SG
Scenario Generator
User's Manual and Examples

SG is generously provided by Steve Luck, formerly of
UCSD Event Related Potentials Laboratory
8/23/90

Most Recent Revision: 5/6/93

1. Introduction

SG is used to generate randomized scenario files that can be used with stimulus presentation programs such as VSPRES, ASPRES, or almost any program that uses text files to specify the order of stimuli. The user specifies groups of stimulus events, formatted appropriately for the stimulus presentation program, and SG simply creates a randomly-ordered sequence of the event groups, randomizing parameters such as inter-stimulus interval, stimulus position, and stimulus identity. Most of the work lies in specifying the groups of stimulus events, so the user must be familiar with the format used by the stimulus presentation programs before using SG. The examples used in this document assume that the stimuli will be presented with VAPP, VSPRES or SBVSPRES which are documented in the VSPRES Usage and Reference Manual.

A relevant example is worth several reams of documentation, so a set of annotated SG examples can be found at the end of this document.

2. Usage

SG requires an event descriptor file, which contains descriptions of event groups and certain optional parameters. The event descriptor file can be created with any ASCII text editor (e.g., VI). The output of SG is an ASCII scenario file that can be used directly by the stimulus presentation program. SG is invoked by typing:

sg EventFile [ScenarioFile] [-p "formatstring"] [-o ofile]

where "EventFile" is the event descriptor file and "ScenarioFile" is the name of the output file (if no scenario filename is specified, the default filename is "sl"). The "-p" and "-o" options will be described below.

3. Event Descriptor Files

The event descriptor file contains information about 2 types of things:
1) Groups of stimulus events; and 2) Groups of constants and random variables that allow SG to create randomized ISIs, positions, and text strings. This information is organized into blocks that

begin with the word “begin” and end with the word “end”. The “end” must be on a line by itself. There will typically be many blocks of event groups but only one block of each type of randomized value.

In most experiments, scenarios consist of several types of trials, each of which occurs several times within each scenario. For example, each scenario in an oddball experiment might consist of 10 occurrences of the letter “A” and 90 occurrences of the letter “B”, presented in random order. In a more complicated experiment, however, the trials within a scenario might themselves consist of multiple stimuli (e.g., a warning stimulus followed by a target), and a scenario would consist of a randomized set of these multiple-stimulus trials. SG allows the user to specify several “event groups”, each of which consists of several lines of text representing the set of events that constitute one trial type. The user also specifies the number of occurrences of each event group, which controls both the probabilities of the various trial types and the total number of trials in each scenario.

Once SG knows what the event groups are and how many of each should be used, it creates a randomized sequence in which each event group appears the appropriate number of times. For each event group in that sequence, SG finds the appropriate event block in the event descriptor file and literally copies the text contained in that block into the scenario file. SG does not make sure that the text in the block is a set of legitimate commands for whatever stimulus presentation program is being used, and it doesn't matter how the text is formatted; the text is simply copied. There is one respect in which the text is not directly copied, however: SG looks for the names of constants and random variables in the text, and replaces them with their values as described below.

3.1 Event Block Format

An event block looks something like this:

begin event 20 “Left Arrow Followed By Left Standard Bar”
800 200 1 text=“<“
2000 200 2 pgi=l”bar.pgi” xoff=-100
end

The first argument after “begin” indicates that this is an event block. The second argument indicates that there should be 20 occurrences of this event type in each scenario. The third argument, which is optional, is a string naming the event group. As far as SG is concerned, any text can occur within an event block, and the text will be copied directly into the scenario file whenever that event type is supposed to occur. As a result, while the order of event groups is randomized, the order of stimuli within an event group is not. For example, consider an event descriptor file containing the following two event blocks:

begin event 3 “A & B”
a
b

end

begin event 2 “C & D”
c
d
end

These event blocks would result. in a scenario that looks something like this (depending upon the exact randomization of event blocks):

c
d
a
b
a
b
c
d
a
b

Note that there are 3 sets of a’s followed by b's, and 2 sets of c’s followed by d's. Although the order of the 2-letter groups is randomized, b always follows a and d always follows c.

There are times when it is desirable to have the number of occurrences of an event group vary randomly across scenario files. This can be achieved by specifying a range instead of the precise number of events. For example, if between 10 and 20 occurrences of an event are desired, the first line of the event block would look like this:

begin event 10-20

SG does not allow the user to specify sequential dependencies for the event groups. There are three ways to control stimulus sequences, however. First, some types of sequential dependencies can be created via the order of events specified within an event block, as in the example above. Second, there is an alternate event block form that allows specification of the precise location of each occurrence of the event group within the scenario. In this alternate form, a list of sequential positions is specified in place of the number of stimuli, and this list is preceded by a “#” sign. For example, the following event group would occur as the first, third, and fifth event group (note that positions start at 0, not 1):

begin event #0,2,4 “A at 1, 3 and 5”
1000 500 1 text=“A”
end

The third way in which sequential dependencies can be created is by use of an "order-of-events" file. When an order-of-events file is used, the order of event groups is specified in that file rather than being generated at random by SG. The order-of-events file is specified on the SG command line with the "-o" option (e.g., **sg edf sl -o orderfile**), and contains a list of numbers corresponding to event blocks. Each event block in the event descriptor file (excluding the "first" and "last" blocks, which are described next) is assigned a number indicating its ordinal position in the event descriptor file (beginning with zero), and the numbers in the order-of-events file correspond to these event block numbers. For example, if the order-of-events file contains the numbers 3, 4, 1, 2 (in that order), then the first event in the scenario file would be the fourth event block from the event descriptor file; the second in the scenario file would be the fifth from the event descriptor file; the third would be the second; and the fourth would be the third. This file overrides the number of occurrences of each event type and any specific sequential positions specified in the event descriptor file. Note that the use of an order-of-events file allows maximum flexibility in ordering the events; however, one must typically write a special-purpose program to create the order-of-events file, so this flexibility is obtained at the cost of additional effort.

3.2 The "First" and "Last" Blocks

There are occasions when one may wish to begin or end a scenario with an event group that doesn't occur at other times in the scenario. For example, a set of instructions may be presented at the beginning of each scenario or some question about the stimuli may be presented at the end of each scenario. Two special event groups, the "first" and "last" event groups, can be used for this purpose (these are actually special cases of the alternate event block form described above). These blocks are just like event blocks, except that they begin **begin first** and **begin last** (with no additional arguments). Here are some examples:

```
begin first
3000 3000 0    text="Attend Left For This Run"
2000 2000 0    text="" #This event results in a 2-sec blank period
end

begin last
5000 5000 0    text="Was there a 'G' in the previous run?" +
               text="Press left for yes" yoff=50 +
               text="Press right for no" yoff=100
end
```

3.3 Specifying Constants

In many experiments, parameters such as the inter-stimulus interval and stimulus duration are the same in different event groups, and are frequently modified during the development of an experiment. SG allows the user to specify a set of constants at the beginning of an event descriptor file that facilitate these modifications and enhance the readability of the event blocks. Each constant has a name and a value, both of which are strings. The name must begin with a

letter and contain only letters and digits, but the value can be any string (the value must be enclosed in double quotes if it contains non-alphanumeric characters). SG is not case-sensitive, so the name "isi3" will be treated the same as the name "ISI3". When SG encounters the name of a constant in an event block, the constant's name is replaced with its value.

The constants are specified in a "constant block", which usually appears at the beginning of an event descriptor file (the word "constant" can be shortened to "const" for all of you Pascal lovers). The following is an example of the constant block format:

begin constant
isi1 1000
dur1 500
isi2 2000
dur2 100
blue "color=2"
red "color=3"
offset "xoff=-20 yoff=40"
end

If the above constant block were specified, the following event block:

begin event 10 "Red A, Blue B"
isi1 dur110 text="A" red offset
isi2 dur2 20 text="B" blue offset
end

would be translated into the scenario file as:

1000 500 10 text="A" color=3 xoff=-20 yoff=40
2000 100 10 text="B" color=2 xoff=-20 yoff=40

When a constant appears in an event block, the character that precedes the name and the character that follows the name must both be non-alphanumeric. For example, if you have a constant named "ap" with a value of "XYXYXY", the string **text="An apple happened aptly to fall ap ap0 0ap"** in an event block would be translated as **text="An apple happened aptly to fall XYXYXY apO Oap"** rather than **text="An XYXYXYple hXYXYXYpened XYXYXYtly to fall XYXYXY XYXYXYO OXYXYXY".**

3.4 The Random Variable Block

Random variables are somewhat similar to constants, in that SG replaces the name of the variable with its value when copying from an event block to the output scenario file. There are some important differences, however. First, the value of a random variable is, as one might expect, random rather than constant. Second, the user doesn't assign arbitrary names to the random

variables. Instead, they are designated by a "%" character followed by a number that specifies the particular random variable (e.g., "%3" designates random variable 3).

A random variable block looks something like this:

begin randvar
1500 2000
200 599
600 1000
end

The first line indicates that this is a random variable block. The next three lines define 3 random variables. The first ranges from 1500 to 2000, the second from 200 to 599, and the third from 600 to 1000. Whenever a random variable is desired, the user should place the string "%n" into the text of the event block, replacing the "n" with an integer indicating which of the random variables should be used (n is between 0 and N-1, where N is the number of random variables defined by the user).

For example, let's randomize the SOA between the cue and target and the interval between trials in a Posner cueing paradigm. Since the ISI represents the interval following a stimulus in the VSPRES program, the ISI for the cue stimulus will determine the SOA, while the ISI for the target stimulus will determine the interval between the target and the next cue. Therefore, the event descriptor file would look something like this:

begin randvar
600 1000
2500 3000
end

begin event 20 "Left Arrow Followed By Left Bar"
%0 200 1 text="<"
%1 200 3 pgi="bar.pgi" xoff=+100
end

begin event 5 "Left Arrow Followed By Right Bar"
%0 200 1 text="<"
%1 200 4 pgi="bar.pgi" xoff=+100
end

begin event 5 "Right Arrow Followed By Left Bar"
%0 200 2 text=">"
%1 200 3 pgi="bar.pgi" xoff=-100
end

begin event 20 "Right Arrow Followed By Right Bar"

%0 200 2 text=“>“
%1 200 4 pgi=“bar.pgi” xoff=+100
end

In this example, each “%0” would be replaced by an integer between 600 and 1000, and each “%l would be replaced by an integer between 2500 and 3000.

One very important detail about random variables is that new random values are created for each occurrence of each event group, not for each occurrence of a random variable specifier. As a result, two instances of “%0” within the same event block will produce two instances of the same value within each occurrence of an event group, but different values between occurrences of the event group. This feature doesn't exist merely to aggravate the user: it's actually very useful in certain situations. Consider a case where the user wants the cue to remain on the screen for the entire SOA. In order to accomplish this, the ISI and stimulus duration must be the same value for each cue, but this value must vary between cues. This can be done with event blocks that look like this:

begin event 20 “Left Arrow Followed By Right Bar”
%0 %0 2 text=“>“
%1 200 4 pgi=“bar.pgi” xoff=+100
end

By default, the values produced by random variables are integers, but fractional decimal values can be produced if a scaling value is placed at the end of the begin randvar line. When SG calculates a random variable, the variable is divided by the scaling value before being printed in the output scenario file. For example, if decimal values between -2 and +2 are desired, the following random variable block would be defined:

begin randvar 1000
-2000 +2000
end

The initial random values in this example would be between -2000 and +2000, but these would be divided by the scaling value of 1000 to produce final values between -2 and +2. When a scaling value is specified, three digits of precision are used to the right of the decimal point (e.g., 321.489 or 0.002), so a scaling value of 1000 will usually be optimal.

3.5 The Random Position Block

In some experiments, the positions of stimuli vary randomly over trials one could accomplish this with random variables, using events such as:

%0 200 1 pgi=“bar.pgi” xoff=%1 yoff=%2

When several randomly positioned stimuli must be placed on the screen at one time, however, truly randomized positions will often result in stimuli that overlap each other. Similarly, the stimuli might occur at the fixation point, which is usually undesirable. Therefore, SG has a random position facility that allows the user to specify the minimum distance between the stimulus positions and their minimum distance from the center of the screen. For each random position, the user specifies the borders of a rectangle surrounding the possible values for that position. The order is left edge, top, right edge, bottom. At the beginning of each occurrence of each event group, a set of random positions is calculated. Whenever the string "*n" is encountered within the event block, SG replaces this string with **xoff=xxx yoff=yyy** where xxx and yyy are coordinates within the rectangle specified for position n. Like random variables, a new set of random positions is calculated for each occurrence of an event block, so multiple references to the same random position specifier within an event block will cause multiple instances of the same position. This can be useful when several different items are plotted simultaneously at each position.

A random position block looks something like this:

begin randpos 40 60
-320 -175 0 +174
0 -175 319 +174
end

The first argument after the "begin" indicates that this is a random position block. The second argument represents the minimum distance between positions in units of pixels. The third argument represents the minimum distance between the positions and the center of the screen, again in units of pixels. A fourth argument may optionally be present to specify a scaling value; as in the random variable blocks, the scaling value is used as a divisor for the randomized values and allows non-integer positions to be used. The two lines that form the main body of the random position block define the borders of two random positions, in this case the left and right halves of the VSPRES display.

If one defines a large enough set of positions and/or a very large amount of space between positions, it is possible for SG to be unable to find a random set of positions that satisfies these constraints. If this occurs, SG will print an error message and exit.

One must be careful not to interpret these positions as necessarily representing the middle of an image, but rather as offsets to the start of the image. If the image is not explicitly centered at the current coordinate, then the images may appear in unintended locations and may overlap each other. For example, if a VSPRES pgi file draws a rectangle with the "orect" or "frect" routines, the upper left corner of the rectangle will be plotted at the coordinates specified by xoff and yoff. The "rmoveto" command can be used to mitigate this problem. For example, the following pgi commands will plot a filled green rectangle 20 pixels wide and 60 pixels long centered at the current coordinate:

setcol 10

rmoveto -10 -30
frect 20 60

In some cases, it is desirable to have several independent sets of random positions. For example, let's say that on some trials there must one item at position (0,-100) and four items randomly distributed over the rest of the screen; on. the other trials, there must be one item at position (0,+100) and another four randomly distributed. A simple approach would be to define 6 positions like this:

begin randpos 40 60
0 -100 0 -100
0 +100 0 +100
-300 -150 +300 +150
-300 -150 +300 +150
-300 -150 +300 +150
-300 -150 +300 +150
end

The problem is that when position (0,-100) is used, the randomly varying positions cannot fall within 40 pixels of (0,+100); likewise, when (0,+100) is used, positions 2-5 cannot fall within 40 pixels of (0,-100). Now this may seem trivial, but there are more complicated extensions of this problem that are more troublesome. In order to get around this limitation, SG allows the specification of up to ten independent random position groups. Each **begin randpos** block encountered by SG is used to create a new set of random positions, and they are numbered by their order of occurrence in the event descriptor file. The first random position block is numbered 0, and is the default block. If a different random position group is needed within an event block, it is specified as "*g.n", where g is replaced by a number between 0 and 9 representing the desired group and n is replaced by a number indicating which position within the group is needed. The following example would print a "#" at position 2 from group 6:

1000 500 2 text="#" *6.2

The default format used by SG to specify a position corresponds to the format required by the VSPRES program, namely **xoff=xxx yoff=yyy**. However, not all stimulus presentation programs utilize this format, so SG provides two alternative means of specifying positions. First, the x- and y-values can be printed directly if the random position is specified as ***#.x** or ***#.y**, where **#** represents the position number. For example, the x-value for random position 3 would be specified as ***3.x**. SG also has a command line option that allows the user to specify the format of the random position output. This option is -p "formatstring", where "formatstring" is a string describing the required format. This string must contain the substrings **[x]** and **[y],** which will be replaced by the randomized x- and y-values. For example, if **-p "position=([x], [y])"** were specified on the command line, a random position with an x-value of 100 and a y-value of 150 would be printed as position=(100,150) in the output scenario file.

3.6 The Random Text Block

In addition to randomizing numeric values, SG can also randomize text. This is useful in a number of situations. As an example, let's consider a case in which an array of four letters is displayed on the screen at locations (-75,0), (-25,0), (+25,0), and (+75,0). The letters can be E, F, L, or T, varying randomly across positions and across trials. Instead of wasting time typing in all 256 possible event types, one could have SG randomly select items from the set of letters and place them in "text=" statements. Each line of a random text block contains a group of text strings, and when SG encounters a "$n" in an event block, the "$n" is replaced with a randomly chosen string from the nth text group. For the previous example, the event descriptor file would look something like this:

```
begin randtext
E F L T
E F L T
E F L T
E F L T
end

begin event 100
500 200 1       text=$0 xoff=-75 +
                text=$1 xoff=-25 +
                text=$2 xoff=+25 +
                text=$3 xoff=+75
end
```

Each line in the random text block represents a different text group and the separate strings in each text group are separated by spaces or tabs. The standard ERPSS argument conventions are used throughout SG, which means that: 1) Strings that contain spaces or tabs must be enclosed in double quotes; 2) A backlash at the end of a line will cause the following line to be included in the virtual line; and 3) There is a maximum of 255 characters per virtual line, and therefore per text group. As an example, the following event descriptor file will produce a scenario in which one of three phrases is printed on each trial:

```
begin randtext
"Starkle, Starkle, Little Twink." \
"Who the hell I am I think." \
"But occifer, I'm not under the alfluence of incohol!"
end

begin event 100 "Random phrase"
1000 750 1      text="$0"
end
```

The random text block is useful for a variety of tasks other than randomizing the printing of text. It can also be used to randomize arbitrary graphic images by specifying pgi and cri filenames, to

choose from a small set of stimulus locations (see example 5.4), to select randomly from a set of digitized sound files for ASPRES, to randomize stimulus colors, etc.

As with the other randomized values, one item from each text group is selected for each occurrence of an event group, so multiple references to a particular random text group within an event block will produce the same string within occurrences of that event block, but different strings between occurrences. There are two limitations on the text groups: 1) The first character on a line can not be a "#" (this would cause the line to be interpreted as a comment); and 2) The first word on a line can not be "end" (this would be interpreted as the end of the block).

There are occasions when it is desirable to sample randomly from a group of strings without replacement. For example, in the example where the letters E, F, L, and T are printed, one might want to exclude cases in which a letter is repeated in the same stimulus array (see Example 5.5). This can be accomplished by placing the string **noreplace** as the last argument on the **begin randtext** line. At the beginning of each event group, SG chooses an index representing a string for each random text group. When the "noreplace" option is specified, SG simply ensures that no two indices are the same. It doesn't really matter to SG whether or not all of the random text groups are the same, but usually they will be. There are two things that do matter, however. First, there must be the same number of items in each random text group. Second, there must be at least as many groups as there are items per group. If these two conditions are not met, then the algorithm for selecting without replacement can't work, and SG will print an error message. These constraints only apply when the "noreplace" option has been specified. When "noreplace" is not desired, the user may specify "replace" on the "begin randtext" line or specify nothing at all.

The random text facilities of SG may not be sufficient for some language experiments, especially when sentences are presented on word at a time. When complicated text randomization is required, the ADDWORDS program may be used as an adjunct to SG (see the ADDWORDS user's manual for details).

3.7 Numeric Functions

In some experiments, the value of one parameter will depend on the value of some other parameter. For example, in a Posner cueing experiment, one may want a variable interval between a cue and the subsequent target, but a constant interval between cues. If we call the cue-target interval "A" and the cue-cue interval "B", then the target-cue interval will be B-A. SG allows the user to specify numeric functions that allow parameters such as these to be computed. Each function has a name (using the same rules as constant names), a parameter list, and a formula. For example, a function named **diff** could be defined to calculate the target-cue interval as follows:

```
begin function
diff(A,B)       B-A
end
```

If the above constant block were specified, the following event block:

begin event 10 “Cue->Target”
300 50 10 text=“Cue”
diff(300,1000) 500 20 text=“Target”
end

would be translated into the scenario file as:

300 50 10 text=“Cue”
700 500 20 text=“Target”

There should be only one function block in an event descriptor file, but up to 50 separate functions can be defined in a function block. Each function can have up to 26 parameters that are passed from the event blocks. The parameters must be named A, B, C, D, etc., in that order. The function's formula uses basically the same form as numeric expressions in the C programming language. One important difference is that, unless parentheses are used to indicate the order of evaluation, the terms are evaluated from left to right (e.g., multiplication and addition have the same precedence). As a result, the expression “A+B*C” is evaluated as “(A+B)*C” rather than “A+(B*C)”. This may change in future versions of SG, so the safest approach is to use parentheses to indicate the order of evaluation.

An important element of the C language that is emulated by SG expressions is the “conditional operator”, which allows a single expression to contain if-then relationships. The syntax of this operator is “expression1 ? expression2 : expression3”. If expression1 is true (i.e., has a non-zero value), then the result is expression2; if expression1 is false (i.e., equals zero), then the result is expression3. For example, the statement “(A > B) ? 10 : 20” would have a value of 10 if A were larger than B, and a value of 20 if A were smaller than or equal to B. All of the usual relational operators (e.g., <, >, ==, !=, etc.) are available, and the result of a relational expression is 1 for true and 0 for false (e.g., the result of (2>1) is 1 and the result of (2<1) is 0). Thus relational operators can be used just like other operators in forming expressions.

SG also supports the standard bitwise operators (&, | , etc.). All values in expressions are normally treated as double-precision floating point numbers, but these numbers are truncated to long integers (32-bit. integers) when bitwise operators or the '%' operator are used. For example, the expression “3.7 | 2” is treated as “3 | 2”, which has a value of 2.0 (‘ | ‘ is the bitwise OR operator).

SG functions may also refer to common numerical functions such as square root, sine, cosine, etc. A list of the supported functions and operators can be found in section 5.

When a function is called during the translation of an event block, it must be passed the appropriate number of parameters. For example, an error would result if the “diff” function described above were called as “diff(1000)” or as “diff(1000,300,100)”. The parameters can be explicit numeric values, as in these examples, constants with numeric values, or random

variables with numeric values. For example, the “diff” function could be called as “diff(%2,isi)” if the random variable “%2” and the constant “isi” were defined.

One must be careful to ensure that all of the parameters sent to functions are numeric. For example, if the constant “isi” is defined as “1000”, then it may safely be sent as a parameter, but if it is defined as “xxx”, then it is not a valid parameter. Similarly, if random positions are used as parameters, they must be specified as the actual x and y coordinates, not as the string “xoff=xvalue yoff=yvalue”, which is the default. For example, to derive the difference between the y value and the x value for random variable 4, the “diff” function would be specified as “diff(*4.x,*4.y)” rather than “diff(*4)”.

By default the result of a function is printed with two digits to the right of the decimal point. The result of “diff(300,1000)” would therefore actually be “700.00” rather than “700”. The number of digits to the right of the decimal point (called the “precision”) can be specified by appending a colon and the number of digits to the parameter list in the function description. For example, to force the “diff” function to return “700” rather than “700.00”, it would be declared as:

begin function
diff(A,B):0 B-A
end

The “:0” in this declaration specifies that the result of the function should be printed with 0 digits to the right of the decimal point (in which case the decimal point is omitted).

3.8 Some Details (Escape Codes; Character Case; Comments)

SG is not case-sensitive, so “begin”, “BEGIN”, and “BeGIn” are treated equally. However, any text within an event block is copied literally to the output scenario file, so the case of the characters within an event block will be propagated to the stimulus presentation program, which may or may not be case-sensitive.

Blank lines are ignored by SG unless they occur within an event block, and should be used liberally to enhance readability.

Any time the special characters “%”, “*”, and “$” are found in the text of an event block, they are replaced by random values. Of course, there are times when one might want these characters to be interpreted literally, as in the following event:

%0 500 1 text=“The price is $10.00”

SG would interpret the “$” as a request for a random text string. In order to overcome this problem, the “^” symbol can be used as an escape character. Whenever a “%”, “*”, or “$” is preceded by a “^”, the symbol is interpreted literally. For example, the above event description should appear in the event descriptor file as:

%0 500 1 text="The price is ^$10.00"

A leading "^" character can also be used to copy a constant's name literally, rather than replacing it with it's value. For example, if a constant named "blank" has been defined, the following event block would cause the word "blank" to be printed instead of it's value:

```
begin event 10 "Print the word 'blank'"
1000 500 12    text="^blank"
end
```

If the "^" did not precede the word "blank", the value of the constant "blank" would appear in the output scenario file rather than the word "blank".

It is possible to override the escape mechanism by using two "^" characters (e.g., **text="The SG escape character is '^^'"**). The VSPRES escape character "\" is not given a special meaning by SG and is copied literally into the scenario file.

In VSPRES, the "#" character indicates the beginning of a comment field and the rest of the line is ignored. A limited version of this feature is available in SG. Any line that begins with a "#" is ignored, and "#" characters within the text of an event block are copied literally into the scenario file and can therefore be used to begin comment fields. However, #-initiated comments are not allowed anywhere else. The following are examples of properly-used comment fields:

```
# Define random positions 40 pixels apart and 60 pixels
# from the center.
begin randpos 40 60
# Position 0 is the left half of the screen.
-320 -175 0 +174
# Position 1 is the left half of the screen.
0 -175 319 +174
end

# First event is the letter E.
begin event 20
1000 500 10 text="E" *0 # This letter will be presented
                        # in the left half of the screen.
end
```

The following are examples of illegal comment fields:

```
begin randpos 40 60 # Not allowed on "begin" line.
-320 -175 0 +174 # We can't have a comment here either.
0 -175 319 +174
end # or here.
```

begin event 20 # Not on a begin line.
1000 500 10 text="E" *0 # But anywhere is OK
 # within an event block.
end

4. Event Descriptor Command Summary

In the following summary, words in boldface are meant literally whereas words in regular typeface represent the name of some parameter which is to be replaced with the value for that parameter. Parameters enclosed in **square** brackets are optional.

begin event [#]NumberOfEvents[-NumberOfEvents] ["EventName"]
Text of event group
- ●
- ●
- ●

end

begin first
Text of first event group
- ●
- ●
- ●

end

begin last
Text of last event group
- ●
- ●
- ●

end

begin const[ant]
ConstantName0 "ConstantValue0"
ConstantName1 "ConstantValue1"
ConstantName2 "ConstantValue2"
- ●
- ●
- ●

end

begin randvar [ScalingValue]
LowerBound0 UpperBound0
LowerBound1 UpperBound1
LowerBound2 UpperBound2
- ●
- ●
- ●

end

begin randpos SpaceBetweenPositions SpaceFromCenter [ScalingValue]
Left0 Top0 Right0 Bottom0
Left1 Top1 Right1 Bottom1
Left2 Top2 Right2 Bottom2
●
●
●
end

begin function
FunctionName0(A,B,C,...)[:precision] FunctionFormula0
FunctionName1 (A,B,C,...)[:precision] FunctionFormula1
FunctionName2 (A,B,C,...)[:precision] FunctionFormula2
●
●
●
end

begin randtext [noreplace] [replace]
TextGroup0
TextGroup1
TextGroup2
●
●
●
end

5. Operators and Built-in Functions

The following operators can be used in SG:

OPERATOR	DESCRIPTION	EXAMPLE
+	Addition of 2 numbers	1+3=4
-	Subtraction of 2 numbers	3-1=2
*	Multiplication of 2 numbers	5*2=10
/	Division of 2 numbers	5/2=2.5
%	Modulus of 2 numbers	5%2=1
+	Unary Addition	+(3)=3
-	Unary Subtraction	-(3)=-3
?/:	Conditional operator	(2>1)?5:10=5

>	Greater than	3>2=True
>=	Greater than or equal to	2>=2=True
<	Less than	3<2=False
<=	Less than or equal to	2<=2=True
==	Equal to	2==2=True
&&	Logical AND	(1<2)&&(5>3)=True
||	Logical OR	(1>2)||(5>3)=True
!	Logical negation	!(1>2)=True
&	Bitwise AND	7&4=4
|	Bitwise OR	6|3=7
^	Bitwise exclusive OR	6^3=5

The following functions can be used within SG function formulas:

FUNCTION	DESCRIPTION	EXAMPLE
sqrt(x)	Square root	sqrt(100)=10
sin(x)	sine (radians)	sin(l)=0.841
cos(x)	Cosine (radians)	cos(l)=0.540
tan(x)	Tangent (radians)	tan(l)=1.557
log(x)	Natural Logarithm	log(10)=2.302
log10(x)	Base 10 Logarithm	log10(10)=l
exp(x)	Power of e	exp(2)=7.389
rnd(x)	Random number (range: 0-x)	rnd(2)=1.18...
trunc(x)	x truncated to an integer	trunc(2.8)=2
round(x)	x rounded to an integer	round(2.8)=3

These functions all require single parameter. For example, the following function would return 2 times the square root of the sum of parameters A and B:

begin function
testfunc(A,B) “2 * sqrt(A + B)”
end

Note that the formula for this function is enclosed in double quotes because it contains spaces.

6. Error Messages

Most error messages are self-explanatory, and are the result of making mistakes in the event descriptor file or exceeding program limitations. The limits are 500 event blocks, 100 constants, 200 random variables, 200 random text variables, and 200 random positions. These limits can be changed by recompiling SG (ask any programmer to do it). There are a few messages that need explanation, and here they are:

Fatal Error: Couldn't satisfy position constraints. This message occurs when SG is unable to generate a set of random positions that are then separated by the user-defined minimum distance. This can be avoided by specifying small spaces between stimulus locations.

Fatal Error: Illegal random variable n. The user has specified a random variable in an event block that was not defined in the randvar block.

Fatal Error: Illegal random position n. The user has specified a random position in an event block that was not defined in the randpos block.

Fatal Error: Illegal random text group u. The user has specified a random text group in an event block that was not defined in the randtext block.

Catastrophic Error: Prepare To Die. This message indicates that SG is unable to read part of the event descriptor file. This can occur if the event descriptor file has been corrupted or if the user encounters a particular bug in SG that has been difficult to locate and exterminate. If this bug should appear, simply try running the program again once or twice. If it persists, save the event descriptor file and call ORKIN.

"filename" already exists. Overwrite? This message occurs when the output scenario file specified by the user already exists. If the user wants to clobber the old file., s/he should type "y" and press ENTER. If not, s/he should type "n" and press ENTER.

Fatal Error: Not enough memory to malloc n bytes. This means that SG was unable to acquire enough memory for the particular event descriptor file.

5. Examples

All of the files required for these examples can be found in "d:\src\video\vsgen\test". The examples start simple and become more and more complex.

5.1. The Oddball Paradigm

This oddball paradigm is very simple and shows the basic setup of the event blocks. In this paradigm, there is a .8-probable non-target stimulus ("XXXX"), a .1-probable non-target stimulus ("WAIT"), and a .1-probable target stimulus ("PUSH"). The event descriptor file is named "oddball" and contains the following text:

```
begin event 80 "Standard"
1000 500        1       text="XXXX"
end

begin event 10 "Rare Non-Target"
1000 500        2       text="WAIT"
end
```

```
begin event 10 "Target"
1000 500        3        text="PUSH"
end
```

5.2. The Basic "Hillyard Attention Paradigm"

This is a simple example which demonstrates the use of a randomized ISI and different event probabilities. In this paradigm, there are two positions at which stimuli may occur, and two types of stimuli. The positions are (-100,0) and (+100,0); the stimuli are "E" and "F". Each scenario consists of 200 trials: 80 left E's (event code 1), 20 left F's (code 2), 80 right E's (code 3), and 20 right F's (code 4). The ISI is randomized between 200 and 400 msec, and stimulus duration is represented in a constant named "dur" with a value of 100 msec. The event descriptor file is named "hillyard" and contains the following text:

```
begin constant
dur 100
end

begin randvar
300 500
end

begin event 80 "Left Standard"
%0 dur        1        text="E" xoff=-100 mon="Left Standard"
end

begin event 20 "Left Target"
%0 dur        2        text="F" xoff=-100 mon="Left Target"
end

begin event 80 "Right Standard"
%0 dur        3        text="E" xoff=100 mon="Right Standard"
end

begin event 20 "Right Target"
%0 dur        4        text="F" xoff=100 mon="Right Target"
end
```

This experiment is best run with a fixation point, which can be done using SBVSPRES and the fixation crosshairs in the file named "plus.cri." The program is invoked as:

```
SBVSPRES h1.scn plus.cri
```

Note that the ISI is randomized by placing the string "%0" where the ISI would normally be found. Note also that stimulus probabilities are controlled via the number of instances of each event group.

5.3. Ron Mangun's Peripheral Cueing Paradigm

This example demonstrates the use of multiple random variables, multiple instances of a random variable within an event block, and event blocks that consist of several stimulus events. In this paradigm, each "trial" consists of a cue on one side and a target bar that usually occurs on the cued side (80% validly cued targets) but sometimes occurs on the opposite side (20% invalidly cued targets). The interval between cue onset and target onset (the "stimulus onset asynchrony" or SOA) can be short (200-600 msec) or long (600-1000 msec). The interval between trials varies from 1250 to 1750 msec. The subject's task is to make a simple reaction time response to the target, independent of its location. This particular paradigm uses a "peripheral cue," an apparent motion near the target location. Four dots form a rectangle around each of the two target locations, and apparent motion is produced by moving the dots on one side closer to the horizontal meridian for 100 msec and then back to their original positions.

The event descriptor file for this example is named "poscue". It requires the presence of three pgi files. "DOT1.pgi" prints the rectangle of dots around one position. "DOT2.pgi" prints the smaller rectangle of dots. "BAR.pgi" prints the target stimulus, a white vertical bar. The position of the dots and the bar is controlled by the VSPRES xoff command. Before proceeding further, it is recommended that the reader runs the scenario with VSPRES to see what it looks like. A sample scenario is contained in a file named "pl.scn" in the examples directory. The pgi files and a portion of the event descriptor file are printed here:

DOT1.pgi	**DOT2.pgi**	**BAR.pgi**
rmoveto 0 -30	**rmoveto 0 -25**	**rmoveto -8 -18**
label ". ."	**label ". ."**	**frect 16 36**
rmoveto 0 +60	**rmoveto 0 +50**	
label ". ."	**label ". ."**	

poscue (Event Descriptor File)

```
begin randvar
# Random variable 0 determines the inter-trial interval.
1250 1750
# Random variable 1 determines the short SOA.
# It varies between 100 and 500 msec, but since this is
# timed from the end of the 100 msec apparent motion, the
# actual SOA is 200 - 600 msec.
100 500
# Random variable 2 determines the long SOA.
500 900
end
```

```
begin constant
# 'CueTime' is the duration/isi of the cue
CueTime "100 100"
* 'TargTime' is the duration/isi of the target
TargTime "200 200"

#The first event puts the fixation point on the screen.
begin first
3000 3000     0       cri="plus.cri" nser
end

begin event   16      "Left Blink, Left Bar, Short SOA"
%0 %0         0       pgi="DOT1.pgi" xoff=-150  mon="Background" +
                      pgi="DOT1.pgi" xoff=+150
CueTime       1       pgi="DOT2.pgi" xoff=-150  mon="Left Blink" +
                      pgi="DOT1.pgi" xoff=+150
%1 %1         0       pgi="DOT1.pgi" xoff=-150  mon="Background" +
                      pgi="DOT1.pgi" xoff=+150
TargTime      2       pgi="DOT1.pgi" xoff=-150  mon="Left Bar" +
                      pgi="DOT1.pgi" xoff=+150  +
                      pgi="BAR.pgi" xoff=-150
end

begin event   16      "Right Blink, Right Bar, Short SOA"
%0 %0         0       pgi="DOT1.pgi" xoff=-150 mon="Background" +
                      pgi="DOT1.pgi" xoff=+150
CueTime       3       pgi="DOT1.pgi" xoff=-150 mon="Right Blink" +
                      pgi="DOT2.pgi" xoff=+150
%1 %1         0       pgi="DOT1.pgi" xoff=-150 mon="Background" +
                      pgi="DOT1.pgi" xoff=+150
TargTime      4       pgi="DOT1.pgi" xoff=-150 mon="Right Bar" +
                      pgi="DOT1.pgi" xoff=+150 +
                      pgi="BAR.pgi" xoff=+150
end
```

One important attribute of this paradigm is that the dot rectangles surrounding the target positions remain on the screen at all times; the screen never goes blank. This is accomplished by using durations that are as long as the corresponding ISIs which in turn is accomplished by using the same random variable for both the ISI and duration parameters. A consequence of this is that VSPRES doesn't have much time to draw the stimuli and there isn't sufficient time to use SBVSPRES to draw a fixation point on each trial. Instead of using SBVSPRES to draw the fixation point, this scenario draws the fixation point on the first event and uses the "nser" option to ensure that VSPRES doesn't erase it.

5.4. The Sperling Stream Attention Paradigm

This experiment demonstrates the use of random text groups for randomizing stimulus identities. On each trial, four letters are presented in a semi-circle on the left or right of a fixation point. The letters are "E" and "F" on non-target trials (p=.80), and one of the four letters is a "T" on target trials (p=.20). Subjects are instructed to attend to one side and press a button whenever they see a "T" on that side. There are 4 event groups: left standard, right standard, left target, and right target.

On standard trials, an "E" or "F" is randomly chosen for each of the four positions. This is accomplished by the use of four random text groups, each having "E" and "F" as possible values. When the command text=$0 occurs within an event block, VSGEN will copy either text="E" or text="F" into the scenario file. On target trials, all four positions are first filled with "E" and "F" characters. Then one of the positions is chosen with a random text group that specifies the offsets for a position. "E" and "F" are printed at that position with color set to 0 to erase whichever character was previously printed at the target location. The "T" is printed. This may seem like a lot of trouble, but it's a lot easier than typing in all of the combinations and permutations of "E", "F", and "T". The event descriptor file is named "sperling" and contains the following text:

```
begin randvar
300 450
end

begin randtext
# These next 4 variables determine whether an E or an F
# is printed at a particular position.
E F
E F
E F
E F
# These variables determine which of the positions
# will contain the target.
"xoff=-72 yoff=-60" "xoff=-100 yoff=-20"
        "xoff=-100 yoff=+20" "xoff=-72 yoff=+60"
"xoff=+72 yoff=-60" "xoff=+100 yoff=-20" \
        "xoff=+100 yoff=+20" "xoff=+72 yoff=+60"
end

begin event 20 "Left Standard"
%0 100          1       text=$0 xoff=-72 yoff=-60 mon="Left Standard" +
                        text=$1 xoff=-100 yoff=-20 +
                        text=$2 xoff=-100 yoff=+20 +
                        text=$3 xoff=-72 yoff=+60
end
```

```
begin event 20          “Right Standard”
%0 100          2       text=$0 xoff=+72 yoff=-60 mon=“Right Standard' +
                        text=$1 xoff=+100 yoff=-20 +
                        text=$2 xoff=+100 yoff=+20 +
                        text=$3 xoff=+72 yoff=+60
end

begin event 20          “Left Target”
%0 100          3       text=$0 xoff=-72 yoff=-60 mon=“Left Target” +
                        text=$1 xoff=-100 yoff=-20 +
                        text=$2 xoff=-100 yoff=+20 +
                        text=$3 xoff=-72 yoff=+60 +
                        text=“E” color=0 $4 +
                        text=“F” color=0 $4 +
                        text=“T” color=7 $4
end

begin event 20          “Right Target”
%0 100          4       text=$0 xoff=+72 yoff=-60 mon=“Right Target” +
                        text=$1 xoff=+100 yoff=-20 +
                        text=$2 xoff=+100 yoff=+20 +
                        text=$3 xoff=+72 yoff=+60 +
                        text=“E” color=0 $5 +
                        text=“F” color=0 $5 +
                        text=“T” color=7 $5
end
```

5.5. Hajo Heinze's Bilateral Probe Paradigm

This experiment demonstrates the “noreplace” option for random text groups and shows how simple sequential relationships can be produced. In this experiment, there are four positions, two on the left side and two on the right. On most trials, a letter (E, F, L, or T) is placed in each of these positions. The subject’s task is to attend to one side and press a button if the two letters on that side are identical. On some trials, a “probe” bar is flashed on one side, covering the positions normally occupied by the letters. Probe trials are always preceded by at least one non-target letter display.

The tricky thing about this design is that we need randomly chosen letters, but no two letters can be the same on non-target trials. This is tantamount to selecting without replacement. The sequential constraint, that probe stimuli must be preceded by at least one non-target letter stimulus, will be accomplished by preceding the probe stimulus by a non-target letter stimulus in the probe event groups. The event descriptor file is named “bilat,” and contains the following text:

```
begin randvar
```

```
300 450
300 450
end

begin randtext noreplace
E F T L
E F T L
E F T L
E F T L
end

begin event 20      "Non-Target"
%0 100      1       text="$0 $1         $2 $3" mon="Non-Target"
end

begin event 20      "Non-Target Followed By Left Probe"
%0 100      1       text="$0 $1         $2 $3" mon="Non-Target"
%1 100      1       pgi="probe.pgi" xoff=-90 mon="Left Probe"
end

begin event 20      "Non-Target Followed By Right Probe"
%0 100      1       text="$0 $1         $2 $3" mon="Non-Target"
%1 100      1       pgi="probe.pgi" xoff=90 mon="Right Probe"
end

begin event 10      "Left Target"
%0 100      1       text="$0 $0         $2 $3" mon="Left Target"
end

begin event 10      "Right Target"
%0 100      1       text="$0 $1         $2 $2" mon="Right Target"
end
```

5.6. The Visual Search Paradigm

This experiment demonstrates the use of random positions. On each trial, 8 colored bars are displayed in random positions. One 50% of trials, four of the bars are blue and horizontal (plotted by the file "BH.pgi") and four are green and vertical (GV.pgi). On 50% of trials, one of the green bars is horizontal (GH.pgi). The green horizontal bar is the target and the subjects press one button on target-present trials and another button on target-absent trials. There are 3 types of trials: target-absent (code 1); left visual field target (code 2); and right visual field target (code 3). We need eight random positions for the non-target bars, outlining the entire screen, and two random positions for the target bar, one for the left side and one for the right. Eight position specifiers with identical borders are needed because the random positions are calculated only once for each occurrence of an event group. If the same random position specifier were used for

all eight bars, then all eight would be placed at the same location. The event descriptor file for this paradigm is named "vsearch," and contains the following text.:

```
begin constant
dur 750
end

begin randvar
1350 1650
end

begin randpos 60 60
# These 8 positions span both sides of the screen.
-200  -125  200  125
-200  -125  200  125
-200  -125  200  125
-200  -125  200  125
-200  -125  200  125
-200  -125  200  125
-200  -125  200  125
-200  -125  200  125
# Position 8 is the left side of the screen.
-200  -125  000  125
# Position 9 is the right side of the screen.
000  -125  200  125
end

begin event 25        "Non-Target"
%0 dur        1       pgi="BH.PGI"       *0 mon="Non-Target" +
                      pgi="BH.PGI"       *1 +
                      pgl="BH.PGI"       *2 +
                      pgi="BH.PGI"       *3 +
                      pgi="GV.PGI"       *4 +
                      pgi="GV.PGI"       *5 +
                      pgi="GV.PGI"       *6 +
                      pgi="GV.PGI"       *7
end

begin event 25        "Left Side Target"
%0 dur        2       pgi="BH.PGI"       *0 mon="Left Target" +
                      pgi="BH.PGI"       *1 +
                      pgi="BH.PGI"       *2 +
                      pgi="BH.PGI"       *3 +
                      pgi="GV.PGI"       *4 +
                      pgi="GV.PGI"       *5 +
```

```
                    pgi="GV.PGI"        *6 +
                    pgi="GH.PGI"        *8
end

begin event 25      "Right Side Target"
%0 dur      3       pgi="BH.PGI"        *0 mon="Right Target" +
                    pgi="BH.PGI"        *1 +
                    pgi="BH.PGI"        *2 +
                    pgi="BH.PGI"        *3 +
                    pgi="GV.PGI"        *4 +
                    pgi="GV.PGI"        *5 +
                    pgi="GV.PGI"        *6 +
                    pgi="GH.PGI"        *9
end
```

ADDWORDS
User's Manual

Steve Luck
UCSD Event Related Potentials Laboratory

1. Introduction

ADDWORDS is used to insert words or other strings into scenario files. It is meant to be used as an adjunct to the SG program, providing more sophisticated word randomization abilities than those provided by SG. However, its basic operation is quite similar to the randomization utilities of SG: ADDWORDS scans a preexisting scenario file for special characters and replaces them with words selected from various lists stored in separate files.

ADDWORDS is especially useful for experiments in which each trial contains a unique word or sentence. SG can be used to create the skeleton of the scenario file, and ADDWORDS can then add the words or sentences to complete the scenario file. If the scenario file needs to be modified, SG's event descriptor file can be modified, a new skeleton scenario created, and the words/sentences re-inserted with ADDWORDS. The alternatives are to edit the scenario file manually, which can be very laborious, or to use SG's random text utility, which is limited in scope.

2. Usage

ADDWORDS is invoked by typing:

addwords InputFile Outputfile [-s[e]] WordFile0 [WordFile1 ...]

where 'InputFile' is the skeleton scenario file, OutputFile is the name of the output file, and WordFile is a file containing a list of words that will be added to the input file to create the output file. Up to 10 separate word files can be used. There are two options that determine the process by which words are selected from the word files; these will be discussed later.

ADDWORDS scans through the input file, copying it directly into the out- put file except when it encounters the tilde character (~). The tilde must be followed by an integer between 0 and 9. The tilde and integer are not copied into the output file; rather, they are replaced by a word selected from the word file specified by the integer. For example, if ADDWORDS encounters ~3 in the input file, a word from word file 3 is copied to the output file (word files numbers begin at 0, so word file 3 is actually the fourth word file specified on the command line).

3. Word Selection Procedures

How does ADDWORDS decide which word to copy from the word file to the output file? By default, the words are simply copied in their order of occurrence, such that the first ~3 in the input file is replaced with the first word in word file 3, the second ~3 with the second word, etc. This should be clarified by the following example.

In this example, the skeleton scenario file will be called "skel" and the output file will be called "s1.scn". Two word files will be used, named "words1" and "words2". The experiment is a semantic priming / lexical decision task in which pairs of words are presented, a prime and a target, and the subject must decide if the target is a real word or a pseudoword. The prime and target can be either related or unrelated. In this example, we will not use different event codes to differentiate the different prime/target combinations (this will be discussed later).

The file "skel" contains the following:

```
200     100     1       text=~0
2000    1000    2       text=~1
200     100     1       text=~0
2000    1000    2       text=~1
200     100     1       text=~0
2000    1000    2       text=~1
200     100     1       text=~0
2000    1000    2       text=~1
200     100     1       text=~0
2000    1000    2       text=~1
200     100     1       text=~0
2000    1000    2       text=~1
```

The file "words1" contains the prime words, as follows:

```
doctor
piano
desk
rabbit
grass
apple
```

The file "words2" contains the target words, as follows:

```
nurse
computer
chair
frinkle
wheel
crube
```

If ADDWORDS is invoked as **addwords skel s1.scn words1 words2**, the. file s1.scn would be created and contain the following:

```
200     100     1       text=doctor
2000    1000    2       text=nurse
200     100     1       text=piano
2000    1000    2       text=computer
200     100     1       text=desk
2000    1000    2       text=chair
200     100     1       text=rabbit
2000    1000    2       text=frinkle
200     100     1       text=grass
2000    1000    2       text=wheel
200     100     1       text=apple
2000    1000    2       text=crube
```

In this example, the first occurrence of ~0 was replaced with the first word from the "words1" file, the second ~0 with the second word from "words1", etc. The first occurrence of ~1 was likewise replaced with the first word from "words2".

The default mode of operation doesn't randomize the order of the word pairs, but it does retain the pairing of primes and targets across files (e.g., doctor and nurse will always be paired together). There are many experiments in which independently randomized lists of words are appropriate, however, and this can be achieved with the "-s" option. This option causes ADDWORDS to "shuffle" (i.e., randomize) the word lists in each file, such that "doctor" might be paired with "frinkle". There is another option, "-se", which causes the word lists to be shuffled into equivalent orders, thereby maintaining the pairing of words across files. If this option were used in the above example, "doctor" would always be paired with "nurse", but this pair would not necessarily be the first word pair.

The three word selection procedures, default, shuffle, and equivalent shuffle, can be illustrated with a simple example. If there were three word files, each containing the numbers 0 through 9, and a skeleton scenario file containing ten lines of "~0 ~1 ~2", then the three selection procedures would produce something like the following:

Default	Shuffle	Equ Shuffle
0 0 0	6 2 9	8 8 8
1 1 1	4 5 5	0 0 0
2 2 2	1 8 2	2 2 2
3 3 3	2 4 3	3 3 3
4 4 4	8 9 4	9 9 9
5 5 5	9 3 0	7 7 7
6 6 6	7 0 1	5 5 5
7 7 7	3 7 8	4 4 4

```
888     017     666
999     566     111
```

4. Words, Sentences, and other Strings

The above examples have assumed that we want to use simple words, one on each line of the word files. Actually, one can have multiple strings on each line, and each string can contain several words. The word files are parsed with the standard ERPSS "argbuf" routines, which are described elsewhere. Essentially, if you want a string to contain any tabs, spaces, or punctuation, it must be enclosed in quotes. For example, if the word list contained

the doctor

then "the" would be treated as one string and "doctor" as another, whereas if the word list contained

"the doctor"

then "the doctor" would be treated as a single string. ADDWORDS doesn't really care about words, per se; everything said previously about "words" actually pertains to strings, which may or may not be single words. Thus, ADDWORDS can easily accommodate the use of sentences as stimuli (e.g. text="This is the sentence.").

It is substantially more difficult to present sentences when each word of every sentence is presented as a single stimulus (e.g. text="This", text="is", text="the", text="sentence."). The difficulty with this format is that there is nothing to tie the words from a single sentence together and to separate the different sentences from each other. These problems become especially difficult when randomization is required at the inter-sentence level but not at the intra-sentence level or when each word must be associated with a particular event code.

There are several ways to deal with word-by-word sentence presentation, two of which will be discussed here. Let's use an example in which sentences are presented one word at a time, at a constant interval of 500 msec between word onsets, and a 2000 msec blank interval between sentences. Each word will have an event code corresponding to its position within the sentence. One method for creating this type of scenario file assumes that randomization is not required. The word file has pairs of strings, each pair consisting of an event code and a word. Since we will not assume equal sentence lengths, a pair of strings with an event code of 0 and the word "END" will be interposed between sentences. The skeleton scenario file would look like this:

```
500 400 ~0      text="~0"
500 400 ~0      text="~0"
500 400 ~0      text="~0"
500 400 ~0      text="~0"
500 400 ~0      text="~0"
500 400 ~0      text="~0"
```

etc.

and the word file would look like this:

1 This
2 is
3 the
4 first
5 sentence.
0 END
1 The
2 second
3 sentence
4 is
5 a
6 different
7 length.
0 END
1 And
2 this
3 is
4 the
5 third
6 sentence.
0 END
etc.

On each line of the skeleton scenario file, the first ~0 will be replaced with the event code from the corresponding line of the word file, and the second ~0 will be replaced with the corresponding word from the same line. After the scenario file has been created with ADDWORDS, the scenario file will need to be edited in order to remove the lines that say:

500 400 0 text="END"

and replace them with lines that say:

2000 1000 0 text=""

This replacement process might seem tedious, but it can be achieved quickly by using the "map" feature of the VI text editor to create a single-keystroke command that will find and replace the next line. To create this command, enter the VI editor and type:

:map q /END^V^MS2000 1000 0 text=""^V^[

The ^V, ^M, and ^[sequences are meant as control characters (e.g. ctrl-V). The above sequence is the sequence that you would type, but VI will strip off the ^V characters as soon as the following character is typed. Once this mapping has been created, typing the "q" key will cause the line-containing the next occurrence of the string "END" to be replaced with the string

2000 1000 0 text=""

A second method for this example allows inter-sentence order to be randomized while intra-sentence order remains constant. To accomplish this, one word file will be used for the first word in each sentence, another for the second word in each sentence, etc. If the maximum sentence length were eight words, then eight word files would be required. Each sentence would be distributed among the eight word files such that the nth line of the mth file contains the mth word of the nth sentence. In order to accommodate different sentence lengths, the string "XXX" would be used on the nth line of the mth file when the nth sentence contains fewer than m words. In other words, if sentence number five has only six words, then line five in word files seven and eight would contain "XXX". For example, if the word files were named "word1", "word2", etc., then the contents of the files might be:

word1	word2	word3	word4	words	word6	word7	word8
This	is	the	first	line.	xxx	xxx	xxx
This	is	number	two.	xxx	xxx	xxx	xxx
And	this	is	the	third	line.	xxx	xxx
etc.							

The skeleton scenario file would look like this:

```
500   400   1 text=~0
500   400   2 text=~1
500   400   3 text=~2
500   400   4 text=~3
500   400   5 text=~4
500   400   6 text=~5
500   400   7 text=~6
500   400   8 text=~7
2000  1000  0 text=""
500   400   1 text=~0
500   400   2 text=~1
500   400   3 text=~2
500   400   4 text=~3
500   400   5 text=~4
500   400   6 text=~5
500   400   7 text=~6
500   400   8 text=~7
2000  1000  0 text=""
etc.
```

After creating the scenario file with ADDWORDS, the scenario file must be edited to remove the lines with "XXX". This can be facilitated with the map command:

:map q /XXX^V^Mdd^V^M

which will cause the next line containing the string "XXX" to be deleted whenever the "q" key is pressed.

It should now be evident that ADDWORDS can insert any text into a scenario file, and not just simple words. For example, if there is a large set of PGI files that need to be presented in order, ADDWORDS can be used to select the filenames. Or if ASPRES is being used, ADDWORDS can be used to insert the file names for digitized sounds. A word file might also consist only of stimulus durations or event codes (which would have been useful for the initial semantic priming / lexical decision example).