AGI Manual for ACT-R 7.6+

Working Draft

Dan Bothell

Table of Contents

Introduction

This document will provide a description of the GUI tools available in ACT-R for producing experiments for models (referred to as the AGI which stands for ACT-R GUI Interface) as well as some of the general ACT-R commands one may use for running models. These commands are used to create many of the experiments which are used for the models included with the ACT-R tutorial. Thus, if you would like to see examples of their use you can look at those models and the corresponding experiment code description documents which accompany the tutorial units. There are also examples of some of the AGI capabilities not used in the tutorial models available in model files located in the examples directory of the ACT-R source code distribution.

Before describing the AGI itself however there will be some brief background provided of the low level interface to ACT-R's perceptual and motor systems, the tools upon which the AGI is built, and some differences in the AGI relative to older versions of ACT-R.

When describing the commands available through the AGI we will be following the same format for specifying the syntax as is used in the ACT-R reference manual, and the details of that are included in the appendix of this manual.

Background

Devices

ACT-R models typically interact with the world through what are called devices. A device is a general mechanism in the ACT-R software which can be associated with a module to provide additional functionality. The AGI is based upon a set of devices which provide the model with the ability to interact with a simulated computer. The motor and speech modules use devices which translate the actions of those modules into appropriate actions for a simulated keyboard, mouse, and microphone, and the vision module uses a device which automatically generates visual features for GUI elements created using the AGI.

While it's possible to use some of those devices separately, the primary purpose of the AGI is to use what is called an "experiment window" device which provides the visual percepts and also installs the keyboard, mouse, and microphone devices automatically.

Virtual Windows

The experiment window device of the AGI is built around a GUI interface we call virtual windows. Virtual windows are based roughly on the windowing system of MCL (Macintosh Common Lisp which is now obsolete but was the dominant Lisp during much of ACT-R's early development), and it implements a windowing interface which is portable across Lisps. It is called virtual because it does not display anything the user can see – it is an abstract interface only visible to the model, but it does include a mechanism though which one can connect the virtual windows to real windows for viewing and to accept real user actions.

Visible Virtual Windows

When the ACT-R Environment is connected to ACT-R it has an option (enabled by default) to connect to the virtual windows interface so that it can display a real representation of a virtual window and allow user interaction. We refer to those displays as visible virtual windows. Thus, even for command line only Lisps it is possible to create and see graphic experiments for the model using the AGI commands when the ACT-R Environment is also used.

UWI

The first attempt to create a set of commands for an ACT-R GUI use was something called the Uniform Windowing Interface (UWI). It was a set of low level windowing functions which were implemented in various Lisps and for the virtual windows to allow them to operate in a portable manner. It was not entirely user friendly and has now been replaced by the AGI. Parts of it still exist in the implementation of the AGI, but it is not recommended for use directly and no documentation is provided on its operation.

Older ACT-R Devices

In prior versions of ACT-R a device was a single construct which provided specific interactions for the perceptual and motor modules through a fixed set of Lisp methods which one had to implement. That approach is no longer available (although like the UWI parts of its mechanisms can still be found if one looks at the underlying AGI implementation), but the more general devices used for the current AGI do provide similar capabilities to those from the monolithic device that was used in prior versions of the AGI.

Real GUI Windows

The older versions of ACT-R also included devices which attempted to provide the same functionality as the AGI did from the native GUI interface widgets of some Lisp systems. That functionality is no longer included with ACT-R for multiple reasons. It would be possible to use the external interface of the virtual windows to provide that in the same way that the visible virtual windows work, and it would also be possible to more directly interface them to the model through the perceptual modules' own interfaces (particularly vision) instead of trying to map through a general abstraction like the virtual windows.

The AGI

The AGI is a high level interface that we provide for GUI construction for models. It is a small set of tools designed to make creating simple experiments for ACT-R models easy. Those tools are able to create interfaces built from text, buttons, lines, and images. The model actions from the output devices it implements can be monitored to record key presses, mouse movement and clicks, and speech output from the model.

This document will describe the commands provided in the AGI as well as some of the associated ACT-R commands for running models. The commands of the AGI are available for use through the dispatcher in the current ACT-R software which makes it possible to use them from languages other than Lisp when connected to ACT-R through the remote interface. Examples of using the AGI can be found in the ACT-R tutorial which provides both Lisp and Python versions of the tasks created with the AGI.

There is one final thing to note however before describing the AGI commands. While it is possible to create real windows using the AGI through the included visible virtual windows of the ACT-R Environment or the external interface of the virtual windows, that is only recommended for use in testing and debugging the task and/or model. The AGI is not recommended for use in creating experiments for human participants because it does not provide any guarantees on the precision or accuracy of the timing data for real user interactions nor does it provide any guarantees on the timing of the display updates. For a model those things are handled relative to ACT-R's clock and thus always occur at the appropriate time for the model, but for real user interactions there are a lot of other factors involved which vary significantly based on the Lisp, OS, and machine being used. Therefore we cannot provide any guarantees on the usefulness of the AGI for human data collection. If one would like to use the AGI for collecting human data then it will be necessary to make sure that you understand the timing issues involved and test it thoroughly to determine the reliability of that information. To help with that, there are some notes about the AGI and timing issues which may be helpful in the appendix of this document.

Visual Scene

The AGI makes some simplifying assumptions about the visual scene. When creating windows for the model the assumption is that they are being displayed on a monitor with a coordinate system based upon the upper left corner of the display being 0,0 and increasing to the right and down, and a window itself also uses coordinates which start at 0,0 in its upper left corner. The location of items is specified using their upper left corner as the reference point relative to the window in which they are drawn (or the whole screen for windows). The windows themselves are not explicitly visible to the model. They only serve as a means of organizing the items which are displayed from the code, and the model can see all of the items in every experiment window which is installed for it. There are no explicit boundaries for the screen or window with respect to the items displayed. Although a negative coordinate value would suggest it is outside of the window boundary that does not prevent the model from seeing it, and the same holds for coordinates which exceed the height or width of the window. Similarly, there is no notion of items being occluded by other items or windows – the model will see all of the items regardless of any overlap which might occur.

Typical Experiment Design

To describe the commands available in the AGI we will first describe a very general experiment design. Then for each step of the experiment we will describe the relevant AGI commands.

Here is a typical procedure for creating a simple experiment:

1. Open a window
2. Clear the display
3. Present some stimuli
4. Wait for a response or fixed delay to pass
5. Record the response
6. Repeat steps 2-5 for different conditions/stimuli
7. Repeat steps 1-6 for multiple participants
8. Analyze the results

That general pattern can be found in most of the tutorial model experiments. Steps 6, 7, and most of step 8 are best done with the iteration constructs and other functions of the language being used, but the AGI provides the tools for carrying out the other steps.

One assumption with that design is that there is one participant performing the task at a time and there is a single window with which that participant is interacting. The AGI (and ACT-R) are not restricted to operating in that fashion, but that is the typical modeling scenario and thus the AGI commands attempt to make that process easy. It is possible to open multiple windows simultaneously with the AGI and make all of those visible to a model as well as run more than one ACT-R model concurrently all interacting with the same AGI window or different AGI windows, but the primary descriptions of the AGI commands will present the simple (single window single user) usage details. Details on running multiple models can be found in the ACT-R reference manual, and there are examples of multiple models interacting with AGI tasks found in the examples/model-task-interfacing directory of the software distribution.

Window Control (Steps 1 and 2)

For most tasks a single window is all that will be required for a model. When there is only one window created for a model all other AGI commands will operate upon that window without needing to specify it directly. If multiple windows are created however one will need to specify which window to use for all of the other commands. When specifying the window it can be done using either the title given to it when it was created or the device list which was returned by the AGI when the window was created.

Open-exp-window

open-exp-window *title* {**:visible** *visible*} {**:width** *w*} {**:height** *h*} {**:x** *x*} {**:y** *y*} -> [device | **nil**]

Remote command name:

open-exp-window *title* { < **visible** *visible,* **width** *w*, **height** *h*, **x** *x*, **y** *y* >}

Arguments and Values:

title ::= a string which can be used to reference the window and will display in the title bar of the visible virtual windows
visible ::= a generalized boolean which indicates whether the window should be displayed through the external window interface if available
w ::= an integer indicating the width of the window in pixels
h ::= an integer indicating the height of the window
x ::= an integer indicating the position of the left edge of the window on the screen
y ::= an integer indicating the position of the top edge of the window on the screen
device ::= a list which is valid for installing as a device in ACT-R

This command takes one required parameter which is the title for an experiment window to create. That title must be a string. If there is already an experiment window open with that title then it clears its contents. If there is not already an experiment window with that title then it creates a new window with the requested title.

The command also accepts several other parameters for configuring the window. The other parameters allow one to specify the height and width of the window in pixels and default to 300 each if not provided (only meaningful if the window is displayed as a visible virtual window), the x and y pixel coordinates of the upper left corner of the window on the screen which also default to 300 each, and visible which indicates whether the window should be displayed visibly if possible. A value of **nil** means do not display it visibly and any other value represents that displaying it is allowed (the default is **t**).

If the provided parameters are all valid values then the command returns a list which can be used to reference the window and is also valid for installing as a device to allow a

model to see the contents of the window. If there is a problem with the parameters then a warning will be printed and **nil** will be returned.

examples:

```
(open-exp-window "Letter recognition")
(open-exp-window "Task" :visible nil)
(open-exp-window "Other" :x 0 :y 0 :width 250 :height 400)

actr.open_exp_window('test', x=0, y=0, visible=True)
actr.open_exp_window('Window', visible=False)
```

Close-exp-window

close-exp-window { *window* } -> [**t** | **nil**]

Remote command name:

close-exp-window

Arguments and Values:

window ::= a string which is the title of a window or the device list returned when opening a window

This command takes no required parameters and has one optional parameter which can indicate a specific window (either by title or device list). If no window is provided then if there is only one open window that one will be used, but if there are more than one or no open windows it will print a warning. It closes the indicated window. Once the window is closed it is no longer possible for a person or model to interact with it. One should close experiment windows when they are no longer needed to avoid potential problems with multiple open windows or memory issues since all objects created for a window persist until the window is closed even if they are not currently displayed. All open windows will be closed automatically when ACT-R is initialized with the clear-all command. If a window is successfully closed then it will return **t**, otherwise it will return **nil**.

Closing a window which is currently installed for a model will cause the model's vision module to reprocess the visual scene.

examples:

```
(close-exp-window)
(close-exp-window "task")

actr.close_exp_window()
actr.close_exp_window('task')
```

Clear-exp-window

clear-exp-window { *window* } -> [**t** | **nil**]

Remote command name:

clear-exp-window

Arguments and Values:

window ::= a string which is the title of a window or the device list returned when opening a window

This command takes no required parameters and has one optional parameter which can indicate a specific window (either by title or device list). If no window is provided then if there is only one open window that one will be used, but if there are more than one or no open windows it will print a warning. It removes all of the items currently displayed in the indicated experiment window and causes all models which have that device installed to reprocess the visual scene. If it successfully clears the window it returns **t** otherwise it returns **nil**.

examples:

```
(clear-exp-window)
(clear-exp-window "window")

actr.clear_exp_window()
actr.clear_exp_window('Task')
```

Displaying Items (step 3)

Displaying items in experiment windows involves two steps: creating an item for the window and adding that item to the window. The AGI includes commands for performing both steps together as well as separate commands for creating and modifying items and adding or removing them from the window.

Item IDs

When creating an item for an experiment window the AGI commands return an ID which can be used to reference that item. That ID will be a list, but the contents of that list are not part of the AGI's API and thus should not be used or modified in any way.

Colors

When creating an item it is possible to specify its color. Colors should be provided as either a string or in Lisp, a symbol can also be used. Any string or symbol may be provided and will be used to create the visual feature for the item and passed to a visible window handler if installed. The vision module only provides chunks for the following colors: black, green, dark-gray, cyan, light-blue, yellow, dark-yellow, light-gray, dark-blue, dark-magenta, white, dark-cyan, blue, purple, gray, dark-green, dark-red, red, brown, pink, and magenta. Any other colors indicated for an item will result in a warning that the chunk is being created by default if it hasn't already been defined in the model. Also, a visible window handler may not recognize every name one provides, and how it handles that is up to the implementer of that window handler. For the visible virtual window handler provided by the ACT-R Environment, it recognizes all of the colors named above which are defined in the vision module, and for any other name it is given it will draw the item in black.

One note on providing colors in Lisp is that in some Lisps there are global variables which are bound to color objects, but those are not valid as colors for the AGI. In particular, in ACL it has variables like red and blue which are bound to the ACL color object for those colors. Those variables are not valid as colors for thebut the symbol with that name is i.e. 'red is a valid color for the AGI but in ACL the global variable red itself is not.]

AGI Items

There are four types of items which can be displayed in an AGI window: text, lines, buttons, and images. Each item has a command for creating one and a command which will create one and add it to a window, and all except images also include a command for modifying one. Whenever an item is added to a window or an item already added to a window is modified that window will cause all models with that window installed to reprocess that item. Creating an item requires specifying the window with which the

item will be associated (either by title or by its device list), and that window will be the only one which can display that item. If there is only one open window then a value of **nil** can be used to indicate that window.

Text

create-text-for-exp-window *win text* {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:color** *c* } {**:font-size** *f* }
-> [id | **nil**]
add-text-to-exp-window *win text* {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:color** *c* } {**:font-size** *f* }
-> [id | **nil**]
modify-text-for-exp-window *text-id* {**:text** *text*} {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:color** *c* }
{**:font-size** *f* } -> [id | **nil**]

Remote command name:

create-text-for-exp-window *win text* <**x** *x,* **y** *y*, **width** *w*, **height** *h*, **color** *c*, **font-size** *f* >
add-text-to-exp-window *win text* <**x** *x,* **y** *y*, **width** *w*, **height** *h*, **color** *c*, **font-size** *f* >
modify-text-for-exp-window *text-id* <**text** *text*, **x** *x,* **y** *y*, **width** *w*, **height** *h*, **color** *c*, **font-size** *f* >

Arguments and Values:

win ::= the title or device list of an experiment window or **nil** if there is only one window
text ::= a string of the text to display
x ::= an integer indicating the x position of the upper left corner of the text on the window
y ::= an integer indicating the y position of the upper left corner of the text on the window
w ::= an integer indicating the width in pixels of a box in which to draw the text
h ::= an integer indicating the height in pixels of a box in which to draw the text
f ::= an integer indicating the size of the font to use to draw the text (in points)
c ::= a string or symbol indicating the color of the text
id ::= a list which can be used to reference this text item
text-id ::= the id of a text item which has been created

These commands are used to draw a text string in an experiment window based on the values of the parameters provided. The x and y parameters specify the pixel coordinate of the upper-left corner of the box in which the text is to be displayed within the window, and the default value for each is 0. The height and width parameters specify the size of the box in which to draw the text, measured in pixels. The default value for height is 20 and for width is 75. The height and width parameters are only potentially meaningful for a visible display of the window – they do not affect the features created for the model. The color parameter specifies in which color the text will be drawn (as described above) and defaults to black. The font-size parameter specifies the size of the font used to draw the text and is measured in points. It defaults to 12 if not specified. It will affect the size of the text item for the visual feature, but it may or may not be used by a visible window handler when displaying the text (the visible virtual windows of the ACT-R Environment do use the font-size when displaying text).

If all of the parameters are valid then the text item is created or modified and its id is returned otherwise a warning will be printed and **nil** is returned.

examples:

```
(add-text-to-exp-window "Task" "Ok" :x 20 :y 30)
(add-text-to-exp-window nil "A" :x 100 :color 'red :width 20 :font-size 20)
(setf *item* (create-text-for-exp-window "Exp. Window" "True"))
(modify-text-for-exp-window *item* :color 'green)

actr.add_text_to_exp_window('Task', 'Ok', x=20, y=30)
actr.add_text_to_exp_window(None, 'A', x=100, color='red', width=20,
                            font-size=20)
item = actr.create_text_for_exp_window('Exp. Window', 'True')
actr.modify_text_for_exp_window(item, color='green')
```

Buttons

create-button-for-exp-window *win* {**:text** *text*} {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:color** *c* } {**:action** *a* } -> [id | **nil**]
add-button-to-exp-window *win* {**:text** *text*} {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:color** *c* } {**:action** *a* } -> [id | **nil**]
modify-button-for-exp-window *b-id* {**:text** *text*} {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:color** *c* } {**:action** *a* } -> [id | **nil**]

Remote command name:

create-button-for-exp-window *win* < **text** *text*, **x** *x,* **y** *y*, **width** *w*, **height** *h*, **color** *c*, **action** *a* >
add-button-to-exp-window *win* < **text** *text*, **x** *x,* **y** *y*, **width** *w*, **height** *h*, **color** *c*, **action** *a* >
modify-button-for-exp-window *b-id* < **text** *text*, **x** *x,* **y** *y*, **width** *w*, **height** *h*, **color** *c*, **action** *a* >

Arguments and Values:

win ::= the title or device list of an experiment window or **nil** if there is only one window
text ::= a string of text to display on the button
x ::= an integer indicating the x position of the upper left corner of the button in the window
y ::= an integer indicating the y position of the upper left corner of the button in the window
w ::= an integer indicating the width in pixels the button
h ::= an integer indicating the height in pixels of the button
c ::= a string or symbol indicating the color of the button
a ::= [fct | cmd | ([fct | cmd] param*)]
fct ::= a Lisp function or symbol naming a function
cmd ::= a string naming an available ACT-R command
param ::= a value to pass to the indicated function or command
id ::= a list which can be used to reference this button item
b-id ::= the id of a button item which has been created

These commands are used to create a button for an experiment window which can perform an action when clicked on with the mouse device (or by a real mouse with the visible virtual windows). The text parameter must be a string and specifies the text to display on the button, and it defaults to "" (an empty string). The x and y parameters specify the pixel coordinate of the upper-left corner of the button in the window and each defaults to 0. The height and width parameters specify the size of the button in pixels, and the height defaults to 18 and the width defaults to 60. The color parameter specifies a background color for the button, and defaults to gray. The action parameter specifies a function or command to be called when this button is pressed. If not provided or set to **nil** then clicking the button will result in printing a warning which says "Button with no valid action clicked at time xxx" where xxx is the current ACT-R time in seconds. The function or command will be called with no parameters unless some are included when creating or modifying the button, in which case those parameters will be passed to it.

If all of the parameters are valid then the button item is created or modified and its id is returned otherwise a warning will be printed and **nil** is returned.

examples:

```
(add-button-to-exp-window "task" :text "Ok" :x 100 :y 150)
(create-button-for-exp-window nil :text "Cancel" :color 'red :action 'cancel)
(modify-button-for-exp-window *but-obj* :action '("new-cmd" 10 t))

actr.add_button_to_exp_window("task", text='Ok', x=100, y=150)
actr.create_button_for_exp_window(None, text='Cancel',color='red',
                                  action='cancel')
actr.modify_button_for_exp_window(but_obj, action=['new-cmd', 10, True])
```

Lines

create-line-for-exp-window *win (x1 y1) (x2 y2)* { *c* } -> [id | **nil**]
add-line-to-exp-window *win (x1 y1) (x2 y2)* { *c* } -> [id | **nil**]
modify-line-for-exp-window *line-id* [*(x1 y1)* | **nil**] [*(x2 y2)* | **nil**] { *c* } -> [id | **nil**]

Remote command name:

create-line-for-exp-window
add-line-to-exp-window
modify-line-for-exp-window

Arguments and Values:

win ::= the title or device list of an experiment window or **nil** if there is only one window
x1 ::= an integer indicating the x position of one end point of the line in the window
y1 ::= an integer indicating the y position of one end point of the line in the window
x2 ::= an integer indicating the x position of one end point of the line in the window
y2 ::= an integer indicating the y position of one end point of the line in the window
c ::= a string or symbol indicating the color of the button

id ::= a list which can be used to reference this line item
line-id ::= the id of a line item which has been created

These commands are used to create a line item for an experiment window. The required parameters are lists with the x and y pixel coordinates for the ends of the line to be drawn. An optional color may be provided. If no color is given it defaults to black. When modifying a line either or both of the position lists may be specified as **nil** to indicate that that position should not be modified.

If all of the parameters are valid then the line item is created or modified and its id is returned otherwise a warning will be printed and **nil** is returned.

examples:

```
(add-line-to-exp-window "task" (list 100 150) (list 50 10))
(create-line-for-exp-window nil (list 0 0) (list 500 100) 'blue)
(modify-line-for-exp-window *line-obj* nil (list 30 30))

actr.add_line_to_exp_window('task', [100, 150], [50, 10])
actr.create_line_for_exp_window(None, [0, 0], [500, 100], 'blue')
actr.modify_line_for_exp_window(line_obj, None, [30, 30])
```

Images

create-image-for-exp-window *win text file* {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:action** *a* }
-> [id | **nil**]
add-image-to-exp-window *win text file* {**:x** *x*} {**:y** *y*} {**:width** *w*} {**:height** *h*} {**:action** *a* }
-> [id | **nil**]

Remote command name:

create-image-for-exp-window *win text file* <**x** *x*, **y** *y*, **width** *w*, **height** *h*, **action** *a* >

add-image-to-exp-window *win text file* <**x** *x*, **y** *y*, **width** *w*, **height** *h*, **action** *a* >

Arguments and Values:

win ::= the title or device list of an experiment window or **nil** if there is only one window
text ::= a string of text to provide as the value of the visual feature for the model
x ::= an integer indicating the x position of the upper left corner of the image in the window
y ::= an integer indicating the y position of the upper left corner of the image in the window
w ::= an integer indicating the width of the box in which to display the image in pixels
h ::= an integer indicating the height of the box in which to display the image in pixels
a ::= [fct | cmd | ([fct | cmd] param*)]
fct ::= a Lisp function or symbol naming a function
cmd ::= a string naming an available ACT-R command

param ::= a value to pass to the indicated function or command
id ::= a list which can be used to reference this image item

These commands are used to create an image item for an experiment window. This is intended only for use with the visible virtual windows of the ACT-R Environment. It allows one to display a .gif file in the visible virtual window. That image can be seen by a model as a single feature with a value indicated when creating the image, and it can be clicked like a button to trigger an action. The required parameters are a string which provides the value for the visual feature made available to the model for the item, and a string which names a .gif file which must be located in the environment/GUI/AGI-images directory of the ACT-R Environment being run. One can also specify the position for the image in the window and the size of the box in which it will be displayed (there is no attempt to stretch or center the image in the box – it is displayed with its upper left corner in the upper left corner of the box specified and will be clipped at the borders if it extends past them and the background color will show if the image does not fill the box). The action parameter specifies a function or command to be called when this image is clicked (which is the entire box whether or not the image actually fills it). If not provided or set to **nil** then clicking the image will result in printing a warning which says "Image *name* with no valid action clicked at position *x*,*y* at time *zzz*." where name is the text value of the box, x and y are the position within the image that the click occurred, and zzz is the current ACT-R time in seconds. If the function or command is specified without any parameters then it will be called with two values which are the text value of the image and a list of the x and y position within the image where the click occurred, but if parameters are included when creating the image those parameters will be passed to the action.

If all of the parameters are valid then the image item is created and its id is returned otherwise a warning will be printed and **nil** is returned.

examples:

```
(add-image-to-exp-window nil "back" "ref-brain.gif" :height 390 :width 390)
(create-image-for-exp-window "WIN" "brain" "ref-brain.gif" :x 10 :y 160
                             :width 128 :height 128 :action "click-brain")

actr.add_image_to_exp_window(None,'back','ref-brain.gif',height=390,width=390)
actr.create_image_for_exp_window('WIN', 'brain', 'ref-brain.gif',x=10,y=160,
                                 width=128, height=128,action='click-brain')
```

General Commands

These commands work for any of the AGI items which have been created to add or remove them from an experiment window.

Remove-items-from-exp-window

remove-items-from-exp-window *win id** -> [**t** | **nil**]

Remote command name:

remove-items-from-exp-window

Arguments and Values:

win ::= the title or device list of an experiment window or **nil** if there is only one window
id ::= the id returned when an AGI item was created or added to an experiment window

This command takes one required parameter which must indicate an experiment window, and then an arbitrary number of parameters which should be AGI item ids. Each of those items is removed from the indicated window. If any of the parameters are not valid then no items are removed, a warning is printed and **nil** is returned. Otherwise all items are removed and **t** is returned.

examples:

```
(remove-items-from-exp-window nil text-id-1 but-id)
actr.remove_items_from_exp_window(None, text_id_1, but_id)
```

Add-items-to-exp-window

add-items-to-exp-window *win id** -> [**t** | **nil**]

Remote command name:

add-items-to-exp-window

Arguments and Values:

win ::= the title or device list of an experiment window or **nil** if there is only one window
id ::= the id returned when an AGI item was created or added to an experiment window

This command takes one required parameter which must indicate an experiment window, and then an arbitrary number of parameters which should be AGI item ids. Each of those items is added to the indicated window. If any of the parameters are not valid then no items are added, a warning is printed and **nil** is returned. Otherwise all items are added and **t** is returned.

examples:

```
(add-items-to-exp-window nil text-id-1 but-id)
actr.add_items_to_exp_window(None, text_id_1, but_id)
```

Waiting on user interaction (step 4)

There are no commands specific to the AGI which are relevant for waiting for an ACT-R model. Many ACT-R commands are relevant to that process, like run, install-device, etc, and those are described in the main reference manual and also used in the tasks included with the ACT-R tutorial. There is however one command which is available in ACT-R and the Python module provided with the ACT-R tutorial which can be useful if you are interacting with a task directly instead of running the model.

Process-events

process-events -> **nil**

This command can be useful when a task is waiting for a person to interact with an experiment window. It takes no parameters and always returns **nil**. It gives the system a chance to handle real user interactions and also gives ACT-R a chance to process them while waiting. It is not necessary when a model is performing the task, but it is recommended that it be called inside any loop which is waiting for a real response.

Response collection (step 5)

When a model performs a key press, mouse click, or speech action the associated devices will generate actions which can be "monitored" to record what has happened, and when a real key press or mouse click occurs on the visible virtual window they will also generate the same actions for monitoring. Thus, the same commands can be used to record both model and human actions. The action monitoring process is not part of the AGI. It is a component of the remote interface built into ACT-R. The details of the underlying process can be found in the remote document in the ACT-R docs directory and the experiment description text in unit 2 of the ACT-R tutorial describes how to use that for monitoring experiment window actions. That will not be covered here, but the actions which can be monitored for the experiment window devices will be described.

Monitorable Actions

Output-key

The output-key action is generated when a key is pressed by a model on the keyboard device or when a person presses a key in a visible virtual window. It is passed two parameters. The first is the name of the model which pressed the key or **nil** if it was a person pressing a key. The second is the string containing the key which was pressed.

Output-speech

The output-speech action is generated when the model performs a speak action by the microphone device (there is no corresponding action for a person interacting with the visible virtual windows). It is called at the beginning of the speech output (when it would first be detectable). It is passed two parameters. The first is the name of the model which performed a speak action, and the second is the string which was spoken.

Move-cursor

The move-cursor action is generated when the model moves the mouse device (there is no corresponding action for a person interacting with the visible virtual windows). It is called with three parameters. The first is the name of the model which moved the mouse. The second is the name of the cursor which was used (will always be the string "mouse" for the device installed with the experiment windows), and the third will be a list of the x and y coordinates to which the mouse was moved. If the :incremental-mouse-moves

parameter is set in the model then this action will be generated for each of the components of a mouse movement.

Click-mouse

The click-mouse action is generated when the model is using the mouse device and performs a punch, peck, or peck-recoil action with a finger on that hand to the location of a button (the default finger positions – it assumes that there are 5 buttons available on the mouse). It is passed three parameters. The first is the name of the model which pressed a mouse button. The second is the list of the x and y coordinates of the mouse position at the time the press occurred, and the third is the name of the finger which performed the press: index, middle, ring, pinkie, or thumb. There is no corresponding action for a person interacting with the visible virtual windows.

Response Time

Get-time

To record the time of an action the get-time command can be used. This command takes one optional parameter and it returns the current time in milliseconds. If the optional parameter is specified as a non-nil value then the time returned is the model's simulated time. If the optional parameter is specified as **nil** then the time is taken from the internal Lisp timer using the function get-internal-real-time. If the optional parameter is not specified, then the model's simulated time is returned. When recording the real time (passing **nil**) the time is not zero referenced with respect to the task (unlike model time which is the current simulated time since the model was last reset). Therefore if collecting real time values one will likely also need to record the time at the start of the trial for reference. Also, although the real time value is measured in milliseconds that does not mean that it is necessarily accurate to that resolution (see the appendix on collecting human data for more details).

examples:

```
?> (get-time)
4931

?> (get-time nil)
67564376340
```

```
>>> actr.get_time()
4931

>>> actr.get_time(None)
67564403993
```

Data analysis (step 8)

To help with data analysis there are two AGI commands provided for performing correlation and mean deviation calculations.

Correlation

This command takes 2 required parameters which must be equal length lists of numbers, and it computes the correlation between those two lists of numbers. That correlation value is returned. There is an optional third parameter which controls whether or not the correlation value is also output. If it is **nil** then no output is written. If it is a string then that string is used as the full pathname for a file which is opened and then has the correlation value written to the end of it (it is created if it does not currently exist). If it is any other value, then the output is written to the ACT-R command trace (if there is a current model) or the ACT-R output trace if there is not a current model. The default is to output the correlation to the appropriate ACT-R trace.

examples:

```
?> (correlation (list 1 2 3 4) (list .3 .5 .9 1.2))
CORRELATION:  0.993
0.9927947

?> CG-USER(371): (correlation (list 1 2 3 4) (list .3 .5 .9 1.2) nil)
0.9927947

>>> actr.correlation([1,2,3,4],[.3,.5,.9,1.2])
CORRELATION:  0.993
0.9927947

>>> actr.correlation([1,2,3,4],[.3,.5,.9,1.2],None)
0.9927947
```

Mean-deviation

This command operates just like correlation, except that the calculation performed is the root mean square deviation between the data sets.

examples:

```
?> (mean-deviation (list 1 2 3 4) (list .3 .5 .9 1.2))
MEAN DEVIATION:  1.936
1.9358461

?> (mean-deviation (list 1 2 3 4) (list .3 .5 .9 1.2) nil)
1.9358461

>>> actr.mean_deviation([1,2,3,4],[.3,.5,.9,1.2])
MEAN DEVIATION:  1.936
1.9358461
```

```
>>> actr.mean_deviation([1,2,3,4],[.3,.5,.9,1.2],None)
1.9358461
```

Miscellaneous

This section contains some additional AGI commands which may be useful for creating experiments.

Permute-list

This command takes one parameter which must be a list and it returns a randomly ordered copy of that list. The randomization is performed using the act-r-random command which means that the setting of the model's :seed parameter will affect the results and it is possible to recreate the same randomized list again when needed.

examples:

```
?> (permute-list '("a" "b" "c" "d" "e"))
("b" "c" "d" "a" "e")

>>> actr.permute_list(['a','b','c','d','e'])
['e', 'c', 'a', 'd', 'b']
```

While

This macro is provided for use in Lisp because this simple looping construct is not part of the ANSI Common Lisp language (there are similar constructs available, but they are often a little more complicated to use and harder to read for those who are not Lisp programmers). It takes an arbitrary number of parameters. The first parameter specifies the test condition, and the rest specify the body of the loop. The test is evaluated and if it returns anything other than **nil** all of the forms in the body are executed in order. This is repeated until the test returns **nil**. Thus, while the test is true (non-**nil**) the body is executed.

examples:

```
(while (null *response*)
  (process-events))
```

Visual Features of AGI Elements for Models

This section will describe how the visual features for the button, line, text, and image items of the AGI displays are constructed for the model. Those features are created from the virtual window interface of the AGI and thus will be the same on all machines. If the visible virtual windows are used, the actual display may vary from machine to machine for various reasons, but that is not the display which the model is actually using.

The visual-location and visual-object chunks for the model are created using the basic visual-location chunk-types which are defined by the vision module:

```
(chunk-type visual-location screen-x screen-y distance kind color value height width size)
```

The visual object chunks are created from specific chunk-types which are subtypes of the default visual-object chunk-type:

```
(chunk-type visual-object screen-pos value status color height width)

(chunk-type (text (:include visual-object)) (text t))
(chunk-type (oval (:include visual-object)) (oval t))
(chunk-type (line (:include visual-object)) (line t) end1-x end1-y end2-x end2-y)
(chunk-type (image (:include visual-object)) (image t))
```

For each of the items we will describe any particular details about how the items are processed and then show examples of creating some items along with the resulting visicon and some visual-location and visual-object chunks which the model receives. For the examples, the window in which the items are placed is at the default location of 300,300.

Before covering the specific items, there are some general features which we will first describe. The screen-x and screen-y position of items is always determined as their center point and represented in global pixel coordinate (that means it takes the position of the window itself into account along with the local window coordinates used when creating the item). The distance is also computed in pixels and is based on the value of the :viewing-distance and :pixels-per-inch parameters in the model. The size is the area of the item in degrees of visual angle squared determined by multiplying the actual height and width after converting them into degrees of visual angle using the :viewing-distance and :pixels-per-inch parameters. Only the visual-object chunk for the item will contain the real value of the item. The value slot of the visual-location chunks will just contain the basic type of the item, and that same value will also be found in the kind slot.

Text items

The default processing for text items is to create a visual feature for each “word” in the text displayed. Words are determined by segmenting the text string based on the classification of the characters as either whitespace, alphanumeric, or other. Whitespace consists of any character which is not graphic-char-p in Lisp as well as the space

character. Alphanumeric characters are any characters which are alphanumericp in Lisp or which have been specified with the add-word-characters command as described in the ACT-R reference manual. Other characters are those which are not whitespace or alphanumeric. A word is a continuous sequence of either alphanumeric or other characters. Thus this string “ab123--word end” would be broken into four separate words “ab123”, “--”, “word”, and “end”.

The size, height, and width values for text are based on the font-size of the text, not on the height and width values specified when creating it (those only describe the box of the visible display in which to draw the text). For the virtual device the values for the 12 point font (the default font size) are: letters are 10 pixels high and each character (including the space) has a fixed width of 7 pixels wide. If there are newlines in the text being displayed that will result in moving the text after the newline down by the current font size pixels and starting again at the specified x position.

example:

Features:

```
(add-text-to-exp-window nil "single" :color 'blue)
(add-text-to-exp-window nil "multiple words" :font-size 20 :x 100 :y 100)
(add-text-to-exp-window nil (format nil "multiple~%lines") :font-size 10
                        :x 200 :y 300)
```

Visicon:

```
Name              Att  Loc             Text  Kind  Color  Width  Value       Height  Size
----------------  ---  --------------  ----  ----  -----  -----  ----------  ------  ---------
VISUAL-LOCATION0  NEW  (322 306 1080)  T     TEXT  BLUE   42     "single"    10      1.18
VISUAL-LOCATION1  NEW  (449 410 1080)  T     TEXT  BLACK  96     "multiple"  17      4.5899997
VISUAL-LOCATION4  NEW  (516 615 1080)  T     TEXT  BLACK  30     "lines"     8       0.68
VISUAL-LOCATION3  NEW  (525 605 1080)  T     TEXT  BLACK  48     "multiple"  8       1.0799999
VISUAL-LOCATION2  NEW  (539 410 1080)  T     TEXT  BLACK  60     "words"     17      2.87
```

Chunks:

```
VISUAL-LOCATION0-0
   KIND  TEXT
   VALUE  TEXT
   COLOR  BLUE
   HEIGHT  10
   WIDTH  42
   SCREEN-X  322
   SCREEN-Y  306
   DISTANCE  1080
   SIZE  1.18


TEXT0-0
   SCREEN-POS  VISUAL-LOCATION0-0
   VALUE  "single"
```

```
   COLOR  BLUE
   HEIGHT  10
   WIDTH  42
   TEXT  T
```

Buttons

A button in the display creates a feature for the button itself as well as features for all of the text displayed on the button. The feature for the button depends on the properties specified when creating the button as well as the :viewing-distance and :pixels-per-inch parameters. The kind and value of the button location chunk will be **oval** (which reflects the shape of the buttons on a Macintosh computer which was where the system was originally developed and has been retained for consistency). The value of the visual object chunk will be the text displayed on the button.

The text value provided for the button will result in text features being generated almost the same as described above for text items. There are few differences between the text features generated from a button and text features generated from text items. Two of those are that the button does not provide a way to specify a font-size or color for the text, thus a button's text items will always be black and computed from a 12 point font. The other difference is that the text for a button is assumed to be placed so that it is centered on the button instead of being aligned with the upper-left corner of the item's placement as they are for text items. That means that if there is a single word in the text it will have the same screen-x and screen-y values as the button itself. If there are multiple words and/or multiple lines then their locations will be arranged such that there are an equal number of lines above and below the screen-y of the button and each line will be centered on the screen-x value of the button.

example:

Features:

```
(add-button-to-exp-window ""  :text "OK" :x 100 :y 100 :height 20 :width 50 :color 'red)
```

Visicon:

```
Name              Att  Loc             Kind  Color  Value  Height  Width  Size  Text  Oval
----------------  ---  --------------  ----  -----  -----  ------  -----  ----  ----  ----
VISUAL-LOCATION0  NEW  (425 410 1080)  OVAL  RED    "OK"   20      50     2.81        T
VISUAL-LOCATION1  NEW  (425 410 1080)  TEXT  BLACK  "OK"   10      14     0.39  T
```

Chunks:

```
VISUAL-LOCATION0-0
   KIND  OVAL
   VALUE  OVAL
   COLOR  RED
```

```
   HEIGHT  20
   WIDTH  50
   SCREEN-X  425
   SCREEN-Y  410
   DISTANCE  1080
   SIZE  2.81

OVAL0-0
   SCREEN-POS  VISUAL-LOCATION0-0
   VALUE  "OK"
   COLOR  RED
   HEIGHT  20
   WIDTH  50
   OVAL  T
```

Lines

Each line added to the display will generate one feature for the model. The height, width, and size of the line are computed based on a rectangle with the given points at opposite corners. The visual object for the line will include additional slots for the pixel coordinates of the end points.

example:

Features:

```
(add-line-to-exp-window "" '(10 10) '(20 20) 'blue)
(add-line-to-exp-window "" '(30 30) '(20 30) 'yellow)
```

Visicon:

(note the names were shortened from visual-location0 and visual-location1 to fit the visicon entries on a single line each)

```
Name  Att  Loc             Line  Kind  End2-Y  End2-X  End1-Y  End1-X  Height  Width  Value  Color   Size
----  ---  --------------  ----  ----  ------  ------  ------  ------  ------  -----  -----  ------  ----
V0    NEW  (315 315 1080)  T     LINE  320     320     310     310     10      10     LINE   BLUE    0.28
V1    NEW  (325 330 1080)  T     LINE  330     320     330     330     0       10     LINE   YELLOW  0.0
```

Chunks:

```
VISUAL-LOCATION0-0
   KIND  LINE
   VALUE  LINE
   COLOR  BLUE
   HEIGHT  10
   WIDTH  10
```

```
   SCREEN-X  315
   SCREEN-Y  315
   DISTANCE  1080
   SIZE  0.28

LINE0-0
   SCREEN-POS  VISUAL-LOCATION0-0
   VALUE  LINE
   COLOR  BLUE
   HEIGHT  10
   WIDTH  10
   LINE  T
   END1-X  310
   END1-Y  310
   END2-X  320
   END2-Y  320
```

Images

Each image added to the display will create a single feature for the model. The kind and value slots of the visual-location chunk will have the value image. The value of the visual object will be the text value provided and the size is based on the height and width provided (not the actual underlying image).

example:

Features:

```
(add-image-to-exp-window "" "image1" "smalllogo.gif" :x 0 :y 0 :height 50 :width 50)
(add-image-to-exp-window "" "image2" "ref-brain.gif" :x 100 :y 100 :height 100 :width 300)
```

Visicon:

```
Name              Att  Loc             Image  Kind   Height  Width  Value     Size
----------------  ---  --------------  -----  -----  ------  -----  --------  ---------
VISUAL-LOCATION0  NEW  (325 325 1080)  T      IMAGE  50      50     "image1"  7.0299997
VISUAL-LOCATION1  NEW  (550 450 1080)  T      IMAGE  100     300    "image2"  83.84
```

Chunks:

```
VISUAL-LOCATION0-0
   KIND  IMAGE
   VALUE  IMAGE
   HEIGHT  50
   WIDTH  50
   SCREEN-X  325
   SCREEN-Y  325
   DISTANCE  1080
   SIZE  7.0299997

IMAGE0-0
   SCREEN-POS  VISUAL-LOCATION0-0
```

```
VALUE  "image1"
HEIGHT  50
WIDTH  50
IMAGE  T
```

Appendix

Issues with using the AGI for human data collection

Because the AGI is intended to work with any ANSI Common Lisp implementation for running ACT-R models, the tools provided are not tuned for high-fidelity human data collection, and as provided, the AGI includes no guarantees as to its performance in that regard. Thus, whether the AGI (or some other tool) is acceptable for any experimental purpose really comes down to researching and testing to determine if it performs within the bounds of what one needs, and timing is typically the important issue with respect to collecting human data. Three things to consider with respect to timing related to the AGI will be described below: the resolution of the timer, the latency between the user actions and when the code can record them, and the latency between when a display change is requested and when it becomes visible to the user.

From a timer resolution aspect the AGI uses the Lisp function get-internal-real-time when not collecting data from a running model and that function does not have any requirements for the resolution it provides. In most Lisps it returns a result that is specified in milliseconds, but often that is not the true resolution of the timer because it may only update once every 50ms or worse in some cases. If that timer is not sufficient then one would need to find out if the particular Lisp being used has any custom functions for accessing a better time source and whether or not that could be used instead of the provided get-time command.

In terms of response detection latency when using the visible virtual windows there are several potential sources of variability and delay. The interactions with the visible virtual windows are processed in the ACT-R Environment (through the Tcl/Tk event handler) and then transmitted over a TCP/IP socket connection to ACT-R where the actions can then be monitored through the ACT-R dispatcher. All of those places (Tcl/Tk, socket communication, and the ACT-R dispatcher) can potentially affect the timing, and they would need to be profiled to determine the effects they have on the results for a given machine.

Latency on display changes is essentially the reverse of response detection – the ACT-R dispatcher sends the commands over a socket connection to Tcl/Tk which handles the display. That chain of actions would also need to be profiled.

All of those pieces of the ACT-R software (the internals of the dispatcher, the socket communication, and the functioning of the ACT-R Environment) are below the level at which support can be provided. Therefore, any investigation and profiling for suitability in your experiment would be up to you to perform.

Command Syntax

When describing a command's syntax the following conventions will be used:

- items appearing in **bold** are to be entered verbatim
- items appearing in *italics* take user-supplied values
- items enclosed in **{**curly braces**}** are optional
- * indicates that any number of items may be supplied
- + indicates that one or more items may be supplied
- | indicates a choice between options which are enclosed in [square brackets]
- (parentheses) denote that the enclosed items are to be in a list
- -> indicates that calling the command on the left of the "arrow" will return the item to the right of the "arrow"
- **::=** indicates that the item on the left of that symbol is of the form given by the expression on the right

An additional indicator will be used when describing the syntax of commands available through the remote interface when they differ from that of the Lisp command.

- <angle brackets> denote an options list for providing optional named parameters to a command and the available parameter names and values are indicated between the angle brackets and separated by commas

Examples

The examples shown for the AGI commands will include both Lisp examples and examples from Python using the actr.py module included with the tutorial. That Python module provides functions for accessing the remote versions of the ACT-R commands and uses named parameters in Python where keyword parameters are used in Lisp. The actual remote syntax is as shown in the description, but the Python functions are "hiding" the details of that to hopefully make it easier to use.

Virtual Window external interface

Details on how to connect an external interface to the virtual windows as is done for the visible virtual windows will be added in the future.

Other Commands

There is currently one other command available with the AGI that may be useful when using the visible virtual windows or other connected real windows. This command is not needed when working with only virtual windows (it has no effect on a virtual window). While this command is provided through the remote interface, it does not have a corresponding function in the Python module included with the tutorial.

Select-exp-window

This command takes no required parameters and has one optional parameter which can indicate a specific window (either by title or device). If no window is provided then it will operate on the only open window if there is one. It will attempt to bring that window to the foreground of the actual display, but that may or may not actually happen for various reasons, like user interface settings or other programs which are obscuring its view.