toggle button if not already one, and its toggled state is changed to on (when *true* is passed) or off (when *false* is passed). When 'toggle' is a non-empty string and does correspond to a boolean debug setting, then the button is changed to a toggle button (if not already one) which reflects and toggles the said boolean debug setting value. When 'toggle' is an empty string or *nil*, the toggle control and toggle state of the button are reset to none.

- SideBarButtonHide(number[, true|false]) hides (without changing its icon/label/tooltip or associated command) the Lua button numbered 'number' in the side bar, or shows it when passed *false*.

- SideBarButtonDisable(number[, true|false]) disables (without changing its icon/label/tooltip or associated command, and without hiding it) the Lua button numbered 'number' in the side bar, or enables it when passed *false*.

- SideBarHide([true|false]) hides the full Lua side bar or shows it when passed *false*.

- SideBarHideOnRightClick([true|false]) causes the side bar to hide whenever any of its buttons is right-clicked, or disables this feature when passed *false*.

- LuaContextMenu(handler_id[, cut_entry[, copy_entry[, paste_entry]]]) sets up the Lua menu entries (labels) for the context menu associated with 'handler_id' (as passed by the OnContextMenu() callback, when the handler declared itself). Each of 'cut_entry', 'copy_entry' and 'paste_entry' Lua strings are used as labels in the context menu for this 'handler_id' and, when they are empty or omitted, it disables/hides the corresponding context menu entry. This function returns a Lua boolean which is *true* when successful or *false* when not (invalid or gone 'handler_id').

- PasteToContextHandler(handler_id[, text]) pastes into the (text or line) editor associated with 'handler_id' the contents of, either, the 'text' Lua string, or the current text held in the viewer clipboard when 'text' is omitted. This function returns a Lua boolean which is *true* when successful or *false* when not (invalid or gone 'handler_id').

- LuaPieMenuSlice(pie_menu_type[, slice_number[, label[, command]]]) sets up a Lua pie menu slice. The Lua pie menu is popped up (if any, else the normal pie menu pops up) whenever you SHIFT-right-click on an object, an avatar, some land or particles (when particles picking is enabled in the viewer). This function allows to setup your own pie menu for the various pick types. 'pie_menu_type' is an integer reflecting the Lua pie type (0 = land pie, 1 = particle pie, 2 = object pie, 3 = agent (self) attachment pie, 4 = avatar (others') pie, 5 = agent (self) pie). If 'pie_menu_type' is passed alone (or if 'slice_number' is 0), then the corresponding Lua pie menu is cleared/reset. If a valid 'slice_number' (Lua integer/number) is passed (1 to 8), then the function affects that slice number in that pie menu type. 'label' (Lua string) is the pie slice label (if omitted or when an empty string is passed, the corresponding pie slice is reset/cleared), while 'command' (Lua string) is the Lua command line associated with that pie slice (if empty, the command is unchanged, allowing to change the label without touching its associated command). The special "nop" (no operation) command may be used and a special "GetPickedObjectID()" Lua function may be used in the pie slice command line, so to get the UUID (as a Lua string) of the picked object (which may be a null UUID, e.g. for Land pie menu/picks).
  NOTE 1: the command line associated with the pie slice is executed in its own Lua context (AKA "state", in Lua jargon).
  NOTE 2: whenever the automation script is stopped or reloaded (via the corresponding entries in the Advanced menu), all existing Lua pie menus are removed/cleared.
  NOTE 3: clicking a Lua pie menu slice also triggers an OnLuaPieMenu() event/callback, when present in the automation script.
  NOTE 4: because the viewer pie menus are initialized only after login, any LuaPieMenuSlice() invocation performed before your avatar is logged in (e.g. from the main body of automation.lua) would be ignored; you should therefore use the OnLogin() callback to initialize Lua pie menu slices from the automation script.

- PickAvatar([allow_multiple, [search_name]]) is only available to the trusted non-threading scripts and pops up an avatar picker floater. 'allow_multiple' is an optional Lua boolean (defaulting to *false*) which governs whether multiple avatars can be selected or not. 'search_name' is an optional Lua string which, when non empty, will trigger a search for that avatar name on the opening of the picker floater. When the floater is closed, the OnPickAvatar() callback of the automation script is called.

- PickInventoryItem(asset_type[, sub_type[, allow_multiple[, exclude_library[, set_immediate[, perms_mask[, callback_on_close]]]]]]) is only available to the trusted non-threading scripts and pops up an inventory item picker floater that will list all inventory items matching the requested 'asset_type' (see Appendix D, Asset types), optionally narrowing the items to 'sub_type' (Lua integer which defaults to -1 for no sub-type, and is only of use with wearable (see Appendix D, Wearable types) and environment settings (see Appendix D, Environment settings types) items. 'allow_multiple' is an optional Lua boolean (defaulting to *false*) which governs whether multiple items can be selected or not. 'exclude_library' is an optional Lua boolean (defaulting to *true*) and which governs whether to show the inventory Library items or not. 'set_immediate' is an optional Lua boolean (added in Cool VL Viewer v1.30.2.15), which, when specified (as either *true* or *false*), causes the “Apply immediately” check box to appear in the floater, and sets its status to either checked (when *true*) or not (when *false*). 'perms_mask' is an optional Lua integer (added in Cool VL Viewer v1.30.2.33) representing the permissions mask to filter the items through (the default is to ignore modify/copy/transfer permissions and list all items); the mask bits are listed in Appendix D, Permission flags. Finally, the 'callback_on_close' optional Lua boolean (added in Cool VL Viewer v1.30.2.33) controls whether the picker should call back Lua with an empty selection when it is closed without a selection committed or not (it defaults to false). When the floater is closed (if 'callback_on_close' was *true*), a selection is committed, or when the “Apply immediately” check box is shown and checked in the floater, the OnPickInventoryItem() callback of the automation script is called.

Debug and persistent settings related functions

- GetDebugSetting(setting_name) attempts to retrieve the value of a debug setting variable. All settings types but Vector3d (not used for any existing setting) and LLSD (only currently used by hidden settings) are supported, and it returns *nil* when it fails (unsupported type). For string, integer, float or boolean settings, the function returns a value of the corresponding Lua type, while for vector, color and rectangle types, it returns as many numbers (floating point ones for Vector3, Color3 and Color4 settings, and integers tor Rect and Color4U ones) as needed (i.e. 3 or 4 numbers). Note that UI colors (corresponding to “settings” that actually pertain to the skin as defined in its colors_base.xml file) component values are integers between 0 and 255 while other colors component values are floats between 0.0 and 1.0. This function always fails (and returns *nil*) if invoked from a Lua command line sent by a scripted object or, for Linux, via D-Bus. (*)

- SetDebugSetting(setting_name[, value|x, y, z|r, g, b[, a]|left, top, right, bottom]) attempts to set the value of a debug setting variable, or to reset it to its default value when only the setting name is passed. All settings types but Vector3d (not used for any existing setting) and LLSD (only currently used by hidden settings) are supported. The function returns *false* when it fails (bad type or incorrect number of parameters for the type, or out of range color component value), or *true* on success. Note that the viewer forbids changing the value of debug settings which are marked as "hidden" (i.e. the ones that cannot be seen/changed from the "Debug settings" floater) and would trigger a Lua error if you attempt to change them via this function; you may however still read the value of hidden settings with GetDebugSetting(). For rectangle (Rect), vector and color settings, you must pass as many numbers as needed (with the last alpha component made optional for colors, which would then be set as opaque). For UI colors it shall be noted that the change is only valid for the current session. UI color (Color4U type) component values must range from 0 to 255 and other color (Color3 and Color3 types) settings component values must range from 0.0 to 1.0. This function always fails (and returns *false*) if invoked from a Lua command line sent by a scripted object or, for Linux, via D-Bus. (*)

- SetGlobalData(data) is only available from the trusted non-threading scripts and it stores an arbitrary 'data' string inside a reserved global debug setting of the viewer, thus allowing to save data/settings for use in a next viewer session. "data" may also be a simple Lua table: valid "simple" tables are made up key/value pairs, where the key is either a number (or an implicit index when the key is omitted) or a string, and the value a Lua string, number, boolean or *nil*; in this case, the function automatically serializes the table and converts it into a base64 value before storing the latter into the reserved global debug setting of the viewer.

- GetGlobalData() is only available from the trusted non-threading scripts and it returns the string or (automatically deserialized and restored) table stored inside a reserved global debug setting of the viewer by the last SetGlobalData() invocation, thus allowing to restore data/settings on a new session start.

(*) Versions of the Cool VL Viewer before v1.28.2.72 and v1.30.0.0 used to check white list settings to determine whether a debug setting could be read or written, and allowed to read or change them from commands sent via LSL and D-Bus; this is no more the case.

- SetPerAccountData(data) is only available from the trusted non-threading scripts and it stores an arbitrary 'data' string inside a reserved per-account debug setting of the viewer, thus allowing to save data/settings for use in a next viewer session with the same avatar. "data" may also be a simple Lua table: valid "simple" tables are made up key/value pairs, where the key is either a number (or an implicit index when the key is omitted) or a string, and the value a Lua string, number, boolean or *nil*; in this case, the function automatically serializes the table and converts it into a base64 value before storing the latter into the reserved per-account debug setting of the viewer.

- GetPerAccountData() is only available from the trusted non-threading scripts and it returns the string or (automatically deserialized and restored) table stored inside a reserved per-account debug setting of the viewer by the last SetPerAccountData() invocation, thus allowing to restore data/settings on a new session start with the same avatar.

- RenderDebugInfo(feature) allows to toggle debug render info features (it is not possible to get several features active at the same time via this function), such as listed in the "Advanced" -> "Rendering" -> "Info display", "Advanced" -> "World" (for wind vectors) and "Advanced" -> "Character" -> "Character debugging". The 'feature' number must be an integer from 0 (features all toggled off) to 32 and corresponds to the index (starting from 1, not 0) of the feature in the LLRenderDebugMask enumeration (see Appendix D, Render debug flags indexes).
  NOTE: these debug features and their index are subject to changes from one viewer branch to the other.

Lua threads related functions

- GetThreadID() is only available from the automation script threads, and returns the thread Id (a positive Lua integer) of the thread.

- HasThread(thread_id) is only available from the automation script and its threads, and returns *true* whenever a thread with Id 'thread_id' (a positive Lua integer) exists and is running, or *false* otherwise.
  NOTE: 0 (the automation script Id) is tolerated for 'thread_id' by this function which would return *false* for it.

- StartThread(source_file[, argv]) is only available from the automation script. It allows to start a new thread by loading the 'source_file' (a Lua string) as a new Lua state and running it in the background. If 'argv' (a "simple" Lua table: see SetGlobalData() above for what it means) is passed, then it is transmitted as an "argv" global variable to the thread. The function returns the thread Id (an integer greater than 0) on success to start the thread or *nil* otherwise. 'source_file' (which may itself contain sub-directory names) is searched in the user_settings/ and in the user_settings/include/ subdirectories, unless it starts with "~/", in which case it is searched in the user home directory instead.
  NOTE: there is currently a limit of 8 simultaneously running threads.

- StopThread(thread_id) is only available from the automation script, and attempts to stop a thread with Id 'thread_id'. It returns *true* whenever such a thread did exist (and was sent the stop request), or *false* otherwise.

- SendSignal(thread_id, signal) is only available from the automation script and its threads, and attempts to send a signal to a thread which Id is 'thread_id' (positive integer, with 0 accepted for the automation script itself). It returns *true* whenever such a thread did exist, got an OnSignal() callback and was sent the signal, or *false* otherwise. The 'signal' is a "simple" Lua table (see SetGlobalData() above for what it means).

- Sleep(milliseconds) is only available from the automation script threads. It sleeps the calling thread for 'milliseconds' (positive Lua integer, 0 included), also resetting the watchdog timer for that thread and checking for any pending signal or thread stopping request.

Chapter 3 – Lua callbacks

Here are the currently recognized callbacks for use in the Lua automation script, triggered by viewer events or as a consequence of Lua function call:

Utility callbacks

- OnAutomationMessage(message) is called each time another Lua command line uses the AutomationMessage(message) function, 'message' (a string or string convertible type) being transmitted as a Lua string from the said command line to the OnAutomationMessage() callback in the automation script. This callback shall not return any value.

- OnAutomationRequest(message) is called each time another Lua command line uses the AutomationRequest(request) function, 'request' (a string or string convertible type) being transmitted as a Lua string from the said command line to the OnAutomationRequest() callback in the automation script which shall return a string in reply.

- OnSignal(from_id, timestamp, signal) is usable in the automation script and its threads. It is invoked whenever a thread (or the automation script) uses SendSignal() to signal us. 'from_id' is the Id of the signaling thread (0 for the automation script), 'timestamp' is the session time stamp (GetFrameTimeSeconds() time stamp), and 'signal' is a "simple" Lua table. This callback shall not return any value.

- OnLogin(location, moved, autologin) is called on successful login into a grid. The 'location' parameter is a Lua table containing the same data as what GetGridSimAndPos() returns. The 'moved' parameter is a Lua boolean which is *true* when the login location was unavailable and the avatar was consequently moved to another place. The 'autologin' parameter is *true* when the login was achieved via the command line transmitted to the viewer on launch, of *false* otherwise (i.e. when the login was performed manually). This callback shall not return any value.

- OnAveragedFPS(average_fps_rate, fps_limited, frame_render_time) is called periodically (based on the value of the "LuaOnAveragedFPSInterval" debug setting, defaulting to 2 seconds) with the averaged frames per second rate during this period and the 'fps_limited' boolean which is *true* whenever the frame rate has been the subject of a capping/limitation (e.g. via the "FrameRateLimit" or "BackgroundYieldTime" debug settings), while frame_render_time provides the actual render time (frame limiting delays not taken into account) in milliseconds, averaged on the last 32 frames (this latter parameter was added in v1.30.2.2). This callback shall not return any value.

Objects related callback

- OnObjectInfoReply(object_id, name, desc, owner_id, group_id) is called each time a reply is received from the server to a GetObjectInfo() request. All values passed to it are of Lua string type. Aside the object, owner and group UUIDs, 'name' is the name of the object and 'desc' its description. This callback shall not return any value.

Agent position and movement related callbacks

- OnRegionChange(region) is called whenever the agent moves to a new region (simulator). The 'region' parameter is a Lua table containing the same data as what GetGridSimAndPos() returns. This callback shall not return any value.

- OnParcelChange(parcel) is called whenever the agent moves to a new parcel. The 'parcel' parameter is a Lua table containing the same data as what GetParcelInfo() returns. This callback shall not return any value.

- OnPositionChange(position) is called each time the agent moves (by more than 3 meters since last call, further throttled to a maximum of one call every 10 seconds). The 'position' parameter is a Lua table containing the following "key"/value pairs: the avatar global position as "global_x" and "global_y" (Lua number values) in meters, the avatar local position in the region as "local_x" and "local_y" (Lua number values) in meters, and finally the avatar "altitude" (Lua number value) in meters. This callback shall not return any value.

- OnTPStateChange(state, failure_reason) is called each time the teleporting state changes. 'state' receives an integer value (see Appendix D, Teleport states) and, when the state change corresponds to a teleport failure, the 'failure_reason' is transmitted as a string (it can be some plain text returned by the server, or a key corresponding to the predefined failure reasons defined in indra/newview/skins/default/xui/en-us/teleport_strings.xml for the "errors" message_set). This callback shall not return any value.

- OnFailedTPSimChange(number_of_residents, x, y, z) is called each time the number of residents changes in the sim that you tried and failed to teleport to. The x,y,z coordinates are the coordinates of the failed TP and may be used directly with TeleportAgentToPos() to retry the TP. This callback shall not return any value.

- OnJoystickButtons(old_state, new_state) is called each time one or several joystick buttons are pressed or released. 'old_state' and 'new_state' are Lua integers holding a bitmap of the buttons state (0 for released, 1 or pressed) with bit 0 = button 0, bit 1 = button 1, ... bit 15 = button 15. 'old_sate' is the former buttons state and 'new_state' is the state of the buttons as this callback is invoked, allowing to see what button(s) got pressed or released. This callback shall not return any value.

- OnAutoPilotFinished(reason, reached, user_cancel) is called each time the auto-pilot feature stops driving your avatar. 'reason' is a Lua string containing the reason for the auto-pilot (e.g. "Sit", "Attach", "Playback", "Lua auto-pilot N", or the UUID of the leader. The string may also be empty for user-triggered auto-pilot), 'reached' is a Lua boolean which is *true* whenever the target was reached or *false* otherwise, and 'user_cancel' is a Lua boolean which is *true* whenever the user cancelled the auto-pilot action or *false* otherwise.

Agent and avatar related callbacks

- OnAgentBaked(baked_textures) is called each time your avatar is rebaked, provided all the wearable layers it wears have been created by you (or have the OpenSim export permission). The 'baked_textures' Lua table then contains key/value pairs with bake names (e.g. "head-baked", "lower-baked", etc) as the keys and the corresponding texture UUIDs (Lua strings) as the values. This callback shall not return any value.

- OnAgentOccupationChange(mode) is called each time the occupation (AFK/Busy/Auto-reply) mode of your agent changes. 'mode' is a Lua number (integer) reflecting the new mode with 0 for "none" (agent present/available), 1 for "AFK", 2 for "Busy" and 3 for "Auto-reply". This callback shall not return any value.
  NOTE: to avoid potential infinite loops, this callback is not triggered when setting the mode via SetAgentOccupation().

- OnAgentPush(pusher_id, push_type, magnitude) is called each time your avatar is pushed by an item/agent. 'pusher_id' is the UUID of the perpetrator of the push (avatar id), 'push_type' is an integer reflecting the type of the push (see Appendix D, Mean collision types) and 'magnitude' represents the force of the push. This callback shall not return any value.

- OnAvatarRezzing(avatar_id) is called each time a new avatar object is created in the viewer (which corresponds to an avatar getting rezzed). This callback shall not return any value.

- OnFriendStatusChange(avatar_id, mask, online) is called each time a friend's status changes. 'avatar_id' is the UUID of the friend avatar, 'mask' is a Lua integer reflecting the type of change which occurred with bit 1 set for ADD, bit 2 for REMOVE, bit 4 for ONLINE, bit 8 for POWERS (see Appendix D, Friends observer events), and 'online' is a boolean reflecting the current online state of the friend avatar. This callback shall not return any value.

Camera related callback

- OnCameraModeChange(mode) is called each time the agent camera mode changes. 'mode' is an integer corresponding to the camera mode (see Appendix D, Camera modes). This callback shall not return any value.

Windlight and extended environment related callback

- OnWindlightChange(sky_settings_name, water_settings_name, daycycle_settings_name) is called each time the Windlight or water presets are changed (other than via the ApplySkySettings() and ApplyWaterSettings() functions). The name of the changed preset (either the sky, the water, or the daycycle one) is passed (the others are empty). Also, the following settings names are passed (in "sky_settings_name") whenever the presets in the "World" -> "Environment settings" sub-menu are used: "region" when the sky settings got reverted to region default, "sunrise", "noon", "sunset" and "midnight" when the region time got set to the corresponding fixed time. This event is also triggered (with "region" or "local" passed as the 'sky_settings_name') whenever you (re)set to region time. The "parcel" and "local" pseudo-setting names are also used and indicate that the parcel Extended Environment or custom/local environment to Windlight translation/rendering is enabled. This callback shall not return any value.

Radar related callbacks

- OnRadar(avatar_id, name, range, marked) is called each time the radar announces an incoming avatar (provided it was configured to do so !), or adds/removes an avatar to/from the Radar floater list. For incoming announces, 'range' is a positive integer: 0 for an arrival in the sim, beyond draw distance, 1 for an arrival within draw distance, 2 for shout range and 3 for chat range. For a newly added avatar (be it located in the agent region or in a neighboring region), 'range' is -1 and for a removed avatar, it is -2. 'marked' is a Lua boolean which is *true* when the avatar was marked by the user in the Radar floater. Starting with the Cool VL Viewer v1.32.2.48, the outgoing announces also trigger this callback, with 'range' set to -10 for a departure from the sim, -11 for an avatar leaving the draw distance, -12 for an avatar leaving shout range and -13 for an avatar leaving chat range. This callback shall not return any value.

- OnRadarSelection(ids) is called each time the selected entries change in the Radar list. The callback gets passed a Lua array of UUIDs corresponding to the selected avatars. This callback shall not return any value.

- OnRadarMark(avatar_id, legacy_name, marked) is called each time a new avatar is marked or un-marked (via the corresponding button) in the Radar. The callback gets passed the avatar UUID (Lua string type), its legacy name (Lua string) and a 'marked' Lua boolean (reflecting whether the avatar is currently marked or not). This callback shall not return any value.

- OnRadarTrack(avatar_id, legacy_name, tracked) is called each time a new avatar starts or stops being tracked in the Radar. The callback gets passed the avatar UUID (Lua string type), its legacy name (Lua string) and a 'tracked' Lua boolean (reflecting whether the avatar is currently tracked or not). This callback shall not return any value.

RestrainedLove related callbacks

- OnRLVHandleCommand(object_id, behav, option, param) is called each time a queued RestrainedLove command is executed, with 'object_id' (Lua string type) being the UUID of the object the command is executed for, and 'behav', 'option', 'param' being the parsed elements of the command (which is always in the form behav:option=param). See the RestrainedLove API Wiki article for the full commands list and syntax. This callback shall not return any value.

- OnRLVAnswerOnChat(object_id, channel, text) is called each time RestrainedLove sends a message to an object in (private) chat, with 'object_id' (Lua string type) being the UUID of the object the 'text' is addressed to (even if all listening objects on the same channel also receive the message), and 'channel' (Lua integer type) being the channel on which the 'text' is sent. This callback shall not return any value.

Chat and instant messaging related callbacks

- OnReceivedChat(type, from_id, is_avatar, name, text) is called each time the viewer receives a chat line. 'type' is a positive number (see Appendix D, Chat types) with 0 for whispers, 1 for normal avatar chat, 2 for shouts, 8 for llOwnerSay(), 9 for scripted objects llInstantMessage(). 'from_id' is the UUID of the chat originator (the owner for chatting objects), is_avatar is a boolean which is *true* when the originator is an avatar, 'name' is the originator object/avatar name and 'text' is the full chat line (name included). This callback shall not return any value.

- OnChatTextColoring(from_id, name, text) is called each time the viewer needs to color a chat line coming from an avatar. 'from_id' is the UUID of the chat originator, 'name' is the originator avatar name and 'text' is the chat line text (name excluded). This callback shall return the color to be applied to the chat line text as a string (containing either the name of a known predefined color, or as a "R,G,B" triplet, with R, G and B as floats in the [0.0, 1.0] range for Red, Green, Blue), or an empty string to let the viewer decide and apply its default rules/colors.

- OnSendChat(text) is called each time you send a chat text from the chat bar(s). The 'text' is passed to the callback (with the ':' to '/me ' substitution already done if needed) and the callback may modify that text if needed. The callback shall return the text to actually be sent by the viewer (i.e. you can use it as a way to substitute sub-strings, or filter text, or even as a command hook). If an empty string is returned, then the viewer will not emit any chat (handy for command hooks). Examples of a pseudo-gesture with parameter and of a draw distance setting command are given in the sample automation script below.

- OnInstantMsg(session_id, origin_id, type, name, text) is called each time an instant message is received. 'type' is an integer value reflecting the IM session type: 0 for peer to peer sessions (private IMs), 1 for group sessions, 2 for conferences. The other parameters are obvious. This callback shall not return any value.

Networking related callback

- OnHTTPReply(handle, success, reply) is called each time an HTTP request sent (*) via either GetHTTP() or PostHTTP() completes (or times out), with 'handle' an Lua integer corresponding to the one returned by GetHTTP() or PostHTTP() when the request was fired, 'success' a Lua boolean which is *true* when the request succeeded or *false* when it failed, and ‘reply' a Lua string containing the body of the server reply on success, or the error message on failure.
  (*) By this automation script or automation thread: each automation script and threads must implement their own OnHTTPReply() callback to deal with their own HTTP requests.

User interface related callbacks

- OnAlertDialog(name, dialog_id, message, buttons) is called each time an alert dialog is spawned, 'name' being the alert name (as seen in the skins/default/xui/en-us/notifications.xml file of the viewer installation directory), 'dialog_id' (Lua string) being the UUID of the alert dialog, 'message' (Lua string) the text displayed in it and 'buttons' a table containing the buttons in key/value pairs, with a number (starting at 1 for the first button) as the key and the name of the button as the string value; note that this name may actually be an URL for dialogs with a clickable URL, and might also be equal to the "<unset>" string when the button slot is empty. This callback shall not return any value.

- OnNotification(name, dialog_id, message) is called each time a (non-script) notification (orange or blue pull-down menu) or notify tip is spawned, 'name' being the notification name (as seen in the skins/default/xui/en-us/notifications.xml file of the viewer installation directory), 'dialog_id' (Lua string) being the UUID of the notification, 'message' (Lua string) the text displayed in it. This callback shall not return any value.

- OnScriptDialog(dialog_id, message, buttons) is called each time a script dialog is spawned, 'dialog_id' (Lua string) being the UUID of the dialog, 'message' (Lua string) the text displayed in the dialog and 'buttons' a table containing the buttons in key/value pairs, with a ~~string~~ number key between ~~"button1" and "button12"~~ 1 and 12 (*), and the value being the button name. This callback shall not return any value.
  (*) The table keys have been changed to numbers (easier to deal with) in the Cool VL Viewer 1.32.0.25 (stable branch) and v1.32.1.8 (experimental branch).

- OnGroupNotification(group_name, group_id, dialog_id, timestamp, sender, subject, message, inventory) is called each time a group notification is received. 'group_name' (Lua string) being the group name, 'group_id' (Lua string) the UUID of the group, 'dialog_id' (Lua string) being the UUID of the notification dialog, 'timestamp' (Lua number) the notification timestamp in seconds since UNIX Epoch, 'sender' (Lua string) the avatar name of the sender, 'subject' (Lua string) the subject line of the notification, 'message' (Lua string) the message body of the notification, and 'inventory' (Lua string) the inventory name of any attached item (an empty string when there is no attached item). This callback shall not return any value.

- OnSLURLDispatch(slurl, nav_type, trusted) is called each time the viewer attempts to dispatch a SLURL (with the exception of SLURLs dispatched with the DispatchSLURL() Lua function), with 'slurl' (Lua string) being the SLURL itself, 'nav_type' (Lua string) can be either of "clicked", "external", "navigated" or an empty string, and 'trusted' (Lua boolean) is *true* when the SLURL was invoked from a “trusted” source or *false* otherwise. This callback shall return either *true* if you allow the SLURL to actually be dispatched by the viewer, or *false* otherwise (in which case, no action will be taken by the viewer and the SLURL will be ignored). This callback can be used to either filter SLURLs based on your own criteria, or to implement your own, custom SLURL scheme (for example to implement a custom HTTP “HUD” that would bear links such as secondlife:///app/lua/my_hud_action/parameter).

- OnURLDispatch(url, target, for_external_browser) is called each time the viewer attempts to dispatch a web URL to open it in either the internal browser (CEF) or via the “external browser” (i.e. the configured default system browser), with 'url' (Lua string) being the URL itself, 'target' (Lua string) being the link HTML target (e.g. "_blank", "_self", etc, but also the viewer-specific "_external" and "_internal" targets which tell the dispatcher what browser to use) or an empty string, and 'for_external_browser' (Lua boolean) is *true* when the URL has been determined to be loaded via the external browser (which depends on both the target and the "UseExternalBrowser" debug setting value), or *false* otherwise. This callback shall return an integer and can be used to either filter URLs based on your own criteria, or to force-redirect them to a given browser, by returning either of -1 (URL allowed to load, no change to final target browser), 0 (URL loading blocked), 1 (allow loading it, but force it to load in the internal browser) or 2 (allow loading it, but force it to load in the external browser).

- OnPickInventoryItem(items, on_close) is called by the inventory items picker floater opened by PickInventoryItem() gets closed (in this case 'on_close' is *true*), or whenever the “Apply immediately” check box of the floater is shown and checked, and an inventory item is selected in the inventory list (in this case 'on_close' is *false*). The 'items' parameter is either *nil* (when no item was selected) or a Lua table containing key/value pairs with the selected inventory items UUIDs as the keys and their name as the values. This callback shall not return any value.
  NOTE: the 'on_close' boolean was added in the Cool VL Viewer v1.30.2.15 (former versions only had this callback called on closing the inventory item picker floater).

- OnPickAvatar(avatars) is called when the avatar picker floater opened by PickAvatar() gets closed. The 'avatars' parameter is either *nil* (when no avatar was selected) or a Lua table containing key/value pairs with the selected avatars UUIDs as the keys and their name as the values. This callback shall not return any value.

- OnLuaFloaterOpen(floater_name, parameter) is called each time a new custom Lua floater is successfully opened. The 'floater_name' and 'parameter' Lua strings correspond to the ones passed to the OpenLuaFloater() function to open that floater. This callback shall not return any value.
  NOTE: when this function is called (from the floater post-build method), the floater is already fully registered and constructed so that all actions on it can be performed (in particular, this function is the place where you could perform control callbacks associations, hide/show and enable/disable actions).

- OnLuaFloaterClose(floater_name, parameter) is called each time a custom Lua floater is closed. The 'floater_name' and 'parameter' Lua strings correspond to the ones passed to the OpenLuaFloater() function to open that floater. This callback shall not return any value.
  NOTE: when this function is called (from the floater destructor), the floater is already unregistered from the custom floaters list so that no action on it can be performed any more.

- OnLuaFloaterAction(floater_name, control_name, value) is called when a control named 'control_name' in a custom Lua floater named 'floater_name' is executing a commit event for which this callback function was registered (it can be registered automatically on OpenLuaFloater() call, if the control name and type are adequate, or on demand, via a SetLuaFloaterCommand() call). 'value' (Lua string) is the current value for that control. This callback shall not return any value.

- OnContextMenu(handler_type, handler_id, operation, clipboard_text) is called when a context menu is opened in a line or text editor which already registered itself as 'handler_type' (Lua string), or at the moment the said editor registers itself as a specific 'handler_type'; currently, the editors doing so are the "chat_input" (chat bar/chat floater) input line, the "chat_history" (chat floater history) text editors (both with and without mutes), the "im_input" (IM panels/floaters) input line, the "im_history" (IM panels/floaters history) text editor, the "text_input" (multi-line chat/IMs input floater) editor, the "notecard" (note card floater) editor, and the "script" (script floater, “Edited script” tab) editor. The text editors and line editors defined in custom Lua floaters and which names follow the convention defined in note 1 of OpenLuaFloater() also declare themselves with a 'handler_type' string equal to the floater name followed with a space and the UI element name. 'handler_id' is a unique number (Lua integer) identifying the corresponding text/line editor (to be used with LuaContextMenu() and PasteToContextHandler()). 'operation' is a Lua integer which is 0 on handler declaration (the moment when it declares its 'handler_type', which may happen before login such as for the chat toolbar handler, or whenever the corresponding floater/panel is opened, such as for IM, chat history, notecard, script, etc), and when LuaContextMenu() has been called (either explicitly, or implicitly by this callback, when it returned *true* for the corresponding 0 operation on this handler) to enable the corresponding custom action for this 'handler_id', 1 for “custom cut”, 2 for “custom copy”, and 3 for “custom paste” actions. 'clipboard_text' is a Lua string containing a copy of the current clipboard text. This call back shall return a Lua boolean which must be *true* if you want it to take the default action for the operation, or *false*, if you do not want it to take any action of its own, but want to deal with the operation in your own code. For operation 0 (handler declaration), the default action is to set all three “custom” context menu entries to “Cut to Lua”, “Copy to Lua” and “Paste from Lua” (enabling the corresponding entries in the context menu for that handler). For operation 3 (“custom paste”, the default action is to paste the contents of the clipboard text, as it is on return from the callback (i.e. you may change that contents with SetClipBoardString() inside the callback to override 'clipboard_text'). There is no default action for operations 1 and 2 (“custom cut” and “custom copy”).

- OnLuaPieMenu(data): is called each time a Lua pie menu slice is clicked. 'data' is a table holding the following "key"/value pairs: "type" is an integer reflecting the Lua pie type (0 = land pie, 1 = particle pie, 2 = object pie, 3 = agent (self) attachment pie, 4 = avatar (others') pie, 5 = agent (self) pie), "slice" is an integer reflecting the number of the pie slice (from 1 to 8) that got clicked, "global_x", "global_y" and "altitude" are Lua numbers reflecting the position of the pick event, "object_id" is a Lua string holding the UUID of the picked object ("object_id" field absent when there was no picked object, such as for a land pick), and for particle pie events, "particle_owner_id" and "particle_source_id" which are Lua strings holding the UUID of, respectively, the owner id (avatar) and particle source id. This callback shall not return any value.

- OnLuaDialogClose(title, button, text) is called each time a Lua dialog floater (opened by any Lua script/command line with MakeDialog(), see above) is closed. 'title' corresponds to the same parameter value as the one used in MakeDialog(), 'button' is the number of the button that got pressed and caused the dialog to close (0 if the dialog was closed with the floater close ("X") button), and 'text' is the text that was contained/entered in the input line. This callback shall not return any value.

- OnSideBarVisibilityChange(visible) is called each time the Lua side bar visibility changes (e.g. whenever SideBarHide() is called, or when the side bar was configured with SideBarHideOnRightClick() and is right clicked) and 'visible' is a boolean indicating whether the side bar is visible or not. Please note that this visibility does not relate to the automatic hiding of the side bar in mouse-look mode (the latter mode always hides the side bar but does not affect its internal visibility status). This callback shall not return any value.

Chapter 4 – Lua automation threads

Starting with the Cool VL Viewer v1.28.0.6 Lua automation threads have been implemented.

Previously, the Cool VL Viewer could only execute (relatively) short sequences of Lua code based on automation script callbacks (or command lines), and that Lua code was executed within the main thread/loop of the viewer, meaning the more complex the code and the more the frame rate would 'hiccup' (given the burst-like load of such Lua callbacks). Also, to avoid freezes or infinite loops, the Lua code executed by the automation script is subject to a watchdog (short) timeout, meaning you cannot perform very complex and long processing this way.

With automation threads, you may now run a full Lua program (and in fact up to 8 Lua programs) in the background (your OS will affect those threads to unused or lightly loaded CPU cores, meaning they won't even slow down the viewer itself !).

A Lua thread is started (from the automation script only) by loading a separate Lua file and executing it as a separate "Lua state". However, that thread still gets the opportunity to call the viewer-specific Lua commands (even non-thread-safe ones, thanks to a special and transparent mechanism), just like if they would be executed by the automation script itself. It may also exchange data with the automation script (and other threads) via "signals".

A Lua thread source is a Lua program file containing at least a ThreadRun() function as an entry point for the thread looping code. ThreadRun() is called at each thread loop and must return a boolean; when the latter is true, the thread keeps running and ThreadRun() is called again after a 1ms "sleep" (which is used both to yield to the OS and allow threads rescheduling by the latter, and to avoid "eating up" a full CPU core when ThreadRun() executes very short sequences of Lua code). Whenever ThreadRun() returns false, the thread is stopped and destroyed.
When it is launched by the automation script (via the StartThread() function), the thread may receive parameters in a global "argv" table.

Note that to avoid infinite loops and to allow timely detection of thread stopping requests from the viewer, the ThreadRun() function execution time is still bound by a 0.5s watchdog. However, a special Sleep() Lua function is available to threads, which resets the watchdog when invoked (because signals and thread stopping requests are checked/processed during the Sleep() call, even when sleeping for 0ms); so, even if each ThreadRun() involves complex/long processing, you can ensure it will not be interrupted by the watchdog under normal operation, by calling Sleep() appropriately.

The Lua thread program file may also contain an OnSignal() callback (and the automation script may also have such a callback to receive data from threads). This callback is entered whenever another thread (or the automation script) invokes SendSignal(), directing it to our thread.
As for the viewer-specific Lua functions which are not thread-safe (e.g. SendChat()), the mechanism I implemented ensures they can be called from threads nonetheless (they cause the thread to pause and set a variable indicating to the viewer main thread that it needs the corresponding code to be ran on its behalf, which is performed during the (badly named) "idle loop" of the viewer).

The print() and warn() Lua functions receive a special treatment: they use an internal print buffer when invoked from a Lua thread and the text gets printed in the viewer chat on the next return of ThreadRun(), on the next invocation of Sleep() or of a non-thread-safe viewer-specific Lua function invocation (whichever happens first).

Here is a (dummy) example of how to use threads:

1.- Add this code to your automation.lua script:

```
function OnAutomationRequest(request)
    -- Starts a new thread
    if request == "thread" then
        local argv = { "foo", "bar", name="My thread" }
        local thead_id = StartThread("thread.lua", argv)
        if thead_id then
            return "Thread started with Id: " .. string.format("%d", thead_id)
        else
            return "Failure to start a new thread"
        end
    end
    -- Stops a thread by Id
    local i, j = string.find(request, "-thread ")
    if i == 1 then
        local thead_id = tonumber(string.sub(request, j + 1))
        if StopThread(thead_id) then
            return "Stopped"
        else
            return "No such thread"
        end
    end
    -- Checks for thread existence, by Id
    local i, j = string.find(request, "thread%? ")
    if i == 1 then
        local thead_id = tonumber(string.sub(request, j + 1))
        return tostring(HasThread(thead_id))
    end
    -- Sends a signal to a thread, by Id
    local i, j = string.find(request, "thread! ")
    if i == 1 then
        local thead_id = tonumber(string.sub(request, j + 1))
        return tostring(SendSignal(thead_id, { "signal" }))
    end
    return ""
end

function OnSignal(from_id, timestamp, sig)
    print("Signal received from thread Id: " .. string.format("%d", from_id) .. "
- Timestamp: " .. tostring(timestamp) .. "s. Contents:")
    for k, v in pairs(sig) do
        print(tostring(k) .. ": " .. tostring(v))
    end
    print("---------------")
end
```

2.- Create a "thread.lua" file and place it in user_settings/include/ (or in user_settings):

```
end_time = 0

function ThreadRun()
    local now = GetFrameTimeSeconds()
    if end_time == 0 then
        end_time = now + 30
        print("Thread started. Now: " .. tostring(now) .. "s")
        if GetThreadID() ~= 1 then
            print("Will exit after: " .. tostring(end_time) .. "s")
        end
        if argv then
            print("Arguments passed:")
            for k, v in pairs(argv) do
                print(tostring(k) .. ": " .. tostring(v))
            end
            print("-----------------")
        end
        local agent_info = GetAgentInfo()
        print("My avatar name / display name: " ..
GetAvatarName(agent_info["id"]) .. " / " .. agent_info["display_name"])
        if HasThread(1) and GetThreadID() ~= 1 then
            print("Pinging thread 1: " .. tostring(SendSignal(1, { "ping" })))
        end
    end
    if GetThreadID() ~= 1 and now > end_time then
        print("Stopping thread. Time: " .. tostring(now) .. "s")
        return false
    end
    return true
end

function OnSignal(from_id, timestamp, sig)
    print("Signal received from thread Id: " .. string.format("%d", from_id) .. "
- Timestamp: " .. tostring(timestamp) .. "s. Contents:")
    for k, v in pairs(sig) do
        print(tostring(k) .. ": " .. tostring(v))
    end
    print("---------------")
    print("Reported to automation script: " .. tostring(SendSignal(0, { "got
it" })))
end
```

3.- Start the viewer, and from the chat, enter:

```
/lua print(AutomationRequest("thread"))
/lua print(AutomationRequest("thread"))
/lua print(AutomationRequest("thread! 2"))
/lua print(AutomationRequest("thread"))
/lua print(AutomationRequest("thread? 1"))
/lua print(AutomationRequest("-thread 1"))
/lua print(AutomationRequest("thread? 1"))
etc…
```

Appendix A – Automation script example

Sample/demonstration automation script (user_settings/automation.lua):

```
-- Constants

-- Side bar buttons assignment
SBB_PREFS = 9
SBB_TPHIST = 7
SBB_SOUNDS = 5
SBB_LANG = 3
SBB_MESHQ = 1
SBB_RLVLOCK = 2
SBB_SHIELD = 4
SBB_FPSLIMIT = 6
SBB_FPSAUTO = 8

-- Minimum draw distance for automatic FPS-based adjustments
MIN_DRAW_DISTANCE = 256

-- Global variables used in the automation script

known_ids = {}
known_sessions = {}
auto_tune_fps = false
tp_started = false
tp_retry = false
max_agents = 20
max_complexity_shield_off = 0
max_complexity_shield_on = 200000
max_area_shield_off = 0
max_area_shield_on = 200
max_memory_shield_off = 0
max_memory_shield_on = 100
shield_on=false
derendered_objects = { "253538b0-56fe-1feb-8089-2cade9c5a413", "6b752a3c-c0a8-8df7-
0d0e-d95c0cda33a1" }
low_dd_regions = { ["Porten Hill"]=true }
protected_attachments = { "hair", "collar", "cuff", "nipple", "penis", "pussy", "hoof",
" ear", "horn", "tail", "muzzle" }
protected_layers = { "shape", "eyes", "hair", "skin", "alpha" }
avatars_colors = {}
account_settings = {}
```

.../...

```
-- Here, we define a custom set of side bar buttons for performing useful
-- tasks or opening commonly used floaters that do not have an associated
-- toolbar button.
-- We also perform tasks that can only happen after successful login.
function OnLogin(location)
     -- Setup the sidebar
    SideBarHideOnRightClick()
    SideBarHide(false)
    SideBarButton(SBB_PREFS, "\u{2699}", "OpenFloater('preferences')", "Opens the
Preferences floater")
    -- Setup the side bar button for spelling language toggling:
    ToggleSpellCheckLanguage(false)
    SideBarButton(SBB_TPHIST, "inv_item_landmark_visited.tga", "OpenFloater('teleport
history')", "Opens the Teleport history")
    SideBarButton(SBB_MESHQ, "inv_item_mesh.tga", "nop", "Toggles mesh queue info")
    SideBarButtonToggle(SBB_MESHQ, "DebugShowMeshQueue")
    SideBarButton(SBB_SOUNDS, "43f0a590-f3d3-48b5-b460-f5b3e6e03626.tga",
"OpenFloater('sounds list')", "Opens the Sounds list floater")
    -- Setup the side bar button for anti-griefers protection toggling:
    OnAutomationMessage("shield-off")
    -- Setup the side bar button for attachments and layers protection:
    local agent = GetAgentInfo()
    if agent.rlv then
        OnAutomationMessage("unprotect")
    end
    -- Initialize FPS limiting buttons
    OnAutomationMessage("fps_unlimit")
    OnAutomationMessage("fps_no_tuning")
    -- Retrieve our per-account settings and validate them (we use a table for
    -- our settings, since it is much simpler than dealing with serialization of
    -- settings to a data string ourselves).
    local settings = GetPerAccountData()
    if type(settings) == "table" then
        account_settings = settings
    end
    -- Restore last Windlight and water settings, if any
    if account_settings.last_sky then
        ApplySkySettings(account_settings.last_sky)
    end
    if account_settings.last_water then
        ApplyWaterSettings(account_settings.last_water)
    end
    -- Derender objects we never want to see
    for i = 1, #derendered_objects, 1 do
        DerenderObject(derendered_objects[i])
    end
     -- Set the draw distance for the login sim, after letting a chance
     -- to the viewer to connect to potential neighbor sims.
    CallbackAfter(5, SetDrawDistance)
     -- Define the Lua pie menu for avatars
    DefineAvatarPieMenu()
 end
```

.../...

```
-- This callback receives the messages sent by the various buttons we setup via
-- the AutomationMessage() function.

function OnAutomationMessage(text)
    -- Used to enable anti-griefers measures
    if text == "shield-on" then
        shield_on=true
        SetDebugSetting("AllowSwapping", false)
        SetDebugSetting("MainMemorySafetyCheck64", true)
        SetDebugSetting("RenderAvatarMaxComplexity", max_complexity_shield_on)
        SetDebugSetting("RenderAutoMuteSurfaceAreaLimit", max_area_shield_on)
        SetDebugSetting("RenderAutoMuteMemoryLimit", max_memory_shield_on)
        SetDebugSetting("KillBogusObjects", true)
        SideBarButton(SBB_SHIELD, "shield_off.png", "AutomationMessage('shield-off')",
"Disables anti-griefers features")
        OpenNotification(0, "Anti-griefer features enabled")
    -- Used to disable anti-griefers measures
    elseif text == "shield-off" then
        shield_on=false
        SetDebugSetting("AllowSwapping", true)
        SetDebugSetting("MainMemorySafetyCheck64", false)
        SetDebugSetting("RenderAvatarMaxComplexity", max_complexity_shield_off)
        SetDebugSetting("RenderAutoMuteSurfaceAreaLimit", max_area_shield_off)
        SetDebugSetting("RenderAutoMuteMemoryLimit", max_memory_shield_off)
        SetDebugSetting("KillBogusObjects", false)
        SideBarButton(SBB_SHIELD, "shield_on.png", "AutomationMessage('shield-on')",
"Enables anti-griefers features")
        OpenNotification(0, "Anti-griefer features disabled")
    -- Used to cancel the auto-TP retries, when enabled
    elseif text == "cancel-auto-tp" then
        tp_retry = false
        OverlayBarLuaButton("", "")
    -- Used to toggle the spell checker language
    elseif text == "language" then
        ToggleSpellCheckLanguage(true)
    -- Used to protect via RLV our outfit from changes
    elseif text == "protect" then
        SideBarButton(SBB_RLVLOCK, "9beb8cdd-3dce-53c2-b28e-e1f3bc2ec0a4.tga",
"AutomationMessage('unprotect')", "Unlock protected layers and attachments")
        SideBarButtonToggle(8, true)
        LockAttachmentsAndLayers()
        OpenNotification(0, "Protected attachments and layers locked")
    -- Used to remove the RLV protection and allow again outfit changes
    elseif text == "unprotect" then
        SideBarButton(SBB_RLVLOCK, "9beb8cdd-3dce-53c2-b28e-e1f3bc2ec0a4.tga",
"AutomationMessage('protect')", "Lock protected layers and attachments")
        SideBarButtonToggle(8, false)
        ExecuteRLV("clear")
        OpenNotification(0, "Protected attachments and layers unlocked")
    -- Used to limit the frame rate to 60 fps
    elseif text == "fps_limit" then
        OnAutomationMessage("fps_no_tuning")
        ToggleFPSLimiting(true)
        OpenNotification(0, "Frame rate limited to 60 fps")
    -- Used to lift the limitation on the frame rate
    elseif text == "fps_unlimit" then
        ToggleFPSLimiting(false)
        OpenNotification(0, "Frame rate now unlimited")
```

.../...

```
    -- Used to limit the frame rate to 60 fps *and* to auto-adjust the draw distance
    -- to be the largest possible without falling below 60fps.
    elseif text == "fps_tuning" then
        OnAutomationMessage("fps_unlimit");
        SideBarButton(SBB_FPSAUTO, "\u{22A1}", "AutomationMessage('fps_no_tuning')",
"Do not tune draw distance based on fps rate.")
        auto_tune_fps = true
        SideBarButtonToggle(SBB_FPSAUTO, true)
        SetDebugSetting("FrameRateLimit", 60)
            OpenNotification(0, "FPS-rate based draw distance tuning turned on")
    -- Used to stop the auto-tuning of the draw distance
    elseif text == "fps_no_tuning" then
        ToggleFPSLimiting(GetDebugSetting("FrameRateLimit"))
        auto_tune_fps = false
        SideBarButtonToggle(SBB_FPSAUTO, false)
        SideBarButton(SBB_FPSAUTO, "\u{22A1}", "AutomationMessage('fps_tuning')", "Tune
draw distance based on fps rate.")
        OpenNotification(0, "FPS-rate based draw distance tuning turned off")
    end
end

-- This callback is invoked each time the side bar visibility is changed. We
-- use it to display the Lua icon in the status bar whenever the side bar gets
-- hidden and we setup the command for that icon so that, when clicked, it
-- shows the side bar (which in turn hides the status bar icon via this
-- callback).
function OnSideBarVisibilityChange(visible)
    if visible then
        StatusBarLuaIcon("")
    else
        StatusBarLuaIcon("SideBarHide(false)", "Shows the Lua buttons side bar")
    end
end

-- This function is used to automatically set the draw distance after TP:
-- If the sim we arrive into is listed in the low_dd_regions table, then the
-- draw distance is set to 128m. If it is a sim without neighbors (island),
-- then the draw distance is set to 512m (to rez everything in sim).
-- In all other cases, the draw distance is set to 256m.
-- Note that the "speed rezzing" feature is also accounted for (i.e. there
-- will not be conflict between this code and the feature: the draw distance
-- final adjustment is done once the speed rezzing adjustments are over).
function SetDrawDistance()
    if GetDebugSetting("SpeedRez") then
        local saved_dd = GetDebugSetting("SavedRenderFarClip")
        if saved_dd > 0 and saved_dd ~= GetDebugSetting("RenderFarClip") then
            -- Disable draw distance auto-adjusting based on fps, if any
            if auto_tune_fps then OnAutomationMessage("fps_no_tuning") end
            CallbackAfter(GetDebugSetting("SpeedRezInterval") + 1, SetDrawDistance)
            return
        end
    end
    local dd = 256
    local location = GetGridSimAndPos()
    local region = location.region
    if low_dd_regions[region] then
        dd = 128
    elseif location.neighbors == 0 then
        dd = 512
    end
    SetDebugSetting("RenderFarClip", dd)
end
```

.../...

```
-- These functions are used to limit the FPS rate and adjust the draw distance

function ToggleFPSLimiting(on)
    if on then
        SideBarButton(SBB_FPSLIMIT, "\u{229F}", "AutomationMessage('fps_unlimit')",
"Lift the limit on the fps rate.")
        SideBarButtonToggle(SBB_FPSLIMIT, true)
        SetDebugSetting("FrameRateLimit", 60)
    else
        SideBarButton(SBB_FPSLIMIT, "\u{229F}", "AutomationMessage('fps_limit')",
"Limit the fps rate.")
        SideBarButtonToggle(SBB_FPSLIMIT, false)
        SetDebugSetting("FrameRateLimit", 0)
    end
end

function OnAveragedFPS(fps, limited, frame_time)
    if not auto_tune_fps then
        return
    end
    -- When below 60fps, we need to reduce the draw distance when possible
    if fps < 60 then
        local dd = GetDebugSetting("RenderFarClip")
        -- Frame rate is way too low, reset to minimum draw distance
        if fps < 30 then
            if dd > MIN_DRAW_DISTANCE then
                SetDebugSetting("RenderFarClip", MIN_DRAW_DISTANCE)
            end
        -- Frame rate below the 60fps target, reduce the draw distance if possible
        elseif dd >= MIN_DRAW_DISTANCE then
            dd = dd - 32
            if dd < MIN_DRAW_DISTANCE then
                dd = MIN_DRAW_DISTANCE
            end
            SetDebugSetting("RenderFarClip", dd)
        end
    -- The actual frame render time is small enough: increase the draw distance
    -- if possible
    elseif frame_time < 8.333 then
        local dd = GetDebugSetting("RenderFarClip")
        if dd < 512 then
            dd = dd + 32
            if dd > 512 then
                dd = 512
            end
            SetDebugSetting("RenderFarClip", dd)
        end
    end
end
```

.../...

```
-- Here we add pseudo commands to:
--  * emulate Firestorm's "/dd" (draw distance) command (with "/dd" alone as an
--    alias for "/dd 256");
--  * toggle the camera front view with "/fc";
--  * adjust the avatar Z offset with "/z " followed by a amount of centimeters
--    (e.g.: "/z -8" to lower your avatar height by 8 cm), with "/z" alone to reset
--    the offset to zero.
-- We also create a pseudo-gesture ("/g " for "greetings") accepting a parameter
-- (the name or title of the resident you want to greet).
function OnSendChat(text)
    if string.sub(text, 1, 1) ~= "/" then
        -- Do not waste time searching for commands if the first character is
        -- not a slash...
        return text
    end
    if text == "/fc" then
        SetDebugSetting("CameraFrontView", not GetDebugSetting("CameraFrontView"))
        return ""
    end
    if text == "/dd" then
        SetDebugSetting("RenderFarClip", 256)
        return ""
    end
    local i, j = string.find(text, "/dd ")
    if i == 1 then
        local distance = tonumber(string.sub(text, j + 1))
        if distance > 512 then
            distance = 512
        elseif distance < 32 then
            distance = 32
        end
        SetDebugSetting("RenderFarClip", distance)
        return ""
    end
    if text == "/z" then
        SetDebugSetting("AvatarOffsetZ", 0)
        return ""
    end
    i, j = string.find(text, "/z ")
    if i == 1 then
        local offset = tonumber(string.sub(text, j + 1)) / 100
        if offset > 9 then
            offset = 9
        elseif offset < -9 then
            offset = 9
        end
        SetDebugSetting("AvatarOffsetZ", offset)
        return ""
    end
    i, j = string.find(text, "/g ")
    if i == 1 then
        text = "/me smiles softly, \"Greetings" .. string.sub(text, j) .. ".\""
    end
    return text
end
```

.../...

```
-- This function toggles the spell checking language between English and
-- French: much easier than doing it via the Preferences menu... Of course,
-- you will have to make sure first that all the related dictionaries have
-- been installed on your system (from "Preferences" -> "Cool features" ->
-- "Chat, IM & text").
function ToggleSpellCheckLanguage(toggle)
    local lang = GetDebugSetting("SpellCheckLanguage")
    if toggle then
        if lang == "fr_fr" then
            lang = "en_us"
        else
            lang = "fr_fr"
        end
        SetDebugSetting("SpellCheckLanguage", lang)
    end
    if lang == "fr_fr" then
        SideBarButton(SBB_LANG, "En", "AutomationMessage('language')", "Toggles the
spell checking language to English (US)")
    else
        SideBarButton(SBB_LANG, "Fr", "AutomationMessage('language')", "Toggles the
spell checking language to French")
    end
end

-- This is just a demonstration of how Lua scripting may be used as a mean of
-- defense against griefing attempts: here, when pushed and the anti-griefer
-- measures are on, the avatar is automatically sat down on ground to prevent
-- further effects of new pushes.
function OnAgentPush(id, t, magnitude)
    if shield_on and mag > 3 then
        AgentSit()
    end
    OpenNotification(0, "Push detected, magnitude = " .. mag)
end

-- This is just a demonstration of IM callbacks usage and of how to use the
-- CallbackAfter() function: it got no real practical use other than being a
-- sample of what can be done.
function InstantMsgReply(session_id, name, text)
    OpenNotification(0, name .. " opened a new IM session: replying now.")
    SendIM(session_id, text)
end

function OnInstantMsg(session_id, origin_id, t, name, text)
    if known_sessions[session_id] then
        return
    end
    known_sessions[session_id] = true
    -- Wait 3 seconds for the IM session to start (important for group sessions) before
replying:
    CallbackAfter(3, InstantMsgReply, session_id, name, "Hello !")
end
```

.../...

```
-- This is just a demonstration of SLURL dispatching and floater opening, all
-- wrapped up in an OnReceivedChat() callback: it got no real practical use
-- other than being a sample of what can be done (and may be dangerous to use
-- "as is", due to the SLURL systematic auto-dispatching).
function OnReceivedChat(t, id, is_avatar, name, text)
    local i, j = string.find(text, "secondlife://")
    if i then
        OpenNotification(0, name .. " sent an SURL, dispatching it")
        DispatchSLURL(string.sub(text, i))
        return
    end
    if known_ids[id] then
        return
    end
    known_ids[id] = true
    if is_avatar then
        OpenNotification(0, name .. " is a new chatting avatar: displaying profile.")
        OpenFloater("avatar info", id)
    else
        OpenNotification(0, name .. " is a new chatting object: inspecting it.")
        OpenFloater("inspect", id)
    end
end

-- Here is a demonstration on how to use the OnChatTextColoring() callback to
-- color incoming chat text from avatars. In this example, the friends' chat
-- text is colored in pink.
function OnChatTextColoring(id, name, text)
    if IsAgentFriend(id) then
        return "pink2"
    end
    return ""
end

-- This is called each time the user loads new Windlight or water presets.
-- We then save the new presets into the per-account settings, so to restore
-- them on next login with the same avatar.
function OnWindlightChange(sky_settings_name, water_settings_name)
    if sky_settings_name ~= "" then
        account_settings.last_sky = sky_settings_name
    end
    if water_settings_name ~= "" then
        account_settings.last_water = water_settings_name
    end
    SetPerAccountData(account_settings)
end
```

.../...

```
-- The code below allows to auto-retry failed teleports (provided the TP global
-- coordinates were known when the TP was first attempted: this is the case for
-- all TPs done from the world map floater, but may not be the case for landmarks
-- TPs and TP invites, at least on the first TP attempt). It illustrates the use
-- of the Lua dialog and overlay bar button, of the AutomationMessage() function
-- and of the TP related callbacks and functions.
function OnTPStateChange(state, reason)
    if state == 0 then -- TELEPORT_NONE
        tp_retry = false
        if tp_started and string.len(reason) > 0 and
           reason ~= "invalid_tport" and
           reason ~= "nolandmark_tport" and
           reason ~= "noaccess_tport" and
           reason ~= "no_inventory_host" then
            if reason == "no_host" then
                -- Auto-retry after sim comes back online
                tp_retry = true
                OverlayBarLuaButton("Cancel auto-TP", "AutomationMessage('cancel-auto-
tp')")
            else
                MakeDialog("Retry TP",
                           "To auto-retry the TP, give the max allowed number of agents
for this sim, else press the Cancel button",
                           max_agents,
                           "Cancel", "", "OK",
                           "DialogClose()", "", "DialogClose()")
            end
        else
            -- Set the draw distance for the arrival sim, after letting a chance
            -- to the viewer to connect to potential neighbor sims.
            CallbackAfter(8, SetDrawDistance)
        end
        tp_started = false
    elseif state == 1 then -- TELEPORT_START
        tp_started = true
        tp_retry = false
    elseif state ~= 2 then -- all other teleport states than TELEPORT_REQUESTED
        tp_started = false
        tp_retry = false
    end
end

function OnLuaDialogClose(title, button, text)
    if title == "Retry TP" and button == 3 then
        max_agent = tonumber(text)
        if max_agent > 0 then
            tp_retry = true
            OverlayBarLuaButton("Cancel auto-TP", "AutomationMessage('cancel-auto-
tp')")
        end
    end
end

function OnFailedTPSimChange(agents, x, y, z)
    if tp_retry and agents < max_agents then
        TeleportAgentToPos(x, y, z)
        OverlayBarLuaButton("", "")
    end
end
```

.../...

```
-- Here is an example of how to use the Lua pie menu, the mini-map and tag
-- colors, and the OnAvatarRezzing() callback.
-- This defines an avatar pie menu with colors you can set for the corresponding
-- avatar's dot in the mini-map and name tag. The color is remembered during the
-- session, even if the avatar gets de-rezzed and re-rezzed.
function DefineAvatarPieMenu()
    LuaPieMenuSlice(4, 1, "Blue", "nop")
    LuaPieMenuSlice(4, 2, "Cyan", "nop")
    LuaPieMenuSlice(4, 3, "Red", "nop")
    LuaPieMenuSlice(4, 4, "Magenta", "nop")
    LuaPieMenuSlice(4, 5, "Yellow", "nop")
    LuaPieMenuSlice(4, 6, "White", "nop")
    LuaPieMenuSlice(4, 7, "Default", "nop")
    LuaPieMenuSlice(4, 8, "Green", "nop")
end

function OnLuaPieMenu(data)
    local color = ""
    if data.slice == 1 then
        color = "blue"
    elseif data.slice == 2 then
        color = "cyan"
    elseif data.slice == 3 then
        color = "red"
    elseif data.slice == 4 then
        color = "magenta"
    elseif data.slice == 5 then
        color = "yellow"
    elseif data.slice == 6 then
        color = "white"
    elseif data.slice == 8 then
        color = "green"
    end
    avatars_colors[data.object_id] = color
    SetAvatarMinimapColor(data.object_id, color)
    SetAvatarNameTagColor(data.object_id, color)
end

-- This callback is used to re-set the configured colors for avatar as
-- they re-rez. It also sets the Radar name color to blue and name
-- tag/mini-map dot to cyan for grid admins.
function OnAvatarRezzing(id)
    local color = avatars_colors[id]
    if color then
        SetAvatarMinimapColor(id, color)
        SetAvatarNameTagColor(id, color)
    elseif IsAdmin(id) == nil then
        -- In case the info for that avatar has not been yet received,
        -- we retry the callback one second later.
        CallbackAfter(1.0, OnAvatarRezzing, id)
    elseif IsAdmin(id) then
        SetAvatarMinimapColor(id, "cyan")
        SetAvatarNameTagColor(id, "cyan")
        -- Here is an example of how to test for the existence of a Lua
        -- function, in case you wish your script to run with older viewer
        -- versions in which that function was not yet implemented...
        if type(SetRadarNameColor) == "function" then
            SetRadarNameColor(id, "blue")
        end
      end
end
```

```
-- This function allows to lock, via RestrainedLove, object attachments and clothing
layers
-- based on their name (for attachments) or type (for clothing layers). Only the joints
and
-- layers corresponding to protected items currently worn are locked.
function LockAttachmentsAndLayers()
    local i, j, k, v, name
    local protected = {}
    local items = GetAgentWearables();
    for k, v in pairs(items) do
        j = string.find(v, "|")
        name = string.lower(string.sub(v, j + 1))
        for i = 1, #protected_layers, 1 do
            if protected_layers[i] == name then
                protected[name] = true
                break
            end
        end
    end
    for k, v in pairs(protected) do
        ExecuteRLV("remoutfit:" .. k .. "=n")
    end
    protected = {}
    items = GetAgentAttachments()
    for k, v in pairs(items) do
        j = string.find(v, "|")
        name = string.lower(string.sub(v, 0, j - 1))
        for i = 1, #protected_attachments, 1 do
            if string.find(name, protected_attachments[i]) then
                protected[string.sub(v, j + 1)] = true
                break
            end
        end
    end
    for k, v in pairs(protected) do
        ExecuteRLV("remattach:" .. k .. "=n")
    end
end
```

Appendix B – Custom Lua floater example

Custom Lua floaters are supported via specific functions. Here is an example of how to use them.

First, add a "floater_lua_test.xml" file to your custom skins directory ("~/.secondlife/cvlv-skins/default/xui/en-us/" under Linux and "%appdata%\SecondLife\skins\default\xui\en-us\" under Windows) containing the following XUI floater definition:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<floater name="lua_floater_test" title="test floater"
 can_close="true" can_drag_on_left="false" can_minimize="true" can_resize="true"
 width="284" height="570" min_width="284" min_height="570">
   <text_editor name="textedit1" font="SansSerif" word_wrap="true"
    left="12" bottom="-86" width="260" height="64" follows="left|top|right" />
   <line_editor name="lineedit1" font="SansSerif" follows="left|top|right"
    tool_tip="Give the path or UUID of the folder you want to set as the root folder
for the inventory panel, then press the 'Set folder' button."
    left="12" bottom_delta="-22" width="260" height="18" />
   <inventory_panel name="inventory1" allow_multi_select="true" border="true"
     left="12" bottom_delta="-166" width="260" height="164" follows="left|top|right" />
   <scroll_list name="list1" multi_select="true"
    background_visible="true" draw_border="true" draw_stripes="true"
draw_heading="true"
    left="12" bottom_delta="-88" width="260" height="86" follows="left|top|right">
      <column name="col0" label="Name" dynamicwidth="true" />
      <column name="col1" label="Date" width="120" />
   </scroll_list>
   <name_list name="namelist1" multi_select="true"
    background_visible="true" draw_border="true" draw_stripes="true"
draw_heading="true"
    left="12" bottom_delta="-88" width="260" height="86" follows="left|top|right|
bottom">
      <column name="name" label="Name" dynamicwidth="true" />
   </name_list>
   <spinner name="spin1" label="Amount" font="SansSerif" label_width="64"
    lua_command="print('Spinner=' .. GetValue())"
    decimal_digits="0" increment="1" min_val="0" max_val="255"
    left="12" bottom_delta="-28" width="120" height="24" follows="left|bottom" />
   <check_box name="check1" initial_value="false"
    label="Confirm" font="SansSerifSmall"
    lua_command="print('Check=' .. GetValue())"
    left_delta="140" bottom_delta="6" width="120" height="20" follows="left|bottom" />
   <slider name="slider1" can_edit_text="false" min_val="0" max_val="1.0"
increment="0.1" decimal_digits="1"
    show_text="true" label="Magnitude"
    left="12" bottom_delta="-20" height="16" width="260" follows="left|bottom"/>
   <radio_group name="radio1" draw_border="false"
    lua_command="print('Radio=' .. GetValue())"
    left="12" bottom_delta="-20" width="260" height="16" follows="left|bottom">
      <radio_item name="0" value="0"
       left="0" bottom="-16" width="40" height="16" follows="left|bottom">
         1st choice
      </radio_item>
      <radio_item name="1" value="1"
       left_delta="42" bottom_delta="0" width="40" height="16" follows="left|bottom">
         2nd choice
      </radio_item>
      <radio_item name="2" value="2"
       left_delta="42" bottom_delta="0" width="40" height="16" follows="left|bottom">
         3rd choice
```

```
      </radio_item>
   </radio_group>
   <combo_box name="combo1" allow_text_entry="false" max_chars="20"
    lua_command="print('Combo=' .. GetValue())"
    left="12" bottom_delta="-24" width="120" height="18" follows="left|bottom">
      <combo_item name="1st" type="string" length="1" value="1">
         1st combo item
      </combo_item>
      <combo_item name="2nd" type="string" length="1" value="2">
         2nd combo item
      </combo_item>
   </combo_box>
   <flyout_button name="flyout1" label="Send text" font="SansSerif"
    height="20" width="120" bottom_delta="0" left_delta="130" follows="left|bottom">
      <flyout_button_item value="send_text" name="send_text">
         Send text
      </flyout_button_item>
      <flyout_button_item value="send_line" name="send_line">
         Send line
      </flyout_button_item>
   </flyout_button>
   <button name="button1" label="Set folder" font="SansSerif"
    tool_tip="Attempts to set the root folder of the inventory panel with the path or
UUID of the folder entered in the input line."
    left="12" bottom_delta="-26" width="80" height="20" follows="bottom|right" />
   <button name="button2" label="Cancel" font="SansSerif"
    left_delta="90" bottom_delta="0" width="80" height="20" follows="bottom|right" />
   <button name="button3" label="OK" font="SansSerif"
    left_delta="90" bottom_delta="0" width="80" height="20" follows="bottom|right" />
</floater>
```

Then, add this code to your automation.lua script:

```
function OnLuaFloaterOpen(name, parameter)
   print("Opened floater: " .. name .. " with parameter: " .. parameter)
   if name == "test" then
      SetLuaFloaterCommand("test","flyout1","if GetValue() == 'send_line'
then;print('Input line=' ..
GetLuaFloaterValue(GetFloaterName(),'lineedit1'));else;print('Text=' ..
GetLuaFloaterValue(GetFloaterName(),'textedit1'));end")

SetLuaFloaterCommand("test","button1","SetLuaFloaterValue(GetFloaterName(),'inventory1'
,GetLuaFloaterValue(GetFloaterName(),'lineedit1'))")
      SetLuaFloaterCommand("test","button2","print('Cancel');FloaterClose()")
      SetLuaFloaterCommand("test","button3","print('Magnitude=' ..
GetLuaFloaterValue(GetFloaterName(),'slider1'));FloaterClose()")
      SetLuaFloaterValue("test","list1","<BOLD>Item 1|<1.0,0.5,0.25>2019-01-08")
      SetLuaFloaterValue("test","list1","<ITALIC>Item 2|<red2>2019-01-09")
      SetLuaFloaterValue("test","list1","<BOLD><ITALIC><blue2>Item 3|2019-01-01")
      SetLuaFloaterValue("test","list1","<green3>Item 4|<ITALIC>2019-01-11")
      SetLuaFloaterValue("test","namelist1",GetAgentInfo().id)
      SetLuaFloaterValue("test","namelist1","<GROUP>d1cc93e1-b51e-d0c2-e699-
6c06e3015e9d")
   end
end
```

.../...

```
function OnLuaFloaterClose(name, parameter)
   print("Closed floater: " .. name .. " - Parameter: " .. parameter)
end

function OnLuaFloaterAction(name, control, value)
   print("Floater: " .. name .. " - Control: " .. control .. " - Value: " .. value)
   if value ~= "" and (string.find(control, "list") or string.find(control,
"inventory")) then
      local values = GetLuaFloaterValues(name, control)
      local i
      for i = 1, #values, 1 do
         print("  --> value #" .. i .. " = " .. values[i])
      end
   end
end
```

Finally, log in with the viewer and type in the chat:

```
/lua OpenLuaFloater("test")
```

to test the result.

Appendix C – LSL to Lua scripting example

This is just an example of what can be done when using LSL and Lua in conjunction (here to automatically invite residents to a group based on a list of names stored in a note card). Do not forget to enable "Advanced" -> "Lua scripting" -> "Accept Lua from LSL scripts".

```
// Cool Group Inviter script v1.00 (c)2021 Henri Beauchamp.
// Released under the GPL license: https://www.gnu.org/licenses/gpl-3.0.en.html

// The UUID of the group you want to invite to.
// (this is an invalid group Id in this example).
string GroupId = "e13c93e1-b51e-d0c2-e889-6cf6e3015ead";
// The UUID of the role in the group (keep empty for the default "Member" role).
string RoleId = "";
// The name of the notecard (which must be placed along this script in the
// inviter object) containing the names of the avatars to invite: one per line.
string Notecard = "*Invited*";
// The Lua prefix configured in the Cool VL Viewer "LuaScriptCommandPrefix"
// debug setting.
string LuaPrefix = "/lua ";

key NotecardQueryId;
key AvatarQueryId;
integer Counter;
integer Channel;
integer Handle;
integer AvatarIndex;
list Invited;

SetHoverText(string msg) {
    llSetText(msg, <0.0, 1.0, 1.0>, 1.0);
}

// This function asks the viewer to perform via Lua a group data request and
// sends the reply to our private 'Channel'. The "roles_list_ok" flag in the
// group data Lua table is either 'nil' (when the agent groups have not yet
// been received, or when the group does not exist, or when the agent is not a
// member of this group), 'false' (roles data not yet fully received) or 'true'
// (all data received by the viewer, and we are therefore ready for invites).
CheckGroupData() {
    string cmd = LuaPrefix + "t=GetAgentGroupData(\"" + GroupId;
    cmd += "\");SendChat(\"/" + (string)Channel + "roles_list_ok = \"";
    cmd += " .. tostring(t[\"roles_list_ok\"]))";
    llOwnerSay(cmd);
}

default {
    state_entry() {
        if (llGetInventoryType(Notecard) == INVENTORY_NOTECARD) {
            NotecardQueryId = AvatarQueryId = NULL_KEY;
            Channel = 30000 - (integer)llFrand(10000.0);
            SetHoverText("Group inviter ready. Touch to start.");
        } else {
            // Leave NotecardQueryId and AvatarQueryId as empty strings
            SetHoverText("Missing \"" + Notecard + "\" notecard.");
        }
    }
```

.../…

```
    on_rez(integer param) {
```

```
    llResetScript();
}

changed(integer change) {
    if (change & CHANGED_INVENTORY) {
        llResetScript();
    }
}

listen(integer chan, string name, key id, string msg) {
    if (msg == "roles_list_ok = nil") {
        if (Counter++ == 0) {
            // Wait for the viewer to receive the agent groups list and
            // check again...
            llSleep(5.0);
            CheckGroupData();
            return;
        }
        SetHoverText("No such group or not a member of this group !");
        llListenRemove(Handle);
        Handle = 0;
    } else if (msg == "roles_list_ok = false") {
        // Wait a bit for the viewer to receive all data and check again...
        llSleep(10.0);
        CheckGroupData();
    } else if (msg == "roles_list_ok = true") {
        SetHoverText("Reading invitations list...");
        Invited = [];
        Counter = 0;
        NotecardQueryId = llGetNotecardLine(Notecard, Counter);
        llListenRemove(Handle);
        Handle = 0;
    }
}

touch_start(integer n) {
    if (llDetectedKey(0) != llGetOwner() || Handle != 0 ||
        NotecardQueryId != NULL_KEY || AvatarQueryId != NULL_KEY) {
        // Either the touch was not from our owner or work is in progress,
        // or the notecard is absent (Ids are then empty strings).
        return;
    }
    SetHoverText("Checking for group data availability...");
    // Open our channel to use for sending Lua commands replies back to us
    Handle = llListen(Channel, "", llGetOwner(), "");
    Counter = 0;    // We need to check at least twice...
    CheckGroupData();
}
```

.../...

```
    dataserver(key query_id, string data) {
        if (query_id == NotecardQueryId) {
            if (data == EOF) {
                NotecardQueryId = NULL_KEY;
                Counter = llGetListLength(Invited);
                if (Counter > 0) {
                    AvatarIndex = 0;
                    AvatarQueryId = llRequestUserKey(llList2String(Invited,
                                                                   AvatarIndex));
                }
            } else {
                data = llStringTrim(data, STRING_TRIM);
                if (llStringLength(data) > 2) {
                    SetHoverText("Adding:\n" + data);
                    Invited += data;
                }
                NotecardQueryId = llGetNotecardLine(Notecard, ++Counter);
            }
        } else if (query_id == AvatarQueryId) {
            if ((key)data != NULL_KEY) {
                string tmp = "Invitation " + (string)(AvatarIndex + 1) + "/";
                tmp += (string)Counter + ":\n";
                tmp += llList2String(Invited, AvatarIndex);
                SetHoverText(tmp);
                tmp = LuaPrefix + "AgentGroupInvite(\"" + data;
                tmp += "\", \"" + GroupId;
                if (RoleId != "") {
                    tmp += "\", \"" + RoleId;
                }
                llOwnerSay(tmp + "\")");
            }
            if (++AvatarIndex < Counter) {
                llSleep(0.55);   // Requests throttled server-side at 1.9 per second
                AvatarQueryId = llRequestUserKey(llList2String(Invited, AvatarIndex));
            }
            else
            {
                SetHoverText("Invitations finished.");
                AvatarQueryId = NULL_KEY;
            }
        }
    }
}
```

Appendix D – Constants

This appendix lists constants found in the viewer code and used or returned by various Lua functions or callbacks. I regrouped them here so to save you from browsing the viewer sources to find them (the name of the source file where they can be found is also given for each constants group).
Note that, with the exception of color names (which are recognized when passed as Lua strings) only the **values** associated with these constants should be used (Lua knows nothing and would not recognize the names of the constants themselves), but nothing prevent you from defining them in your Lua scripts (using global variables, for example in #include Lua files) as they are listed below, so to use them by name.

Chat types (indra/newview/llchat.h):

```
CHAT_TYPE_WHISPER = 0
CHAT_TYPE_NORMAL = 1
CHAT_TYPE_SHOUT = 2
CHAT_TYPE_START = 4
CHAT_TYPE_STOP = 5
CHAT_TYPE_DEBUG_MSG = 6
CHAT_TYPE_REGION = 7
CHAT_TYPE_OWNER = 8
CHAT_TYPE_DIRECT = 9    (sent via by the llRegionSayTo() LSL function)
```

Friends observer events (indra/newview/llavatartracker.h):

```
NONE = 0          (no event)
ADD = 1           (friend added)
REMOVE = 2        (friend removed)
ONLINE = 4        (friend online status changed)
POWERS = 8        (friend powers changed)
```

Mean collision types (indra/newview/hbfloaterbump.h):

```
MEAN_INVALID = 0                    (should not be seen happening)
MEAN_BUMP = 1                       (avatar bumps)
MEAN_LLPUSHOBJECT = 2               (object pushes)
MEAN_SELECTED_OBJECT_COLLIDE = 3    (selected/edited objects pushes)
MEAN_SCRIPTED_OBJECT_COLLIDE = 4    (scripted pushes)
MEAN_PHYSICAL_OBJECT_COLLIDE = 5    (physical objects pushes)
```

Camera modes (indra/newview/llagent.h):

```
CAMERA_MODE_THIRD_PERSON = 0
CAMERA_MODE_MOUSELOOK = 1
CAMERA_MODE_CUSTOMIZE_AVATAR = 2
CAMERA_MODE_FOLLOW = 3
```

Asset types (indra/llcommon/llassettype.h):

```
AT_TEXTURE              = 0
AT_SOUND                = 1
AT_CALLINGCARD          = 2
AT_LANDMARK             = 3
AT_CLOTHING             = 5
AT_OBJECT               = 6
AT_NOTECARD             = 7
AT_CATEGORY             = 8         (Inventory folder)
AT_LSL_TEXT             = 10
AT_TEXTURE_TGA          = 12
AT_BODYPART             = 13
AT_SOUND_WAV            = 17
AT_ANIMATION            = 20
AT_GESTURE              = 21
AT_LINK                 = 24
AT_LINK_FOLDER          = 25
AT_MARKETPLACE_FOLDER   = 26
AT_SETTINGS             = 56        (Environment settings)
AT_MATERIAL             = 57        (PBR material)
AT_NONE                 = -1
```

Wearable types (indra/llappearance/llwearabletype.h):

```
WT_SHAPE                = 0
WT_SKIN                 = 1
WT_HAIR                 = 2
WT_EYES                 = 3
WT_SHIRT                = 4
WT_PANTS                = 5
WT_SHOES                = 6
WT_SOCKS                = 7
WT_JACKET               = 8
WT_GLOVES               = 9
WT_UNDERSHIRT           = 10
WT_UNDERPANTS           = 11
WT_SKIRT                = 12
WT_ALPHA                = 13
WT_TATTOO               = 14
WT_PHYSICS              = 15
WT_UNIVERSAL            = 16
WT_NONE                 = -1
```

Environment settings types (indra/llinventory/llsettingstype.h):

```
ST_SKY                  = 0
ST_WATER                = 1
ST_DAYCYCLE             = 2
ST_NONE                 = -1
```

Permission flags (indra/llinventory/llpermissionsflags.h):

```
PERM_TRANSFER = 0x00002000    (bit 13 set)
PERM_MODIFY   = 0x00004000    (bit 14 set)
PERM_COPY     = 0x00008000    (bit 15 set)
PERM_EXPORT   = 0x00010000    (bit 16 set, OpenSim extension not used in SL)
PERM_MOVE     = 0x00080000    (bit 19 set, can grab/translate/rotate object)
PERM_NONE     = 0
PERM_ALL      = 0x7fffffff
PERM_ITEM_UNRESTRICTED = PERM_MODIFY | PERM_COPY | PERM_TRANSFER
```

Teleport states (indra/newview/llagent.h):

```
-- Transition to REQUESTED. Viewer has sent a TeleportRequest to the source
-- simulator.
TELEPORT_START = 1

-- Waiting for source simulator to respond with TeleportFinish.
TELEPORT_REQUESTED = 2

-- Viewer has received destination location from source simulator
TELEPORT_MOVING = 3

-- Transition to ARRIVING. Viewer has received avatar update, etc,
-- from destination simulator
TELEPORT_START_ARRIVAL = 4

-- Waiting while content "pre-caches"
TELEPORT_ARRIVING = 5

-- Teleporting in-sim without showing the progress screen
TELEPORT_LOCAL = 6

-- Viewer not yet ready to receive reliably the TeleportFinish message; TP has
-- been queued (Cool VL Viewer specific state for TP failures workaround).
TELEPORT_QUEUED = 10
```

Region flags (indra/llmessage/llregionflags.h):

The region flags value is a 64 bits unsigned integer, with the following possible flags (which may be ORed together):

```
REGION_FLAGS_ALLOW_DAMAGE                = 0x000000001
REGION_FLAGS_ALLOW_LANDMARK              = 0x000000002
REGION_FLAGS_ALLOW_SET_HOME              = 0x000000004
REGION_FLAGS_RESET_HOME_ON_TELEPORT      = 0x000000008
REGION_FLAGS_SUN_FIXED                   = 0x000000010     (deprecated in SL)
REGION_FLAGS_ALLOW_ACCESS_OVERRIDE       = 0x000000020
REGION_FLAGS_BLOCK_TERRAFORM             = 0x000000040
REGION_FLAGS_BLOCK_LAND_RESELL           = 0x000000080
-- Bits 10 and 11 are not currently used
REGION_FLAGS_SANDBOX                     = 0x000000100
REGION_FLAGS_ALLOW_ENVIRONMENT_OVERRIDE  = 0x000000200
REGION_FLAGS_SKIP_COLLISIONS             = 0x000001000
REGION_FLAGS_SKIP_SCRIPTS                = 0x000002000
REGION_FLAGS_SKIP_PHYSICS                = 0x000004000
REGION_FLAGS_EXTERNALLY_VISIBLE          = 0x000008000
REGION_FLAGS_ALLOW_RETURN_ENCROACHING_OBJECT         = 0x000010000
REGION_FLAGS_ALLOW_RETURN_ENCROACHING_ESTATE_OBJECT  = 0x000020000
REGION_FLAGS_BLOCK_DWELL                 = 0x000040000
REGION_FLAGS_BLOCK_FLY                   = 0x000080000
REGION_FLAGS_ALLOW_DIRECT_TELEPORT       = 0x000100000
REGION_FLAGS_ESTATE_SKIP_SCRIPTS         = 0x000200000
REGION_FLAGS_RESTRICT_PUSHOBJECT         = 0x000400000
-- Bits 24 and 25 are not currently used
REGION_FLAGS_ALLOW_PARCEL_CHANGES        = 0x004000000
REGION_FLAGS_BLOCK_FLYOVER               = 0x008000000
REGION_FLAGS_ALLOW_VOICE                 = 0x010000000
REGION_FLAGS_BLOCK_PARCEL_SEARCH         = 0x020000000
REGION_FLAGS_DENY_AGEUNVERIFIED          = 0x040000000
REGION_FLAGS_DENY_BOTS                   = 0x080000000
-- Bit 32 is not currently used
REGION_FLAGS_ENABLE_ANIMATION_TRACKING   = 0x200000000
```

Parcel flags (indra/llinventory/llparcelflags.h):

The parcel flags value is a 32 bits unsigned integer, with the following possible flags (which may be ORed together):

```
PF_ALLOW_FLY                = 0x00000001
PF_ALLOW_OTHER_SCRIPTS      = 0x00000002
PF_FOR_SALE                 = 0x00000004      (Can buy this land)
PF_ALLOW_LANDMARK           = 0x00000008
PF_ALLOW_TERRAFORM          = 0x00000010
PF_ALLOW_DAMAGE             = 0x00000020
PF_CREATE_OBJECTS           = 0x00000040
PF_FOR_SALE_OBJECTS         = 0x00000080      (Can buy all objects on this land)
PF_USE_ACCESS_GROUP         = 0x00000100
PF_USE_ACCESS_LIST          = 0x00000200
PF_USE_BAN_LIST             = 0x00000400
PF_USE_PASS_LIST            = 0x00000800
PF_SHOW_DIRECTORY           = 0x00001000
PF_ALLOW_DEED_TO_GROUP      = 0x00002000
PF_CONTRIBUTE_WITH_DEED     = 0x00004000
PF_SOUND_LOCAL              = 0x00008000      (Hear sounds in this parcel only)
PF_SELL_PARCEL_OBJECTS      = 0x00010000      (Objects on land are sold with it)
PF_ALLOW_PUBLISH            = 0x00020000      (Allow parcel info in search)
```

```
PF_MATURE_PUBLISH            = 0x00040000      (Parcel info is for mature search)
PF_URL_WEB_PAGE              = 0x00080000      (The media URL is an HTML page)
PF_URL_RAW_HTML              = 0x00100000      (The media URL is raw HTML code)
PF_RESTRICT_PUSHOBJECT       = 0x00200000      (Restricted object push)
PF_DENY_ANONYMOUS            = 0x00400000      (Deny all non identified accounts)
-- Bit 23 and 24 are not currently used
PF_ALLOW_GROUP_SCRIPTS       = 0x02000000      (Allow scripts owned by group)
PF_CREATE_GROUP_OBJECTS      = 0x04000000      (Allow object creation by group)
PF_ALLOW_ALL_OBJECT_ENTRY    = 0x08000000      (Allow all objects parcel entry)
PF_ALLOW_GROUP_OBJECT_ENTRY  = 0x10000000      (Allow group objects entry)
PF_ALLOW_VOICE_CHAT          = 0x20000000      (Allow voice chat on this parcel)
PF_USE_ESTATE_VOICE_CHAN     = 0x40000000      (Use Estate voice channel)
PF_DENY_AGEUNVERIFIED        = 0x80000000      (Deny age-unverified entrance)
```

Render debug flags indexes (indra/newview/llpipeline.h):

For ease of use, the render debug flags are set by Lua via their index (and can only be disabled all at once, by passing a 0 index), corresponding to the values found in LLRenderDebugMask. Note that these indexes are subject to changes from one viewer branch to the other (for example, RENDER_DEBUG_REFLECTION_PROBES_IDX is only honored in the PBR branch of the Cool VL Viewer).

```
RENDER_DEBUG_COMPOSITION_IDX       = 1
RENDER_DEBUG_VERIFY_IDX            = 2
RENDER_DEBUG_BBOXES_IDX            = 3
RENDER_DEBUG_OCTREE_IDX            = 4
RENDER_DEBUG_WIND_VECTORS_IDX      = 5
RENDER_DEBUG_OCCLUSION_IDX         = 6
RENDER_DEBUG_POINTS_IDX            = 7
RENDER_DEBUG_TEXTURE_PRIORITY_IDX  = 8
RENDER_DEBUG_TEXTURE_AREA_IDX      = 9
RENDER_DEBUG_FACE_AREA_IDX         = 10
RENDER_DEBUG_PARTICLES_IDX         = 11
RENDER_DEBUG_TEXTURE_ANIM_IDX      = 12
RENDER_DEBUG_LIGHTS_IDX            = 13
RENDER_DEBUG_BATCH_SIZE _IDX       = 14
RENDER_DEBUG_RAYCAST_IDX           = 15
RENDER_DEBUG_AVATAR_DRAW_INFO_IDX  = 16
RENDER_DEBUG_SHADOW_FRUSTA_IDX     = 17
RENDER_DEBUG_SCULPTED_IDX          = 18
RENDER_DEBUG_AVATAR_VOLUME_IDX     = 19
RENDER_DEBUG_AVATAR_JOINTS_IDX     = 20
RENDER_DEBUG_AGENT_TARGET_IDX      = 21
RENDER_DEBUG_UPDATE_TYPE_IDX       = 22
RENDER_DEBUG_PHYSICS_SHAPES_IDX    = 23
RENDER_DEBUG_NORMALS_IDX           = 24
RENDER_DEBUG_LOD_INFO_IDX          = 25
RENDER_DEBUG_RENDER_COMPLEXITY_IDX = 26
RENDER_DEBUG_ATTACHMENT_INFO_IDX   = 27
RENDER_DEBUG_TEXTURE_SIZE_IDX      = 28
RENDER_DEBUG_REFLECTION_PROBES_IDX = 29
```

Agent control flags (indra/newview/llagent.h):

The agent control flags 32 bits value is a bit field. The bit numbers in the controls flag are defined as follow (e.g. test for bit 13 in the control flags to see if the agent is flying):

```
CONTROL_AT_POS_INDEX          = 0
CONTROL_AT_NEG_INDEX          = 1
CONTROL_LEFT_POS_INDEX        = 2
CONTROL_LEFT_NEG_INDEX        = 3
CONTROL_UP_POS_INDEX          = 4
CONTROL_UP_NEG_INDEX          = 5
CONTROL_PITCH_POS_INDEX       = 6
CONTROL_PITCH_NEG_INDEX       = 7
CONTROL_YAW_POS_INDEX         = 8
CONTROL_YAW_NEG_INDEX         = 9
CONTROL_FAST_AT_INDEX         = 10
CONTROL_FAST_LEFT_INDEX       = 11
CONTROL_FAST_UP_INDEX         = 12
CONTROL_FLY_INDEX             = 13
CONTROL_STOP_INDEX            = 14
CONTROL_FINISH_ANIM_INDEX     = 15
CONTROL_STAND_UP_INDEX        = 16
CONTROL_SIT_ON_GROUND_INDEX   = 17
CONTROL_MOUSELOOK_INDEX       = 18
CONTROL_NUDGE_AT_POS_INDEX    = 19
CONTROL_NUDGE_AT_NEG_INDEX    = 20
CONTROL_NUDGE_LEFT_POS_INDEX  = 21
CONTROL_NUDGE_LEFT_NEG_INDEX  = 22
CONTROL_NUDGE_UP_POS_INDEX    = 23
CONTROL_NUDGE_UP_NEG_INDEX    = 24
CONTROL_TURN_LEFT_INDEX       = 25
CONTROL_TURN_RIGHT_INDEX      = 26
CONTROL_AWAY_INDEX            = 27
CONTROL_LBUTTON_DOWN_INDEX    = 28
CONTROL_LBUTTON_UP_INDEX      = 29
CONTROL_ML_LBUTTON_DOWN_INDEX = 30
CONTROL_ML_LBUTTON_UP_INDEX   = 31
TOTAL_CONTROLS                = 32
```

This translates in the following hexadecimal values (which may be ORed) for the control flags:

```
AGENT_CONTROL_AT_POS          = 0x00000001
AGENT_CONTROL_AT_NEG          = 0x00000002
AGENT_CONTROL_LEFT_POS        = 0x00000004
AGENT_CONTROL_LEFT_NEG        = 0x00000008
AGENT_CONTROL_UP_POS          = 0x00000010
AGENT_CONTROL_UP_NEG          = 0x00000020
AGENT_CONTROL_PITCH_POS       = 0x00000040
AGENT_CONTROL_PITCH_NEG       = 0x00000080
AGENT_CONTROL_YAW_POS         = 0x00000100
AGENT_CONTROL_YAW_NEG         = 0x00000200

AGENT_CONTROL_FAST_AT         = 0x00000400
AGENT_CONTROL_FAST_LEFT       = 0x00000800
AGENT_CONTROL_FAST_UP         = 0x00001000

AGENT_CONTROL_FLY             = 0x00002000
AGENT_CONTROL_STOP            = 0x00004000
AGENT_CONTROL_FINISH_ANIM     = 0x00008000
AGENT_CONTROL_STAND_UP        = 0x00010000
AGENT_CONTROL_SIT_ON_GROUND   = 0x00020000
AGENT_CONTROL_MOUSELOOK       = 0x00040000
```

```
AGENT_CONTROL_NUDGE_AT_POS    = 0x00080000
AGENT_CONTROL_NUDGE_AT_NEG    = 0x00100000
AGENT_CONTROL_NUDGE_LEFT_POS  = 0x00200000
AGENT_CONTROL_NUDGE_LEFT_NEG  = 0x00400000
AGENT_CONTROL_NUDGE_UP_POS    = 0x00800000
AGENT_CONTROL_NUDGE_UP_NEG    = 0x01000000
AGENT_CONTROL_TURN_LEFT       = 0x02000000
AGENT_CONTROL_TURN_RIGHT      = 0x04000000

AGENT_CONTROL_AWAY            = 0x08000000

AGENT_CONTROL_LBUTTON_DOWN    = 0x10000000
AGENT_CONTROL_LBUTTON_UP      = 0x20000000
AGENT_CONTROL_ML_LBUTTON_DOWN = 0x40000000
AGENT_CONTROL_ML_LBUTTON_UP   = 0x80000000
```

Some combinations (ORing) of the above hexadecimal values are used to determine what type of controls the agent is subjected to:

```
AGENT_CONTROL_AT = AGENT_CONTROL_AT_POS | AGENT_CONTROL_AT_NEG |
                   AGENT_CONTROL_NUDGE_AT_POS | AGENT_CONTROL_NUDGE_AT_NEG

AGENT_CONTROL_LEFT = AGENT_CONTROL_LEFT_POS | AGENT_CONTROL_LEFT_NEG |
                     AGENT_CONTROL_NUDGE_LEFT_POS | AGENT_CONTROL_NUDGE_LEFT_NEG

AGENT_CONTROL_UP = AGENT_CONTROL_UP_POS | AGENT_CONTROL_UP_NEG |
                   AGENT_CONTROL_NUDGE_UP_POS | AGENT_CONTROL_NUDGE_UP_NEG

AGENT_CONTROL_HORIZONTAL = AGENT_CONTROL_AT | AGENT_CONTROL_LEFT

AGENT_CONTROL_NOT_USED_BY_LSL = AGENT_CONTROL_FLY | AGENT_CONTROL_STOP |
                                AGENT_CONTROL_FINISH_ANIM |
                                AGENT_CONTROL_STAND_UP |
                                AGENT_CONTROL_SIT_ON_GROUND |
                                AGENT_CONTROL_MOUSELOOK | AGENT_CONTROL_AWAY

AGENT_CONTROL_MOVEMENT = AGENT_CONTROL_AT | AGENT_CONTROL_LEFT |
                         AGENT_CONTROL_UP

AGENT_CONTROL_ROTATION = AGENT_CONTROL_PITCH_POS | AGENT_CONTROL_PITCH_NEG |
                         AGENT_CONTROL_YAW_POS | AGENT_CONTROL_YAW_NEG

AGENT_CONTROL_NUDGE = AGENT_CONTROL_NUDGE_AT_POS | AGENT_CONTROL_NUDGE_AT_NEG |
                      AGENT_CONTROL_NUDGE_LEFT_POS |
                      AGENT_CONTROL_NUDGE_LEFT_NEG;
```

Group powers (indra/newview/roles_constants.h):

The group powers is a 64 bits unsigned integer which is a bitmap in which each bit represents a given power. The powers values below may therefore be ORed together to define the full power map in a group.

```
GP_NO_POWERS  = 0
GP_ALL_POWERS = 0xffffffffffffffff

-- Membership
GP_MEMBER_INVITE        = 1 << 1
GP_MEMBER_EJECT         = 1 << 2
GP_MEMBER_OPTIONS       = 1 << 3
GP_MEMBER_VISIBLE_IN_DIR  = 1 << 47

-- Roles
GP_ROLE_CREATE                = 1 << 4
GP_ROLE_DELETE                = 1 << 5
GP_ROLE_PROPERTIES            = 1 << 6
GP_ROLE_ASSIGN_MEMBER_LIMITED = 1 << 7
GP_ROLE_ASSIGN_MEMBER         = 1 << 8
GP_ROLE_REMOVE_MEMBER         = 1 << 9
GP_ROLE_CHANGE_ACTIONS        = 1 << 10

-- Group identity
GP_GROUP_CHANGE_IDENTITY      = 1 << 11

-- Parcel management
GP_LAND_DEED                  = 1 << 12
GP_LAND_RELEASE               = 1 << 13
GP_LAND_SET_SALE_INFO         = 1 << 14
GP_LAND_DIVIDE_JOIN           = 1 << 15

-- Parcel identity
GP_LAND_FIND_PLACES           = 1 << 17
GP_LAND_CHANGE_IDENTITY       = 1 << 18
GP_LAND_SET_LANDING_POINT     = 1 << 19

-- Parcel settings
GP_LAND_CHANGE_MEDIA          = 1 << 20
GP_LAND_EDIT                  = 1 << 21
GP_LAND_OPTIONS               = 1 << 22

-- Parcel powers
GP_LAND_ALLOW_EDIT_LAND       = 1 << 23
GP_LAND_ALLOW_FLY             = 1 << 24
GP_LAND_ALLOW_CREATE          = 1 << 25
GP_LAND_ALLOW_LANDMARK        = 1 << 26
GP_LAND_ALLOW_SET_HOME        = 1 << 28
GP_LAND_ALLOW_HOLD_EVENT      = 1 << 41
GP_LAND_ALLOW_ENVIRONMENT     = 1 << 46
GP_LAND_GARDENING             = 1 << 35

-- Parcel access
GP_LAND_MANAGE_ALLOWED        = 1 << 29
GP_LAND_MANAGE_BANNED         = 1 << 30
GP_LAND_MANAGE_PASSES         = 1 << 31
GP_LAND_ADMIN                 = 1 << 32
```

```
-- Parcel contents
GP_LAND_RETURN_GROUP_SET      = 1 << 33
GP_LAND_RETURN_NON_GROUP      = 1 << 34
GP_LAND_RETURN_GROUP_OWNED    = 1 << 48

-- Object management
GP_OBJECT_DEED                = 1 << 36
GP_OBJECT_MANIPULATE          = 1 << 38
GP_OBJECT_SET_SALE            = 1 << 39

-- Accounting
GP_ACCOUNTING_ACCOUNTABLE     = 1 << 40

-- Group chat moderation
GP_SESSION_JOIN               = 1 << 16
GP_SESSION_VOICE              = 1 << 27
GP_SESSION_MODERATOR          = 1 << 37
GP_GROUP_BAN_ACCESS           = 1 << 51

-- Experience tools related
GP_EXPERIENCE_ADMIN           = 1 << 49
GP_EXPERIENCE_CREATOR         = 1 << 50
```

Predefined color names (indra/llmath/llcolor4.cpp):

Each name (Lua string) corresponds to a *red*, *green*, *blue*, *alpha* quadruplet of floating point component values (between 0.0 and 1.0), as follow:

```
"red" is 1.0, 0.0, 0.0, 1.0
"green" is 0.0, 1.0, 0.0, 1.0
"blue" is 0.0, 0.0, 1.0, 1.0
"black" is 0.0, 0.0, 0.0, 1.0
"yellow" is 1.0, 1.0, 0.0, 1.0
"magenta" is 1.0, 0.0, 1.0, 1.0
"cyan" is 0.0, 1.0, 1.0, 1.0
"white" is 1.0, 1.0, 1.0, 1.0
"smoke" is 0.5, 0.5, 0.5, 0.5
"grey" is 0.5, 0.5, 0.5, 1.0
"orange" is 1.0, 0.5, 0.0, 1.0
"purple" is 0.6, 0.2, 0.8, 1.0
"pink" is 1.0, 0.5, 0.8, 1.0
"transparent" is 0.0, 0.0, 0.0, 0.0

"grey1" is 0.8, 0.8, 0.8, 1.0
"grey2" is 0.6, 0.6, 0.6, 1.0
"grey3" is 0.4, 0.4, 0.4, 1.0
"grey4" is 0.3, 0.3, 0.3, 1.0
"grey5" is 0.125, 0.125, 0.125, 1.0

"red0" is 0.5, 0.0, 0.0, 1.0
"red1" is 1.0, 0.0, 0.0, 1.0
"red2" is 0.6, 0.0, 0.0, 1.0
"red3" is 1.0, 0.2, 0.2, 1.0
"red4" is 0.5, 0.1, 0.1, 1.0
"red5" is 0.8, 0.1, 0.0, 1.0

"green0" is 0.0, 0.5, 0.0, 1.0
"green1" is 0.0, 1.0, 0.0, 1.0
"green2" is 0.0, 0.6, 0.0, 1.0
"green3" is 0.0, 0.4, 0.0, 1.0
"green4" is 0.0, 1.0, 0.4, 1.0
"green5" is 0.2, 0.6, 0.4, 1.0
"green6" is 0.4, 0.6, 0.2, 1.0
"green7" is 0.6, 1.0, 0.4, 1.0
"green8" is 0.4, 1.0, 0.6, 1.0
"green9" is 0.6, 1.0, 0.6, 1.0

"blue0" is 0.0, 0.0, 0.5, 1.0
"blue1" is 0.0, 0.0, 1.0, 1.0
"blue2" is 0.0, 0.4, 1.0, 1.0
"blue3" is 0.2, 0.2, 0.8, 1.0
"blue4" is 0.0, 0.0, 0.6, 1.0
"blue5" is 0.4, 0.2, 1.0, 1.0
"blue6" is 0.4, 0.5, 1.0, 1.0
"blue7" is 0.0, 0.0, 0.5, 1.0

"yellow1" is 1.0, 1.0, 0.0, 1.0
"yellow2" is 0.6, 0.6, 0.0, 1.0
"yellow3" is 0.8, 1.0, 0.2, 1.0
"yellow4" is 1.0, 1.0, 0.4, 1.0
"yellow5" is 0.6, 0.4, 0.2, 1.0
"yellow6" is 1.0, 0.8, 0.4, 1.0
"yellow7" is 0.8, 0.8, 0.0, 1.0
"yellow8" is 0.8, 0.8, 0.2, 1.0
"yellow9" is 0.8, 0.8, 0.4, 1.0
```

```
"orange1" is 1.0, 0.8, 0.0, 1.0
"orange2" is 1.0, 0.6, 0.0, 1.0
"orange3" is 1.0, 0.4, 0.2, 1.0
"orange4" is 0.8, 0.4, 0.0, 1.0
"orange5" is 0.9, 0.5, 0.0, 1.0
"orange6" is 1.0, 0.8, 0.2, 1.0

"magenta1" is 1.0, 0.0, 1.0, 1.0
"magenta2" is 0.6, 0.2, 0.4, 1.0
"magenta3" is 1.0, 0.4, 0.6, 1.0
"magenta4" is 1.0, 0.2, 0.8, 1.0

"purple1" is 0.6, 0.2, 0.8, 1.0
"purple2" is 0.8, 0.2, 1.0, 1.0
"purple3" is 0.6, 0.0, 1.0, 1.0
"purple4" is 0.4, 0.0, 0.8, 1.0
"purple5" is 0.6, 0.0, 0.8, 1.0
"purple6" is 0.8, 0.0, 0.6, 1.0

"pink1" is 1.0, 0.5, 0.8, 1.0
"pink2" is 1.0, 0.8, 0.9, 1.0

"cyan1" is 0.0, 1.0, 1.0, 1.0
"cyan2" is 0.4, 0.8, 0.8, 1.0
"cyan3" is 0.0, 1.0, 0.6, 1.0
"cyan4" is 0.6, 1.0, 1.0, 1.0
"cyan5" is 0.2, 0.6, 1.0, 1.0
"cyan6" is 0.2, 0.6, 0.6, 1.0
```

Alphabetical Index

AddMute....13
AgentAutoPilotFollow....18
AgentAutoPilotLoad....18
AgentAutoPilotRecord....18
AgentAutoPilotRemove....18
AgentAutoPilotReplay....18
AgentAutoPilotSave....18
AgentAutoPilotStop....18
AgentAutoPilotToPos....17
AgentGroupInvite....12, 63
AgentPuppetryStart....19
AgentPuppetryStop....19
AgentRotate....17
AgentSit....17, 53
AgentStand....17
AlertDialogResponse....24
ApplyDaySettings....21
ApplySkySettings....21, 48
ApplyWaterSettings....21, 48
Automation script....
  automation.lua....3
  Load a new automation script....3
  Re-load the currently running automation script....3
  Stop automation script....3
AutomationMessage....6, 36, 55
AutomationRequest....7, 36, 46
BlockSound....13
BrowseToURL....22
CallbackAfter....6, 48, 55
CancelNotification....24
CloseFloater....25
CloseIMSession....12
CloseLuaFloater....26
Constants....
  Agent control flags....69
  Asset types....65
  Camera modes....64
  Chat types....64
  Environment settings types....65
  Friends observer events....64
  Group powers....71
  Mean collision types....64
  Parcel flags....67
  Permission flags....66
  Predefined color names....73
  Region flags....67
  Render debug flags indexes....68
  Teleport states....66
  Wearable types....65
Debug settings....
  BackgroundYieldTime....7, 36
  FrameRateLimit....36
  LuaAcceptDbusCommands....4
  LuaAcceptScriptCommands....3
  LuaAutomationScript....3
  LuaCommandPrefix....3
  LuaFloaterRect....25
  LuaMaxAgentPosHistorySize....17
  LuaOnAveragedFPSInterval....36
  LuaScriptCommandPrefix....3, 61

LuaTimeout....4
LuaTimeoutForScriptFile....4
PuppetryCamera....19
PuppetryCameraOption....19
RestrainedLoveLuaNoBlacklist....22
UseExternalBrowser....42
DecodeBase64....8
DecodeJSON....8
DeleteInventoryLink....20
DerenderObject....8, 48
DispatchSLURL....22, 41, 54
EncodeBase64....8
EncodeJSON....8
ExecuteRLV....22, 49, 57
FindInventoryObject....19
ForceQuit....8
GetAgentAttachments....20, 57
GetAgentFriends....9
GetAgentGroupData....11, 61
GetAgentGroups....11
GetAgentInfo....9
GetAgentPosHistory....17
GetAgentPushes....9
GetAgentRotation....17
GetAgentWearables....20, 57
GetAvatarName....10
GetBlockedSounds....13
GetCameraMode....15
GetClipBoardString....7
GetDebugSetting....33, 53
GetDerenderedObjects....8
GetEESettingsList....21
GetEnvironmentStatus....22
GetFloaterControls....28
GetFloaterCtrlState....29
GetFloaterInstances....28.
GetFloaterList....29
GetFloaterName....59
GetFrameTimeSeconds....7
GetGlobalData....33
GetGridSimAndPos....16, 50
GetGroupName....11
GetHTTP....23, 40
GetIMSession....12
GetLuaFloaterListLine....27
GetLuaFloaterValue....27, 59
GetLuaFloaterValues....27, 60
GetObjectInfo....8, 36
GetParcelInfo....16
GetPerAccountData....34, 48
GetRadarData....14
GetRadarList....14
GetSourceFileName....6
GetThreadID....35, 46
GetTimeSinceEpoch....7
GetTimeStamp....7
GetWatchdogState....6
GetWLSettingsList....22
GiveInventory....19
GroupNotificationAcceptOffer....24
HasThread....35, 46
IsAdmin....10
IsAgentFriend....9
IsAgentGroup....11

IsAvatar....9
IsBlockedSound....13
IsMuted....13
IsObject....8
IsUUID....7
ListInventoryFolder....20
Lua commands sources....
  Accept Lua from LSL scripts....3
  dbuslua.tk....4
  Lua command lines....3
  Lua commands from D-Bus....4
Lua console....3
Lua UI callback pseudo-functions....
  DialogClose....25, 55
  FloaterClose....27, 59
  GetDialogInput....25
  GetFloaterName....27
  GetFloaterParam....27
  GetPickedObjectID....31
  GetValue....27, 58
  nop....31
  SetDialogInput....25
lua_command....26
LuaContextMenu....31, 43
LuaPieMenuSlice....6, 31, 56
MakeDialog....25, 55
MakeInventoryLink....20
MinimizeWindow....7
MoveToInventoryFolder....20
NewInventoryFolder....20
NotificationResponse....24
OnAgentBaked....38
OnAgentOccupationChange....38
OnAgentPush....38, 53
OnAlertDialog....24, 41
OnAutomationMessage....6, 36, 49
OnAutomationRequest....7, 36
OnAutoPilotFinished....17, 37
OnAvatarRezzing....38, 56
OnAveragedFPS....36
OnCameraModeChange....38
OnChatTextColoring....40, 54
OnContextMenu....31, 43
OnFailedTPSimChange....37, 55
OnFriendStatusChange....38
OnGroupNotification....41
OnHTTPReply....23, 40
OnInstantMsg....40
OnJoystickButtons....37
OnLogin....36, 48
OnLuaDialogClose....25, 43, 55
OnLuaFloaterAction....26., 42, 60
OnLuaFloaterClose....25, 42, 60
OnLuaFloaterOpen....25., 42, 59
OnLuaPieMenu....31, 43, 56
OnNotification....41
OnObjectInfoReply....8, 36
OnParcelChange....37
OnPickAvatar....31, 42
OnPickInventoryItem....32, 42
OnPositionChange....17, 37
OnRadar....39
OnRadarMark....39
OnRadarSelection....39

OnRadarTrack....39
OnReceivedChat....40, 54
OnRegionChange....37
OnRLVAnswerOnChat....39
OnRLVHandleCommand....39
OnScriptDialog....41
OnSendChat....40, 52
OnSideBarVisibilityChange....43, 50
OnSignal....36, 45
OnSLURLDispatch....22, 41
OnTPStateChange....37, 55
OnURLDispatch....42
OnWindlightChange....38, 54
OpenFloater....24, 48
OpenLuaFloater....25, 60
OpenNotification....24, 49
OverlayBarLuaButton....6, 29, 49, 55
PasteToContextHandler....31, 43
PickAvatar....31, 42
PickInventoryItem....32, 42
PlayUISound....24
PostHTTP....23, 40
print....6, 44
RemoveMute....13
RenderDebugInfo....34
ScriptDialogResponse....24
SendChat....12, 61
SendIM....12, 53
SendSignal....35, 44, 46
SetAgentGroup....12
SetAgentOccupation....9
SetAgentTyping....12
SetAvatarMinimapColor....10, 56
SetAvatarNameTagColor....10, 56
SetCameraFocus....15
SetCameraMode....15
SetClipBoardString....7, 43
SetDayTime....21
SetDebugSetting....33, 53
SetGlobalData....33
SetLuaFloaterCommand....27
SetLuaFloaterEnabled....26
SetLuaFloaterInvFilter....28
SetLuaFloaterValue....25, 27, 59
SetLuaFloaterVisible....26
SetPerAccountData....34, 54
SetRadarMarkChar....15
SetRadarMarkColor....15
SetRadarNameColor....14, 56
SetRadarToolTip....14
SetRadarTracking....15
SetVisualMute....13
ShowFloater....28
ShowLuaFloater....26
SideBarButton....6, 30, 48
SideBarButtonDisable....30
SideBarButtonHide....30
SideBarButtonToggle....30, 48
SideBarHide....30, 43, 48
SideBarHideOnRightClick....30, 43, 48
Sleep....35, 44
StartThread....35, 44
StatusBarLuaIcon....6, 29, 50
StopThread....35, 45

TeleportAgentHome....17
TeleportAgentToPos....17, 55
ThreadRun....44, 46
trusted scripts....5
VIEWER_NAME....5
VIEWER_VERNUM....5
VIEWER_VERSION....5
warn....6, 44
  @off....6
  @on....6
  @prefix....6
  @tochat....6
Watchdog....4