Behavior Research Methods, Instruments, & Computers
2002, 34 (1), 55-68

Using Matlab to generate families of similar Attneave shapes

CHARLES A. COLLIN and PATRICIA A. McMULLEN
Dalhousie University, Halifax, Nova Scotia, Canada

We present a program for Matlab that quickly generates Attneave-style random polygons and families of similar polygons. The function allows a great deal of user control over various aspects of the shape generation process. It also has the ability to detect and eliminate shapes that do not match a variety of user-entered parameters regarding the lengths of the shapes' sides, vertex angles, and topological form. The function eliminates the time-consuming task of generating such shapes by hand and should allow their broader use in behavioral research. The Matlab script function can be downloaded at www.dal.ca/~mcmullen/downloads.html.

A number of domains in behavior research present participants with meaningless two-dimensional (2-D) visual shape stimuli. These include studies on shape recognition, mental rotation, and the nature of mental representation. In many cases, the selection or creation of stimuli in these studies has been quite arbitrary, making assertions about the generalizability of the results difficult. Attneave (1957; Attneave & Arnoult, 1956) long ago proposed several methods for designing novel 2-D shapes in an algorithmic and well-defined manner. These methods have the advantage that they create shape stimuli in prescribed ways, allowing for more precise characterization of the population of shapes from which they are drawn. Attneave also suggested ways to create well-defined *shape families*, which are groups of 2-D shapes having varying degrees of similarity to one another. More informal methods, such as arbitrarily drawing figures by hand, make it hard to quantify the objective visual characteristics of the stimuli and thus make it difficult to create groups of similar shapes in a controlled manner.

The most basic and widely used of Attneave's (1957; Attneave & Arnoult, 1956) methods for generating shapes consisted of the following steps.

1. Randomly place a set of N scattered points (where N is the number of sides of the shape) in a coordinate space, using a table of random numbers and a 100×100 sheet of graph paper.

2. Join the outer points in a convex hull (although Attneave did not use this term himself). The convex hull is a mathematical concept defined as the smallest subset of a set of points that, when joined, will surround all the points and have convex angles at all vertices. To visualize what a convex hull is, imagine the points are nails driven halfway into a board and that elastic has been allowed to contract around the set of nails. Some of the nails will touch the elastic, forming the hull, and others will be inside it. All the angles on the hull will be convex as viewed from outside the shape.

3. Connect the points inside the hull to randomly chosen points on the outside in such a way that no lines cross one another.

An example of this process is shown in Figure 1. By standardizing the number of points (and thus the number of sides) of a group of shapes, Attneave (1957; Attneave & Arnoult, 1956) suggested that one could create shapes of equal complexity. There are some obvious exceptions to this, such as a perfectly square or rectangular shape, which benefit from greater *goodness*, in Garner's (1970) sense of the term, but aside from these accidental cases, Attneave's premise seems sound. Certainly, his techniques allow for better quantification and reproduction of shape stimuli than do more arbitrary methods.

As was mentioned above, another advantage of Attneave's (1957; Attneave & Arnoult, 1956) methods is that they allow for the creation of shape *families*, which are sets of shapes based on the same prototype. These can be useful stimuli in a number of domains, including research on the effects of similarity on mental rotation (Cooper, 1975; Folk & Luce, 1987) and studies of how prototypes and exemplars are processed (e.g., Marsolek, 1996; Posner & Keele, 1968). Attneave suggested a number of methods for creating such families. The most basic of these simply involved creating each family member by moving the vertices of the prototype shape in a random direction by a random distance. Figure 2 shows several examples of Attneave shape families, with the prototype at the top and family members below.

Attneave's shapes have been employed in many research studies (e.g., Bethell-Fox & Shepard, 1988; Cooper, 1975; Cooper & Podgorny, 1976; Cornoldi & Longoni, 1977;

This work was supported by grants from the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche to C.A.C. and from the Human Frontiers Science Program (RG 0161/1999-B, P. McMullen PI) to P.A.M. We thank Erin M. Harley, Ira H. Bernstein, and Jonathan Vaughan for useful comments on an earlier version of the manuscript. We also thank Nikolaus Troje for help in testing the program on IBM-compatible platforms. Correspondence concerning this article should be addressed to C. A. Collin, Psychology Department, Dalhousie University, Halifax, NS, B3H 4J1 Canada (e-mail: ccollin@is.dal.ca).

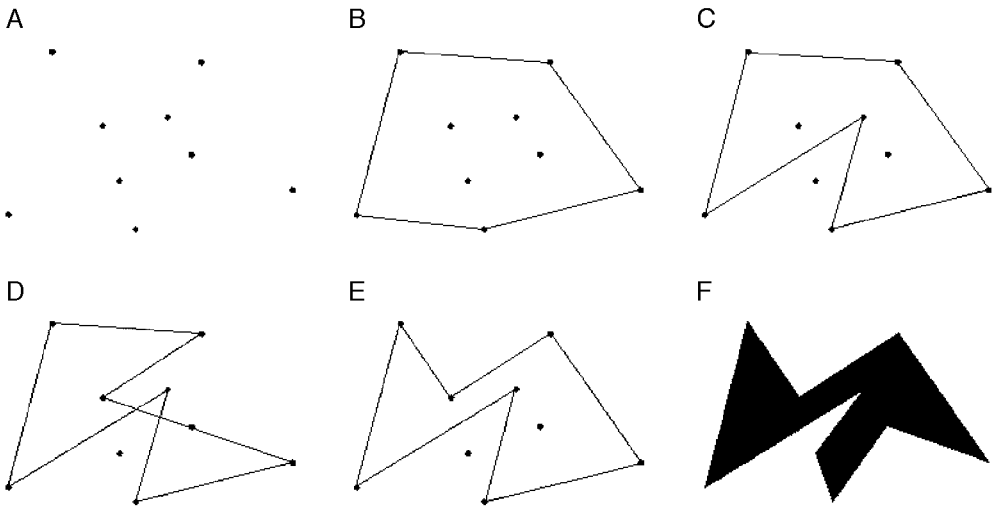

Figure 1. The creation of an Attneave shape. (A) First, a set of random points is created. (B) These points are surrounded by a convex hull. (C) A point inside the hull is joined to the outside. (D) An unacceptable joining of an inside point to the outside due to line crossings. This joining will be rejected, and a new one will be tried. (E) All the points inside the shape are connected to points on the hull in such a way that no lines cross. (F) The final shape.

Folk & Luce, 1987; Klein, 1982; Willis & Dornbush, 1968; Wu, Sun, Wu, & Xu, 1991), but their use is most likely limited by the time-consuming task of creating them. In this paper, we present a method for using Matlab (The Mathworks, Inc., Natick, MA; www.mathworks.com) to rapidly generate families of Attneave-type polygons with a good deal of user control and flexibility. Matlab is a popular mathematics and visualization software extensively

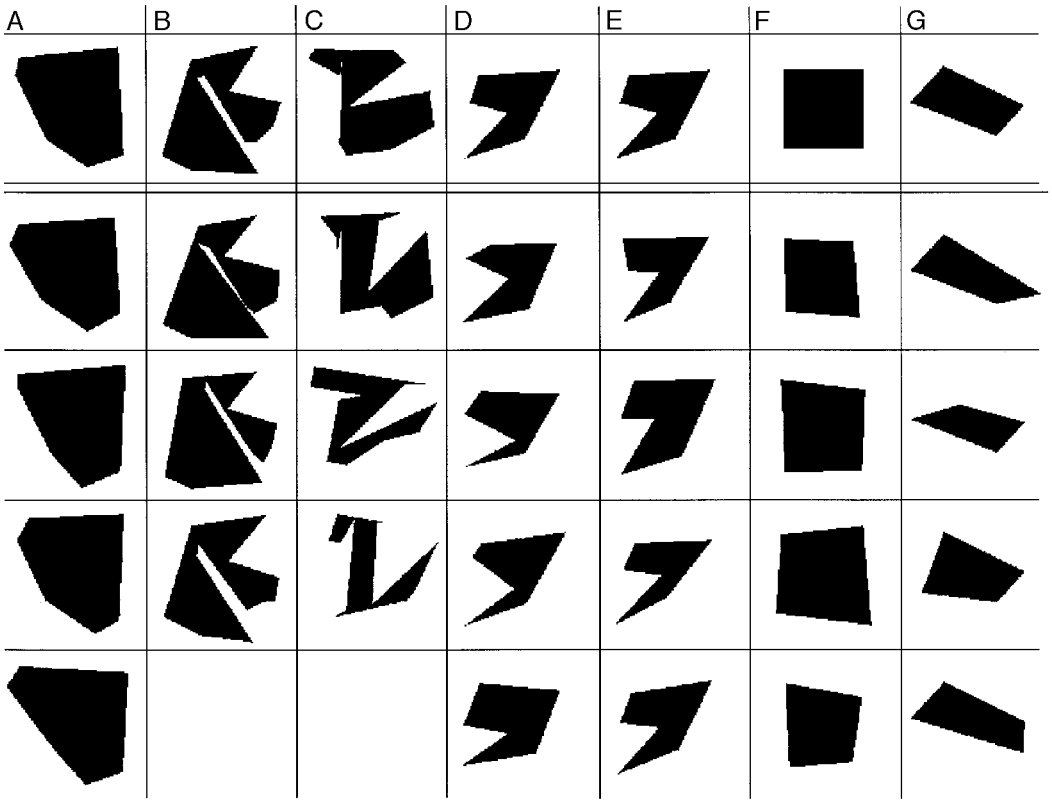

Figure 2. Several examples of shape families output by *ShapeFamily.m*. Prototypes for each family appear at the top. See the text for details.

used by psychophysicists and other behavior researchers to prepare and present stimuli.

ShapeFamily.m is a Matlab function that creates a set of Attneave-type polygons according to a variety of user specifications that will be described in detail below. It is designed for use with Matlab 5.2 on Macintosh or Matlab 6.0 on IBM-compatible computers. The function will likely work with other versions of Matlab and on other operating systems but has not been tested on them. The code for the function is given in the Appendix. It may also be downloaded from our Web page at www.dal.ca/~mcmullen/downloads.html. The downloadable version contains a help file, which may be accessed by calling

```
»help ShapeFamily
```

at the Matlab prompt. The only Matlab toolbox required to run this function is *Image Processing*. In general terms, *ShapeFamily.m* operates by generating a set of random coordinates for a prototype shape and then applying random offsets to these in order to create *family members* of varying similarity to the prototype. Similarity is controlled by varying the average distance a vertex is moved when family members are created. This factor is controlled by the user, as is the number of sides of the prototype and family members and the number of family members generated. Perhaps the most important aspect of the function is that each generated shape can be checked to make sure it satisfies a number of criteria regarding lengths of sides, angles at vertices, and topological integrity. Below, we describe how to use the function in detail and then explain the algorithm in general terms.

Using *ShapeFamily.m*

To use *ShapeFamily.m*, place the function in the Matlab folder (or a folder in Matlab's path). Then, at the Matlab prompt, type a command in the form

```
» [Xf, Yf, Xp, Yp] = ShapeFamily('paramter1',
value1, 'parameter2', value2, . . .);
```

The return values (Xf, Yf, Xp, and Yp) are optional, as are the input parameters. There are 16 possible input parameters, each of which is discussed in detail below. If no input values are provided at the command line, a dialog box will appear where these values may be entered. The return values Xf and Yf are matrices containing the coordinates of the generated shape family. All return values are in the range of 0 to 1. Each row contains the coordinates defining one member of the shape family. For example, one can view member *n* of a family by calling

```
» fill(Xf(n), Yf(n));
```

where *n* is an integer from 1 to the number of family members generated (see the parameter description for *NMembers*, below). Xp and Yp are the coordinates of the prototype from which the family is derived. These are in the same format as the family members.

There are 16 input parameters that may be modified to control how a shape family is generated. Any number of these values may be entered in a single command line, and they may be entered in any order, although the value must always follow its associated parameter immediately. The values control such things as the number of sides the generated shapes will have, the number of shapes created, and so on. The use of each parameter is detailed below.

NMembers. This is the number of family members to be generated. Any number may be requested. A value of 0 or lower will result in only a prototype being generated. The default value is 4.

NSides. This is the number of sides the prototype and each family member will have. Attneave (1957) referred to this as the *complexity* of the shape. Any number of sides 3 or greater may be requested, but as the number rises the function may take a long time to run, especially if *TopolMethod* (see below) is set to "fast" or "complete." A maximum of 24 is recommended. The default value is 6.

NPts2Shift. This is the number of vertices (points) to shift when making new family members. Generally, this should be equal to *NSides*, but some authors have used stimuli they refer to as *mutants*, where the original prototype shape has only one vertex moved. To make such mutants, set this parameter to 1. The default value is 6.

PtsMethod. This is the method of choosing which vertices to shift. It is only relevant if *NPts2Shift* is less than *NSides.* Three possible values may be entered: "r" for random, "c" for constant, or "s" for sequential. The random method arbitrarily chooses a new set of points to shift each time. The constant method shifts the same points each time. The sequential method incrementally moves through all the points in the shape, shifting around the shape by one vertex for each new family member. The default value is "r."

ShiftLims. This is a vector containing two values, a minimum and maximum amount by which vertices are shifted when family members are made. Both values must be in the range of 0 to 1. Note that the shift value may be constrained to a single distance by making these equal. The default value for this parameter is [0 1].

LengthLims. This is a vector containing two values, the minimum and maximum length of sides for the prototype and family members. Both values must be in the range of 0 to 1. Because *ShapeFamily.m* works by simply rejecting shapes that do not fit within these limits, the function may take a great deal of time to run if these two values are too close to one another. It is not recommended that the lower limit be greater than .2, nor the higher limit less than .8. If the lower limit is set too small, some sides may be too short to see, resulting in fewer visible sides than expected. If it is too large, it may be impossible for the function to generate a shape within the allowed space (all the shapes are generated within an absolute coordinate space, so all points must have coordinates in the range of 0 to 1). The upper limit should generally be left at 1; however, it may be useful to lower it as far as .5 if many-sided polygons are being generated, since the shorter sides will tend to allow the shape to fit into the allowed space more readily. The default value for this parameter is [.05 1].

AngleLims. This is a vector containing two values, the minimum and maximum acute angles, in degrees, at any

vertex of the prototype or family members. If the angle at a vertex is too wide, the sides may not be visibly differentiable, resulting in fewer visible sides than expected. If the angle is too acute, this may result in a *spike* that contains no volume and is not properly part of the shape. As with *LengthLims*, it is not recommended that the minimum and maximum values be too similar, or the function will take a long time to run. Recommended values are 5 and 175 or 10 and 170, respectively. The default value of this parameter is [5 175].

FamilyName. This is simply a label for the shape family. If images of the shapes are being generated (see *MakePix*, below), this acts as the first part of the image file names. Any string is allowed, so "Smith," "One," or "S&V22" are all acceptable. The default value is "shape."

FamilyRes. This determines the degree of *family resemblance* among the shapes generated. *ShapeFamily.m* creates family members by shifting the locations of the prototype's vertices in random directions by a random distance. The distance is drawn from a flat probability distribution with a range of 0 to 1 and then multiplied by 1 − *FamilyRes.* Thus, the higher this parameter, the lower the average distance a vertex is shifted when a family member is created. For shapes having some degree of subjective similarity, recommended values are from .80 to .99. The default value is .9.

TopolMethod. ShapeFamily.m can check the generated shapes to see if they have good topological integrity—that is, that they are single shapes with no easily differentiable parts or holes. To accomplish this, *ShapeFamily.m* applies a number of pixel-wise erosions to the shapes it generates and then checks to see whether this divides them into separate pieces. There are three possible values for this parameter: "n" for "none," "f" for "fast," and "c" for "complete." The *fast* method is much quicker but may miss some unacceptable cases, especially where many-sided polygons are being generated. The *complete* method is recommended but can take a great deal of time, especially where many-sided polygons are being generated. More on the specific algorithm used by this feature is given in a later section. The default value is "n."

NErosions. This is the number of pixel-wise erosions applied to a shape when checking its integrity. As more erosions are applied, thicker bridges between parts will be severed, resulting in only highly integral shapes being accepted for output. The default value for this parameter is 3.

CrossCheck. If *CrossCheck* is set to "y," *ShapeFamily* checks to make sure that no lines cross in the shape being generated. This is in accordance with Attneave and Arnoult's (1956) Method 1. *CrossCheck* may also be set to "n," in which case line crossings are allowed in the output shapes and new *emergent* vertices may be created (see Figure 1D). Setting *CrossCheck* to "n" will approximate Attneave and Arnoult's Method 2, although certain constraints on point joining are not implemented. The default value is "y."

MakePix. If *MakePix* is set to "y," *ShapeFamily.m* will save the generated family of shapes as a set of TIFF files. These are created in the folder where *ShapeFamily.m* is executed. The file names follow the naming convention <Family Name><Member Number>.tiff, where <Family Name> is a string passed via the *FamilyName* parameter (see above) and <Member Number> is a sequential two-digit number from 1 to *NMembers.* The prototype image is saved to a file called <Family Name>Proto.tiff. The default value of this parameter is "y." One can view these images in Matlab. For instance, to view the second member of a family of shapes whose *FamilyName* (see above) is "MyShapes," use the following commands:

```
»x=imread('MyShapes02.tiff');
»imshow(x);
```

ImageSize. The size (in pixels) of the images generated when *MakePix* is "y." The images are always square. The default is 256.

PrototypeX and *PrototypeY.* Normally, *ShapeFamily.m* generates its own prototype shape and then derives the family members from it. However, it is possible to pass the function a set of *x* and *y*-coordinates for a prototype, in which case it will use these as the starting point for a family. The passed values in this case are vectors of coordinates in the 0 to 1 range, with one coordinate per vertex. There are no default values for these parameters; they are randomly generated if none is passed to the function.

Examples

The following gives some examples of parameter sets that might be passed to *ShapeFamily.m*, and the results. The resulting shapes are shown in Figure 2, with the prototype shape at the top and the family members below.

```
» [Xf, Yf, Xp, Yp]=ShapeFamily;
```

Simply entering the command without any passed parameters will cause a dialog window to be activated. The user may enter values for all the different parameters here. Simply hitting the "okay" button without modifying any parameters will cause a family of shapes to be generated in accordance with the default values. This means that four family members will be generated, with six sides each; the sides of these shapes will be a minimum of .05 absolute units long, the acute angles at all vertices will be between 5° and 175°, and so on. The coordinates for the shapes will be stored in Xf and Yf, with one shape per row. The prototype will be stored in Xp and Yp. See Figure 2, column A for an example of shapes generated with the default parameter values.

```
» [Xf, Yf, Xp, Yp]=ShapeFamily('NSides', 12,
'NPts2Shift', 12, 'NMembers', 3);
```

This will generate a prototype and a family of three shapes, all having 12 sides. Each family member will be created by shifting all the vertices in random directions by a random distance (see Figure 2, column B).

```
» [Xf, Yf, Xp, Yp]=ShapeFamily('FamilyRes', 0.7,
'NSides', 12, 'NPts2Shift', 12, 'NMembers', 3);
```

This is the same as the previous example, except that the family resemblance has been set to .7, meaning the shapes in the family will not be very similar to one another. The shapes in the family will be created by moving the vertices of the prototype an average of .1 absolute units, up to twice the length of a side, so the similarity in shape will tend to be low (see Figure 2, column C).

```
» [Xf1, Yf1, Xp, Yp]=ShapeFamily('Family
  Name', 'One');
```

and

```
» [Xf2, Yf2]=ShapeFamily('FamilyName',
  'Two', 'PrototypeX', Xp, 'PrototypeY', Yp);
```

Executing these two commands in sequence will create two families based on the same prototype (see Figure 2, columns D and E).

```
» [Xf, Yf]=ShapeFamily('PrototypeX', [.2 .2 .8 .8],
  'PrototypeY', [.2 .8 .8 .2]);
```

This will generate a family of shapes with a square as the prototype (see Figure 2, column F).

```
» [Xf, Yf, Xp, Yp]=ShapeFamily('NSides', 4,
  'NMembers', 4, 'NPts2Shift', 1, 'PtsMethod', 's');
```

This command will generate a family of 4 four-sided shapes, each of which will be a *mutant*, as used by Cooper and Podgorny (1976). That is, each will have only a single vertex moved. In addition, because the point selection method is sequential, each family member will have a different point moved (see Figure 2, column G).

The Algorithm

The algorithm used by *ShapeFamily.m* is quite similar to the first one suggested by Attneave and Arnoult (1956). The main difference is that this program does not use discrete units for coordinate values. Also, the program has the ability to automatically filter out shapes that do not match certain criteria regarding lengths of sides, angles at vertices, and topological integrity.

The first part of the function generates the prototype shape by randomly scattering a set of points in a 2-D space. The coordinate values of these points are drawn from a flat distribution in the range of 0 to 1. Alternatively, the user may pass in a set of prototype coordinates via the *PrototypeX* and *PrototypeY* parameters. A convex hull is then generated around the set of points. Each point that is not on the hull is then connected to two points that are. This is done by inserting the inside point's coordinates into the list of convex hull coordinates at a random place and then checking to see whether the shape thus defined has any lines that cross. If crossing lines are created, a new insertion place in the list is attempted. This is continued until all possible insertion places are tried. There will always be at least one insertion placement that does not result in line crossings. Once a good insertion point is found, the inside coordinate is added to the list of outer hull coordinates. Then the next inside point is placed randomly in the list, and the process is started again. This continues until all the points have been connected in a single shape with no crossing lines. The prototype shape thus generated is then checked to see if it fits criteria for length of sides, acute angles at vertices, and topological integrity. If it does not, new vertices are generated, and the process starts again.

The length of the sides is determined in a straightforward manner: It is the linear distance between each pair of adjacent points in the list of vertex coordinates. If any length is outside the values given in *LengthLims*, the prototype shape is rejected. The angle check relies on the law of cosines, which, for any set of three points—x1, y1, x2, y2, and x3, y3—gives the acute angle at x2, y2 as the arccosine of the dot product of the vectors (i.e., x1, y1 to x2, y2 and x3, y3 to x2, y2) over the dot product of their norms. These two checks are incorporated mainly in order to eliminate situations in which two or more nearly collinear sides appear to form a single side or in which the length of a side is so small as to be invisible. They ensure that all *n*-sided figures will visually appear *n*-sided.

The purpose of the topological check is to ensure that the output figures will each consist of a single unitary shape and not a set of differentiable parts joined at single points or narrow bridges. The function accomplishes this by first creating binary images of the shapes. These images consist of "on" pixels inside the shape and "off" pixels outside it (see Figure 3A). The image is then eroded a number of times (equal to *NErosions*), and its Euler number is checked (see Figure 3B). Erosion is a morphological operation that removes the outer layer of pixels from a shape. Doing this several times has the effect of severing narrow bridges and junctions that may join one or more distinguishable parts in a shape. The Euler number is checked following this procedure to see whether the shape still consists of a single contiguous area. The Euler number is a topological quantity equal to the number of shapes in an image (i.e., the number of contiguous patches of "on" pixels) minus the number of holes in those shapes (i.e., the number of patches of "off" pixels inside the patches of "on" pixels).

When the *TopolMethod* parameter is set to "fast," the function simply checks that the Euler number equals 1 following

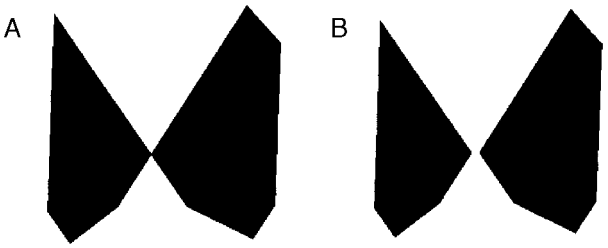

Figure 3. Pixel-wise erosion serves to sever two distinguishable parts of a shape. (A) The shape before erosion is topologically unitary, in that it consists of a single contiguous patch of "on" pixels, but it has two obvious parts. (B) The parts have been separated by implementing three consecutive erosion operations.

the erosions. This is generally a reliable method of ensuring that a single integral shape is generated, but it may miss some unacceptable cases, such as when a shape consists of two distinguishable parts, one of which has a hole in it (2 − 1 = 1, so the Euler number will be 1 even though the shape is not acceptable). When *TopolMethod* is set to "thorough," this problem is eliminated by checking the Euler number before and after a flood fill has been performed on the image. The flood fill starts at the image origin and has the effect of linking any separate shapes in the image into a single solid background, leaving only the holes. The absolute Euler number at this point will always equal one more than the number of holes. The *thorough* method eliminates any generated figure that contains any holes, as well as any figure containing more than one part (where parts are defined as those areas that become separate following the erosions).

If the prototype shape fails to fulfill any of the criteria above, it is rejected, and a new one is created (unless the prototype is passed by the user, in which case the program exits with an error). Once an acceptable prototype is generated, it is used to derive a family of similar shapes. Each member of the shape family is generated by moving the vertices of the prototype by random amounts in random directions. The distances are chosen from a flat distribution in the range of 0 to 1 and then are multiplied by 1 − *FamilyRes*. Any shift distances falling outside the values in *ShiftLims* are changed to equal the minimum or the maximum value. The shift directions are chosen from a flat distribution covering 360º. If *Pts2Shift* is less than *NSides*, a subset of vertices is selected for shifting for each family member. Depending on *PtsMethod*'s value, these points may be selected randomly, held constant, or incremented sequentially. Each of the shapes thus generated is checked to see whether any of its lines cross. If not, they are put through the same checks for lengths of sides, angles at vertices, and topological integrity as the prototype shape. Family members that fail any of these checks are rejected, and replacements are created for them.

Conclusions

Attneave's (1957; Attneave & Arnoult, 1956) methods for generating shapes fill a need in behavioral research for quantifiable shape stimuli. *ShapeFamily.m* provides a rapid and well-controlled method for generating shapes, using his algorithm, and allows automatic checking of a number of criteria regarding the qualities of the shapes generated. It is our hope that this will prove useful to other behavior researchers and will allow a wider use of Attneave's techniques.

REFERENCES

ATTNEAVE, F. (1957). Physical determinants of the judged complexity of shapes. *Journal of Experimental Psychology*, **53**, 221-227.

ATTNEAVE, F., & ARNOULT, M. D. (1956). The quantitative study of shape and pattern perception. *Psychological Bulletin*, **53**, 452-471.

BETHELL-FOX, C. E., & SHEPARD, R. N. (1988). Mental rotation: Effects of stimulus complexity and familiarity. *Journal of Experimental Psychology: Human Perception & Performance*, **14**, 12-23.

COOPER, L. A. (1975). Mental rotation of random two-dimensional shapes. *Cognitive Psychology*, **7**, 20-43.

COOPER, L. A., & PODGORNY, P. (1976). Mental transformations and visual comparison processes: Effects of complexity and similarity. *Journal of Experimental Psychology: Human Perception & Performance*, **2**, 503-514.

CORNOLDI, C., & LONGONI, A. (1977). The MP–DP effect and the influence of distinct repetitions on recognition. *Italian Journal of Psychology*, **4**, 65-76.

FOLK, M. D., & LUCE, R. D. (1987). Effects of stimulus complexity on mental rotation rate of polygons. *Journal of Experimental Psychology: Human Perception & Performance*, **13**, 395-404.

GARNER, W. R. (1970). Good patterns have few alternatives. *American Scientist*, **58**, 34-42.

KLEIN, R. (1982). Patterns of perceived similarity cannot be generalized from long to short exposure durations and vice versa. *Perception & Psychophysics*, **32**, 15-18.

MARSOLEK, C. J. (1996). Dissociable neural subsystems underlie abstract and specific object recognition. *Psychological Science*, **10**, 111-118.

POSNER, M. I., & KEELE, S. W. (1968). On the genesis of abstract ideas. *Journal of Experimental Psychology*, **77**, 353-363.

WILLIS, E., & DORNBUSH, R. L. (1968). Preference for visual complexity. *Child Development*, **39**, 639-646.

WU, Z., SUN, C., WU, Z., & XU, S. (1991). Age differences in the imagination of figures. *Psychological Science* [China], **2**, 1-6.

APPENDIX

```
function varargout=ShapeFamily(varargin);
% No help is included in this version.  Help text is included in the
% downloadable version available at www.dal.ca/~mcmullen/downloads.html

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%   ASSIGN PARAMETERS   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% First, create table of parameter names, values and prompts
Parameters = {                                                          ...
'NMembers',    '4',     'Number of family members to generate (0-50): ';   ...
'NSides',      '6',     'Number of sides per shape (3-24+): ';             ...
'NPts2Shift',  '6',     'Number of vertices to shift (0 to # of sides) : ';...
'PtsMethod',   'r',     'Shifted point selection method (r, c, or s) ';    ...
'ShiftLims',   '0 1',   'Min&Max vertex shifts (0-1): ';                   ...
'LengthLims',  '.05 1', 'Min&Max side lengths (0-1): ';                    ...
'AngleLims',   '5 175', 'Min&Max vertex angles (0-180): ';                 ...
'FamilyName',  'shape', 'Family name: ';                                   ...
'FamilyRes',   '.9',    'Family resemblance coefficient (0-1): ';          ...
```

APPENDIX (Continued)

```
'TopolMethod', 'n',     'Topology checking method (f, c, or n)?';          ...
'NErosions',   '3',     'Number of erosions to seperate parts (0-5+): ';   ...
'CrossCheck',  'y',     'Check for line crossings? ("y" or "n"): ';        ...
'MakePix',     'y',     'Make pictures? ("y" or "n"): ';                   ...
'ImageSize',   '256',   'Size of output images in pixels: ';               ...
'PrototypeX',  '',      'X coordinates of prototype shape: ';              ...
'PrototypeY',  '',      'Y coordinates of prototype shape: ';              ...
};

% If no parameters were input at the command lines, pop up windows
% to get their values.
if nargin == 0
   ParamInputs = inputdlg(Parameters(:,3), 'Enter Shape Parameters', 1,    ...
   Parameters(:,2));

   [Parameters{:,2}] = deal(ParamInputs{:});
end

% Now assign each parameter in the table to its value
for par = 1:size(Parameters,1)
   if isempty(str2num(Parameters{par,2})) % for string entries
      eval([Parameters{par,1} '=''' Parameters{par,2} ''';']);
   else                                   % for number entries
      eval([Parameters{par,1} '=[' Parameters{par,2} '];']);
   end
end

% If there were inputs at the command line, use those
for x=1:2:nargin
   if ~exist(varargin{x})        % check for non-parameter entries
      error(['''' varargin{x} ''' is not a recognized parameter.']);
   elseif ischar(varargin{x+1})  % if a string entry
      eval([varargin{x} '=''' varargin{x+1} ''';']);
   elseif ~ischar(varargin{x+1}) % if a numerical entry
      eval([varargin{x} '=[' num2str(varargin{x+1}) '];']);
   end
end

rand('state', fix(1e6*sum(clock))); % Seed the random number generator

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  GENERATE PROTOTYPE SHAPE  %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% IF NO PROTOTYPE HAS BEEN HANDED TO THE FUNCTION, IT GENERATES ONE
if isempty(PrototypeX)

   ProtoTypeOkay=0;
   while ~ProtoTypeOkay
      ProtoX = rand(1,NSides);  % make random points
      ProtoY = rand(1,NSides);

      ConvexDex = convhull(ProtoX,ProtoY);  % Get the convex hull indexes.

      % Get the coordinates of points on the convex hull
      ProtoXConv = ProtoX(ConvexDex(1:end-1));
      ProtoYConv = ProtoY(ConvexDex(1:end-1));

      % Get the indexes of points inside the hull
      InDex = inpolygon(ProtoX, ProtoY, ProtoXConv, ProtoYConv);

      % Get the coordinates of points inside the hull
      ProtoXIn = ProtoX(InDex==1);
      ProtoYIn = ProtoY(InDex==1);
```

APPENDIX (Continued)

```
% Shuffle the order of the inside points
% They will be attached to the outer hull in the shuffled order
[null, sdex] = sort(rand(size(ProtoXIn)));
ProtoXIn = ProtoXIn(sdex);
ProtoYIn = ProtoYIn(sdex);

% TestX and TestY are the coordinates to be tried out.  To begin
% they are assigned to be equal to the points on the Convex Hull.
TestX = ProtoXConv; TestY = ProtoYConv;

% For each point inside the hull, try inserting its coordinates into the
% list of coordinates for points that are on the hull.  The place of
% insertion into the list is random.  Generate the polygon thus defined
% and check if it has any crossing lines.  If it does, try the next list
% insertion placement, otherwise move on to insert the next point that

% inside is the polygon.
ProtoTypeOkay = 1;
ProtoTypeFailed = 0;
for InsertDex = 1:length(ProtoXIn)

   % Create random order in which to try insertion points.
   InsertOrder = shuffle(1:length(TestX));

   for p = 1:length(InsertOrder)

      % Insert the new coordinate in the list of coordinates already
      % on the polygon
      TestX(InsertOrder(p)+1:end+1) = TestX(InsertOrder(p):end);
      TestY(InsertOrder(p)+1:end+1) = TestY(InsertOrder(p):end);

      TestX(InsertOrder(p)) = ProtoXIn(InsertDex);
      TestY(InsertOrder(p)) = ProtoYIn(InsertDex);

      % Check to see if the new shape has line crossings in it
      CrossOkay = CheckCross([TestX, TestX(1)], [TestY, TestY(1)],   ...
      CrossCheck);

      % If there is a crossing, undo the insertion
      if ~CrossOkay
         TestX(InsertOrder(p):end-1) = TestX(InsertOrder(p)+1:end);
         TestY(InsertOrder(p):end-1) = TestY(InsertOrder(p)+1:end);
         TestX = TestX(1:end-1);
         TestY = TestY(1:end-1);
      else
         % if the point is okay with the currently attempted insertion
         % break out and move on to place the next inside point.
         break;
      end
   end

end

% If the prototype has been successfully created by the above,
% check it for other criteria, rejecting and starting anew if
% any are not met.
if ProtoTypeOkay

   LengthOkay = 1; AngleOkay = 1; TopolOkay = 1;

   LengthOkay = CheckLength(TestX,TestY, LengthLims(1), LengthLims(2));
   if LengthOkay
      AngleOkay  = CheckAngle(TestX, TestY, AngleLims(1), AngleLims(2));
      if AngleOkay
         TopolOkay  = CheckTopol(TestX, TestY, NErosions, TopolMethod);
      end
   end
```

APPENDIX (Continued)

```
         if any([~LengthOkay, ~AngleOkay, ~TopolOkay]);
            ProtoTypeOkay = 0;
         end
      end

   end

   % If the set of coordinates passes all criteria, they become the
   % prototype coordinates.
   Xp = TestX;
   Yp = TestY;

% IF A PROTOTYPE HAS BEEN HANDED TO THE FUNCTION, IT CHECKS IF IT PASSES
% THE SHAPE CRITERIA AND USES IT
else

   % Assign input coordinates to prototype coordinates
   Xp = PrototypeX;  Yp = PrototypeY;

   NSides = length(Xp);  % Derive # of sides from length of passed prototype

   % Check if the prototype meets criteria
   LengthOkay = 1; AngleOkay = 1; TopolOkay = 1; CrossOkay = 1;

   LengthOkay = CheckLength(Xp,Yp, LengthLims(1), LengthLims(2));
   if LengthOkay
      AngleOkay  = CheckAngle(Xp,Yp, AngleLims(1), AngleLims(2));
      if AngleOkay
         TopolOkay  = CheckTopol(Xp,Yp, NErosions, TopolMethod);
         if TopolOkay
            CrossOkay  = CheckCross([Xp, Xp(1)], [Yp, Yp(1)], CrossCheck);
         end
      end
   end

   % Display error if prototype is not acceptable
   if     length(Xp) ~= length(Yp)
      error('Length of ProtoTypeX and ProtoTypeY must be the same');
   elseif (Xp>1 | Xp<0) | (Yp>1 | Yp<0)
      error('All entered prototype coordinates must be in range 0 to 1');
   elseif ~LengthOkay
      error('Line(s) in prototype are not within length limits');
   elseif ~AngleOkay
      error('Angle(s) in prototype are not within angle limits');
   elseif ~TopolOkay
      error('Prototype has failed topological check');
   elseif ~CrossOkay
      error('Lines in prototype cross one another');
   end

end

% Print the prototype image out to a TIFF file
if MakePix == 'y'
   img = flipud(~roipoly(repmat(0, ImageSize, ImageSize),                    ...
   Xp * ImageSize, Yp * ImageSize));
   imwrite(img, [FamilyName, 'Proto.tiff'], 'tiff', 'compression', 'ccitt');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%   GENERATE FAMILY MEMBERS   %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

APPENDIX (Continued)

```
Xf = ''; Yf = ''; % Assign empty matrices for shape coordinates

LengthOkay = 0; AngleOkay = 0; TopolOkay = 0;

% For the 'constant' or 'sequential' shifted point selection methods,
% generate a random set of points to be shifted.
if (PtsMethod == 'c') | (PtsMethod == 's')
   PointIndexes = 1:NSides;
   [null, sdex] = sort(rand(size(PointIndexes)));
   PointIndexes = PointIndexes(sdex);

   StaticPtIndexes = PointIndexes(1:(NSides-NPts2Shift));
end

MembersMade = 0;
while MembersMade < NMembers

   while 1
      % For each family member, generate a new set of coordinate
      % offsets (shifts).
      xyShiftLengths = rand(1,NSides) * (1-FamilyRes);

      % Check to make sure the shifts are within the desired limits
      xyShiftLengths(find(xyShiftLengths > ShiftLims(2) )) = ShiftLims(2);
      xyShiftLengths(find(xyShiftLengths < ShiftLims(1) )) = ShiftLims(1);

      % If a random set of points are being shifted on each family
      % member, generate a new set of points each time.
      if PtsMethod == 'r'
         PointIndexes = 1:NSides;
         [null, sdex] = sort(rand(size(PointIndexes)));
         PointIndexes = PointIndexes(sdex);
         StaticPtIndexes = PointIndexes(1:(NSides-NPts2Shift));
      end

      xyShiftLengths(StaticPtIndexes) = 0;

      % Create shift angles (0 to 360) and then derive x & y shifts
      xyShiftAngles = rand(1,NSides) * pi * 2;

      xshifts = cos(xyShiftAngles) .* xyShiftLengths;
      yshifts = sin(xyShiftAngles) .* xyShiftLengths;

      % Apply the shifts to the prototype's coordinates
      % to generate a new family member.
      XMem = Xp + xshifts;
      YMem = Yp + yshifts;

      % Check that coordinates are all still in 0 to 1 range.
      if all([XMem>0, XMem<1, YMem>0, YMem<1]);
         break;
      end
   end

   % Check that the family member fits the various shape criteria.
   LengthOkay = 1; AngleOkay = 1; TopolOkay = 1; CrossOkay = 1;

   LengthOkay = CheckLength(XMem, YMem, LengthLims(1), LengthLims(2));
   if LengthOkay
      AngleOkay  = CheckAngle(XMem, YMem, AngleLims(1), AngleLims(2));
      if AngleOkay
         TopolOkay  = CheckTopol(XMem, YMem, NErosions, TopolMethod);
         if TopolOkay
            CrossOkay  = CheckCross([XMem,XMem(1)],[YMem,YMem(1)], CrossCheck);
         end
      end
   end
```

APPENDIX (Continued)

```
   if all([LengthOkay, AngleOkay, TopolOkay, CrossOkay])
      MembersMade = MembersMade + 1;

      Xf = [Xf; XMem];  % Save coordinates of generated
      Yf = [Yf; YMem];  % family members.

      % If points to be shifted are being sequentially moved around
      % for each family member
      if PtsMethod == 's'
         StaticPtIndexes = StaticPtIndexes+1;
         StaticPtIndexes(find(StaticPtIndexes > NSides)) = 1;
      end

      % Print the image out to a TIFF file
      if MakePix == 'y'
         img = flipud(~roipoly(repmat(0, ImageSize, ImageSize),            ...
         XMem*ImageSize, YMem*ImageSize));

         imwrite(img, [FamilyName, sprintf('%.2d',MembersMade) '.tiff'],   ...
         'tiff', 'compression', 'ccitt');
      end

   end
end

% Finish up and provide output arguments
close('all');
if nargout >= 2
   varargout{1} = Xf;
   varargout{2} = Yf;
end
if nargout == 4
   varargout{3} = Xp;
   varargout{4} = Yp;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  MINOR FUNCTIONS  %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

% FUNCTION CheckAngle
function okay = CheckAngle(X, Y, MinAngle, MaxAngle);
% CheckAngle takes a set of points contained in the vectors X and Y,
% and checks them to make sure the angles between all sets of 3 adjacent
% points are within MinAngle and MaxAngle.

okay = 1;
if (MinAngle == 0) & (MaxAngle == 180)
   return;
end

X = [X, X(1:2)];  % Add the first two coords on the end in order to
Y = [Y, Y(1:2)];  % check the wrap-around of the polygon.

for a = 1:size(X,2) -2
   warning off; % to get rid of divide by zero warnings
   if getangle(X(a),Y(a), X(a+1),Y(a+1), X(a+2),Y(a+2)) <MinAngle |       ...
      getangle(X(a),Y(a), X(a+1),Y(a+1), X(a+2),Y(a+2)) >MaxAngle
      okay = 0;
      return;
   end
   warning on;
end
```

APPENDIX (Continued)

```
% FUNCTION CheckLength
function okay = CheckLength(X,Y, MinLength, MaxLength);
% CheckLength takes a set of coordinates contained in the vectors X and
% Y, and checks them to make sure the distance between any two adjacent
% points is greater than MinLength and less than MaxLength.

okay = 1;

if (MinLength == 0) & (MaxLength == 1)
   return;
end

X = [X, X(1)];    % Add the first point onto the end in order to
Y = [Y, Y(1)];    % check the wrap-around of the polygon.

for a = 1:size(X,2) -1
   if (getdist(X(a), Y(a), X(a+1), Y(a+1)) < MinLength) |                    ...
      (getdist(X(a), Y(a), X(a+1), Y(a+1)) > MaxLength)
      okay = 0;
      return;
   end
end

% FUNCTION CheckCross
function okay = CheckCross(X,Y, CrossCheck);
% Checks to see if any of the lines in the polygon defined by X,Y
% cross one another.  The poly must be closed, so the last coordinate
% in the set much match the first.

okay = 1;

if CrossCheck == 'n'
   return;
end

tol = 0.000001;  % Tolerance value for vertex position
% Check each pair of lines in the polygon to see if they cross
for g = 1:length(X)-1
   for h = 1:length(Y)-1
      % First, calculate the equations of the two lines
      % (warnings are suppressed due to possibility of vertical lines)
      warning off;
      b1 = (Y(g+1)-Y(g))/(X(g+1)-X(g));
      b2 = (Y(h+1)-Y(h))/(X(h+1)-X(h));
      warning on;
      % If either line is vertical set slope to "very high" instead of Inf
      if b1 == Inf
         b1 = 1000000;
      end
      if b2 == Inf
         b2 = 1000000;
      end
      % If lines are parallel, offset one slightly to avoid intersection
      % at infinity
      if b1 == b2
         b1 = b1 + 0.000001;
      end

      % Calculate intercepts of both lines
      a1 = Y(g) - b1*X(g);
      a2 = Y(h) - b2*X(h);
```

APPENDIX (Continued)

```
      % Calculate point where lines will intersect
      xi = -(a1-a2) / (b1-b2);
      yi = a1 + b1 * xi;

      % If the intersection point is within the limits of both lines
      % and the point is not a vertex of the polygon, then there is a
      % crossing of lines.
       if all([((X(g)-xi)*(xi-X(g+1)) >= 0), ((X(h)-xi)*(xi-X(h+1)) >= 0), ...
          ((Y(g)-yi) * (yi-Y(g+1)) >= 0), ((Y(h)-yi) * (yi-Y(h+1)) >= 0),  ...
          (abs(xi-X(g)) > tol & abs(yi-Y(g)) > tol),                       ...
          (abs(xi-X(g+1)) > tol & abs(yi-Y(g+1)) > tol),                   ...
          (abs(xi-X(h)) > tol & abs(yi-Y(h)) > tol),                       ...
          (abs(xi-X(h+1)) > tol & abs(yi-Y(h+1)) > tol)]);
         okay = 0;
         break;
      end
   end

   if okay == 0
      break;
   end
end

% FUNCTION CheckTopol
function okay = CheckTopol(X,Y,NErode, TopolMethod);
% CheckTopol takes a set of coordinates passed in the vectors X and Y,
% generates the polygon they represent and checks to make sure it is a
% single integrated shape with no holes in it.

okay = 1;
% If TopolMethod is 'none' return immediately
if TopolMethod == 'n'
   return;
end

% Generate image of the polygon
img = roipoly(repmat(0,256,256), X*256,Y*256);

% First do the fast check, checking if Euler number is initially 1.
% If not, shape is not okay.
if (bweuler(img) ~= 1)
   okay = 0; return;
end
% If shape passed first test, erode it a few times, checking after each
% erosion.  If at any point, Euler ~= 1, shape is not okay.
for E = 1:NErode
   if bweuler(bwmorph(img, 'erode', E)) ~= 1
      okay = 0; return;
   end
end

% Then the full check if requested. Flood fill from image origin and check
% the number of holes in the image.  If not 0, shape is not okay.
if TopolMethod == 'f' % full
   if ((bweuler(bwfill(img, 1,1,4))-1) * -1) ~= 0
      okay = 0; return;
   end
end

% FUNCTION getdist
function dist = getdist(x1,y1,x2,y2);
dist = ((x1-x2)^2 + (y1-y2)^2)^.5;
```

APPENDIX (Continued)

```
% FUNCTION getangle
function angle = getangle(x1,y1,x2,y2,x3,y3);
% Given three points in cartesian space, x1 y1, x2 y2, and x3 y3,
% returns the acute angle at x2 y2 in degrees.

p1 = [x1 y1];  p2 = [x2 y2];  p3 = [x3 y3];
v1 = p1 - p2;
v2 = p3 - p2;

angle = acos(v1*v2' / (norm(v1)*norm(v2))) * 180/pi;


% FUNCTION shuffle
function [out,dex] = Shuffle(in)
% Randomly shuffles positions of elements of in
[dummy,dex] = sort(rand(size(in)));
out = in(dex);
```

(Manuscript received March 12, 2001;
revision accepted for publication October 14, 2001.)