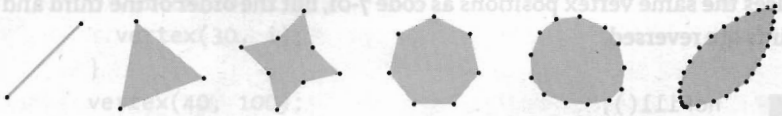
Shape 2: Vertices

This unit focuses on drawing lines and shapes from sequences of vertices.

Syntax introduced:
`beginShape()`, `endShape()`, `vertex()`
`curveVertex()`, `bezierVertex()`

The geometric primitives introduced in Shape 1 provide extraordinary visual potential, but a programmer may often desire more complex shapes. Fortunately, there are many ways to define visual form with software. This unit introduces a way to define shapes as a series of coordinates, called vertices. A vertex is a position defined by an x- and y-coordinate. A line has two vertices, a triangle has three, a quadrilateral has four, and so on. Organic shapes such as blobs or the outline of a leaf are constructed by positioning many vertices in spatial patterns:



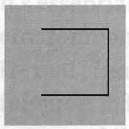
These shapes are simple compared to the possibilities. In contemporary video games, for example, highly realistic characters and environmental elements may be made up of more than 15,000 vertices. They represent more advanced uses of this technique, but they are created using similar principles.

Vertex

To create a shape from vertex points, first use the `beginShape()` function, then specify a series of points with the `vertex()` function and complete the shape with `endShape()`. The `beginShape()` and `endShape()` functions must always be used in pairs. The `vertex()` function has two parameters to define the x-coordinate and y-coordinate:

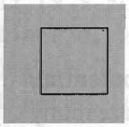
```
vertex(x, y)
```

By default, all shapes drawn with the `vertex()` function are filled white and have a black outline connecting all points except the first and last. The `fill()`, `stroke()`, `noFill()`, `noStroke()`, and `strokeWeight()` functions control the attributes of shapes drawn with the `vertex()` function, just as they do for those drawn with the shape functions discussed in Shape 1 (p. 23). To close the shape, use the `CLOSE` constant as a parameter for `endShape()`.



```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

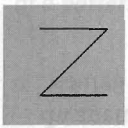
7-01



```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape(CLOSE);
```

7-02

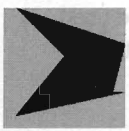
The order of the vertex positions changes the way the shape is drawn. The following example uses the same vertex positions as code 7-01, but the order of the third and fourth points are reversed.



```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(30, 75);
vertex(85, 75);
endShape();
```

7-03

Adding more vertex points reveals more of the potential of these functions. The following examples show variations of turning off the fill and stroke attributes and embedding `vertex()` functions within a `for` structure.

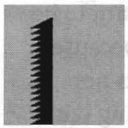


```
fill(0);
noStroke();
smooth();
beginShape();
vertex(10, 0);
vertex(100, 30);
vertex(90, 70);
vertex(100, 70);
vertex(10, 90);
vertex(50, 40);
endShape();
```

7-04

7-05

```
noFill();
smooth();
strokeWeight(20);
beginShape();
vertex(52, 29);
vertex(74, 35);
vertex(60, 52);
vertex(61, 75);
vertex(40, 69);
vertex(19, 75);
endShape();
```



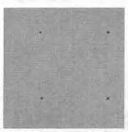
7-06

```
noStroke();
fill(0);
beginShape();
vertex(40, 10);
for (int i = 20; i <= 100; i += 5) {
  vertex(20, i);
  vertex(30, i);
}
vertex(40, 100);
endShape();
```

A shape can have thousands of vertex points, but drawing too many points can slow down your programs.

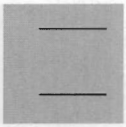
Points, Lines

The `beginShape()` function can accept different parameters to define what to draw from the vertex data. The same points can be used to create a series of points, an unfilled shape, or a continuous line. The parameters `POINTS` and `LINES` are used to create different configurations of points and lines from the coordinates defined in the `vertex()` functions. Remember to type these parameters in uppercase letters because Processing is case-sensitive (p. 20).



7-07

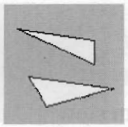
```
// Draws a point at each vertex
beginShape(POINTS);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```



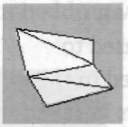
```
// Draws a line between each pair of vertices
beginShape(LINES);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

Shapes

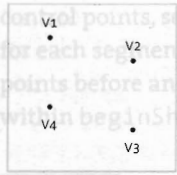
Use the parameters TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN, QUADS, and QUAD_STRIP with beginShape() to create other kinds of shapes. It's important to be aware of the spatial order of the vertex points when using these parameters because they affect how a shape is rendered. If the order required for each parameter is not followed, the expected shape will not draw. It's easy to change between working with TRIANGLES and a TRIANGLE_STRIP because the vertices can remain in the same spatial order, but this is not the case for changing between QUADS and a QUAD_STRIP. Refer to the examples below and the facing diagram for more information.



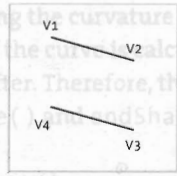
```
// Connects each grouping of three vertices as a triangle
beginShape(TRIANGLES);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```



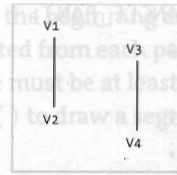
```
// Starting with the third vertex, connects each
// subsequent vertex to the previous two
beginShape(TRIANGLE_STRIP);
vertex(75, 30);
vertex(10, 20);
vertex(75, 50);
vertex(20, 60);
vertex(90, 70);
vertex(35, 85);
endShape();
```



POINTS



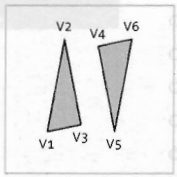
LINES



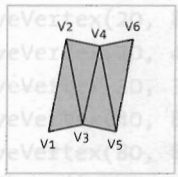
LINES

POINTS, LINES

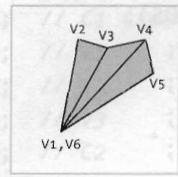
The same data can be interpreted as a sequence of points or lines. The spatial order of the points affects what is drawn when using `LINES`.



TRIANGLES



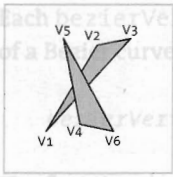
TRIANGLE_STRIP



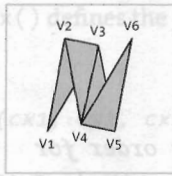
TRIANGLE_FAN

TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP

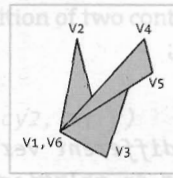
Groups of three vertices are drawn as individual triangles or a connected group.



TRIANGLES

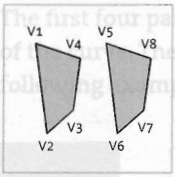


TRIANGLE_STRIP

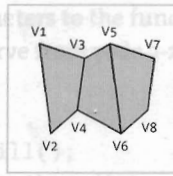


TRIANGLE_FAN

Unexpected results occur if the defined order is not followed.



QUADS

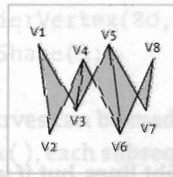


QUAD_STRIP

QUADS, QUAD_STRIP

Groups of four vertices are drawn as individual quads or a connected group. The spatial order determines whether a quad or a "bow" is drawn. Note that the order is reversed for `QUADS` *and* `QUAD_STRIP`.

QUADS



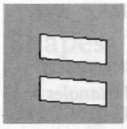
QUAD_STRIP

Parameters for `beginShape()`

There are eight options for the `MODE` parameter of the `beginShape()` function, and each interprets vertex data in a different way. The notation V1, V2, V3, etc., represents the order and position of each vertex point.

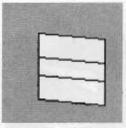
```
beginShape(TRIANGLE_FAN);
vertex(10, 20);
vertex(75, 30);
vertex(75, 50);
vertex(90, 70);
vertex(10, 20);
endShape();
```

7-11



```
beginShape(QUADS);
vertex(30, 25);
vertex(85, 30);
vertex(85, 50);
vertex(30, 45);
vertex(30, 60);
vertex(85, 65);
vertex(85, 85);
vertex(30, 80);
endShape();
```

7-12



```
// Notice the different vertex order for
// this example in relation to example 7-12
beginShape(QUAD_STRIP);
vertex(30, 25);
vertex(85, 30);
vertex(30, 45);
vertex(85, 50);
vertex(30, 60);
vertex(85, 65);
vertex(30, 80);
vertex(85, 85);
endShape();
```

7-13

Curves

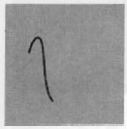
The `vertex()` function works well for drawing straight lines, but if you want to create shapes made of curves, the two functions `curveVertex()` and `bezierVertex()` can be used to connect points with curves. These functions can be run between `beginShape()` and `endShape()` only when `beginShape()` has no parameter.

The `curveVertex()` function is used to set a series of points that connect with a curve. It has two parameters that set the x-coordinate and y-coordinate of the vertex.

```
curveVertex(x, y)
```

The first and last `curveVertex()` within a `beginShape()` and `endShape()` act as

control points, setting the curvature for the beginning and end of the line. The curvature for each segment of the curve is calculated from each pair of points in consideration of points before and after. Therefore, there must be at least four `curveVertex()` functions within `beginShape()` and `endShape()` to draw a segment.



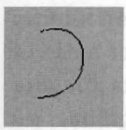
7-14

```
smooth();
noFill();
beginShape();
curveVertex(20, 80);  // C1 (see p.76)
curveVertex(20, 40);  // V1
curveVertex(30, 30);  // V2
curveVertex(40, 80);  // V3
curveVertex(80, 80);  // C2
endShape();
```

Each `bezierVertex()` defines the position of two control points and one anchor point of a Bézier curve:

```
bezierVertex(cx1, cy1, cx2, cy2, x, y)
```

The first time `bezierVertex()` is used within `beginShape()`, it must be prefaced with `vertex()` to set the first anchor point. The line is drawn between the point defined by `vertex()` and the point defined by the x and y parameters to `bezierVertex()`. The first four parameters to the function position the control points to define the shape of the curve. The curve from code 2-21 (p. 30) was converted to this technique to yield the following example:



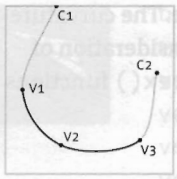
7-15

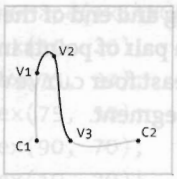
```
noFill();
beginShape();
vertex(32, 20);                          // V1 (see p.76)
bezierVertex(80, 5,  80, 75,  30, 75);   // C1, C2, V2
endShape();
```

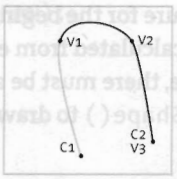
Long, continuous curves can be made with `bezierVertex()`. After the first `vertex()` and `bezierVertex()`, each subsequent call to the function continues the shape by connecting each new point to the previous point.

7-16

```
smooth();
noFill();
beginShape();
vertex(15, 30);                              // V1 (see p.76)
bezierVertex(20, -5, 70, 5, 40, 35);         // C1, C2, V2
bezierVertex(5, 70, 45, 105, 70, 70);        // C3, C4, V3
endShape();
```

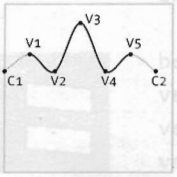


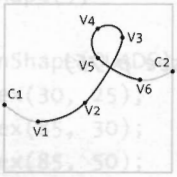


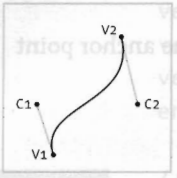


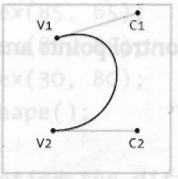
Curve vertices

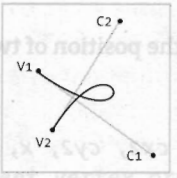
The `curveVertex()` function defines coordinates that are connected with curved shapes. The first and last points are control points that define the shape of the curve at the end and beginning.





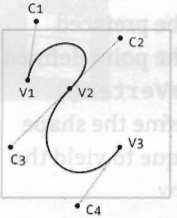


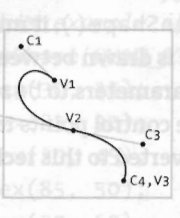


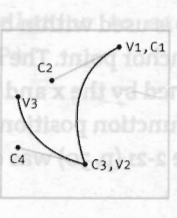


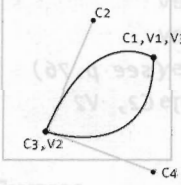
Bézier vertices

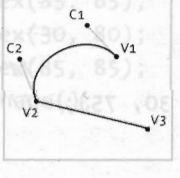
Bézier curves are defined by vertex points and control points used as parameters to the `bezierVertex()` function. The control points define the shape of the curves that are drawn between the vertex points.

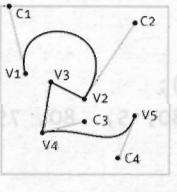












Curves

These curves are converted to software with the `vertex()`, `curveVertex()`, and `bezierVertex()` functions. The notation V0, V1, V2, etc., represents the order and position of each vertex point, and the notation C1, C2, C3, etc., represents the control points. Some of these curves are translated to software in codes 7-14 to 7-18.

To make a sharp turn, use the same position to specify the vertex and the following control point. To close the shape, use the same position to specify the first and last vertex.

7-17

```
smooth();
noStroke();
beginShape();
vertex(90, 39);                          // V1 (see p.76)
bezierVertex(90, 39, 54, 17, 26, 83);    // C1, C2, V2
bezierVertex(26, 83, 90, 107, 90, 39);   // C3, C4, V3
endShape();
```

Place the `vertex()` function within `bezierVertex()` functions to break the sequence of curves and draw a straight line.

7-18

```
smooth();
noFill();
beginShape();
vertex(15, 40);                          // V1 (see p.76)
bezierVertex(5, 0, 80, 0, 50, 55);       // C1, C2, V2
vertex(30, 45);                          // V3
vertex(25, 75);                          // V4
bezierVertex(50, 70, 75, 90, 80, 70);    // C3, C4, V5
endShape();
```

A good technique for creating complex shapes with `beginShape()` and `endShape()` is to draw them first in a vector drawing program such as Inkscape or Illustrator. The coordinates can be read as numbers in this environment and then used in Processing. Another strategy for drawing intricate shapes is to create them in a vector-drawing program and then import the coordinates as a file. Processing includes a simple library for reading SVG files. Other libraries that support more formats and greater complexity can be found on the Processing website at *www.processing.org/reference/libraries.*

Exercises

1. *Use `beginShape()` to draw a shape of your own design.*
2. *Use different parameters for `beginShape()` to change the way a series of vertices are drawn.*
3. *Draw a complex curved shape of your own design using `bezierVertex()`.*