Unit 3: Attention

This unit is concerned with developing a better understanding of how perceptual attention works in ACT-R, particularly as it is concerned with visual attention.

3.1 Visual Locations

When a visual display such as this

V N T Z

C R Y K

W J G F

is presented to ACT-R a representation of all the visual information is immediately accessible in a visual icon. One can view the contents of this icon using the "Visicon" button in the environment or with the command **print-visicon**:

```
> (print-visicon)

Loc        Att   Kind          Value             Color          ID
---------  ---   ------------  ----------------  -------------  ------------
( 80 107)  NEW   TEXT          "v"               BLACK          VISUAL-LOCATION0
( 80 157)  NEW   TEXT          "c"               BLACK          VISUAL-LOCATION1
( 80 207)  NEW   TEXT          "w"               BLACK          VISUAL-LOCATION2
(130 107)  NEW   TEXT          "n"               BLACK          VISUAL-LOCATION3
(130 157)  NEW   TEXT          "r"               BLACK          VISUAL-LOCATION4
(130 207)  NEW   TEXT          "j"               BLACK          VISUAL-LOCATION5
(180 107)  NEW   TEXT          "t"               BLACK          VISUAL-LOCATION6
(180 157)  NEW   TEXT          "y"               BLACK          VISUAL-LOCATION7
(180 207)  NEW   TEXT          "g"               BLACK          VISUAL-LOCATION8
(230 107)  NEW   TEXT          "z"               BLACK          VISUAL-LOCATION9
(230 157)  NEW   TEXT          "k"               BLACK          VISUAL-LOCATION10
(230 207)  NEW   TEXT          "f"               BLACK          VISUAL-LOCATION11
```

This prints the information of all the features that are available for the model to see. For each feature it shows the screen location, attentional status, general object type, specific value information, color, and a name by which it will be referenced. This low-level feature set is what is searched when a visual-location request is made.

3.1.1 Visual Location Requests

When requesting the visual location of an object there are many slots that can be specified in the request. In the last unit we only used the request parameter :attended. We will expand on the use of :attended in this unit. We will also provide details on specifying the slots in a visual-location request, and show another request parameter for the **visual-location** buffer called :nearest.

3.1.2 The Attended Test in More Detail

The :attended request parameter was introduced in unit 2. It tests whether or not the model has attended the object at that location, and the possible values are **new**, **nil**, and **t**. Very often we use the fact that attention tags elements in the visual display as attended or not to enable us to draw attention to the previously unattended elements. Consider the following production:

```
(p find-random-letter
   =goal>
     isa      read-letters
     state    find
==>
   +visual-location>
     :attended nil
   =goal>
     state    attending)
```

In its action, this production requests the location of an object that has not yet been attended. Otherwise, it places no preference on the location to be selected. When there is more than one item in the visicon that matches the request, the one most recently added to the visual icon (the newest one) will be chosen. If multiple items also match on their recency, then one will be picked randomly among those. If there are no objects which meet the constraints, then the **error** state will be set for the **visual-location** buffer. After a feature is attended (with a **visual** buffer request to move-attention), it will be tagged as attended **t** and this production's request for a visual-location will not return the location of such an object.

3.1.2.1 Finsts

There is a limit to the number of objects which can be tagged as attended **t**, and there is also a time limit on how long an item will remain marked as attended **t**. These attentional markers are called finsts (INSTantiation FINgers) and are based on the work of Zenon Pylyshyn. The number of finsts and the length of time that they persist can be set with the parameters :visual-num-finsts and :visual-finst-span respectively.

The default number of finsts is four, and the default decay time is three seconds. Thus, with these default settings, at any time there can be no more than four objects marked as attended **t**, and after three seconds the attended state of an item will revert from **t** to **nil**. Also, when attention is shifted to an item that would require more finsts than there are available the oldest one is reused for the new item i.e. if there are four items marked with finsts as attended **t** and you move attention to a fifth item the first item that had been marked as attended **t** will now be marked as attended **nil** and the fifth item will be marked as attended **t**. Because the default value is small, productions like the one above are not very useful for modeling tasks with a large number of items on the screen because the model will end up revisiting items very quickly. One solution is to always set :visual-num-finsts to a value that works for your task. However, changing architectural parameters like that is not recommended without a good reason, and keeping the number of parameters one needs to change for a model as small as possible is generally desired.

After discussing some of the other specifications one can use in a request we will come back to ways to work with the limited set of finsts.

3.1.3 Visual-location slots

The chunk-type used to specify the location chunks for the default ACT-R device is created like this:

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

In the tutorial we will only be using the built-in ACT-R device which is designed around interacting with a 2-D screen. All of those slots are used in creating the features for the default interface objects. Here we will describe how those slots are used in the default device.

The screen-x and screen-y slots represent the location based on its x and y position on the screen and are measured in pixels. The upper left corner is screen-x 0 and screen-y 0 with x increasing from left to right and y increasing from top to bottom. The distance slot will always have a value of 1080, which is also measured in pixels, and represents a distance of 15 inches from the model to the screen with a screen resolution of 72 pixels per inch.

The height and width slots hold the dimensions of the item measured in pixels. The size slot holds the approximate area covered by the item measured in degrees of visual angle squared. These values provide the general shape and size of the item on the display.

The color slot holds a representation of the color of the item. This will be a symbolic value like black or red and are defined as chunks by the vision module.

The kind and value slots provide a description of the item but may not contain all of the specific information needed to fully describe the item. To get the specific information the model will have to attend to the item. The kind slot specifies a general classification of the item, like text or line. The value slot holds some description which is assumed to be available without attending to the item and often will be the same as the kind slot. The value column shown in the visicon for a feature is the specific information which will be available after attending the item, and is usually not the same as the value which will be provided in the visual-location for the feature. For example, the visual-location chunks placed into the buffer for the text letters above will have a value slot of text instead of the specific letter.

It is possible for one to specify abstract devices for a model to interact with instead of a simple computer screen. When doing so, one may create additional chunk-types to represent the visual-location information which may have different slots to hold the relevant information. Those additional slots can be requested in the same ways as any of the default slots as described below. Creating new devices however is beyond the scope of the tutorial (see the extending-actr slides in the docs directory of the distribution and the models in the examples directory for information on creating new devices).

3.1.4 Visual-location request specification

One can specify constraints for a visual-location request using any of the slots in the visual-location chunk-type (or any chunk-type which has been used to create features for the device). Any of the slots may be specified using any of the modifiers (-, <, >, <=, or >=) in much the same way one specifies a retrieval request. Each of the slots may be specified any number of times. In addition, there are some special tests which one can use that will be described below. All of the constraints specified will be used to find a visual-location in the visicon to be placed into the **visual-location** buffer. If there is no visual-location in the visicon which satisfies all of the constraints then the **visual-location** buffer will indicate an error state.

3.1.4.1 Exact values

If you know the exact values for the slots you are interested in then you can specify those values directly:

```
+visual-location>
   screen-x 50
   screen-y 124
   color    black
```

You may also use the negation test, -, with the values to indicate that you want a location which does not have that value:

```
+visual-location>
   color    black
 - kind     text
```

Often however, one does not know the specific information about the location of visual items in advance and things need to be specified more generally in the model.

3.1.4.2 General values

When the slot being tested holds a number it is also possible to use the slot modifiers <, <=, >, and >= along with specifying the value. Thus to request a location that is to the right of screen-x 50 and at or above screen-y 124 one could use the request:

```
+visual-location>
 >  screen-x 50
 <= screen-y 124
```

In fact, one could use two modifiers for each of the slots to restrict a request to a specific range of values. For instance to request an object which was located somewhere within a box bounded by the corners 10,10 and 100,150 one could specify:

```
+visual-location>
  > screen-x 10
  < screen-x 100
  > screen-y 10
  < screen-y 150
```

3.1.4.3 Production variables

It is also possible to use variables from the production in the requests instead of specific values. Consider this production which uses a value from a slot in the goal to test the color:

```
(p find-by-color
   =goal>
    target =color
==>
  +visual-location>
    color  =color
)
```

Variables from the production can be used just like specific values along with the modifiers. Assuming that the LHS of the production binds =x, =y, and =kind this would be a valid request:

```
  +visual-location>
    kind     =kind
 <  screen-x =x
 -  screen-x 0
 >= screen-y =y
 <  screen-y 400
```

3.1.4.4 Relative values

If you are not concerned with any specific values, but care more about relative properties then there are also ways to specify that. You can use the values **lowest** and **highest** in the specification of any slot which has a numeric value. Of the chunks which match the other constraints the one with the numerically lowest or highest value for that slot will then be the one found.

In terms of screen-x and screen-y, remember that x coordinates increase from left to right, so **lowest** corresponds to leftmost and **highest** rightmost, while y coordinates increase from top to bottom, so **lowest** means topmost and **highest** means bottommost.

If this is used in combination with :attended it can allow the model to find things on the screen in an ordered manner. For instance, to read the screen from left to right a model could use:

```
+visual-location>
   :attended nil
   screen-x lowest
```

assuming that it also moves attention to the items so that they become attended and that there are sufficient finsts to tag everything.

There is one note about using **lowest** and **highest** when more than one slot is specified in that way for example:

```
+visual-location>
   width    highest
   screen-x lowest
   color    red
```

First, all of the non-relative values are used to determine the set of items to be tested for relative values. Then the relative tests are performed one at a time in the order provided to reduce the matching set. Thus, the specification above would first consider all items which were red because that is a constant value. Then it would reduce that to the set of items with the highest width (widest) and then of those it would pick the one with the lowest screen-x coordinate (leftmost). That may not produce the same result as this request for the same set of visicon chunks:

```
+visual-location>
   screen-x lowest
   width    highest
   color    red
```

This request will again start with all red items. Then it will find those with the lowest x coordinate and among those will choose the widest.

3.1.4.5 The current value

It is also possible to use the special value **current** in a request. That means the value of the slot must be the same as the value for the location of the currently attended object (the one attention was last shifted to with a move-attention request). This request would find a location which had the same screen-x value as the current one:

```
+visual-location>
   screen-x current
```

You can also use the value **current** with the modifiers. The following test will find a location which is up and to the right of the currently attended object in a different color:

```
+visual-location>
   > screen-x current
   < screen-y current
   - color    current
```

If the model does not have a currently attended object (it has not yet attended to anything) then the tests for current are ignored.

3.1.4.6 Request variables

A special component of the visual-location requests is the ability to use variables to compare the particular values in a visual-location to each other in the same way that the LHS tests of a production use variables to match chunks. If a value for a slot in a visual-location request starts with the character & then it is considered to be a variable in the request in the same way that values starting with = are considered to be variables on the LHS of a production.

This request:

```
+visual-location>
   isa      visual-location
   height   &height
   width    &height
```

would attempt to find a location which has the same value in the height and width slots. The request variables can be combined with the modifiers and any of the other values allowed to be used in the requests. Here is an example which may not be the most practical, but shows most of the components in use together:

```
+visual-location>
   screen-x current
   screen-x &x
 > screen-y 100
   screen-y lowest
 - screen-y &x
```

That request would try to find a location which had a screen-x value which was the same as the currently attended location and a screen-y value which was the lowest one greater than 100 but not the same as the screen-x value.

Request variables are probably not very useful with the default slots of a visual-location, but could become very useful when one creates other devices and feature representations.

3.1.5 The :nearest request parameter

Like :attended, there is another request parameter available in visual-location requests. The :nearest request parameter can be used to find the items closest to the currently attended location, or some other location. To find the location of the object nearest to the currently attended location we can again use the value **current**:

```
+visual-location>
   :nearest current
```

It is also possible to specify any location chunk for the nearest test, and the location of the object nearest to that location will be returned:

```
+visual-location>
   :nearest  =some-location
```

If there are constraints other than nearest specified then they are all tested first. The nearest of the locations that matches all of the other constraints is the one that will be placed into the buffer. Specifically, the nearest is determined by the straight line distance using the screen-x, screen-y, and distance coordinates of the locations.

3.1.6 Ordered Search

Above it was noted that a production using this **visual-location** request (in conjunction with appropriate attention shifts) could be used to read words on the screen from left to right:

```
(p read-next-word
   =goal>
      state     find
==>
   +visual-location>
      :attended nil
      screen-x  lowest
   =goal>
      state     attend
)
```

However, if there are fewer finsts available than words to be read that production will result in a loop that reads only one more word than there are finsts. For instance, if there are six words on the line and the model only has four finsts (the default) then when it attends the fifth word the finst on the first word will be removed to use because it is the oldest. Then the sixth request will result in finding the location of the first word again because it is no longer marked as attended. If it is attended it will get the finst from the second word, and so on.

By using the tests for current and lowest one could have the model perform the search from left to right without using the :attended test:

```
(p read-next-word
   =goal>
      state    find
==>
   +visual-location>
    > screen-x current
      screen-x lowest
   =goal>
      state    attend
)
```

That will always be able to find the next word to the right of the currently attended one.

If items were to be read a line at a time one could add ‘screen-y current’ to that request along with an additional production for moving to the next line when the end of the current one is reached. If requests are combined with the :nearest request parameter, a variety of other ordered search strategies are also possible.

3.2 The Sperling Task

If you open the **sperling** model, you will see an example of the effects of visual attention. This model contains functions for administering the Sperling experiment where subjects are briefly presented with a set of letters and must try to report them. Subjects see displays of 12 letters such as:

V N T Z

C R Y K

W J G F

This model reproduces the partial report version of the experiment. In this condition, subjects are cued sometime after the display comes on as to which of the three rows they must report. The delay of the cue is either 0, .15, .3, or 1 second after the display appears. Then, after 1 second of total display time, the screen is cleared and the subject is to report the letters from the cued row. In the version we have implemented the responses are to be typed in and the space bar pressed to indicate completion of the reporting. For the cueing, the original experiment used a tone with a different frequency for each row and the model will hear simulated tones while it is doing the task. This task does not have a version which you can run through as a person because of complications with presenting real tones reliably across Lisps and machines.

In the original experiment the display is only presented for 50 ms and it is generally believed that there is an iconic visual memory that holds the stimuli for some time after onset which the participants are then processing. ACT-R’s vision module does not have such an iconic visual memory. Thus, for this task we have simulated this for ACT-R by having the display actually stay on for longer than 50ms. It will be visible for a random period of time between 0.9 to 1.1 seconds to simulate that effect. There were also some other differences to the actual visual conditions of the original task relative to what we are using for the model, but this simplified representation is sufficient for the purpose of demonstrating attention with this model.

One thing to note about this model is that it does not use the imaginal module, as described in the previous unit, to hold the problem representation separate from the control state. Instead, all of the task relevant information will be kept in the **goal** buffer chunk. This is done primarily to keep the productions simpler so as to keep the focus in this unit on the details of the attention mechanisms.

The following is the trace of ACT-R's performance of one trial of the task. In this trace the sound is presented .15 seconds after onset of the display and the target row was the middle one. This trace was generated with the :trace-detail set to low to avoid lots of the details for now:

```
> (do-sperling-trial .15)
0.000   GOAL           SET-BUFFER-CHUNK GOAL GOAL REQUESTED NIL
0.000   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1-0 REQUESTED NIL
0.050   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.185   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.185   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION3-0
0.200   AUDIO          SET-BUFFER-CHUNK AURAL-LOCATION AUDIO-EVENT0 REQUESTED NIL
0.235   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
0.285   PROCEDURAL     PRODUCTION-FIRED DETECTED-SOUND
0.320   VISION         SET-BUFFER-CHUNK VISUAL TEXT1
0.370   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.370   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION2-0
0.420   PROCEDURAL     PRODUCTION-FIRED ATTEND-LOW
0.505   VISION         SET-BUFFER-CHUNK VISUAL TEXT2
0.555   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.555   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0
0.570   AUDIO          SET-BUFFER-CHUNK AURAL TONE0
0.605   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
0.655   PROCEDURAL     PRODUCTION-FIRED SOUND-RESPOND-MEDIUM
0.690   VISION         SET-BUFFER-CHUNK VISUAL TEXT3
0.740   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.740   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION4-0
0.790   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
0.875   VISION         SET-BUFFER-CHUNK VISUAL TEXT4
0.925   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.925   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1-0
0.975   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
1.110   PROCEDURAL     PRODUCTION-FIRED START-REPORT
1.110   GOAL           SET-BUFFER-CHUNK GOAL CHUNK0
1.110   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL TEXT0-0
1.160   PROCEDURAL     PRODUCTION-FIRED DO-REPORT
1.160   MOTOR          PRESS-KEY KEY c
1.160   DECLARATIVE    SET-BUFFER-CHUNK RETRIEVAL TEXT4-0
1.760   PROCEDURAL     PRODUCTION-FIRED DO-REPORT
1.760   MOTOR          PRESS-KEY KEY r
1.760   DECLARATIVE    RETRIEVAL-FAILURE
2.260   PROCEDURAL     PRODUCTION-FIRED STOP-REPORT
2.260   MOTOR          PRESS-KEY KEY SPACE
2.560   ------         Stopped because no events left to process

answers: ("K" "Y" "R" "C")
responses: ("R" "C")
2
```

While the sound is presented at .150 seconds into the run it does not affect the model until **sound-respond-medium** fires at .655 seconds into the run to encode the tone. One of the things we will discuss is what determines the delay of that response. Prior to that time the model is finding letters anywhere on the screen. After the sound is encoded the search is restricted to the target row. After the display disappears, the production **start-report** fires which initiates the keying of the letters that have been encoded from the target row.

3.3 Visual Attention

As in the models from the last unit there are three steps that the model must perform to encode visual objects. It must find the location of an object, shift attention to that location, and then harvest the chunk which encodes the attended object. In the last unit this was done with three separate productions, but in this unit, because the model is trying to do this as quickly as possible the encoding and request to find the next are combined into a single production. In addition, for the first item's location there is no production that does an initial find.

3.3.1 Buffer Stuffing

Notice that the first production to fire in this model is attend-medium:

```
0.050   PROCEDURAL    PRODUCTION-FIRED ATTEND-MEDIUM
```

Here is the definition of that production:

```
(p attend-medium
   =goal>
     isa         read-letters
     state       attending
   =visual-location>
     isa         visual-location
   > screen-y    154
   < screen-y    166

   ?visual>
     state       free
==>
   =goal>
     location    medium
     state       encode
   +visual>
     cmd         move-attention
     screen-pos =visual-location)
```

Notice that it tests a chunk in the **visual-location** buffer, and it matches and fires even though there has not been a request to put a chunk into the **visual-location** buffer. However, there is a line in the trace prior to that which indicates that a visual-location was found:

```
0.000   VISION        SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1-0 REQUESTED NIL
```

This process is referred to as buffer stuffing and it occurs for both visual and aural percepts. It is intended as a simple approximation of a bottom-up mechanism of attention. When the **visual-location** buffer is empty and the model processes the display it can automatically place the

location of one of the visual objects into the **visual-location** buffer. The "requested nil" at the end of the line in the trace indicates that this setting of the chunk in the buffer was not the result a production's request.

You can specify the conditions used to determine which location, if any, gets selected for the **visual-location** buffer stuffing using the same conditions you would use to specify a **visual-location** request in a production. Thus, when the screen is processed, if there is a visual-location that matches that specification and the **visual-location** buffer is empty, then that location will be stuffed into the **visual-location** buffer.

The default specification for a visual-location to be stuffed into the buffer is :attended new and screen-x lowest. If you go back and run the previous units' models you can see that before the first production fires to request a visual-location there is in fact already one in the buffer, and it is the leftmost new item on the screen.

Using buffer stuffing allows the model to detect changes to the screen automatically. The alternative method would be to continually request a location that was marked as :attended new, notice that there was a failure to find one, and request again until one was found.

One thing to keep in mind is that buffer stuffing will only occur if the buffer is empty. If the model is busy doing something with a chunk in the **visual-location** buffer then it will not automatically notice a change to the display. If you want to take advantage of buffer stuffing in a model then you must make sure that all requested chunks are cleared from the buffer (the vision module will erase a stuffed chunk automatically from the **visual-location** buffer after .5 seconds by default). That is typically not a problem because the strict harvesting mechanism that was described in the last unit causes buffers to be cleared automatically when they are tested in a production.

3.3.2 Testing and Requesting Locations with Slot Modifiers

Something else to notice about this production is that the buffer test of the **visual-location** buffer shows modifiers being used when testing slots for values. These tests allow you to do a comparison when the slot value is a number, and the match is successful if the comparison is true. The first one (>) is a greater-than test. If the chunk in the **visual-location** buffer has a value in the screen-y slot that is greater than 154, it is a successful match. The second test (<) is a less-than test, and works in a similar fashion. If the screen-y slot value is less than 166 it is a successful match. Testing on a range of values like this is important for the visual locations because the exact location of a piece of text in the icon is determined by its "center" which is dependant on the font type and size. Thus, instead of figuring out exactly where the text is at in the icon (which can vary from letter to letter or even for a particular letter under different fonts) the model is written to accept the text in a range of positions.

After attention shifts to a letter on the screen, the production **encode-row-and-find** harvests the visual representation of the object, marks it with its row designation for future reference, and requests the next location:

```
(p encode-row-and-find
   =goal>
     isa       read-letters
     location  =pos
     upper-y   =uy
     lower-y   =ly
   =visual>
==>
   =visual>
     status    =pos
   -visual>
   =goal>
     location  nil
     state     attending
   +visual-location>
     :attended nil
   > screen-y  =uy
   < screen-y  =ly)
```

Note that this production places the row of the letter (=pos having values high, medium, and low) into the status slot of the visual object currently in the **visual** buffer. Later, when reporting, the model will restrict itself to recalling items from the designated row.

In addition to modifying the chunk in the **visual** buffer, it also explicitly clears the **visual** buffer. This is done so that the modified chunk goes into declarative memory. Remember that declarative memory holds the chunks that have been cleared from the buffers. Typically, strict harvesting will clear the buffers automatically, but because the chunk in the **visual** buffer is modified on the RHS of this production it will not be automatically cleared. Thus, to ensure that this chunk enters declarative memory at this time we explicitly clear the buffer.

The production also updates the goal chunk to remove the location slot and update the state slot, and makes a request for a new visual location. The visual-location request uses the < and > modifiers for the screen-y slot to restrict the visual search to a particular region of the screen. The range is defined by the values from the upper-y and lower-y slots of the chunk in the **goal** buffer. The initial values for the upper-y and lower-y slots are shown in the initial goal:

```
(goal isa read-letters state attending upper-y 0 lower-y 300)
```

and include the whole window, thus the location of any letter that is unattended will be potentially chosen. When the tone is encoded those slots will be updated so that only the target row's letters will be found.

3.3.3 Finsts in Use

There is one important feature to emphasize about this model, which may be useful in the assignment to follow. The model does not repeat letters because of the :attended nil test in the requests to the **visual-location** buffer.

Look back at the visual icon for the sperling task displayed above. You will note that all the characters are initially tagged as attended **new**. That means that they have not yet been attended and that they have been added to the icon recently. The time that items remain marked as **new** is parameterized and defaults to .5 seconds (it can be changed with the :visual-onset-span parameter). After that time if they still have not been attended they will be tagged as attended **nil**. This allows attention to be sensitive to the onset of an item. As we saw in the previous unit, visual attention has to be shifted to the object before a representation of it is built in the **visual** buffer and it can be accessed by a production. This corresponds to the research in visual attention showing that preattentively we have access to features of an object but we do not have access to its identity. This preattentive access to the objects is available through the **visual-location** buffer. When we move the model's attention to an object its attentional status is changed. So if the model moves its attention to the w and then the n we would get the following (assuming that it took less than .5 seconds to do so otherwise all the other items would be marked as attended **nil**):

```
> (print-visicon)

Loc        Att   Kind           Value              Color           ID
---------  ---   -------------  -----------------  --------------  -------------
( 80 107)  NEW   TEXT           "v"                BLACK           VISUAL-LOCATION0
( 80 157)  NEW   TEXT           "c"                BLACK           VISUAL-LOCATION1
( 80 207)  T     TEXT           "w"                BLACK           VISUAL-LOCATION2
(130 107)  T     TEXT           "n"                BLACK           VISUAL-LOCATION3
(130 157)  NEW   TEXT           "r"                BLACK           VISUAL-LOCATION4
(130 207)  NEW   TEXT           "j"                BLACK           VISUAL-LOCATION5
(180 107)  NEW   TEXT           "t"                BLACK           VISUAL-LOCATION6
(180 157)  NEW   TEXT           "y"                BLACK           VISUAL-LOCATION7
(180 207)  NEW   TEXT           "g"                BLACK           VISUAL-LOCATION8
(230 107)  NEW   TEXT           "z"                BLACK           VISUAL-LOCATION9
(230 157)  NEW   TEXT           "k"                BLACK           VISUAL-LOCATION10
(230 207)  NEW   TEXT           "f"                BLACK           VISUAL-LOCATION11
```

where the T's for these elements indicate that they have now been attended.

To keep this unit simple the number of finsts and the finst duration will be set to values large enough that it does not have to be considered. This unit is concerned with how the minimum time to search the display determines the behavior of the system and needing to specify a search strategy into the productions would only complicate things.

3.4 Auditory Attention

There are a number of productions responsible for processing the auditory message and they serve as our first introduction to the audio module. As in the visual case, there is an **aural-location** to hold the location of an aural message and an **aural** buffer to hold the sound that is attended. However, unlike the visual system we typical need only two steps to encode a sound and not three. This is because usually the auditory field of the model is not crowded with sounds and we can rely on buffer stuffing to place the sound's location into the **aural-location** buffer. If a new

sound is presented, and the **aural-location** buffer is empty, then the audio-event for that sound (the auditory equivalent of a visual-location) is placed into the buffer automatically. However, there is a delay between the initial onset of the sound and when the audio-event becomes available. The length of the delay depends on the type of sound being presented (tone, digit, word, or other) and represents the time necessary to encode its content. This is unlike the visual-locations which are immediately available.

In this task the model will hear one of the three possible tones on each trial. The default time it takes the model's audio module to encode a tone sound is .050 seconds. The **detected-sound** production responds to the appearance of an audio-event in the **aural-location** buffer:

```
(p detected-sound
   =aural-location>
   ?aural>
     state   free
   ==>
   +aural>
     event   =aural-location)
```

Notice that this production does not test the **goal** buffer. If there is a chunk in the **aural-location** buffer and the **aural** state is free this production can fire. It is not specific to this, or any task. On its RHS it makes a request to the **aural** buffer specifying the event slot. That is a request to shift attention and encode the event provided. The result of that encoding will be a chunk with slots specified by the chunk-type sound being placed into the **aural** buffer:

```
(chunk-type sound kind content event)
```

The kind slot will be used to indicate the type of sound encoded which could be tone, digit, or word by default, but custom kinds of sound can also be generated for a model. The value of the content slot will be a representation of the sound heard, and how that is encoded is different for different kinds of sounds (tones encode the frequency, words are encoded as strings, and digits are encoded as the number). The event slot contains a chunk which relates to the event that was used to attend the sound.

Our model for this task has three different productions to process the encoded sound chunks, one for each of high, medium, and low tones. The following is the production for the low tone:

```
(p sound-respond-low
   =goal>
     isa      read-letters
     tone     nil
   =aural>
     isa      sound
     content  500
==>
   =goal>
     tone     low
     upper-y  205
```

```
      lower-y  215)
```

The content slot of a tone sound encodes the frequency of the tone. For this experiment a 500 Hertz sound is considered low, a 1000 Hertz sound medium, and a 2000 Hertz sound high. On the RHS this production records the type of tone presented in the goal and updates the restrictions on the y coordinates for the search to constrain it to the appropriate row.

It takes some time for the impact of the tone to make itself felt on the information processing. Consider this portion of the trace in the case where the tone was sounded .150 seconds after the onset of the display:

```
0.135   VISION         SET-BUFFER-CHUNK VISUAL TEXT0
0.185   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.185   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION3-0
0.200   AUDIO          SET-BUFFER-CHUNK AURAL-LOCATION AUDIO-EVENT0 REQUESTED NIL
0.235   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
0.285   PROCEDURAL     PRODUCTION-FIRED DETECTED-SOUND
```

Although the sound was initiated at .150 seconds, it takes .050 seconds to detect the nature of the sound. Thus, its event appears in the **aural-location** buffer at .200 seconds. At .235 seconds **detected-sound** can be selected in response to the event that happened. It could not be selected sooner because the **attend-high** production was selected at .185 seconds (before the tone was available) and takes 50 milliseconds to complete. When the **detected-sound** production completes at .285 seconds aural attention is shifted to the sound.

```
0.285   PROCEDURAL     PRODUCTION-FIRED DETECTED-SOUND
...
0.570   AUDIO          SET-BUFFER-CHUNK AURAL TONE0
0.605   PROCEDURAL     PRODUCTION-FIRED ATTEND-HIGH
0.655   PROCEDURAL     PRODUCTION-FIRED SOUND-RESPOND-MEDIUM
0.690   VISION         SET-BUFFER-CHUNK VISUAL TEXT3
0.740   PROCEDURAL     PRODUCTION-FIRED ENCODE-ROW-AND-FIND
0.740   VISION         SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION4-0
0.790   PROCEDURAL     PRODUCTION-FIRED ATTEND-MEDIUM
```

Attending to and fully encoding the sound takes .285 seconds since we see that the encoded sound chunk is not placed into the **aural** buffer until time .570 seconds. The production **sound-respond-medium** is then selected at .605 seconds (after the **attend-high** production completes which had been selected before the sound chunk was available) and fires at .655 seconds. The next production to fire is **encode-row-and-find**. It encodes the last letter that was read and issues a request to look at a letter that is in the correct row this time, instead of an arbitrary letter. Thus, even though the sound is given at .150 seconds it is not until .690 seconds, when **encode-row-and-find** is selected, that it has any impact on the processing of the visual array.

3.5 Typing and Control

The production that initiates typing the answers is **start-report**:

```
(P start-report
   =goal>
```

```
    isa     read-letters
    tone    =tone
  ?visual>
    state   free
  ==>
  +goal>
    isa     report-row
    row     =tone
  +retrieval>
    status  =tone)
```

This causes a new chunk to be placed into the **goal** buffer rather than a modification to the chunk that is currently there (as indicated by the +goal rather than an =goal). The goal module's requests create new chunks the same way the imaginal module's requests do except that there is no time cost for creating a new goal chunk. This production also a retrieval request for a letter in the target row (indicated by the status slot as was set by the **encode-row-and-find** production).

This production's conditions are fairly general and it can match at many points in the model's run, but we do not want it to apply as long as there are letters to be perceived. We only want this rule to apply when there is nothing else to do. Each production has a quantity associated with it called its utility. The productions' utilities determine which production gets selected during conflict resolution if there is more than one that matches. We will discuss utility in more detail in later units. For now, the important thing to know is that the production with the highest utility among those that match is the one selected. Thus, we can make this production less preferred by setting its utility value low. The function for setting production parameters is **spp** (set production parameters). It is similar to **sgp** which is used for the general parameters as discussed previously. The utility of a production is set with the :u parameter, so the following call found in the model sets the utility of the **start-report** production to -2:

```
(spp start-report :u -2)
```

The default utility is 0. So, this production will not be selected as long as there are other productions with a higher utility that match, and in particular that will be as long as there is still something in the target row on the screen to be processed by the productions that encode the screen.

Also note that the productions that process the sound are given higher utility values than the default in the model:

```
(spp detected-sound  :u 10)
(spp sound-respond-low :u 10)
(spp sound-respond-medium :u 10)
(spp sound-respond-high :u 10)
```

This is so that the sound will be processed as soon as possible – these productions will be preferred over any others that match at the same time.

Once the report starts, the following production is responsible for reporting all the letters in the target row:

```
(P do-report
   =goal>
     isa      report-row
     row      =tone
   =retrieval>
     status   =tone
     value    =val
   ?manual>
     state    free
   ==>
   +manual>
     cmd      press-key
     key      =val
   +retrieval>
     status   =tone
     :recently-retrieved  nil)
```

This production fires when an item has been retrieved and the motor module is free. As actions, it presses the key corresponding to the letter retrieved and requests a retrieval of another letter. Notice that it does not modify the chunk in the **goal** buffer (which is the only buffer that does not get cleared by strict harvesting) and thus can fire again once the other conditions are met. Here is a portion of the trace showing the results of this production firing and its selection to fire again:

```
1.160   PROCEDURAL    PRODUCTION-FIRED DO-REPORT
1.160   MOTOR         PRESS-KEY KEY c
1.160   DECLARATIVE   SET-BUFFER-CHUNK RETRIEVAL TEXT4-0
1.760   PROCEDURAL    PRODUCTION-FIRED DO-REPORT
```

Something new that you may notice in that production is the request parameter in the retrieval request (:recently-retrieved). We will discuss that in the next section.

When there are no more letters to be reported (a retrieval failure occurs because the model can not retrieve any more letters from the target row), the following production applies to terminate processing:

```
(p stop-report
   =goal>
     isa     report-row
     row     =row
   ?retrieval>
     buffer  failure
   ?manual>
     state   free
==>
   +manual>
     cmd     press-key
```

```
    key     space
  -goal>)
```

It presses the space key to indicate that it is done and then clears the chunk from the **goal** buffer to stop the model.

3.6 Declarative Finsts

While doing this task the model only needs to report the letters it has seen once each. One way to do that easily is to indicate which chunks have been retrieved previously so that they are not retrieved again. However, one cannot modify the chunks in declarative memory. Modifying the chunk in the **retrieval** buffer will result in a new chunk being added to declarative memory with that modified information, but the original unmodified chunk will also still be there. Thus some other mechanism must be used.

The way this model handles that is by taking advantage of the declarative finsts built into the declarative memory module. Like the vision system, the declarative system marks items that have been retrieved with tags that can be tested against in the retrieval request. These finsts are not part of the chunk, but can be tested for with the :recently-retrieved request parameter in a retrieval request as shown in the **do-report** production:

```
  +retrieval>
     status   =tone
     :recently-retrieved nil
```

If it is specified as **nil**, then only a chunk that has not been recently retrieved (marked with a finst) will be retrieved. In this way the model can exhaustively search declarative memory for items without repeating. That is not always necessary and there are other ways to model such tasks, but it is a convenient mechanism that can be used when needed.

Like the visual system, the number and duration of the declarative finsts is also configurable through parameters. The default is four declarative finsts which last 3 seconds each. Those are set using the :declarative-num-finsts and :declarative-finst-span parameters respectively. In this model the default of four finsts is sufficient, but the duration of 3 seconds is potentially too short because of the time it takes to make the responses. Thus in this model the span is set to 10 seconds to avoid any potential problems:

```
(sgp :v t :declarative-finst-span 10)
```

3.7 Data Fitting

One can see the average performance of the model run over a large number of trials by using the function **run-sperling** and giving it the number of trials one wants to see run. However, there are a few changes to the model that one should make first. The first thing to change is to remove the sgp call that sets the :seed parameter which causes the model to always perform the same trial in

the same way, otherwise the performance is going to be identical on every trial. The easiest way to remove that call is to place a semi-colon at the beginning of the line like this:

```
;(sgp :seed (100 0))
```

A semi-colon in a Lisp file designates a comment and everything on the line after the semi-colon is ignored.

After making that change the model will be presented with different trials and perform differently from trial to trial (after the model is saved and reloaded).

There are other changes that can be made to the model and experiment code to make it run the simulation faster (take less real time to complete) without changing the simulated timing results.

The first is to turn off the trace by setting the :v parameter to **nil**:

```
(sgp :v nil :declarative-finst-span 10)
```

You will also want to turn off the printing of the answers and responses which is controlled by a global variable called *show-responses* in this task and it too should be set to **nil**:

```
(setf *show-responses* nil)
```

It can be sped up even more by making the model use a virtual window instead of a real one. A virtual window is an abstraction of a real window (a real window is a displayed window which both a person and a model can interact with) that the model can see and interact with as if it were a real window without the overhead of actually displaying and updating it on the screen. The other advantage of a virtual window for the model is that when interacting with a virtual window it does not have to be constrained to operate in real time and can run as fast as possible. Those changes will require making changes to the Lisp code that controls the experiment, and thus the details of how to do that are in the unit 3 experiment code document.

When one calls run-sperling with all of these changes, one sees something similar to:

```
> (run-sperling 100)
CORRELATION:  0.997
MEAN DEVIATION:  0.115

Condition    Current Participant   Original Experiment
 0.00 sec.            3.20                 3.03
 0.15 sec.            2.43                 2.40
 0.30 sec.            2.17                 2.03
 1.00 sec.            1.56                 1.50
```

This prints out the correlation and mean deviation between the experimental data and the average of the 100 ACT-R simulated runs. Also printed out are the original data from the Sperling experiment.

From this point on in the tutorial we will compare the performance of the models on the tasks to the data collected from people doing the tasks to provide a measure of how well the models compare to human performance. For the assignment models, you should be able to produce

models that compare to human performance at least as well as the model results shown in the units.

3.8 The Subitizing Task

Your assignment for this unit is to write a model for a subitizing task. This is an experiment where you are presented with a set of marks on the screen (in this case Xs) and you have to count how many there are. If you load the **subitize** model you can run yourself in this experiment by calling the subitize function and providing the symbol human:

```
(subitize 'human)
```

you will be presented with 10 trials in which you will see from 1 to 10 objects on the screen. The trials will occur in a random order. You should press the number key that corresponds to the number of items on the screen unless there are 10 objects in which case you should type 0. The following is the outcome from one of my runs through the task:

```
> (subitize 'human)

CORRELATION:  0.956
MEAN DEVIATION:  0.367
Items    Current Participant   Original Experiment
    1          0.80(T  )         0.60
    2          0.93(T  )         0.65
    3          0.91(T  )         0.70
    4          1.16(T  )         0.86
    5          1.46(T  )         1.12
    6          1.84(T  )         1.50
    7          1.75(T  )         1.79
    8          2.85(T  )         2.13
    9          2.73(T  )         2.15
   10          2.58(T  )         2.58
```

This provides a comparison between my data and the data from an experiment by Jensen, Reese, & Reese (1950). The value in parenthesis after the time will be either T or NIL indicating whether or not the answer the participant gave was correct (T is correct, and NIL is incorrect).

3.8.1 The Vocal System

We have already seen that the default ACT-R mechanism for pressing keys can take a considerable amount of time and can vary based on which key is pressed. That could have an effect on the results of this model. One solution would be to more explicitly control the hand movements to provide faster and consistent responses, but that is beyond the scope of this unit. For this task the model will instead provide a vocal response i.e. it will speak the number of items on the screen instead of pressing a key, which is how the participants in the data being modeled also responded. This is done by making a request to the speech module (through the **vocal** buffer) and is very similar to the requests to the motor module through the **manual** buffer which we have already seen.

Here is the production in the Sperling model that presses a key:

```
(P do-report
   =goal>
     isa      report-row
     row      =tone
   =retrieval>
     status   =tone
     value    =val
   ?manual>
     state    free
   ==>
   +manual>
     cmd      press-key
     key      =val
   +retrieval>
     status   =tone
     :recently-retrieved  nil)
```

With the following changes it would speak the response instead (note however that the **sperling** experiment is not written to accept a vocal response so it will not properly score those responses if you attempt to run the model with these modifications):

```
(P do-report
   =goal>
     isa      report-row
     row      =tone
   =retrieval>
     status   =tone
     value    =val
   ?vocal>
     state    free
   ==>
   +vocal>
     cmd      speak
     key      =val
   +retrieval>
     status   =tone
     :recently-retrieved  nil)
```

The primary change is that instead of the **manual** buffer we use the **vocal** buffer. On the LHS we query the **vocal** buffer to make sure that the speech module is not currently in use:

```
  ?vocal>
      state    free
```

Then on the RHS we make a request of the **vocal** buffer to speak the response:

```
  +vocal>
     cmd      speak
     string   =val
```

The cmd slot indicates the action to perform, in this case speak, and the string slot specifies the text to be spoken. The default timing for speech acts is .200 seconds per assumed syllable based on the length of the string to speak, and that works well for this task.

3.8.2 Exhaustively Searching the Visual Icon

When the model is doing this task it will need to exhaustively search the display. It can use the ability of the visual system to tag those elements that have been attended and not go back to them -- just as in the Sperling task. To make the assignment easier, the number of finsts has been set to 10 in the starting model. Thus, your model only needs to use the :attended specification in the visual-location requests. Once you have a model working that way you may want to try changing it so that it will also work with the default number of finst.

Another thing which the model will have to do is detect when there are no more unattended visual locations (or no location found when using a search strategy other than just the attended status). This will be signaled by a failure when a request is made of the **visual-location** buffer that cannot be satisfied. That is the same as when the **retrieval** buffer reports a failure when no chunk that matches a retrieval request can be retrieved. A query of the buffer for a failure is true in that situation, and the way for a production to test for that would be to have a query like this on the left-hand side:

```
(p no-location-found
...
   ?visual-location>
     buffer   failure
...
==>
...)
```

3.8.3 The Assignment

Your task is to write a model for the subitizing task that always responds correctly by **speaking** the number of items on the display, and does an approximate job of reproducing the human data. The following are the results from my ACT-R model:

```
CORRELATION:  0.980
MEAN DEVIATION:  0.230
Items    Current Participant   Original Experiment
```

```
 1        0.54  (T  )             0.60
 2        0.77  (T  )             0.65
 3        1.01  (T  )             0.70
 4        1.24  (T  )             0.86
 5        1.48  (T  )             1.12
 6        1.71  (T  )             1.50
 7        1.95  (T  )             1.79
 8        2.18  (T  )             2.13
 9        2.42  (T  )             2.15
10        2.65  (T  )             2.58
```

You can see this does a fair job of reproducing the range of the data. However, the human data shows little effect of set size (approx. 0.05-0.10 seconds) in the range 1-4 and a larger effect (approx. 0.3 seconds) above 4 in contrast to this model which increases about .23 seconds for each item. The small effect for little displays probably reflects the ability to perceive small numbers of objects as familiar patterns and the larger effect for large displays probably reflects the time to retrieve counting facts. Both of those effects could be modeled, but would require ACT-R mechanisms which have not been described to this point in the tutorial. Therefore the linear response pattern produced by this model is a sufficient approximation for our current purposes, and provides a fit to the data that you should aspire to match.

In the starting model you are provided there are chunks that encode numbers and their ordering from 0 to 10:

```
(add-dm (one isa chunk)(two isa chunk)
        (three isa chunk)(four isa chunk)
        (five isa chunk)(six isa chunk)
        (seven isa chunk)(eight isa chunk)
        (nine isa chunk)(ten isa chunk)
        (zero isa chunk) (eleven isa chunk)
        (start isa chunk)
        (n0 isa number-fact identity zero next one value "0")
        (n1 isa number-fact identity one next two value "1")
        (n2 isa number-fact identity two next three value "2")
        (n3 isa number-fact identity three next four value "3")
        (n4 isa number-fact identity four next five value "4")
        (n5 isa number-fact identity five next six value "5")
        (n6 isa number-fact identity six next seven value "6")
        (n7 isa number-fact identity seven next eight value "7")
        (n8 isa number-fact identity eight next nine value "8")
        (n9 isa number-fact identity nine next ten value "9")
        (n10 isa number-fact identity ten next eleven value "0")
        (goal isa count state start))
```

The number facts also contain a slot called value that holds the string of the number which can be used by the model to speak it.

The chunk-type provided for the goal chunk is:

```
(chunk-type count count state)
```

It has a slot to maintain the current count and a slot to hold the current model state. An initial goal chunk which has a state slot value of start is also set initially. As with the demonstration model for this unit, you may use only the **goal** buffer for holding the task information instead of

splitting the representation between the **goal** and **imaginal** buffers. Also, as always, the provided chunk-types and chunks are only a recommended starting point and one is free to use other representations and control mechanisms.

There are two functions provided to run the experiment for the model. The **subitize** function can be called without any parameters to perform one pass through all of the trials in a random order. Because there is no randomness in the timing of the experiment and we have not enabled any variability in the model's actions, it is not necessary to run the model multiple times and average the results to assess the model's performance. The other function is called **subitize-trial** and can be used to run a single trial. It takes one parameter, which is the number of items to display, and it will run the model through that single trial and return a list of the time of the response and whether or not the answer given was correct.

```
> (subitize-trial 4)
(1.24 T)
```

As with the other models you have worked with so far, this model will be reset before each trial. Thus, you do not need to have the model detect the screen change to know when to transition to the next trial because it will always start the trial with the initial goal chunk. Also, like the sperling task, this experiment starts with the ACT-R trace enabled and runs by default with a real window and in real time. If you would like to make the task complete faster you can disable the trace as described above and change it to use a virtual window and not run in real time as described in the experiment description document for this unit.

References

Sperling, G.A. (1960). The information available in brief visual presentation [Special issue]. *Psychological Monographs, 74* (498).

Jensen, E. M., Reese, E. P., & Reese, T. W. (1950). The subitizing and counting of visually presented fields of dots. *Journal of Psychology, 30*, 363-392.