Lua scripting and viewer automation feature

Table of Contents

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".

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).

- 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).

- 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).

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.

Miscellaneous

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

- `VIEWER_NAME` contains the viewer name ("Cool VL Viewer").
- `VIEWER_VERSION` contains the viewer version (e.g. "1.26.20.5").

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 !

Chapter 2 – Lua commands

Here are the currently implemented viewer-specific Lua commands:

Printing text

- The print() Lua command is replaced with a viewer-specific one that prints into the viewer chat history and chat console. The printed messages are all prefixed with "Lua: " and colored with the "system" messages color.

- The warn() Lua command (new in Lua v5.4) is supported and, by default, causes a "Lua: WARNING: ..." message to be printed into the viewer chat history and chat console. The printed messages are 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).

- 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 v5.3), 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 via 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 via 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.

- 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.

- GetClipBoardString() returns the current string contained in the paste string buffer of the viewer (i.e. any string that got copied into the clipboard from a viewer input line or text editor).

- 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.

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 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.
  NOTE: this function is only available from the automation script (and its threads), since it requires the presence of an appropriate Lua callback in the automation script to be of any use.

- 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 (as defined in indra/newview/llagent.h, ECameraMode enum), "control_flags" is a Lua integer reflecting the current control flags (as defined in indra/llcommon/llagentconstants.h, AGENT_CONTROL_* constants), "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*), "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.

- 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]) sets the mini-map avatar dot 'color' (which must be a Lua string containing either the name of a known color, as defined in indra/llmath/llcolor4.cpp, 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]) sets the avatar tag text to 'color' (which must be a Lua string containing either the name of a known color, as defined in indra/llmath/llcolor4.cpp, 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.

- 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 (as defined in indra/newview/roles_constants.h, GP_* constants).
  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.
  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).

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 automation script and its threads (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.

- 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

- 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 value, "name" key, with the avatar legacy name as a string value, "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 value, "global_x" and ""global_y" keys, each associated with a Lua number value reflecting the avatar position on the grid, "altitude" key associated with the Lua number value reflecting the avatar altitude, "marked" key with a boolean value which is *true* if the avatar is flagged as marked in the Radar, or *false* otherwise, "focused" with a boolean value which is *true* if the camera is focused on that avatar via the Radar, or *false* otherwise, "drawn" with a boolean value which is *true* if the avatar is within draw distance, or *false* otherwise, "in_sim" which is *true* if the avatar is present in the same sim as yours, or *false* otherwise, and "entry_age" with a Lua number reflecting the age of the entry as a number of seconds since last refresh by the Radar.

- 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 (corresponding to the ECameraMode enumeration in indra/newview/llagent.h).

- 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, corresponding to the flags defined in indra/llmessage/llregionflags.h), 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, and finally the avatar "altitude" (Lua number value) in meters.
  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, corresponding to the flags defined in indra/llinventory/llparcelflags.h), 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]]) takes global grid coordinates (the altitude, 'z', may be omitted) and attempts to teleport the agent (your avatar) over there. If a boolean is passed as the 'preserve_look_at' fourth parameter ('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.

- 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 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 (as defined in indra/llcommon/llassettype.h) 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.

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"), "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 workds 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). I'm debating whether making all dispatch requests un-trusted when they come from LSL scripts (some dispatch actions are unavailable to un-trusted SLURLs), even though the latter scripts are always scripts you own (feel free to advocate for or against it in in the feature request thread linked to below).

User interface related functions

- ScriptDialogResponse(dialog_id, btn_name) replies to the script dialog 'dialog_id' (string type), like if the button 'btn_name' got pressed. Returns a *true* boolean if successful (dialog still valid and button present in it) or *false* otherwise.

- 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 "area search", "avatar info" (needs an avatar id), "camera controls", "chat", "debug settings", "experiences", "friends", "group info" (needs a group id), "groups", "inspect" (needs an object or avatar id), "instant messages", "inventory", "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", "search", "snapshot", "sounds list", "teleport history".

- CloseFloater(name) closes a specific floater. Currently recognized names are "area search", "camera controls", "chat", "debug settings", "experiences", "friends", "groups", "inspect avatar", "inspect object", "instant messages", "inventory", "map", "media filter", "mini map", "mute list", "nearby media", "notifications", path-finding "characters" and "linksets", "preferences", "pushes", "radar", "search", "snapshot", "sounds list", "teleport history".

- 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]) tries and opens a custom floater which XUI definition should be held in an XML file placed in the 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" which is a reserved name (used by the MakeDialog() function). The optional 'parameter' (Lua string) is associated with the floater and passed to the Lua OnLuaFloaterOpen() and OnLuaFloaterClose() automation script callbacks (see below). 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"), 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, the OnLuaFloaterOpen() automation script callback is called (when present) at the end of the floater post-build method.

- 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.

- 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.

- 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 UTF-8 special symbols, 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 icon 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 tooltip are modified without changing the existing command (which allows to change a side bar button aspect without modifying its action).
  NOTE 1: icons are searched for in the skin folder and skin overlay folders, including the 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 it 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 which was white-listed via the "LuaSettingsWhiteList" debug setting variable, 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*.

- 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.

- PickInventoryItem(asset_type[, sub_type[, allow_multiple[, exclude_library]]]) is only available to the automation script and pops up an inventory item picker floater that will list all inventory items matching the requested 'asset_type' (Lua integer; see the indra/llcommon/llassettype.h source file for valid LLAssetType values in the EType enum), optionally narrowing the items to 'sub_type' (Lua integer which defaults to -1 for no sub-type, and is only of use with wearable and environment settings items: see the EType enums in the indra/llappearance/llwearabletype.h and indra/llinventory/llsettingstype.h source files for valid sub-type values). 'allow_multiple' is an optional Lua boolean (defaulting to *false*) which governs whether multiple items can be selected or not, and 'exclude_library' is another optional Lua boolean (defaulting to *true*) and which governs whether to show the inventory Library items or not. When the floater is closed, the OnPickInventoryItem() callback of the automation script is called.

- PickAvatar([allow_multiple]) is only available to the automation script 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. When the floater is closed, the OnPickAvatar() callback of the automation script is called.

- 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.

Debug and persistent settings related functions

- GetDebugSetting(setting_name) attempts to retrieve the value of a debug setting variable (only integer, boolean, float and string types variables are currently supported). This fails (and returns *nil*) if the setting name is not white-listed in either the "LuaSettingsWhiteList" or "LuaSettingsReadOnlyWhiteList" debug setting variables.

- SetDebugSetting(setting_name, value) attempts to set the value of a debug setting variable (only integer, boolean, float and string types variables are currently supported). This fails (and returns *false* instead of *true* on success) if the setting name is not white-listed in the "LuaSettingsWhiteList" debug setting variable. 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 it via this function; you may however still read the value of hidden settings with GetDebugSetting().

- SetGlobalData(data) is only available from the automation script 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 automation script 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

- SetPerAccountData(data) is only available from the automation script 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 automation script 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 (defined in indra/newview/llpipeline.h).
  NOTE: some features are not available/enabled in the Cool VL viewer and some names are, alas, rather misleading (e.g. the feature corresponding to the avatar complexity score is "RENDER_DEBUG_AVATAR_DRAW_INFO" and not "RENDER_DEBUG_RENDER_COMPLEXITY"). Especially interesting feature numbers are 18 (avatar complexity info) and 30 ("RENDER_DEBUG_ATTACHMENT_INFO" corresponding to attachments memory usage and surface area info).

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) is called on successful login into a grid. The 'location' parameter is a Lua table containing the same data as what GetGridSimAndPos() returns. 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 GetObecjtInfo() 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 (corresponding to the ETeleportState enumeration defined in indra/newview/llagent.h) 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 (as defined in indra/newview/llmeancollisiondata.h for EMeanCollisionType: 1 for avatar bumps, 2 for object pushes, 3 for selected objects pushes, 4 for scripted pushes, 5 for physical objects pushes) 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 refelecting the type of chage that occured with bit 1 set for ADD, bit 2 for REMOVE, bit 4 for ONLINE, bit 8 for POWERS (see the enum in the LLFriendObserver class which is defined in indra/newview/llcallingcard.h), 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 ECameraMode enumeration defined in indra/newview/llagent.h. 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 EEP parcel or custom/local environments to Windlight translation/rendering is enabled. This callback shall not return any value.

Radar related callbacks

- 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.

- OnRadar(avatar_id, name, range, marked) is called each time the radar announces an incoming avatar (provided it was configured to do so !). '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, 3 for chat range. 'marked' is a Lua boolean which is *true* when the avatar was marked by the user in the Radar floater. 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 (corresponding to the e_chat_type enum in indra/newview/llchat.h) 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 color, as defined in indra/llmath/llcolor4.cpp, 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.

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.

User interface related callbacks

- OnNotification(name, 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), 'id' (Lua string) being the UUID of the notification, 'message' (Lua string) the text displayed in it. This callback shall not return any value.
  NOTE: for now alerts and group notifications do not trigger such an event (I did not see any use for it, but if you have a use case, let me know), and notifications or notify tips spawned via OpenNotification() are also voluntarily filtered out (to avoid potential infinite loops).

- 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 key between "button1" and "button12" and the value being the button name. This callback shall not return any value.

- OnPickInventoryItem(items) is called when the inventory items picker floater opened by PickInventoryItem() gets closed. 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.

- 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.

- 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):

```
-- Global variables used in the automation script

known_ids = {}
known_sessions = {}
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(1, "\u{2699}", "OpenFloater('preferences')", "Opens the Preferences
floater")
    -- Setup the side bar button for spelling language toggling:
    ToggleSpellCheckLanguage(false)
    SideBarButton(3, "inv_item_landmark_visited.tga", "OpenFloater('teleport
history')", "Opens the Teleport history")
    SideBarButton(4, "inv_item_mesh.tga", "nop", "Toggles mesh queue info")
    SideBarButtonToggle(4, "DebugShowMeshQueue")
    SideBarButton(5, "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
    -- 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 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 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(2, "En", "AutomationMessage('language')", "Toggles the spell
checking language to English (US)")
    else
        SideBarButton(2, "Fr", "AutomationMessage('language')", "Toggles the spell
checking language to French")
    end
end
```

.../...

```
-- This callback receives the messages sent by the various buttons we setup via
-- the AutomationMessage() function.
function OnAutomationMessage(text)
    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(6, "shield_off.png", "AutomationMessage('shield-off')", "Disables
anti-griefers features")
        OpenNotification(0, "Anti-griefer features enabled")
    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(6, "shield_on.png", "AutomationMessage('shield-on')", "Enables
anti-griefers features")
        OpenNotification(0, "Anti-griefer features disabled")
    elseif text == "cancel-auto-tp" then
        tp_retry = false
        OverlayBarLuaButton("", "")
    elseif text == "language" then
        ToggleSpellCheckLanguage(true)
    elseif text == "protect" then
        SideBarButton(8, "9beb8cdd-3dce-53c2-b28e-e1f3bc2ec0a4.tga",
"AutomationMessage('unprotect')", "Unlock protected layers and attachments")
        SideBarButtonToggle(8, true)
        LockAttachmentsAndLayers()
        OpenNotification(0, "Protected attachments and layers locked")
    elseif text == "unprotect" then
        SideBarButton(8, "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")
    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
```

.../...

```
-- 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 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 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 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
            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
```

.../...

```
-- 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 = tonumbe(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
```

.../...

```
-- 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


-- 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

function OnAvatarRezzing(id)
    local color = avatars_colors[id]
    if color then
        SetAvatarMinimapColor(id, color)
        SetAvatarNameTagColor(id, color)
    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/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"
    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"
    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"
    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"
    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","spin1","print('Spinner=' .. GetValue())")
      SetLuaFloaterCommand("test","check1","print('Check=' .. GetValue())")
      SetLuaFloaterCommand("test","combo1","print('Combo=' .. GetValue())")
      SetLuaFloaterCommand("test","radio1","print('Radio=' .. GetValue())")
      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.

Index

Alphabetical Index