Andrew Glassner's Notebook

Crop Art, Part 2

Andrew Glassner

In the September/October 2004 issue, I presented Part 1 of my discussion of crop circles. We looked at their history and some interesting geometry, and then reconstructed a famous formation using classical ruler-and-compass techniques.

This time I'll present a much easier way to capture the geometric structure of many crop circle formations, using a special-purpose language.

Crop

Most crop circle formations have the same underlying geometric ideas appearing over and over. Inspired by this observation, I created a little language called *Crop* for describing the geometry of these often-beautiful designs.

Crop is compact and easy to read. It can also create handy, explicit instructions for creating a design. We'll see an example of that in Part 3 of this column.

The design of Crop is simple and its goals are modest: to represent crop circle formations efficiently. It's definitely not a general-purpose programming language.

Crop uses a postfix syntax, which makes it easy to write and parse. It's the same style of language I used in my January/February 2003 column to create Andrew's Weaving Language. The idea is that values are pushed onto and popped off of a stack as the code is interpreted.

You can define variables within Crop, just as in any other language. The collection of all the variables that exist at any given time, along with their values, is called the *dictionary*. Initially the dictionary starts off empty, but there are a few special variables that get computed for you on the fly when you need them, as I'll discuss below.

There are three distinct types of objects in Crop: *scalars* (floating-point or integer numbers), *points* (represented by pairs of numbers), and *objects* (circles, ellipses, and polygons). In the following discussion, I'll label variables starting with the letters `s`, `p`, or `o` to identify their type (I'll also use `i` to refer to scalars that must be integers).

Crop is written in plain text, using tokens separated by white space. Any string of characters bounded by white space that doesn't have a predefined meaning is simply pushed on the stack as that string of characters. Anywhere a single space would do, you can insert as many spaces, tabs, carriage returns, or other white space as you like to improve legibility. The language is case-insensitive for all commands, but it's case sensitive for variable names.

Table 1 summarizes all of the commands in Crop. Let's look at some of them in batches.

Useful math

The first four commands add, subtract, multiply, and divide pairs of scalars. They pop two scalars off the stack, apply the operation, and push the result back on top:

```
s1 s2 +
s1 s2 –
s1 s2 *
s1 s2 /
```

Points are obviously important for describing formations. We create a point by naming two scalars and then bundling them together with `makePoint`:

```
sx sy makePoint
```

This pops two scalars, representing the *x* and *y* coordinates, and pushes back a single entity that represents a 2D point. You can use the shortcut symbol # to represent the origin, located at point (0, 0).

We can also do math on points. We can add and subtract points, and multiply or divide them by scalars:

```
p1 p2 p+
p1 p2 p–
s1 p1 p*
s1 p1 p/
```

Note that the operators here are all prefixed with the letter `p`, giving us for example `p+` instead of simply +. If you try to add two points with + rather than `p+`, you'll get an error. I think that for this language, there's value to explicitly distinguishing these operators.

We can find how far apart two points are by pushing them onto the stack and then calling `distance`, which pushes their distance back onto the stack:

```
p1 p2 distance
```

Names and geometric objects

We can name objects in Crop using the `name` command. This takes whatever object is on top of the stack (a scalar, point, or object) and assigns it to the given name:

 Published by the IEEE Computer Society

Table 1. A summary of commands in the Crop language.

Command	Usage Summary	Action
+	s1 s2 +	Add two scalars
–	s1 s2 –	Subtract two scalars
*	s1 s2 *	Multiply two scalars
/	s1 s2 /	Divide two scalars
makePoint	s1 s2 makePoint p	Create a point object
p+	p1 p2 p+	Add two points
p–	p1 p2 p–	Subtract two points
p*	s1 p1 p*	Multiply a point by a scalar
p/	s1 p1 p/	Divide a point by a scalar
distance	p1 p2 distance s	The distance between points
+	pso name name	Name an object
line	< p0 p1 p2 … pn > line	Draw lines connecting points
circle	pc < s0 s1 … sn > circle	Draw concentric circles
ellipse	pp pq < s0 s1 … sn > ellipse	Draw ellipses
ngon	pc in angle < r0 r1 … rn > ngon	Draw ngons
makeLine	p1 p2 makeLine	Make a line object
makeCircle	pc sr makeCircle	Make a circle object
makeEllipse	pp pq sr makeEllipse	Make an ellipse object
makeNgon	pc in sa sr makeNgon	Make an ngon object
trope	pa pb sa sb trope	Find a point using a triangle-rope
pwalk	o1 p1 sd pwalk	Walk around an object's perimeter
pspin	o1 p1 sd pspin	Rotate around an object's center
ngonloop	[commands] pc in sr sa ngonloop	Repeat the loop commands for each vertex in an ngon
pop	pop	Pop and discard the top of the stack
dup	dup	Duplicate the item on top of the stack
printDict	printDict	Print the dictionary (for debugging)
printStack	printStack	Print the stack (for debugging)
//	// comment	Comment to end of line
#	#	A shortcut for the point (0, 0)
%	%	A shortcut for ngonloops to rotate 180/*n* degrees
V	in V	Coordinates of specified vertex in closest loop
V0	V0, V1, V–1, …	Coordinates of vertices in closest loop
V0′	V0′, V1′, V–1′, V2′, …	Vertices in enclosing loop(s)
LC	LC	The iteration count in closest loop
LC′	LC′, LC″, …	The iteration count in enclosing loop(s)

```
pso variable-name name
```

For example, `3.14 pi name` creates a value for `pi`, and `p1 p2 p+ 2 p/ midpoint name` sets the variable `midpoint` to the point halfway between `p1` and `p2`. You can redefine a name any time by just assigning a new value to it.

Okay, that finishes up the foundation. Let's make some geometry!

The four stars of our geometry world are `line`, `circle`, `ellipse`, and `ngon`. Each of these commands uses a *list* as one of its arguments. A list is simply a sequence of objects separated by white space and delimited by angle brackets. The angle brackets are necessary even if the list has only one element.

The `line` command joins up points that are specified in a list:

```
< p0 p1 p2 … pn > line
```

It draws a line from *p0* to *p1*, then to *p2*, and so on, to *pn*.

The `circle` command draws a series of circles that share a common center, but have different radii:

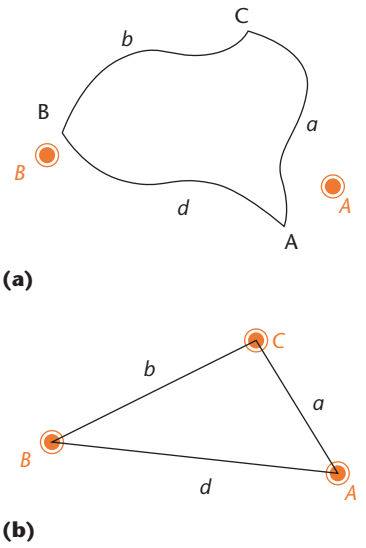
```
pc < s0 s1 … sn > circle
```

The point *pc* is the common center for all of the circles. Each scalar in the list causes a circle of that radius to be drawn. The radii do not have to be in any specific order, and they may repeat (which will simply cause the same circle to be drawn more than once).

A close geometrical cousin to the circle, the `ellipse` command takes two points *pp* and *pq* for the ellipses' foci, and a list of scalars for the string lengths that can be tied to the foci to produce the ellipse (see Part 1 of this column for geometrical information on building an ellipse in the field):

```
pp pq < s0 s1 … sn > ellipse
```

An *n*-gon is a regular, convex polygon with *n* sides. This means that each side has the same length, and the sides meet at each vertex at the same angle. The `ngon`



1 Idea behind the `trope` command. (a) A rope is marked at points A, B, and C, so that the distances between them along the rope are a = AC, b = BC, and d = AB. (b) When rope points A and B are on top of ground points A and B, and the rope is pulled taut, we have found point C.

command draws a set of n-gons with in points, centered at point pc, and with radii drawn from a list. A radius is defined to be the distance from the center of the n-gon to any vertex. The n-gon is drawn by default with the first vertex sitting on the positive x-axis. The sa parameter tells the system how many degrees to rotate the n-gon clockwise:

```
pc in sa < r0 r1 ... rn > ngon
```

A useful shortcut exists for the sa field of an **ngon** command (we'll see its use in related commands in a bit). If you use a percent sign for this angle, this will rotate the polygon so that the vertices line up with where the old side midpoints were. That is, if the polygon has in sides, then % in the sa parameter is equivalent to $180/in$ degrees. For example, in a square % means 45 degrees, and for a hexagon it means 30 degrees. I chose the percent sign (two circles and a line) to remind us of an edge and two vertices.

The commands we've just seen actually draw objects. Sometimes it's handy to create these objects so we can use their geometry to make other objects, but not draw them. The commands **makeLine**, **makeCircle**, **makeEllipse**, and **makeNgon** do just that. These commands are like their corresponding versions from above, except that they don't take lists for arguments, since each command makes a single object. Typically when you make an object with these commands you'll then assign it a name so it can be used later:

```
p1 p2 makeLine
pc sr makeCircle
pp pq sr makeEllipse
pc in sa sr makeNgon
```

Finding a point

Sometimes in the field you need to locate a certain point, perhaps for the center of a circle or n-gon. Often the best way to define that point is with respect to two points you already know, and a distance from each of those. In that case, you could find the missing point using what I call a *triangle rope*.

Suppose that your unknown point C is a distance a from point A and a distance b from point B. Since you know where points A and B are located, you also know the distance d between them. Take a rope of length $a + b + d$ and knot the ends together, making a loop. Use tape and a bit of paper to mark the knot with a big letter A. Now measure off a length d of rope in either direction from the knot, and mark it with a big B. Now continue from that point around the rope and measure off a distance b. Put tape on the rope that marks it as point C. If you want to check your work, make sure that the amount of rope left over has length a. Figure 1a shows the idea.

Now in the field, have one person stand at point A and hold the rope at the knot, and have another person at point B hold the rope at the tape marked B. A third person holds the knot at the tape marked C pulls the rope away from the other two people; when the rope is taut, this person is at point C, as in Figure 1b. The person pulling the rope can find two taut points, one on either side of the line AB. You'll want to make sure as you're walking around in the field that you're standing on the correct side of the line for the point you're trying to locate.

The Crop command that embodies this process is **trope**. It takes as arguments the two points **pa** and **pb** and the distances **sa** and **sb** that tell how far away the unknown point is, respectively, from these two points:

```
pa pb sa sb trope
```

Figure 2 shows the geometry behind **trope**. The system draws a circle of radius a around point A, and a circle of radius b around point B, and tries to intersect them. The result could be that there are no points of intersection, just one (if they happen to be tangent), or two distinct points. If no points of intersection exist between the two circles, the system reports an error, and returns the origin point (0, 0). If there's just one point, the system returns that. Otherwise, it imagines a line from A to B, and it returns the point that's on the left side of that line.

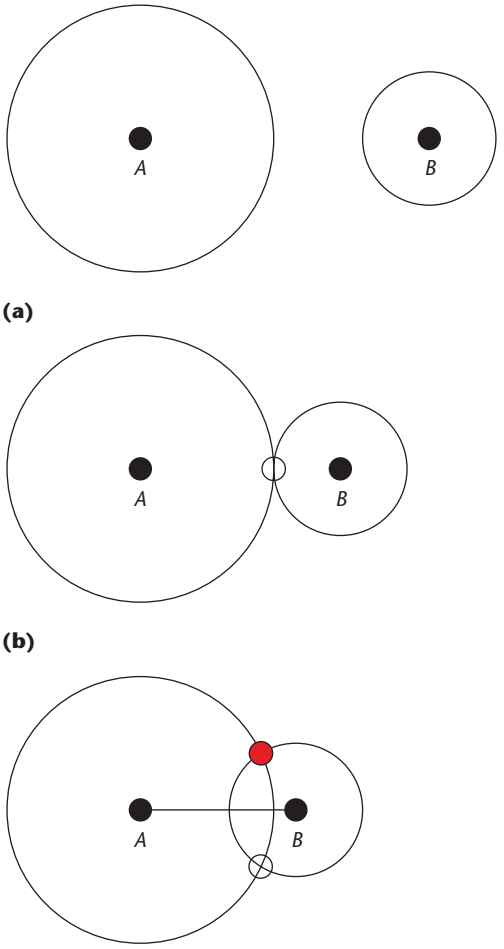
Finding the point of intersection of two circles is straightforward. Figure 3 shows the geometry. We can see from the figure that $a^2 = p^2 + h^2$ and $b^2 + q^2 + h^2$. Rewriting these for h^2 we get $h^2 = a^2 - p^2$ and $h^2 = b^2 - q^2$. Since these are both h^2, we have $a^2 - p^2 = b^2 - q^2$. Because $d = p + q$, then $q = d - p$. Substituting this value for q into the last expression and simplifying

$$\begin{aligned} a^2 - p^2 &= b^2 - q^2 \\ &= b^2 - (d-p)^2 \\ &= b^2 - d^2 + 2dp - p^2 \end{aligned}$$

Removing p^2 from both sides and solving for p, we find

$$p = \frac{a^2 - b^2 + d^2}{2d}$$

We can now find the point Q as $Q = A + (p/d)(B - A)$.



2 (a) No intersections between circles. (b) One intersection between circles. (c) Two intersections between circles. The system returns the point on the left side of the line from *A* to *B*.

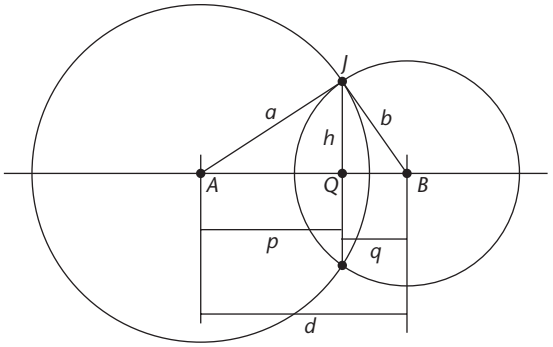
To find J, we first compute $h - \sqrt{a^2 - p^2}$. Now make a vector **S** that's perpendicular to AB, so $\mathbf{S} = (-(B_y - A_y), B_x - A_x)$. Normalize **S** to unit length (call it $\hat{\mathbf{S}}$) and now find $J = Q + h\hat{\mathbf{S}}$.

Another way to create a point is with the **pwalk** command. This command takes an object *o1*, a starting point *p1*, and a distance *sd*, and "walks" the starting point clockwise around the perimeter of that object. After it's covered the desired distance, the system pushes the newly computed print onto the stack. Note that the distance measured isn't the straight line between the starting point and the ending point, but instead is the distance as measured along the object's perimeter.

```
o1 p1 sd pwalk
```

The **pwalk** command needs to start with a point already on the object's perimeter. If the input point *p1* isn't on the perimeter, **pwalk** finds the point on the object's perimeter that's nearest to *p1* and starts with that instead.

A closely related command is **pspin**. This command



3 The geometry of `trope`. Circle A has center *A* and radius *a*. Circle B has center *B* and radius *b*. The distance between them is *d*, and they meet at point *J*. The line perpendicular to *AB* through *J* meets the line *AB* at *Q*.

takes an object *o1*, a starting point *p1*, and an angle *sd* by which to rotate that point clockwise around the object's center:

```
o1 p1 sd pspin
```

Loops

Many crop formations are built on a structure based on one or more regular polygons. A common idiom is that a similar construction is carried out at each vertex of these polygons. This suggests the use of a loop that repeats for each polygon vertex. I call this an *ngonloop*.

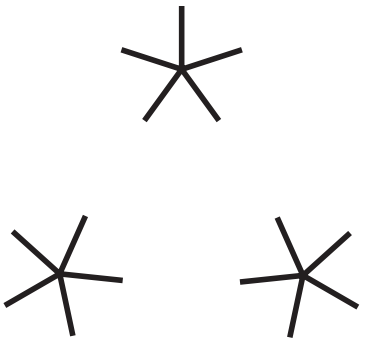
In many ways, this command is the heart of Crop:

```
[ commands ] pc in sr sa ngonloop
```

The arguments are the same as those in the **ngon** command: they define a polygon with a center at *pc*, made of *in* points, with a radius *sr*, rotated clockwise by an angle *sa*. As before, *sr* is the distance from the center to each vertex, and *sa* is an angle in degrees to rotate the polygon clockwise. And the special character % used for *sa* again means to rotate the polygon 180/*in* degrees. The commands between the square brackets get executed *in* times, once for each vertex in the polygon.

While the loop is executing, variables beginning with the letter **V** take on a special meaning: they're point objects whose values are the polygon's vertices. The system takes the rest of the name of the variable and interprets it as a number (if you use variables in the loop that begin with V and then continue, but aren't immediately followed by an integer, you'll get an error). If you use the variable **V** all by itself, then the system assumes that there's an integer on the top of the stack that should be used as the value for the variable. For example, rather than say **V4** you could say **2 2 + V**, which refers to the same vertex.

The variable **V0** has the value of the current vertex (that is, it's a point). The variable **V1** is the next point clockwise around the polygon, **V2** is the one after that, and so on. Variable **V–1** is the point preceding the current point (that is, counter-clockwise from **V0**), **V–2** is the one counter-clockwise from that one, and so on. The indices are "wrapped around" the polygon, so negative



4 Crop expression and its result.

numbers, or numbers greater than *in*, are taken modulo *in*. Note that `V–2` is a variable name, not an arithmetic expression (both because Crop isn't an infix language, and because `V–2` has no spaces).

For example, suppose we've created a polygon with *sa* = 0, so the first vertex is on the *x*-axis. Then the first time through the loop, the variable `V0` refers to that vertex, and `V1` refers to the first vertex clockwise from the *X* axis. The second time through the loop, `V0` has the value of the first vertex clockwise of the *X* axis, `V1` has the value that `V2` used to have, and so on.

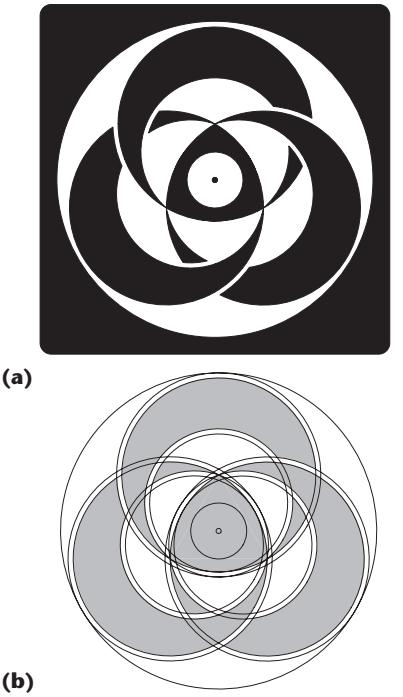
Another special variable is `LC`, which stands for LoopCount. This is an integer that tells us how many times the loop has been completed. The first time through the loop, `LC` is zero. The next time it's one, and so on. For example, to refer to the vertex on the *x*-axis at any point during the loop, you could use the idiom `LC –1 * V`. So the fourth time through the loop, `LC` has the value 3, this little expression evaluates to –3, and the variable is read as `V–3`.

You can nest loops if you want by putting one `ngonloop` inside another. Each polygon loop can of course have a different center, radius, angle, and number of vertices. A common idiom is to place the center of an inner polygon on the vertices of an outer one.

The variables `V0`, `V1`, `V–1`, and so on, as well as `LC`, refer to the innermost loop in which they appear. If you want to refer to the vertices of the polygon in an outer loop, append a prime to the variable name. Thus `V1′` refers to the vertex after the current one in the polygon one loop up, and `V–3″` refers to the vertex three steps before the current one in the polygon two loops up. Similarly, `LC′` refers to the loop count in the loop one up, and `LC″` goes two loops up.

For example, suppose we want to place a small pentagon on each vertex of a big triangle, and draw a line from each vertex of each pentagon to the vertex of the triangle it's centered upon. To make things more interesting, let's rotate each pentagon so that its first vertex lies on the line from the triangle's center to the pentagon's center, as in Figure 4. We could write

```
[
    [ < V0 V0′ > line ] V0 5 3 LC 120 *
       ngonloop
] # 3 6 0 ngonloop
```



(a)

(b)

```
[ V0 < 3.46 3.64 > circle ] # 3 2 0
 ngonloop
[ V0 < 2.46 2.64 > circle ] # 3 1 0
 ngonloop
# < 1 5.64 0.09 > circle
```

(c)

5 (a) Folly Barn 2001 formation. (b) Schematic of (a). (c) Crop code for (a).

In the innermost loop, `V0` refers to the current vertex of the pentagon, and `V0′` refers to the current vertex of the triangle. The expression `LC 120 *` rotates the pentagon based on how many times we've gone through the triangle loop.

Housekeeping

There are a few commands in Crop that are useful for housekeeping and debugging. These commands take no arguments:

- `pop` pops the top element off the stack and discards it;
- `dup` pops the top element off the stack and pushes it back on twice, effectively duplicating the top-of-stack element;
- `printDict` prints the complete current dictionary to the output;
- `printStack` prints the current stack to the output; and
- `// comment` makes a comment line.

Anything after a pair of double slashes is considered a comment until the end of the line.

This wraps up the Crop language as it stands today. Crop is a completely phenomenological language,

designed to match the formations that I've looked at and tried to replicate compactly. I've tried to keep it as simple and small as possible, while also remaining legible and easy to understand. I encourage readers to extend the language if they think of other commands that are simple and useful.

Strictly speaking, the language I've described here should be referred to as Crop 1.1. The first version of Crop appears in my book *Morphs, Mallards, and Montages: Computer-Aided Imagination* (AK Peters, 2004). Both versions work perfectly well, but some of the commands described here are a bit more refined than those in the book, resulting in a more consistent and compact notation.

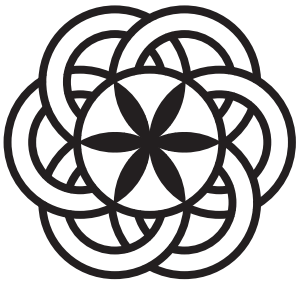
Crop examples

Let's look at some examples of the Crop language as applied to formations actually found in the field. Figures 5 through 11 show some images of actual crop formations, and the Crop code that describes them.

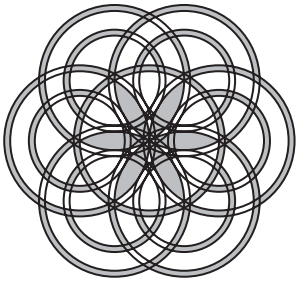
These examples are based on formations that were actually observed (and photographed), but for copyright purposes, I didn't use photographs of those formations. Instead, I used Adobe Photoshop to create simulations of formations. Since I could, and I thought it would be fun, I used a variety of different media in which to form the patterns.

These examples wrap up our discussion of the Crop language. Next time I'll talk about how I used this system to actually create a formation in the real world. ■

Readers may contact Andrew Glassner at andrew@glassner.com.



(a)



(b)

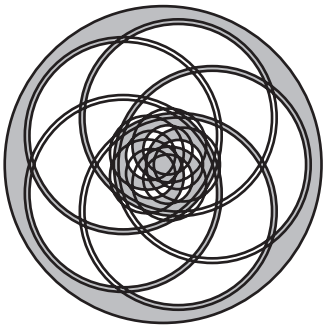
```
1.0 Ar name
1.08565 Br name
0.732051 Cr name
0.646402 Dr name
[ V0 < Ar Br Cr Dr > circle ] # 6 Ar 0
 ngonloop
# < Ar Br > circle
```

(c)

(d)

6 (a) Tegdown Hill 2003 formation. (b) Schematic of (a). (c) Crop code for (a). (d) Constructed formation.

(a)



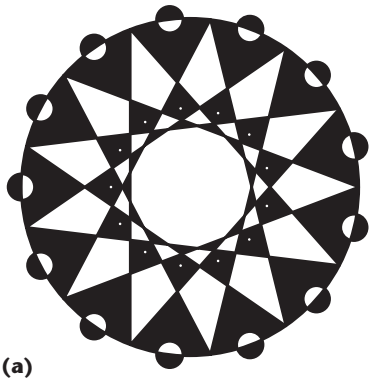
(b)

```
1 Ar name
Ar 5 * 64 / Br name
Ar 30 * 64 / Cr name
Ar 64 / Dr name
# < Ar > circle
[ < V0 V5 > line ] # 13 Ar 0
 ngonloop
[ V0 < Br > circle ] # 13 Ar %
 ngonloop
[ V0 < Dr > circle ] # 13 Cr %
 ngonloop
```

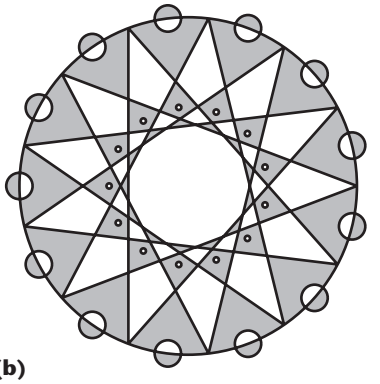
(c)

(d)

7 (a) Windmill Hill 2003 formation. (b) Schematic of (a). (c) Crop code for (a). (d) Constructed formation.



(a)



(b)

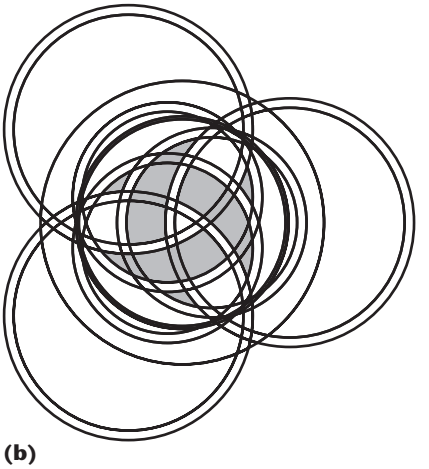
8 (a) West Stowell 2003 formation. (b) Schematic of (a). (c) Crop code for (a). (d) Constructed formation.

```
[ V0 < 2.89 3.21 > circle ] # 3 1 0
 ngonloop
[ V0 < 3.89 4.21 > circle ] # 3 3.6 0
 ngonloop
# < 3.5 3.6 4.8 > circle
```

(c)

(d)

(a)



(b)

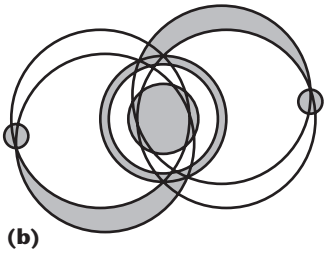
9 (a) Barbury Castle 1999 formation. (b) Schematic of (a). (c) Crop code for (a). (d) Constructed formation.

```
1 Ar name
0.866 Br name
0.555 Cr name
1.41421 Dr name
0.09754 Er name
141.37 Ar * angle1 name //
  (2*pi*360/16) * Ar
# < Ar Br Cr > circle
# < Ar > makeCircle C1 name
[
 V0 < Dr > circle
 V0 < Dr > makeCircle C2 name
 C1 V0 angle1 pspin J name
 J < Dr > circle
 V0 J Dr Dr trope K name
 C2 K Er pwalk L name
 L < Er > circle
] # 2 Ar 0 ngonloop
```

(c)

(d)

(a)



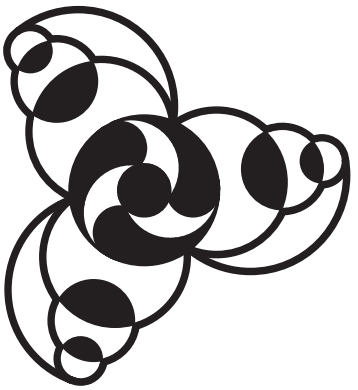
(b)

```
1 Ar name
0.866 Br name
0.555 Cr name
1.41421 Dr name
0.09754 Er name
141.37 Ar * angle1 name //
  (2*pi*360/16) * Ar
# < Ar Br Cr > circle
# < Ar > makeCircle C1 name
[
 V0 < Dr > circle
 V0 < Dr > makeCircle C2 name
 C1 V0 angle1 pspin J name
 J < Dr > circle
 V0 J Dr Dr trope K name
 C2 K Er pwalk L name
 L < Er > circle
] # 2 Ar 0 ngonloop
```

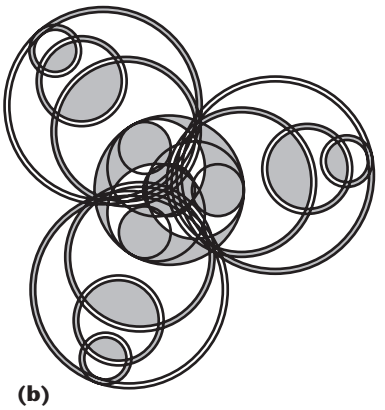
(c)

(d)

10
(a) Sompting 2002 formation. (b) Schematic of (a). (c) Crop code for (a). (d) Constructed formation.



(a)



(b)

```
1 Ar name
0.181332 Br name
0.818668 Cr name
1.06849 s name
# < Br > circle
[
 V0 < Br > circle
 # V0 13 38 trope A name
 A < Ar Ar s * > circle
 # V0 48 73 trope B name
 B < Br Br s * > circle
 # V0 69 94 trope C name
 C < Cr Cr s * > circle
 # V0 31 56 trope D name
 D < Dr Dr s * > circle
] # 3 Cr 0 ngonloop
[
 V0 < Cr > circle
] # 3 Br 0 ngonloop
```

(c)

(d)

11 (a) Ivinghoe Beacon 2002 formation. (b) Schematic of (a). (c) Crop code for (a). (d) Constructed formation.