# AGI Manual
*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 commands themselves however there will be some brief background discussion of the low level interface to ACT-R's perceptual and motor systems and the tools upon which the AGI is built.

# Background

## A Device

An ACT-R model interacts with the world through what is called a device.  A device is an abstract representation of the world for the model (which typically takes the form of a simulated computer monitor, keyboard, and mouse) implemented as an object in Lisp.  There are a handful of methods that must be defined on that object so that ACT-R will be able to "see" and "manipulate" that device.  In general, defining such a device object is all that is necessary to produce something with which an ACT-R model can interact, and the slides titled "extending-actr" found in the docs directory of the ACT-R distribution describe the details of how one can create a new device.  When one needs to specify a new or different "world" for an ACT-R model then creating a device is the way to do so, but for simple experiments it's often easier to just work with the devices which are already available in ACT-R.  If one of the following devices is used for the model then no additional code will be necessary to provide the vision and motor capabilities of the model.

## Real GUI Windows

Built into ACT-R are devices for some limited interaction with the native Lisp GUIs and interface widgets of Allegro Common Lisp, Clozure Common Lisp, and LispWorks.  Thus if one builds an interface with the native tools for those Lisps using the subset of widgets supported (or adds the necessary support for other widgets) it is likely that the model will be able to interact with that interface with very little extra effort.  There are some potential difficulties with that however.  The first is that you have to learn how to use the GUI tools of the Lisp system you are using if you do not know them already.  Another is that the GUI systems of the Lisps are not compatible i.e. if you build it in CCL it is only going to work in CCL.  The last is that we do not provide any documentation for the specific low-level Lisp code needed to implement those devices other than what is found in the code which implements them (they are located in the devices directory of the ACT-R distribution).  The AGI commands described below will create real GUI windows in those Lisps with which the model can interact through the provided device.

## Virtual Windows

Virtual windows are an abstract representation, 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 development), built into ACT-R that implements a windowing interface which is portable across Lisps.  It is called virtual because it does not display anything – it is an abstract interface only visible to the model and which only accepts actions from the model (except under the situation described below).  Because the virtual windows do not open real windows or handle real user interaction they are often much

faster than the native windows on a given system, and can be used when speed and/or portability is important.  The virtual windows can be created and used directly if desired, but there is no documentation available on doing so other than in the code which implements them.  The recommended way to use them is through the AGI commands.

## Visible Virtual Windows

If the ACT-R Environment is connected to ACT-R it has an option (enabled by default) to allow the virtual windows to be displayed and interacted with by a real user as well as the model.  The AGI command for creating a window will create visible virtual windows if they are available instead of native windows in the Lisp being used. 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.  The visible virtual windows use the same device for the model as the virtual windows.

## The UWI

The first attempt to create a set of commands for 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.  It is still implemented in the code and is used underneath the AGI functions, but it is not recommended for use directly and no documentation is provided on its operation.

# The AGI

The high level interface that we provide for GUI construction for models is the AGI.  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 which use any of the devices included with ACT-R (real windows in the supported Lisps, virtual windows, and the visible virtual windows). and include three types of objects which can be placed into those windows: text, buttons, and lines.  It also provides methods which can be defined for collecting the keypress and mouse button actions performed within those windows.

This document will describe the functions provided in the AGI as well as some of the important ACT-R functions for running models.  Examples of the AGI in action can be found in the ACT-R tutorial and the examples directory of the ACT-R  software distribution.

There is one final thing to note however before describing the AGI commands.  Creating a visible interface with the AGI does make it possible for the experimenter to interact with the same experiment as the model, but 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 functions for real user interactions nor does it provide any guarantees on the timing of the display changes.  For a model those actions are handled relative to ACT-R's internal 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 the recommendation is to make sure you understand the timing issues involved and then test it thoroughly using the target machine.  To help with that, there is some information about the AGI and timing issues which may be helpful at the end of this document.

# 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 already present in ANSI Common Lisp, 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 as well as run more than one ACT-R model concurrently, but the primary descriptions of the AGI commands will present the simple (single window) usage details. Information on using multiple AGI windows will be included in a later section. For details on running multiple models one should consult 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 commands will operate upon that window and it is referred to as the "current" window.  If multiple windows are created however one will need to specify the window to use for all of the other commands and there is no current window.  That can be done using either the title specified when creating the window or the window object returned when the window is created.

## Open-exp-window

This command takes one required parameter which is the title for an experiment window to create.  That title must be a string or symbol and will be displayed in the title bar of the window along with the current model's name (with the provided devices the model cannot see the title bar of the window).  If there is already an experiment window open with that title in the current model then it clears its contents and brings it to the foreground.  If there is not already an experiment window with that title then it opens a new window with the requested title and brings it to the foreground.

The command also accepts several keyword parameters for configuring the window. The possible keyword parameters are :height and :width which specify the size of the window in pixels and default to 300 each, :x and :y which specify the screen coordinates of the upper left corner of the window in pixels and also default to 300 each, and :visible which can be either **t** or **nil** (the default is **t**).  If :visible is **t** it specifies that a real or visible virtual window be opened, and if it is **nil** it means that a virtual window will be opened.

There is also a keyword parameter called :class which can be used to create the window using a custom window class instead of the default classes for those looking to extend the operation of the system.  The only documentation for using that capability is to note that the specified class must be a subtype of rpm-window and additionally a subtype of visible-virtual-window if that is how the windows are being shown.  Beyond that, extending the system in that manor will require one to be familiar with the underlying interface mechanisms involved either from the Lisp being used or the ACT-R virtual windows which are only documented in the code itself.

If the provided parameters are all valid values then the function returns the corresponding window object.  If there is a problem with the parameters, or in creating or clearing the window 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 *title* :x 0 :y 0 :width 250 :height 400)
```

## Select-exp-window

This command takes no required parameters and has one optional parameter which can indicate a specific window (either by title or object). If no window is provided then it will operate on the current window. It will bring that window to the foreground.

Note: The select-exp-window and open-exp-window commands may not always bring a visible virtual window to the foreground because of issues with which application has the focus in some operating systems.

**examples:**

```
(select-exp-window)
(select-exp-window "Letter recognition")
```

## Close-exp-window

This command takes no required parameters and has one optional parameter which can indicate a specific window (either by title or object). If no window is provided then it will operate on the current window. It closes that experiment 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. However, all open windows will be closed automatically when ACT-R is initialized with the clear-all command which should prevent problems when loading different models. If desired, ACT-R can also close the windows when the system is reset by setting the system parameter :close-exp-windows-on-reset to t.

**examples:**

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

## Clear-exp-window

This command takes no required parameters and has one optional parameter which can indicate a specific window (either by title or object). If no window is provided then it will operate on the current window. It removes all of the items from that experiment window.

**examples:**

```
(clear-exp-window)
(clear-exp-window *exp-window*)
```

# Displaying Items (step 3)

Displaying items in experiment windows effectively involves two steps: creating an item for the window and adding that item to the window.  The most commonly used commands in the AGI perform both steps together, but there are also commands for creating items without displaying them, for removing specific items without clearing the entire display, and a command which allows one to put created or removed items on the display.  When creating the items, the AGI commands will construct an appropriate object for the type of window involved (one of the supported real windows, virtual, or visible virtual) and that item will be returned from the function.  The type of those returned items will be different for each type of window, and the details of those item objects is not part of the AGI's API.  Thus, the recommendation is to treat those objects as unmodifiable except through the use of the AGI modifying commands for the items.

## Colors

For all of the item creation commands there is an option to specify a color for the item.  When providing a color it must be a symbol naming the color and not a Lisp specific color object, even if it is a real window.  The following color names are supported across platforms and window types: black, blue, red, green, white, pink, yellow, gray, light-blue, dark-green, purple, brown, light-gray, or dark-gray.  If an unsupported color name is provided then the results depend on whether the window is visible or virtual.  For a virtual window the provided color name will be used and made available to the model, but for a visible window an unsupported color will be drawn in black and that is what the model will see as well.

## Item Specific Commands

For each of the provided display items (text, buttons, and lines) there are three commands.  Those commands can be used to create an item without displaying it, create an item and display it in a window, or modify an item which has been created.  The commands for each purpose take the same set of parameters for a given item type and return the corresponding item.

### Text

**add-text-to-exp-window**
**create-text-for-exp-window**
**modify-text-for-exp-window**

These commands take several keyword parameters and are used to draw a text string in an experiment window based on the values of those parameters.   The :window keyword

parameter can be used to specify the window to use for the add and create commands and the current window will be used if no :window parameter is specified. The modify command requires the item to be modified as its only required parameter. The return value is a text object which is appropriate for the indicated experiment window.

The :text parameter specifies the text string to display and defaults to "". 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, 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 in pixels. The default value for :height is 20 and for :width is 75. [One thing to note about the :height and :width parameters is that although the text shown in a real window or a visible virtual window may be clipped at the borders of the box specified a model will always see the entire text string regardless of the box boundaries.] The :color parameter specifies in which color the text will be drawn 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.

**examples:**

```
(add-text-to-exp-window :text "Ok")
(create-text-for-exp-window :text "a" :x 150 :y 150)
(modify-text-for-exp-window *text-item* :text "b" :color 'blue)
```

## Buttons

**add-button-to-exp-window**
**create-button-for-exp-window**
**modify-button-for-exp-window**

These commands take several keyword parameters and are used to place a button in an experiment window. The :window keyword parameter can be used to specify the window to use for the add and create commands and the current window will be used if no :window parameter is specified. The modify command requires the item to be modified as its only required parameter. The return value is a button object which is appropriate for the current experiment window.

The :text parameter must be a string and specifies the text to display on the button and it defaults to "Ok". The :x and :y parameters specify the pixel coordinate of the upper-left corner of the button and each defaults to 0. The :height and :width parameters specify the size of the button in pixels, and :height defaults to 18 and :width defaults to 60. The :action parameter specifies a function to be called when this button is pressed and defaults to **nil** (no function to call). When provided, that function will be called with the button object itself as the only parameter. The :color parameter specifies a background color for the button, and defaults to gray. When specifying a color for buttons in visible windows one may not actually see the color displayed because of issues with the Lisp or OS being

used, but the model will see the appropriate color regardless of what is shown on the display.

**examples:**

```
(add-button-to-exp-window :text "Ok" :x 100 :y 150)
(create-button-for-exp-window :text "Cancel" :color 'red)
(modify-button-for-exp-window *but-obj* :text "x" :action (lambda (button)
                                                   (setf *result* 'x)))
```

## Lines

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

These commands take two required parameters and two keyword parameters and are used to place a line in an experiment window.   The :window keyword parameter can be used to specify the window to use for the add and create commands and the current window will be used if no :window parameter is specified.  The modify command requires the item to be modified as an additional (first) required parameter.  The return value is a line object which is appropriate for the current experiment window.

This two required parameters specify the pixel coordinates of the end points of the line and each should be a two element list of the x and y coordinate respectively for one end of the line.  The keyword parameter :color can be used to specify the color in which to draw the line.

When modifying a line, if only the color needs to be changed the start and end points may both be specified as **nil**, but if either coordinate pair is to be changed both must be specified.

**examples:**

```
(add-line-to-exp-window (list 75 35)  (list 125 35))
(create-line-for-exp-window '(0 0) '(10 35) :color 'green)
(modify-line-for-exp-window *line-obj* '(0 0) '(10 35) :color 'blue)
```

## General Commands

These commands work for any of the items which have been created using the commands above.

## Remove-items-from-exp-window

This command takes an arbitrary number of parameters and an optional keyword parameter :window which specifies the window to use (if not provided then the current window is used).  Each of those items is removed from the current window.

**examples:**

```
(let ((a (add-text-to-exp-window :text "a"))
      (b (add-button-to-exp-window :text "b" :x 50)))
  (remove-items-from-exp-window a b))
```

## Add-items-to-exp-window

This command takes an arbitrary number of parameters each of which must be an object that has been created for the current experiment window.  Each of those items is then added to the current display if it is not already there.

**examples:**

```
(let ((a (create-text-for-exp-window :text "a"))
      (b (create-button-for-exp-window :text "b" :x 50)))
  (add-items-to-exp-window a b))
```

# Waiting on user interaction (step 4)

There are no commands in the AGI which are relevant for waiting for an ACT-R model, but there are some AGI (and Lisp) commands which are useful when a person is interacting with the experiment windows. There are however several ACT-R commands which are relevant for interfacing a model to a task. Those ACT-R commands are described fully in the ACT-R Reference Manual, but this section will include a brief description of some of the most important ones.

## Install-device

This command takes one parameter which must be an valid device (for example an experiment window created with open-exp-window). This tells the model which device it is interacting with. All of the models actions (key presses, mouse movement and mouse clicks) will be sent to this device and the contents of this device will be what the model can "see".

**examples:**

```
(let ((w (open-experiment-window "test")))
  (install-device w))
```

## Proc-display

This command can take one keyword parameter, :clear. Calling this command tells the model to process the currently installed device for visual information i.e. this command makes the model "look" at the current contents of the installed device. Whenever the window is changed you must call proc-display again to make sure the model becomes aware of those changes. The re-encoding and buffer stuffing mechanisms described in the tutorial only happen after this command is called. If the :clear parameter is specified as **t** it will cause the model to treat the window as all new items – everything there will be considered unattended.

**examples:**

```
(proc-display)
(proc-display :clear t)
```

## Run & run-full-time

These commands take one required parameter which is the time to run a model in seconds and a keyword parameter called :real-time. The run command will run the model until

either the requested amount of time passes, or there is nothing left for the model to do (no productions will fire and there are no pending actions that can change the state). The run-full-time command will run the model for exactly the specified amount of time regardless of whether it has any actions to perform. If the keyword parameter :real-time is specified as **t**, then these commands will run the model in step with real time instead of being allowed to run as fast as possible in its own simulated time. If the :real-time parameter is a number then that indicates a scaling of the real time performance to use e.g. a value of 2 will attempt to run the model twice as fast as real time. Note, that if real windows are being used then the :real-time parameter should always be specified as **t**.

**examples:**

```
(run 10)
(run-full-time 5 :real-time t)
```

## Allow-event-manager

This command is for use when a person is interacting with an experiment window. It takes one parameter, which must be an experiment window. It calls the appropriate function for the current Lisp and operating system to handle user interaction when a person is doing the task. Its use is necessary to give the system a chance to handle the real user interactions otherwise the data collection methods may never be called. It is not necessary when a model is performing the task. It is recommended that it be called frequently until the user has responded.

**examples:**

```
(let ((w (open-experiment-window "test")))
  (loop
      (when (user-response-received) (return))
      (allow-event-manager w)))
```

## Sleep

Sleep is a function defined in ANSI Common Lisp, but because it is used for experiment generation in the tutorial tasks it is being included here. The Lisp specification for sleep says it takes one parameter, seconds, which is a non-negative real, and it causes execution to cease and become dormant for approximately the seconds of real time indicated by seconds, whereupon execution is resumed. An important thing to note is that this is only useful when a person is doing a task. **The sleep function will have no effect upon the timing of actions from the model's perspective**, but it will increase the time it takes to run the model from the user's perspective.

**examples:**

```
(sleep .5)
```

# Response collection (step 5)

When a key press or mouse click occurs in an experiment window (generated by either a person or the model) the method rpm-window-key-event-handler or rpm-window-click-event-handler respectively will be called.  Thus, to record such responses you must define those methods on the rpm-window class (a super class of all the windows which can be generated by the open-exp-window command).  Those definitions will look like this:

```
(defmethod rpm-window-key-event-handler ((win rpm-window) key)
  ...)

(defmethod rpm-window-click-event-handler ((win rpm-window) pos)
  ...)
```

with your code to record the responses inside of them.

The parameters passed to the rpm-window-key-event-handler will be the window in which the key press occurs and the key that is pressed.  For the 'normal' keys (letters, numbers, and simple punctuation keys) the key parameter will be the character of that item, but for other things (function keys, arrow keys, etc) it may be a system dependent character or possibly a symbol representation of that key.  Thus, you will have to be careful when using such keys to make sure that your method can properly handle those items.

The parameters passed to the rpm-window-click-event-handler will be the window in which the mouse click occurs and the position of the mouse when the click occurred.  The position will be a two element vector of the x and y pixel coordinates within the window of the mouse pointer at the time of the click.

## Get-time

This command takes an 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.  The time from the internal timer is not zero referenced with respect to the task, so one needs to make sure and record the time at the start of the trial for reference when necessary.  Although the time is measured in milliseconds for a person that does not mean that it is necessarily accurate to that resolution (see the section on collecting human data for more details).

**examples:**

```
(get-time)
(get-time nil)
```

# 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 'collections' of numbers.  The numbers can be in arrays or lists and the two parameters do not need to be in the same format (one could be an array and the other a list).  The only requirement is that they have the same number of numbers in them.  This function then extracts those numbers and computes the correlation between the two sets of numbers.  That correlation value is returned.  There is a keyword parameter :output which defaults to t.  When :output is t the correlation is printed to *standard-output*.  If :output is **nil** then nothing is printed, and if it is a string, stream, or pathname then that is used to open a stream (if necessary) to which the results are written.

**examples:**

```
(let ((results (perform-task)))
  (correlation results '(.6 .65 .7 .86 1.12 1.5 1.79 2.13 2.15 2.58)))

(correlation (list 1 2 3 4) (list .3 .5 .9 1.2) :output nil)

(correlation (vector 1 2 3) (list 4 3 2))
```

## Mean-deviation

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

**examples:**

```
(let ((results (perform-task)))
  (mean-deviation results '(.6 .65 .7 .86 1.12 1.5 1.79 2.13 2.15 2.58)))

(mean-deviation (list 1 2 3 4) (list .3 .5 .9 1.2) :output nil)

(mean-deviation (vector 5 9 3) (list 1 2 9))
```

## Miscellaneous

This section contains some additional AGI functions 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 if needed.

**examples:**

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

### While

This is a looping construct which may be more familiar to 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*)
   (allow-event-manager window))
```

# Visual Features of AGI Elements for Models

When ACT-R processes a display that was generated though the AGI the visual features that are generated for buttons, lines, and text are processed similarly regardless of which (if any) display is being used.  However, there are some differences between the different devices based on the specific details of the particular system and thus there may be some minor difference in the features the model sees when the same code is run under different circumstances.  There may also be some discrepancies between what the model sees and what is actually displayed on the screen when a real window is used in some circumstances because of limitations or issues with the windowing system being used. This section will describe how the visual features are constructed for the model, what differences may result in the model representation based on the default visual processing mechanisms, and what inconsistencies one may encounter with the real displays.

## 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 Lisp character which is not graphic-char-p as well as the character #\space.  Alphanumeric characters are any characters which are alphanumericp or which have been specified with the add-word-characters command (add-word-characters can change a whitespace character into an alphanumeric).  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 slots of the visual-location representing a word are set in two ways.  One set of slots will receive values that are consistent among the various devices provided with ACT-R and the other set of slots will receive values based on characteristics particular to the specific device which is being used.  The slots which will be set consistently are:

**kind** – the value **text**

**value** – also the value **text** (Note however that a visual-location request may specify a specific string for the value slot in a request and that will be compared to the actual text on the display.  Thus, the model can request the location of an "A" on the screen, but the chunk which encodes the features will not contain the "A" because to get that information requires a shift of attention.)

**color** – the color specified when creating the text item (or **black** if no color was provided)

**distance** – the current default distance in pixels based on the value of the :viewing-distance and :pixels-per-inch parameters

The other set of slots represent the position and size information based on the specific font which is being used for the display and the value of the device parameters which can

be set to describe the display.  The font values are likely to vary among devices for the same text display.  Here the details as to how the values are derived will be provided as well as the default settings for the virtual device.

In determining the location of each word the first thing to consider is how the text is being displayed.  The assumption for the AGI text items is that it starts inside the top left corner of the box specified by the x and y coordinates given for the text item.  Any newlines in the text result in moving the following text down one "line" and starting again at the left side of the box. [Note, the height and width values provided for a text item do not actually constrain the feature generation for the model, but may truncate the results actually shown on the screen for some devices.]  Based on the font being used for the display the following values are determined: how tall the font is (the ascent value), how tall a line of text is (the ascent plus the descent), how wide each word is, and how wide the space character is.  Using those values along with the pixels per inch of the display, as specified by the :pixels-per-inch parameter (which defaults to 72) the remaining slots of the visual-location chunk are set as follows:

**height** – how tall the font is

**width** – the width of the word in pixels

**size** – the area of the word in degrees of visual angle squared determined by multiplying the height and width after converting them into degrees of visual angle based on the viewing distance and pixels per inch settings

**screen-x** – the x coordinate of the center of the word based on its position (determined using the x position of the text item and the width of the words and spaces that precede it on the current line) and its width

**screen-y** – the y coordinate of the center of the word based on its position (determined using the y position of the text item and how many lines of text are above it) and the font height

For the virtual device the default values for the 12 point font information are: 10 pixels high, each character (including the space) is 7 pixels wide, and a line is 12 pixels high.

Not all of the visible devices properly display text with newlines.  In some cases one may only see the first line of the text, and in other cases all of the text may be displayed on one line.  However, the model will always see the text as if it is formatted as described above.

## 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 button's feature will have the following properties:

**kind** – the value **oval**

**value** – the value **oval**

**color** – the color specified when creating the button or **gray** if no color was specified

**distance** – the current default distance in pixels based on the value of the :viewing-distance and :pixels-per-inch parameters

**height** – the height specified when creating the button or 18 if no height provided

**width** – the width specified when creating the button or 60 if no width provided

**size** – the area of the button in degrees of visual angle squared determined by multiplying the height and width after converting them into degrees of visual angle based on the viewing distance and pixels per inch settings

**screen-x** – the x coordinate of the center of the button based on its position and its width

**screen-y** – the y coordinate of the center of the button based on its position and its height

The text value provided for the button (which defaults to "Ok" if no text parameter is given when creating the button) will result in text features being generated almost the same as described above for text items. There are two differences between the text features generated from a button and text features generated from text items. The first is that the color of a text item from a button will always be **black**. The other has to do with how the x and y positions are determined.

The text features for a button are assumed to be placed so that they are 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.

There are some discrepancies in how the real windows display buttons among the various device and OS combinations relative to what the model sees. First, the same possible issues for text items apply to the text on buttons as well. The model can see all the text displayed on the button regardless of how large the button actually is, whereas in the real display some text may be clipped at the button boundary. Also, when there are newlines in the text for a button it may or may not display in multiple lines centered appropriately on the button. Another issue with buttons is that they may or may not actually be drawn as large as the height and width indicate they should, but the model will always see them in the size specified. Finally, the button may or may not display in the color indicated. In some situations the text on the button will be drawn in the color for the button instead of the button itself being filled with the color specified. Even in those situations where the text is drawn in the button's color the model will still see the text as **black** and the button as the appropriate color.

## Lines

Each line added to the display will generate one feature for the model. The feature for the line depends on the properties specified when creating the line as well as the :viewing-distance and :pixels-per-inch parameters.  A line's feature will have the following properties:

**kind** – the value **line**

**value** – the value **line**

**color** – the color specified when creating the line or **black** if no color was specified

**distance** – the current default distance in pixels based on the value of the :viewing-distance and :pixels-per-inch parameters

**height** – the difference between the y coordinates of the endpoints of the line

**width** – the difference between the x coordinates of the endpoints of the line

**size** – the area of the line in degrees of visual angle squared determined by multiplying the height and width after converting them into degrees of visual angle based on the viewing distance and pixels per inch settings

**screen-x** – the mid-point between the x coordinates of the endpoints of the line

**screen-y** – the mid-point between the y coordinates of the endpoints of the line


When displaying lines in real windows there are a couple of issues that may arise.  One is that the lines may not be displayed until the system event manager updates the display. Thus it may be necessary to call allow-event-manager after adding lines to a real display. The other is that for some devices lines drawn along the borders of the window may not be visible i.e. those with both x coordinates equal to zero or the width of the window minus one or both y coordinates equal to zero or the height of the window minus one.

# Multiple AGI Windows

The description of the commands above mention the possibility of using multiple windows.  This section will provide a little more detail on how to do so.  The important issue with respect to the commands is the "current" window, and the current window is relative to the current model.  First we will describe how that works when there is only one model.

If there is only one experiment window open then it will be the current window by default and the commands will act upon it.  If more than one window is opened in the context of a single model then there will not be a default current window.  In that case the commands must specify which window is the current one to use.  That is done by including the window object or window title of the desired window to the command as indicated above (either as an optional parameter or a keyword parameter).  Here is a very simple example showing two windows being opened and adding a single item to each followed by the first one then being brought to the front:

```
(let ((w1 (open-exp-window "A"))
      (w2 (open-exp-window 'b :x 0 :y 0)))
  (add-text-to-exp-window :text "X" :window w1)
  (add-button-to-exp-window :text "Ok" :window 'b)
  (select-exp-window "A"))
```

The same rules for current windows apply when there are multiple models currently defined.  The important distinction is that each model will have its own current experiment window.  When a window is created with open-exp-window it is considered to belong to the current model at that time and each model will have a set of windows which belong to it.  When opening experiment windows the title of a window is only meaningful within a particular model.  Thus, more than one model can open a window with the same title, but within a model there can only be one window with each title.  However, just because an experiment window belongs to a particular model does not preclude other models from interacting with that window – any window can be installed as the current device for any model.  The only potential issue with that is making sure that all the commands which manipulate that window properly indicate it as being the current one which will require specifying the window object (not the title) when accessing it from a model other than the owning one.

Examples of using multiple experiment windows with multiple models can be found in the "examples/agi" and "examples/model-task-interfacing" directories of the software distribution.

## Mouse Cursors

When there are multiple experiment windows and/or multiple models there are some issues to be aware of with respect to how models interact with the mouse cursor.  If the

windows which are opened are real windows within Lisp then all models which are using those windows will be using the single real mouse cursor for interaction with all of those windows. Virtual windows however, whether visible or not, will each have their own virtual cursor which is shared by all models that have that window installed as their current device. Whenever a mouse cursor is shared among models extra care should be taken with those models when performing actions with the mouse because there is no guarantee that it will stay where the model moves it, and even actions like clicking a button could "fail" since the cursor could be moved off of the button between the model's request to press the button and the model's finger actually depressing it.

## AGI Summary

The AGI was primarily designed to support the ACT-R tutorial.  As such, it is fairly minimal in what it provides, but that was one of the goals – to keep it simple.  Hopefully, people find the AGI useful, and if you have any questions, suggestions, or comments about the AGI feel free to contact me at db30@andrew.cmu.edu.

## Issues with using the AGI for human data collection

Because the AGI is intended to work with many different ANSI Common Lisp implementations the tools provided are not tuned for high-fidelity human data collection in any particular one.  Thus, as provided, the AGI includes no guarantees as to its performance.  So, essentially, 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.  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 which 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, things are more complicated because how user actions are handled for real interfaces vary from Lisp to Lisp.  For information on that when using one of the native GUI interfaces one would have to consult any documentation available for the corresponding Lisp, and may need to also investigate the device code in ACT-R (which is below the level at which support is provided).  In some past investigation it was found that the default event handler which is used by the AGI in one Lisp had a latency which would be significant for some tasks (60-120ms) and getting "good" timing data required writing a low-level event handler replacement.  That may not be true anymore, but can only be determined with appropriate investigation.  If the visible virtual windows are being used then things could be significantly worse because that response has to be processed in the ACT-R Environment (through the Tcl/Tk event handler) and then transmitted to the AGI in Lisp over a TCP/IP socket connection which is monitored by a background thread running in the Lisp.  In either case, there are a lot of factors involved which affect the response detection latency which will need to be investigated.

Latency on display changes is basically the same as for the response detection – there could be significant delays depending on the mechanisms involved.  Again, one will have to consult the specific Lisp documentation for details on native GUIs, and for the visible virtuals there is a TCP/IP socket connection in the chain of events that must happen before something gets to the display.

Because of the uncertainty in the performance we do not recommend using the AGI for human data collection without thorough testing by the user on the target machine to assess its performance.