This final article in the "Data visualization with Processing" series explores some of Processing's more advanced features, starting with an introduction to 2-D and 3-D graphics and lighting features. Then explore physics applications with graphical visualization, learn about Processing's networking features, and develop a simple application that visualizes data from the Internet.

Part 1 and Part 2 explore many of the capabilities of the Processing language, from basic 2-D graphics and text to image processing. This final installment ties up the loose ends and looks at how Processing handles 3-D graphics, lighting, and networking. The networking discussion provides a way to give your Processing application some useful data to work with for visualization.

Let's begin with a discussion of transformations and the various Processing capabilities that provide them.

Transformations

If you've developed graphical applications, you're probably familiar with linear algebra and the matrix operations required to implement things like translation, scaling, and rotation. Processing provides functions to simplify operations such as these with a single function call, hiding the matrix math that occurs under the covers.

Coordinate system translation

Translation simply changes the coordinate system so that the upper-left corner exists at a new offset. It is an additive operation in that the translation defines the offset for new operations within the display window. For example, the code in Listing 1 draws a square using the rect function, then specifies a change to the coordinate system using translate. Any new shapes drawn after the translate function appears using the offset; so in this case, the rect function actually creates a square at 100, 100, 150, 150, as shown in Figure 1, cell A.
**Listing 1. Using `translate()` to alter the coordinate system**

```java
size(200, 200);

rect(0, 0, 50, 50);
translate(100, 100);
rect(0, 0, 50, 50);
```

**Scaling**

The `scale` function, as the name implies, magnifies drawing operations. For example, if you request a scale of 2.0, your object will be 200 percent larger than the original (in each dimension). Listing 2 illustrates this concept, with the resulting output shown in Figure 1, cell B.

**Listing 2. Using `scale()` to magnify shape draw operations**

```java
size(200, 200);

rect(0, 0, 50, 50);
translate(75, 75);
scale(2.0);
rect(0, 0, 50, 50);
```

**Rotation**

The `rotate` function rotates a shape around the upper-left corner (the 0,0 coordinate). For this reason, rotating a shape doesn't actually rotate in place. Listing 3 shows the rotation operation as applied to a square. You draw your square first, then specify the rotation operation that is applied to the subsequent shapes drawn to the display window. The result is shown in Figure 1, cell C.

**Listing 3. Using `rotate()` to rotate an object**

```java
size(200, 200);

rect(100, 100, 50, 50);
rotate(PI/16);
rect(100, 100, 50, 50);
```

If you wanted to rotate the object in place, you would need to update the coordinate system to account for this. To do this, use `translate` to adjust the coordinate system, then redraw your rotated square (with updated \(x\) and \(y\) coordinates), as shown in Listing 4. The result is shown in Figure 1, cell D.

**Listing 4. Using `translate()` and `rotate()`**

```java
size(200, 200);

rect(100, 100, 50, 50);
translate(100, 100);
rotate(PI/16);
rect(0, 0, 50, 50);
```
The code in Listing 4 didn't actually rotate the object in place around its center, but around its upper-left corner. You can rotate your square by defining how you draw your object. The default draw mode for rectangles is **CORNERS**, which means that the upper-left corner is defined by the $x$ and $y$ parameters of `rect`. If you change the mode to **CENTER** (where the $x$ and $y$ parameters of `rect` define the center), you can draw your square and rotate it around its center. This is shown in Listing 5, with the result shown in Figure 1, cell E.

### Listing 5. Using `translate()` and `rotate()` on centers

```java
size(200, 200);
rectMode(CENTER);
rect(100, 100, 50, 50);
translate(100, 100);
rotate(PI/16);
rect(0, 0, 50, 50);
```

Now, apply these concepts to build an image reminiscent of those of the Spirograph. In this example, you rotate a square on its center, but do not fill the square. The square is rotated 16 times around the center, with the result shown in Figure 1, cell F (with the simple application shown in Listing 6).

### Listing 6. More fun with translation and rotation

```java
size(200, 200);
rectMode(CENTER);
noFill();
translate(100, 100);
for (int i = 1 ; i < 16 ; i++) {
  rotate( (PI/16)*i );
  rect(0, 0, 100, 100);
}
```

Processing hides the complexity of the matrix operations and instead presents a simple set of functions with useful graphical operations such as scaling and rotation.

### Figure 1. Images resulting from the code in Listings 1-6
Managing transformations
With the basic transformations covered, let's explore one additional topic that’s useful when you draw multiple transformations and objects. Recall that once a transformation function is called, subsequent shape draws use that transformation. Additional transformations can be performed, and these are applied to the current set of matrices defined by the previous operations. For example, calling translate within the context of another translate means that the second translate is based on the coordinate system of the first translate, not the original. You can save the internal coordinate matrices through a call to pushMatrix, then restore them using popMatrix. As these functions use stack semantics, you can layer transformations for shapes and later remove them to restore the original matrices.

As an example, Listing 7 shows pushMatrix and popMatrix. After some initialization, you push the current coordinate matrices to the stack. Then, you create a new coordinate system that’s centered within your display using translate and store this matrix. Next, rotate and store this matrix; then, offset from the current coordinate system with translate (which is applied from the previously rotated and translated matrices), and draw a red ellipse. Pop the stack and draw a green rectangle to remove the previous translation and scaling. Pop again and draw a blue rectangle (which no longer includes the rotation transformation). Then, pop back to the original matrix and draw the gray square. This scheme permits the layering of transformations on top of one another while providing the ability to restore previously used matrices for other objects in the display. Note that the indentation shown in Listing 7 is there only for readability and is not required.

Working directly with matrices
If you need to work with the matrices directly, you can use printMatrix to view the current matrix, applyMatrix to multiply the current matrix by a user-defined one, and resetMatrix to reset the current matrix to the identify matrix.

Listing 7. Saving and restoring matrices
```java
size(200, 200);
rectMode(CENTER);
noFill();
smooth();
strokeWeight(2);
colorMode(RGB, 100);
pushMatrix();
   translate(100, 100);
pushMatrix();
   rotate(PI/4);
pushMatrix();
   translate(20, 20);
   scale(2.0);
   stroke(255, 0, 0); // Red
   ellipse(0, 0, 50, 10);
popMatrix();
   stroke(0, 255, 0); // Green
   rect(0, 0, 50, 25);
popMatrix();
   stroke(0, 0, 255); // Blue
```
The result of the Listing 7 code is shown in Figure 2.

**Figure 2. Demonstrating pushMatrix and popMatrix**

![Diagram showing pushMatrix and popMatrix effects](image)

### 3-D graphics

Now, let's extend visualization into the third dimension (on a 2-D plane) by exploring some of the APIs that Processing provides for 3-D. Be sure to see Resources to dig deeper into the gory details.

The example shown in Listing 8 provides a simple 3-D program that uses the `box` function to create an object in the display window. As the name suggests, the `box` function creates a box in the display (as shown, with equal dimensions). It's possible to create rectangles with `box` just by adding a few parameters (width, height, and depth).

The only other new elements here are the `rotate` functions. These functions allow you to rotate around a given axis. The first box rotates around the $y$-axis, and the second box rotates around the $x$-axis. The parameter for the rotation is determined from the mouse for the particular axis. The mouse value is read, then mapped (via the `map` function) into the range $-\pi$ to $\pi$. 

```java
rect(0, 0, 75, 50);
popMatrix();
stroke(128, 128, 128); // Gray
rect(0, 0, 50, 50);
```
Listing 8. Simple 3-D example with mouse-based rotation

```java
void setup() {
  size(200, 200, P3D);
  noFill();
  smooth();
}

void draw() {
  background(0);
  translate(width/2, height/2, -(width/2));
  rotateY(map(mouseX, 0, width, -PI, PI));
  stroke(100);
  box(150);
  rotateX(map(mouseY, 0, height, -PI, PI));
  stroke(150);
  box(75);
}
```

The output from Listing 8 (at least a static portion of it) is shown in Figure 3.

**Figure 3. Static output from Listing 8**

It's also possible to add lighting, and Processing provides a set of functions that make doing so simple. I demonstrate one here with another 3-D object construction function, and then discuss some of the others. Like the `box` function, you can use `sphere` to create a 3-D sphere in the display window. The argument to `sphere` represents the radius of the sphere.

Listing 9 demonstrates the `pointLight` function. This function creates a light source defined by the latter three arguments of `pointLight` (its location in the space). The first three arguments define the color of the light.

**Listing 9. Sphere and light**

```java
size(100, 100, P3D);
background(0);
noStroke();
pointLight(50, 100, 180, 80, 20, 40);
translate(20, 50, 0);
sphere(40);
```

Figure 4 provides the output of the simple Processing application in Listing 9.
Figure 4. Demonstration of pointLight

The `pointLight` function provides one of the simpler light functions (other than perhaps `ambientLight`), but Processing provides more control over lighting. The `spotLight` function adds a spotlight with considerable controls. For example, in addition to the location, direction, and color of the light, it's possible to control the angle of the `spotlight` cone and the center bias of the cone. The `directionalLight` function permits light to be focused in a particular direction and provides more natural lighting that changes based on the direction and angle of the light.

Processing provides much more complex means of constructing objects using vertices. Shapes can be constructed in this way, with the ability to texturize them. Check out Resources for details on this aspect of Processing.

Physics

The previous 3-D examples used static models, but if you want some projectile physics, you can extend this example into real time. Listing 10 provides a simple simulation of box being shot from a cannon. The kinematics equations are outside the scope of this article, but check out Resources for more information.

The `setup` function creates the display window and initializes the starting position of the box (the origin). In draw, you use `background` to clear the display, and then calculate the direction cosines for the orientation of the cannon. Next, given your time variable, you calculate the position of the box within the display window along the three dimensions. Just to add some interesting effects, rotate the box around the x- and y-axis at different rates.

You use the `pointLight` function to create two light sources. On the left, place a blue light; on the right, place a red light. You then emit the current location of the box in the 3-D space to the console for debugging.

Finally, place the box in the display. Begin by pushing the current set of matrices onto the stack, then call `translate` to change the displacement for your box. Doing so allows you to place x slightly to the right in the window, invert the y-axis (because in the Processing coordinate system, y grows down), and finally to project z outward (also by inverting it). Recall that in Processing's coordinate system, x grows to the right, y grows down, and -z grows away. Finally, apply your x and y rotations, place your box in the display, and pop the current set of matrices.
Listing 10. Projectile simulation with Processing

```java
float x, y, z;          // Current Position
float velocity = 120.0; // Muzzle velocity;
float alpha = 30.0;     // Angle from y-axis
float gamma = 60.0;     // Angle from x-axis
float g = 9.8;          // Acceleration due to gravity (m/s^2)
float time = 0.0;
float dt = 0.1;
float rotX = 0.0, rotY = 0.0;

void setup() {
    size(300, 400, P3D);
    smooth();
x = 0.0; y = 0.0; z = 0.0;
}

void draw() {
    float b, Lx, Ly, Lz;
    time += dt;
    background(0);
    // Calculate cosines for the cannon orientation (static)
b = cos((90.0-alpha) * 3.14/180.0);
Lx = b * cos(gamma * 3.14/180.0);
Ly = cos(alpha * 3.14/180.0);
Lz = b * sin(gamma * 3.14/180.0);
    // Calculate the position of the box at the given time
x = velocity * Lx * time;
y = (cos(alpha*3.14/180.0)) + (velocity * Ly * time) -
(0.5 * g * time * time);
z = velocity * Lz * time;
    // Rotate the box around the x- and y-axis.
rotX += PI/256.0;
rotY += PI/128.0;
    // Create two light sources (one blue and one red)
pointLight(0, 100, 255, 0, 0, 0);
pointLight(255, 0, 0, 400, 400, 0);
println("x " + x + " y " + y + " z " + z");
    // Place the box in the display
pushMatrix();
    translate(100, 400-y, -z);
    rotateX(rotX); rotateY(rotY);
    box(90);
    popMatrix();
}
```

With this small amount of code, you've implemented a simple simulation of a rotating projectile. Time steps of this simulation are shown in Figure 5, although the real-time visualization is much more interesting.
Networking

Developing networking applications in Processing is similar to the approaches that Ruby and Python take compared to the standard API in C (the Berkeley Socket API). The networking functions are implemented in a library and present two classes for application development. The first is the `Server` class, which you use to create servers; the second is the `Client` class, which you use to create clients. There's also a set of events methods that you use as an aid for application development. For example, the server event is a callback method that is invoked when a new client connects to the server. The client event is a callback method that is invoked when the server has sent bytes to the client.

Listing 11 shows a client that communicates with a peer web server for the purpose of identifying its server software. This is done simply with the HTTP `HEAD` method, which returns meta-information about the client's request. For example, clients typically issue an HTTP `GET` request to return a file from a web server. The `HEAD` request returns no HTTP message body but rather the request file's meta-information (size, etc.). This is the simplest way to gather the information that you're interested in.

Listing 11 begins by creating a new client class. You define the server and port to which you want to connect (in this case, the web server port for IBM). When you have a connection to the web server, send the HTTP request using the `write` method. In the `draw` method, check to see whether any bytes are available to read; if there are, tokenize them into an array. Then, you search the array for the field in the HTTP response for the server type (which looks like `Server: Apache`). When you find the server field, emit the next token, which will be the server string itself (in this example, `IBM_HTTP_Server`).

Figure 5. Portions of the output from Listing 10
Listing 11. A simple client socket example

```
import processing.net.*;
Client myClient;
String inString;

void setup() {
    myClient = new Client(this, "www.ibm.com", 80);
    myClient.write("HEAD / HTTP/1.0
\n");
}

void draw() {
    if (myClient.available() > 0) {
        inString = myClient.readString();
        String[] tokens = splitTokens(inString, ":\n ");
        for (int i = 0; i < tokens.length-1; i++) {
            if (tokens[i].equals("Server")) println(tokens[i+1]);
        }
    }
}
```

The `Server` class provides a similar set of methods in addition to `disconnect`, which disconnects a particular client, and `stop`, which disconnects all clients and stops the server. See the Processing reference (see Resources) for more details on this API.

Building a networking mashup

Let's look at another client socket example that takes data from the Internet and displays it. For the example, you'll incorporate ideas from other topics you've learned about in this final article to visualize data from a web service. You'll use your socket client to attach to the Yahoo! Traffic REST API and extract traffic information for a given location (see Listing 12). After you've connected to the remote server (part of the instantiation of the `Client` class), you write your request. This is a REST request and specifies the service with which you want to communicate (Yahoo!'s MapsService) along with the location information that you're interested in (Sunnyvale, Calif.). With your request complete, you create your display and load your font.

The `draw` function has two section of code. The first is a receipt of the HTTP response (which appears first). In this section, you receive characters as they're available, then tokenize the current result to search for the `Title` token (actually `<Title>`, but you've stripped the brackets as part of tokenization). When the `Title` token is found, the next string will be the title of the traffic incident.

The next section emits the traffic incident string to the display, if it has been received (which you know, because it has a non-zero length). As with the previous 3-D example, you apply scaling and rotation to the output to make it a bit more interesting. The result is a string that slowly rotates away to infinity.

Listing 12. Simple client to extract traffic incident data

```
import processing.net.*;
Client myClient;
String input="";
String displayString="";
float myScale=1.0, myRatioX = 0.0, myRatioY = 0.0;

void setup() {
    myClient = new Client(this, "local.yahooapis.com", 80);
```
void draw() {
  background(100);

  if (myClient.available() > 0) {
    input += myClient.readString();
    String tokens[] = splitTokens(input, "<\n");
    for (int i = 0 ; i < tokens.length ; i++) {
      if (tokens[i].equals("Title")) {
        displayString = tokens[i+1];
      }
    }
  }

  if ((displayString.length() > 0) && (myScale > 0)) {
    myRatioX += (PI/300);
    myRatioY += (PI/460);
    myScale -= 0.005;
    scale(myScale);
    pushMatrix();
    translate( 50, height/2, 0);
    rotateX(myRatioX);
    rotateY(myRatioY);
    rotateZ(myRatioY/4);
    text(displayString, 0, 0);
    popMatrix();
  }
}

The result of Listing 12 is shown in Figure 6. This image was taken 23 frames in, so it's just beginning its movement.

**Figure 6. Output from Listing 12**

**Going further**

Processing is not only a great language and environment for visualization but a great example of what's possible with open source technology. Processing is used not only by engineers and
Data visualization with Processing, Part 3: 2-D, 3-D, physics, and networking

scientists for data visualization but also by artists and those interested in learning programming and visual design.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)