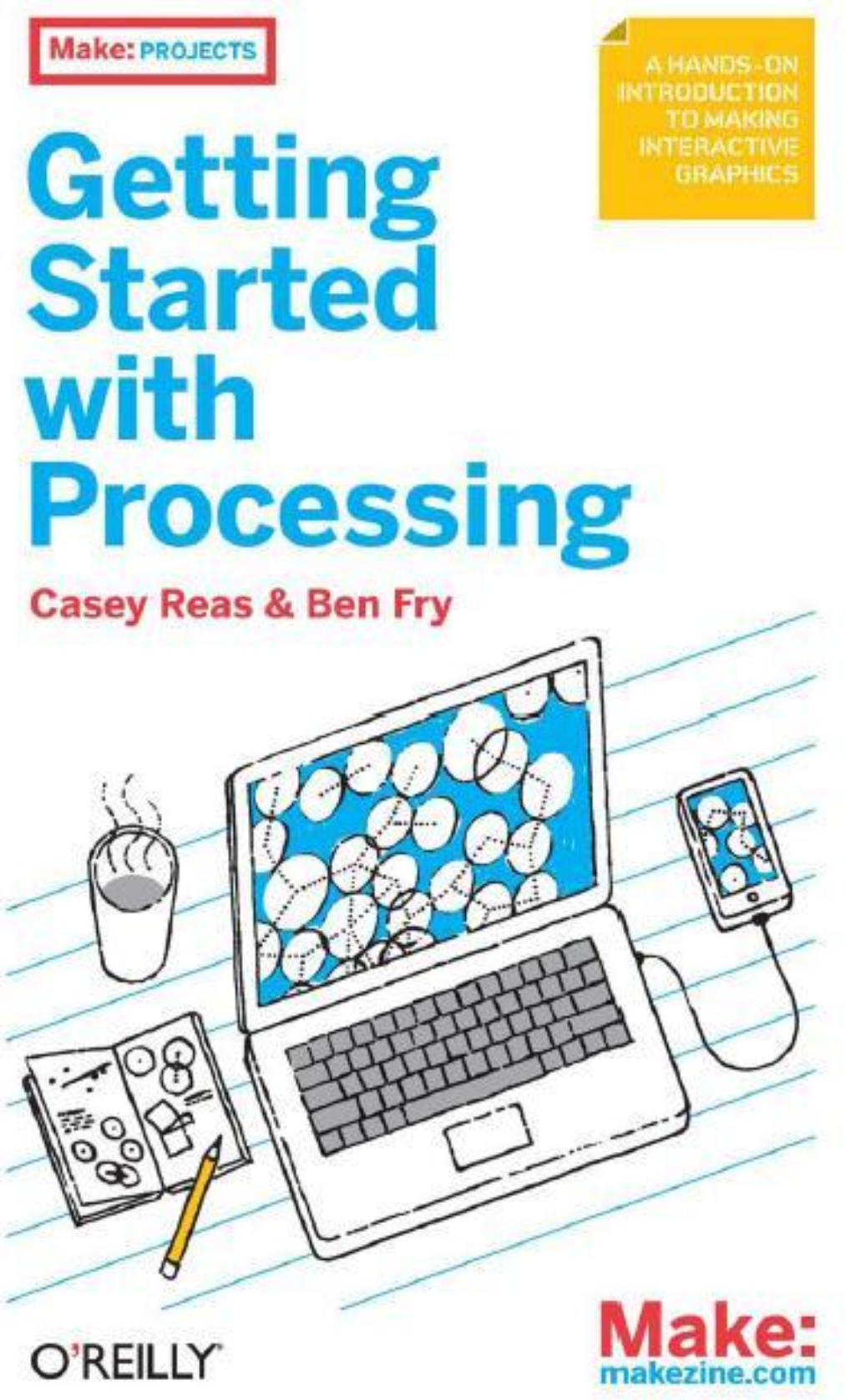


Guía de inicio a la programación en Processing.
Traducción al libro Getting started width processing
Traducción realizada por Ana maría Ospina y Johnny Alexander Sepúlveda

Capitulo 1

1 Hola

Processing es un software para escribir imágenes, animaciones, e interacciones. La idea es escribir una sola línea de código, y tener un círculo que aparece en la pantalla. Agrega unas cuantas líneas de código, y el círculo ahora sigue el mouse. Otra línea de código, y el círculo cambia de color cuando el mouse es presionado. A esto lo llamamos *dibujar* con código. Escribes una línea, luego agregas otra, luego otra, y así sucesivamente. EL resultado es un programa creado.

Los cursos de programación son enfocados típicamente en la teoría y la estructura. Nada visual; una interface o una animación son considerados un postre para ser disfrutado solo después de terminar los vegetales tras varias semanas de estudio de algoritmos y métodos. A través de los años, hemos visto a varios amigos intentar tomar muchos cursos y abandonarlo después de la lectura o después de un tiempo. La curiosidad inicial que ellos tenían acerca de hacer trabajar el computador para ellos se fue perdiendo, porque no pudieron ver un camino desde lo que tenían que aprender, a lo que ellos querían crear.

Processing ofrece una forma para aprender programación a través de la creación de gráficos interactivos. Existen varias formas posibles para enseñar código, pero a menudo los estudiantes encuentran un estimulo y una motivación a una respuesta visual inmediata. La capacidad de Processing para proporcionar esa realimentación ha hecho una forma popular para aprender programación, y su énfasis en las imágenes, dibujos y la comunidad son discutidos en las siguientes páginas.

Bocetos y Prototipos

Los bocetos son una forma de pensar; son juguetones y rápidos. Los objetivos básicos son explorar varias ideas en un corto tiempo. Usualmente en nuestro propio trabajo, empezamos con el boceto en un papel y luego movemos los resultados a un código. Usualmente las ideas para animación e interacción son dibujadas, las mejores ideas son seleccionadas y combinadas en prototipos (Figura 1-1). Es un proceso cíclico para hacer, probar, y mejorar lo que se mueve adelante y atrás entre el papel y la pantalla.

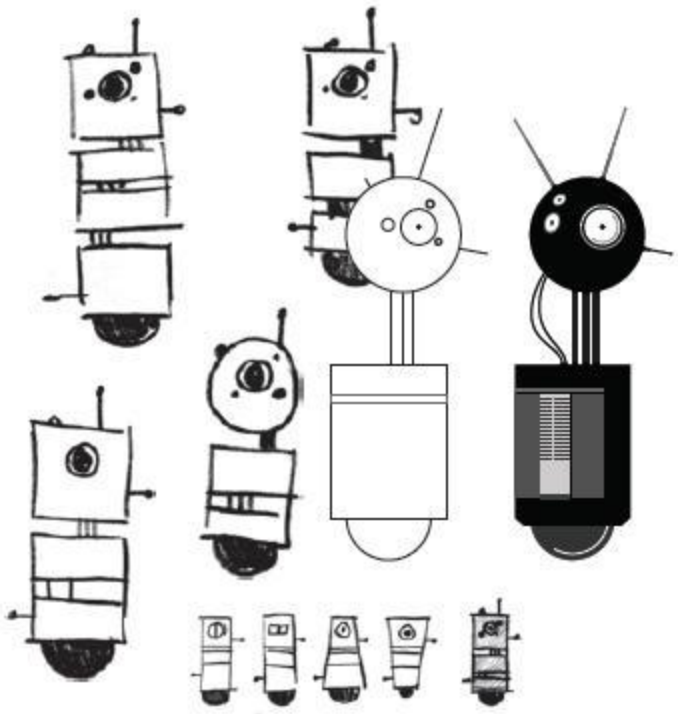


Figura 1-1. Como movimos los dibujos desde el boceto a la pantalla, nuevas posibilidades surgen.

Flexibilidad

Como un cinturón de utilidad en un software. Processing consiste en varias herramientas que trabajan juntas en diferentes combinaciones. Como resultado, puede ser usada para hacks rápidos o para una investigación más profunda. Porque un programa de Processing puede ser una corta línea o miles de líneas. Más de 100 bibliotecas amplían a Processing aún más en los dominios, incluyendo el sonido, la visión computarizada, y la fabricación digital (Figura 1-2).

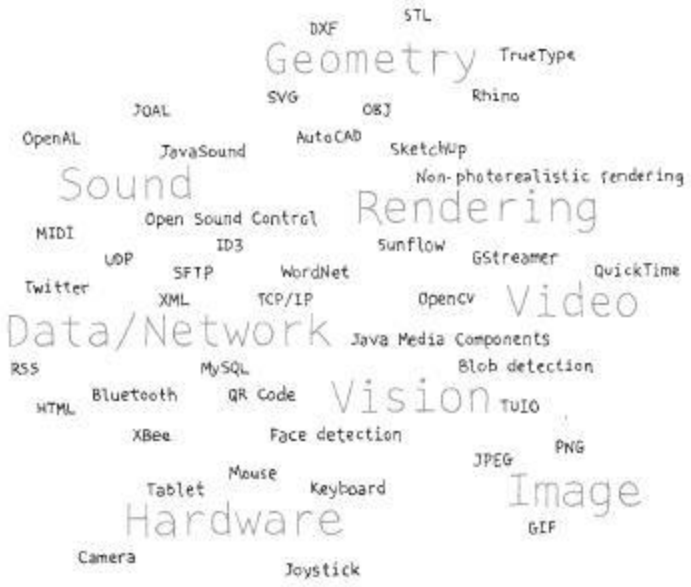


Figura 1-2. Varios tipos de información pueden fluir dentro y fuera de Processing.

Gigantes

La gente ha estado haciendo fotos con computadores desde 1960, y hay más para aprender de esta historia (figura 1-3). En la vida, todos estamos en los hombros de los gigantes, y los titanes para Processing son los grandes pensadores que están desde el diseño, los gráficos por computador, el arte, la arquitectura, la estadística, y los espacios. Échale un vistazo al *Sketchpad* (1963) de Ivan Sutherland, el *Dynabook* (1968) de Alan Kay, y varios artistas invitados en *Artist and computer* (Harmony books, 1976) de Ruth Leavitt. Los archivos ACM SIGGRAPH proporcionan una fascinante vislumbre en la historia del software y los gráficos.

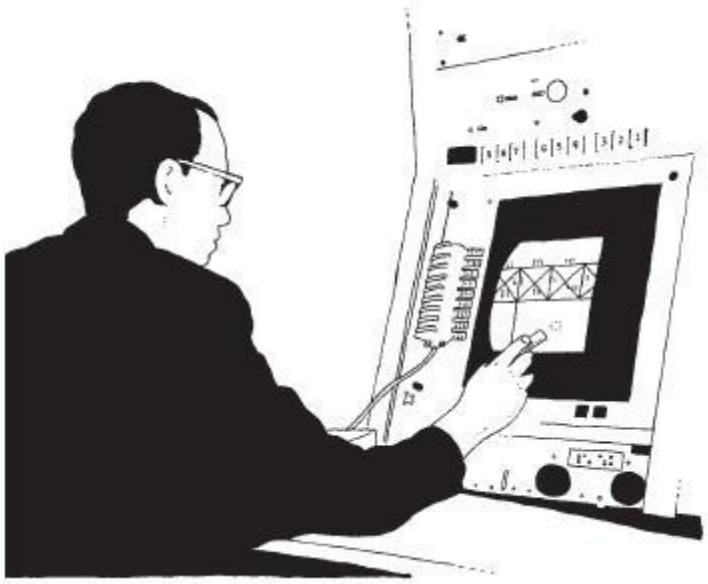


Figura 1-3. Processing fue inspirado por grandes e individuales ideas a lo largo de las últimas cuatro décadas.

El Árbol Familiar

Como los lenguajes humanos, Los lenguajes de programación pertenecen a familias de lenguas parecidas. Processing es un dialecto de un lenguaje de programación llamada Java; La sintaxis del lenguaje es casi idéntica, pero Processing agrega características personales relacionadas con gráficos e interacción (Figura 1-4). Los elementos gráficos de Processing están relacionados con PostScript (una fundación de PDF) y OpenGL (una especificación de gráficos 3D). Debido a estas características comunes, aprender Processing es un paso de nivel de entrada para programar en otros lenguajes y usar diferentes herramientas de software.

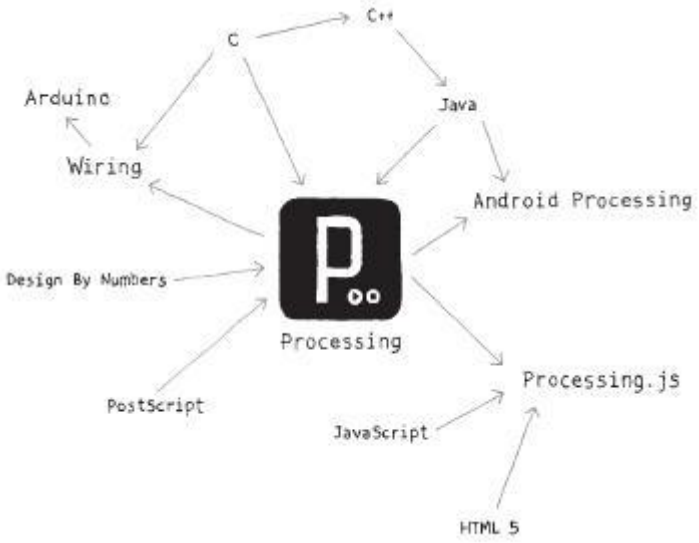


Figura 1-4. Processing tiene una gran familia de lenguajes relacionados y desarrolladores de programación.

Participar

Miles de personas usan Processing cada día. Como ellos, tú puedes descargar Processing sin ningún costo. Incluso tienes la opción de modificar el código de Processing para que se adapte a tus necesidades. Processing es un proyecto *FLOSS* (que es, *free/libre/open source software)*, y en el espíritu de la comunidad te animamos a que participes compartiendo tus proyectos y conocimiento, online en Processing.org y en los varios sitios de redes sociales que hospedan el contenido de Processing (Figura 1-5). Existen sitios vinculados desde el website Processing.org.

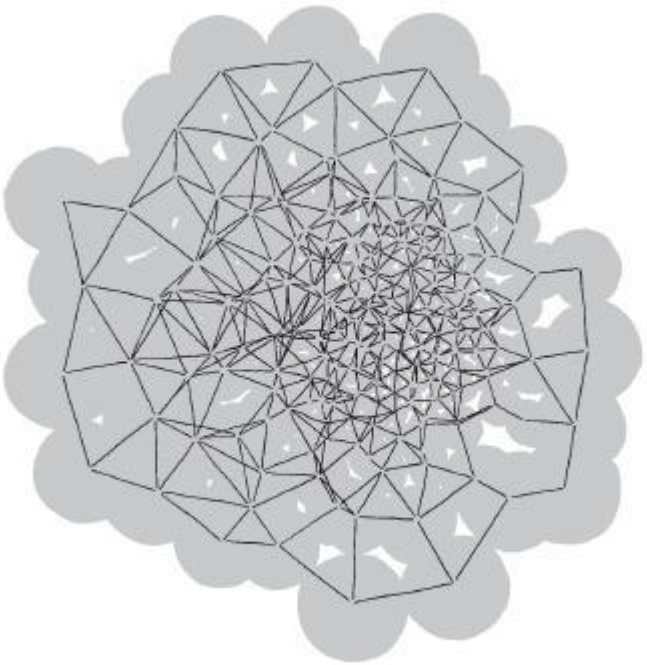


Figura 1-5. Processing es alimentado por miles de personas que contribuyen a través de internet. Esta es nuestra interpretación de cómo se relacionan el uno con el otro.

CAPITULO 2

2 Empezando a codificar

Para sacar el máximo de este libro, necesitas hacer más que sólo leer las palabras. Necesitas experimentar y practicar. No puedes aprender a programar sólo leyendo acerca de ello, necesitas hacerlo. Para empezar, descarga Processing y haz tu primer dibujo.

Empieza visitando http://processing.org/download y selecciona la versión Mac, Windows o Linux, dependiendo de la máquina que tengas. La instalación en cada sistema operativo es sencilla:

>>>> En Windows, tendrás un archivo .zip. Haz doble click, y arrastra la carpeta dentro de una ubicación en tu disco duro. Podría ser *Archivos de programa* o simplemente tu escritorio, pero lo importante es que la carpeta *Processing* sea colocada fuera de ese archivo .zip. Luego das doble click a *processing.exe* para empezar.

>>>> En Mac OSX, la versión esta en un archivo de imagen de disket (.dmg). Arrastra el icono de Processing a la carpeta Applications. Si algunas veces usas ese computador y no es posible modificar la carpeta Applications, solo arrastra la aplicación al escritorio. Entonces haga doble click en el icono de Processing para empezar.

>>>> La versión de Linux es un archivo .tar.gz, que debería ser familiar para los usuarios de Linux. Descarga el archivo en tu directorio Home, entonces abre una ventana de terminal, y escribe:

tar xvfz processing-xxxx.tgz

(Reemplaza xxxx con el resto del nombre del archivo, que es su versión en número.) Cuando dejamos creada la carpeta con el nombre Processing-1.0 o similar. Luego cambiamos ese directorio:

cd processing-xxxx

Y lo corremos:

./processing

Con alguna suerte, la principal ventana de Processing ahora podrá ser visible (Figure 2-1). Todos los setups son diferentes, si el programa no empieza, podrás empezar de otra manera, podrás visitar la página de disparos de problemas para posibles soluciones (foro): *http://wiki.processing.org/index.php/Troubleshooting.*

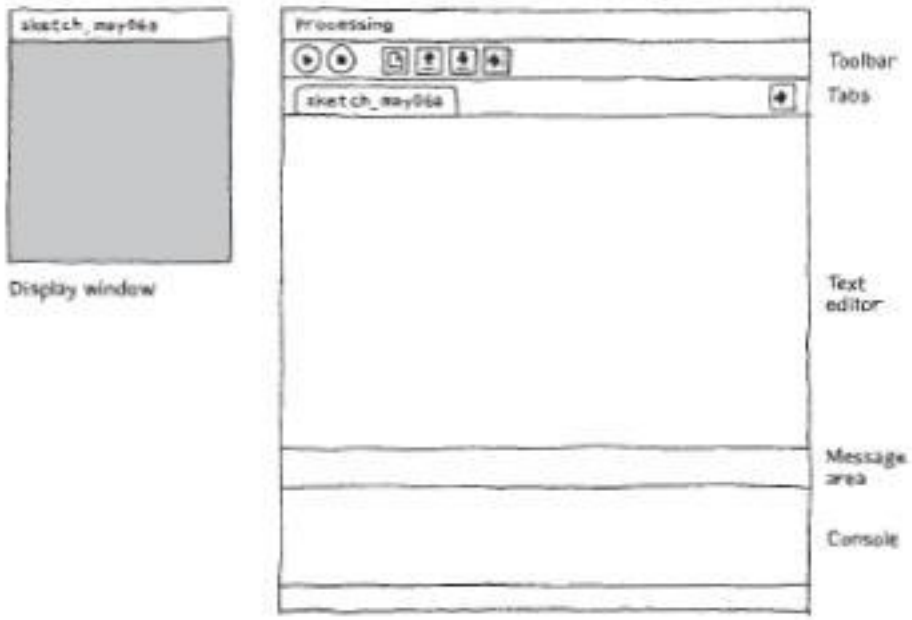


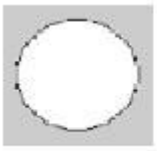
Figura 2-1. El ambiente de desarrollo de Processing.

Tu primer programa

Ahora que el entorno de desarrollo de Processing está corriendo (o PDE). No hay mucho que decir; el área larga es el editor de texto, y hay una fila de botones a través de la parte superior, esta es la barra de herramientas. Por debajo del editor esta el área de mensajes, y por debajo de esta está la consola. El área de mensajes es usada para una línea de mensajes, y la consola es usada para algunos detalles técnicos.

Ejemplo 2-1 Dibujando un círculo.

En el editor, escriba lo siguiente:



```
ellipse(50, 50, 80, 80);
```

Esa línea de código es un método para "dibujar un círculo, con el centro a partir de 50 píxeles por encima desde la izquierda y 50 píxeles desde la parte superior hacia abajo." Haz click en el botón correr, que se ve de la siguiente manera:

Si tienes todo escrito correctamente, podrás ver una imagen de un círculo. Si tu escritura es incorrecta, se mostrará en el área un mensaje de error de color rojo. Si esto sucede, es mejor que copies el código del ejemplo exactamente. Los números deberían estar contenidos dentro de los paréntesis y tener comas dentro de cada uno de ellos, la línea debería finalizar con un punto y coma.

Una de las cosas complicadas para empezar con la programación, es que tienes que estar siempre especificando la sintaxis de la línea. El software Processing no siempre es lo suficientemente inteligente para saber qué quieres decir, y puede ser bastante exigente sobre la colocación de la puntuación. Puedes acostumbrarte con la práctica.

Después, seguiremos adelante con nuestro sketch que se pondrá mas interesante.

Ejemplo 2-2 Haciendo círculos

Borra el texto del ejemplo anterior, y trata de probar éste:

```
void setup() {
 size(480, 120);
 smooth();
}

void draw() {
 if (mousePressed) {
   fill(0);
  } else {
   fill(255);
 }
 ellipse(mouseX, mouseY, 80, 80);
}
```

Este programa crea una ventana de 480 pixeles de ancho y 120 pixeles de alto, y empieza a dibujar círculos en la posición del mouse. Cuando un botón del mouse es presionado, el color del círculo cambia a negro. Explicamos más detalladamente los elementos de este programa. Por ahora, corre el código, mueve el mouse y clickea la experiencia.

Demostración

Hasta ahora solo hemos explicado con el botón de correr, aunque probablemente adivinaremos que hace el botón de stop, que está al lado:

Si no queremos usar estos botones, siempre puedes usar los del menú del sketch, que muestran los atajos Ctrl-R (o Cmd-R en Mac) para correr. Debajo de run en el menú del sketch está Present, que borra el resto de la pantalla para presentarse por sí mismo

Puedes usar también Present desde la barra de herramientas, manteniendo pulsada la tecla Shift y clickeando el botón run.

Guardar

El próximo comando importante es salvar o guardar. Es la flecha que apunta hacia abajo en la barra de herramientas.\

También puedes encontrar bajo el menú File, que por defecto, todos tus programas están guardados en "skecthbook", que es una carpeta que guarda tus programas para un fácil acceso. Clickeando sobre el botón abrir en la barra de herramientas (la flecha apuntando hacia arriba) podremos abrir una lista de todos tus programas (sketches) en tu carpeta sketchbook, también hay una lista con ejemplos instalados en el software de Processing:

A menudo es una buena idea salvar tus sketches, ya que se trata de diferentes cosas, salvarlos con diferentes nombres es ideal para volver a una versión anterior, Esto es especialmente útil cuando algo se rompe. También puedes observar donde está ubicado el skecth en tu disco con la opción Show en la carpeta bajo el Skecth menu.

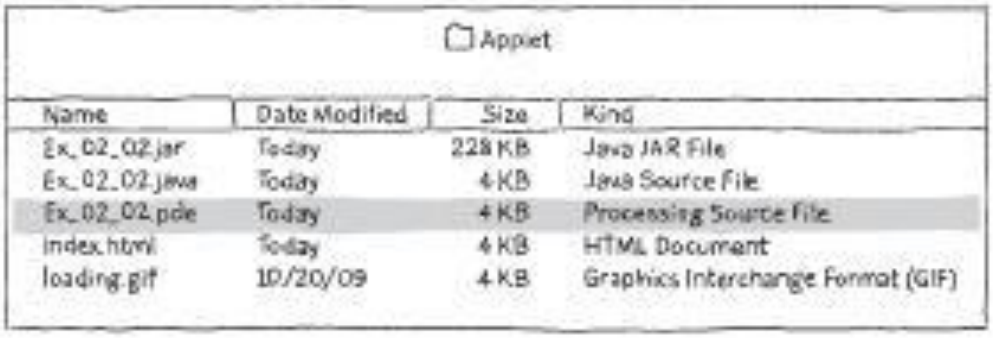
También puedes crear un nuevo skecth presionando el botón New en la barra de herramientas:

Esto reemplazará el sketch presente en la ventana con un skecth limpio. Manteniendo pulsada la tecla Shift, este nuevo botón creará un nuevo skecth en su propia ventana, igualmente podemos hacer esto seleccionando File → new. El botón abre una nueva ventana, trabajando de la misma manera.

Aportando

Otro tema de Processing es compartir tu trabajo. El botón de exportar en la barra de herramientas:

Junta todo tu código es una única carpeta llamada *applet* que se puede descargar en un servidor web (figura 2-2). Después de exportar, la carpeta Applet puede abrirse en el escritorio. El archivo PDE es el código fuente, El archivo JAR es el programa, El archivo HTML es una página web, y el archivo GIF es una muestra en el navegador web cuando el archivo es cargado. Si haces doble click en el archivo .html, esto lanzará tu navegador web y mostrará el skecth que hayas creado en una página web.



Nota: La carpeta applet es borrada y recreada cada vez que se use el comando exportar. Así que es seguro mover la carpeta a otro lugar donde después de tú hagas un cambio en el archivo HTML o cualquier cosa adentro.

También puedes encontrar Export, junto con su hermano Export a aplicación. Por debajo del archivo menu. Exportar a aplicación crea una aplicación para tu opción en Mac, Windows y Linux, esto es una manera fácil de hacer tus propios contenidos, doble click para obtener tus proyectos en diferentes versiones. (Figure 2-3)

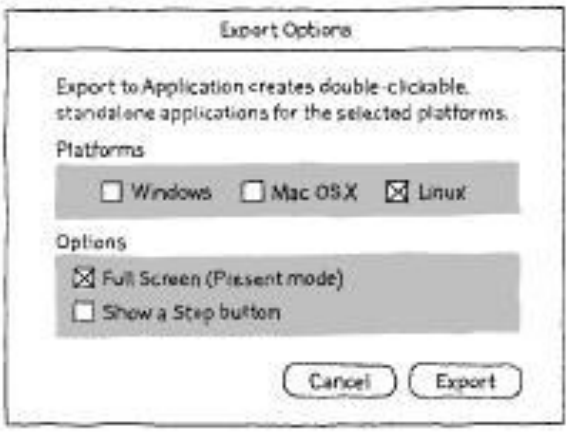


Figura 2-3 El menú exportar a aplicación.

Manteniendo presionada la tecla Shitf cuando presionas el botón de exportar en la barra de herramientas, es otra forma de exportar la aplicación.

Ejemplos y referencia

Aprender a programar con Processing implica explorar muchos códigos: correrlos, alterarlos, romperlos, y repasarlos hasta tener una nueva forma en algo nuevo. Con esto en mente, en la descarga del software Processing incluye decenas de ejemplos de demostración, con diferentes características del software. Para abrir los ejemplos, selecciona ejemplos desde el menú File y clickea el icono abierto en el PDE. Los ejemplos están agrupados por categorías según su función, forma, funcionamiento, e imagen. Encontrando tópicos interesantes en la lista y probando ejemplos.

Si ves una parte del programa con la cual no estás familiarizado y es de color naranja (esto es un método de una parte del Processing), selecciona esa palabra, click derecho (en Windows o ctrl-click en Mac) y selecciona "find in Reference" y entraras al menu de ayuda. Abrirás la referencia en un navegador web para el elemento del código seleccionado. La referencia también está disponible online en http://www.processing.org/reference/.

La referencia de Processing explica cada elemento de código con una descripción y ejemplos. La referencia usualmente son programas muy cortos (usualmente cuatro o cinco líneas) y son más fáciles de entender que los códigos largos en la carpeta de ejemplos. Recomendamos abrir la referencia cuando estés leyendo este libro y cuando estés programando, puedes estar navegando por tópicos y alfabéticamente; algunas veces es más rápido hacer un texto de búsqueda con tu ventana de navegación.

La referencia está escrita pensando en las mentes principiantes; esperamos que estén claros y comprensibles. Estamos agradecidos con la gente que ha depurado errores durante años y ha informado de ellos. Si usted piensa que puede mejorar una referencia o encontró algún error, Por favor háganoslo saber clickeando en el link de la parte superior de cada página de referencia.

CAPÍTULO 3

3 Dibujo

Primero, dibujar en una pantalla de computador es como trabajar en un papel cuadriculado. Se empieza como un proceso técnico muy cuidadoso, mientras los nuevos conceptos son introducidos; dibujar formas simples con software que se expandirán a la animación y a la interacción. Pero antes de que hagamos este salto, necesitamos empezar por el principio.

Una pantalla de computador es una red de luces con elementos llamados Pixeles. Cada pixel tiene una posición en la pantalla definido por coordenadas. En Processing, la x coordina la distancia desde el borde izquierdo de la ventana de display y la Y coordina la distancia desde el borde de arriba.

Escribimos las coordenadas de un pixel así: *(x,y)*. Así que, si la pantalla es de 200 X 200 píxeles, la superior izquierda es de (0,0), el centro está en (100,100), y la izquierda de abajo es (199,199). Estos números parecen ser confusos: cómo vamos desde 0 a 199 en lugar de 1 a 200? La respuesta es que usualmente en el código, contamos desde 0 porque es más fácil para hacer los cálculos en los que vamos a entrar más adelante.

La ventana de display es creada y las imágenes son dibujadas adentro de ella a través de elementos de código llamados funciones. Las funciones son los ladrillos básicos de un programa de Processing. El comportamiento de una función es definida por sus parámetros. Por ejemplo, al menos cada programa de Processing tiene una función de *size()* para establecer el ancho y el alto de la ventana de display. (Si el programa no tiene una función de *size()* la dimensión será establecida en 100 X 100 pixeles.)

Ejemplo 3-1: Dibujar una ventana

La función *size()* tiene dos parámetros: El primero establece el ancho de la ventana y el segundo establece el alto. Para dibujar una ventana que sea de 800 pixeles de ancho y 600 de alto, se escribe:

```
size( 800, 600 );
```

Corra esta línea de código para ver el resultado. Ponga diferentes valores para ver que es posible. Intente con números muy pequeños y números más grandes que los de su pantalla.

Ejemplo 3-2: Dibujar un punto

Para establecer el color de un solo pixel entre la ventana del display, usamos la función *point()*. Esta tiene dos parámetros que definen su posición: La coordenada x seguida de la coordenada y. Para dibujar una pequeña ventana y un punto en el centro de la pantalla,

según el size(480, 120); se debería escribir después de size() la función point(240,60);

```
size(480, 120);
point(240, 60);
```

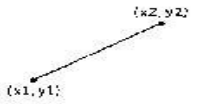
Intenta escribir un programa que ponga un punto en cada esquina de la ventana de display y uno en el centro. Intenta colocando puntos lado a lado para hacer líneas horizontales, verticales y diagonales.

Formas básicas

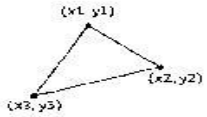
Processing incluye un grupo de funciones para dibujar formas básicas (ver la figura 3-1). Las formas simples como las líneas pueden ser combinadas para crear formas más complejas como una hoja o una cara. Para dibujar una sola línea, necesitamos cuatro parámetros: dos para empezar la colocación y dos para el final.

Figura 3

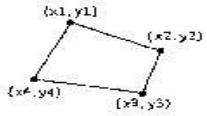
Line(x1, y1, x2, x2)



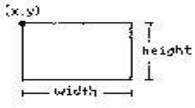
Triangle(x1, y1, x2, y2, x3, y3)



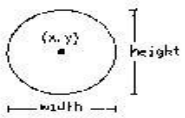
quad(x1,y1,x2,y2,x3,y3,x4,y4)



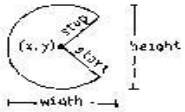
rect(x, y, width, height)



ellipse(x, y, width, height)

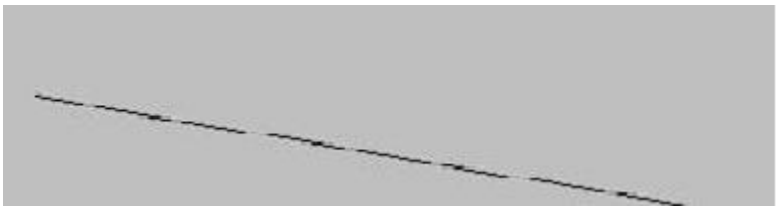


arc(x,y,width,height,start,stop)



Ejemplo 3-3: Dibujar una línea

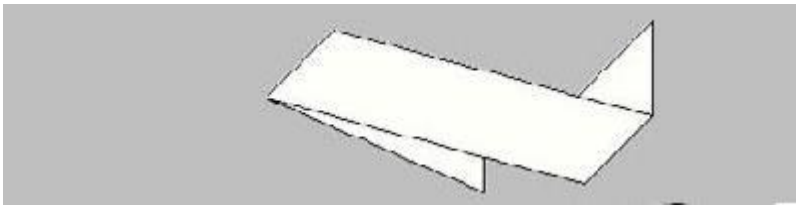
Para dibujar una línea entre las coordinadas (20,50) y (420,110), intenta:



```
Size (480,120);
Line (20,50,420,110);
```

Ejemplo 3-4: dibujar formas básicas

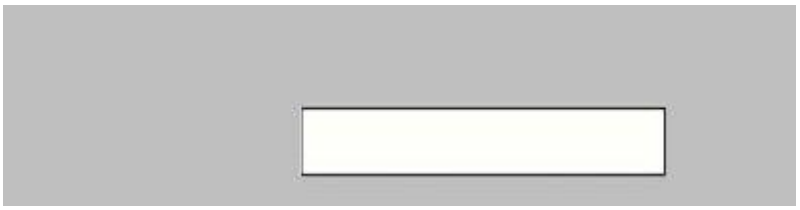
Siguiendo este patrón, un triángulo necesita seis parámetros y un cuadrilátero necesita 8 (un par por cada punto):



```
size(480, 120);
quad(158,55,199,14,392,66,351,107);
triangle(347,54,392,9,392,66);
triangle(158,55,290,91,290,112);
```

Ejemplo 3-5: dibujar un rectángulo

Los rectángulos y círculos son definidos con cuatro parámetros: el primero y el segundo son para las coordenadas X y Y del punto de anclaje, el tercero para el ancho, y el cuarto para el largo. Para hacer un rectángulo con las coordenadas (180,60) con un ancho de 220 pixeles y un largo de 40, usa la función rect() así:



```
size(480,120);
rect(180,60,220,40);
```

Ejemplo 3-6: Dibujar un circulo

Las coordenadas X y Y para un rectángulo son las de la esquina superior izquierda, pero en un circulo son las del centro de la forma. En este ejemplo, notamos que la coordenada Y para la primera elipse está fuera de la ventana. Los objetos pueden ser dibujados parcialmente (o enteramente) fuera de la ventana sin ningún error.

```
size (480,120);
ellipse (278,-100,400,400);
ellipse (120,100,110,110);
ellipse (412,60,18,18);
```

Processing no tiene funciones separadas para hacer cuadrados y círculos. Para hacer estas formas, use el mismo valor de los parámetros de ellipse () y rect() para el ancho y el largo.

Ejemplo 3-7: Dibujar partes de una elipse

La función arc() dibuja una parte de una elipse:

```
size (480,120);
arc (90,60,80,80,0, HALF_PI);
arc (190,60,80,80,0, PI+HALF_PI);
arc (290,60,80,80, PI, TWO_PI+HALF_PI);
arc (390,60,80,80, QUARTER_PI, PI+QUARTER_PI);
```

El primer y el segundo parámetro establece la ubicación, el tercero y el cuarto establecen el ancho y el largo. El quinto parámetro establece el ángulo para empezar el arco, y el sexto establece la parada del ángulo. Los ángulos son establecidos en radianes, en lugar de grados. Los radianes son ángulos de medición basados en el valor de pi (3.14159). La figura 3-2 muestra como ambos se relacionan. Tal como se presenta en este ejemplo, cuatro valores de radian son reemplazados por nombres especiales, para estos fue agregada una parte de Processing. Los valores PI, QUARTER_PI, HALF_PI, y TWO_PI pueden ser usados para reemplazar el valor de 180, 45, 90, 360, radianes.

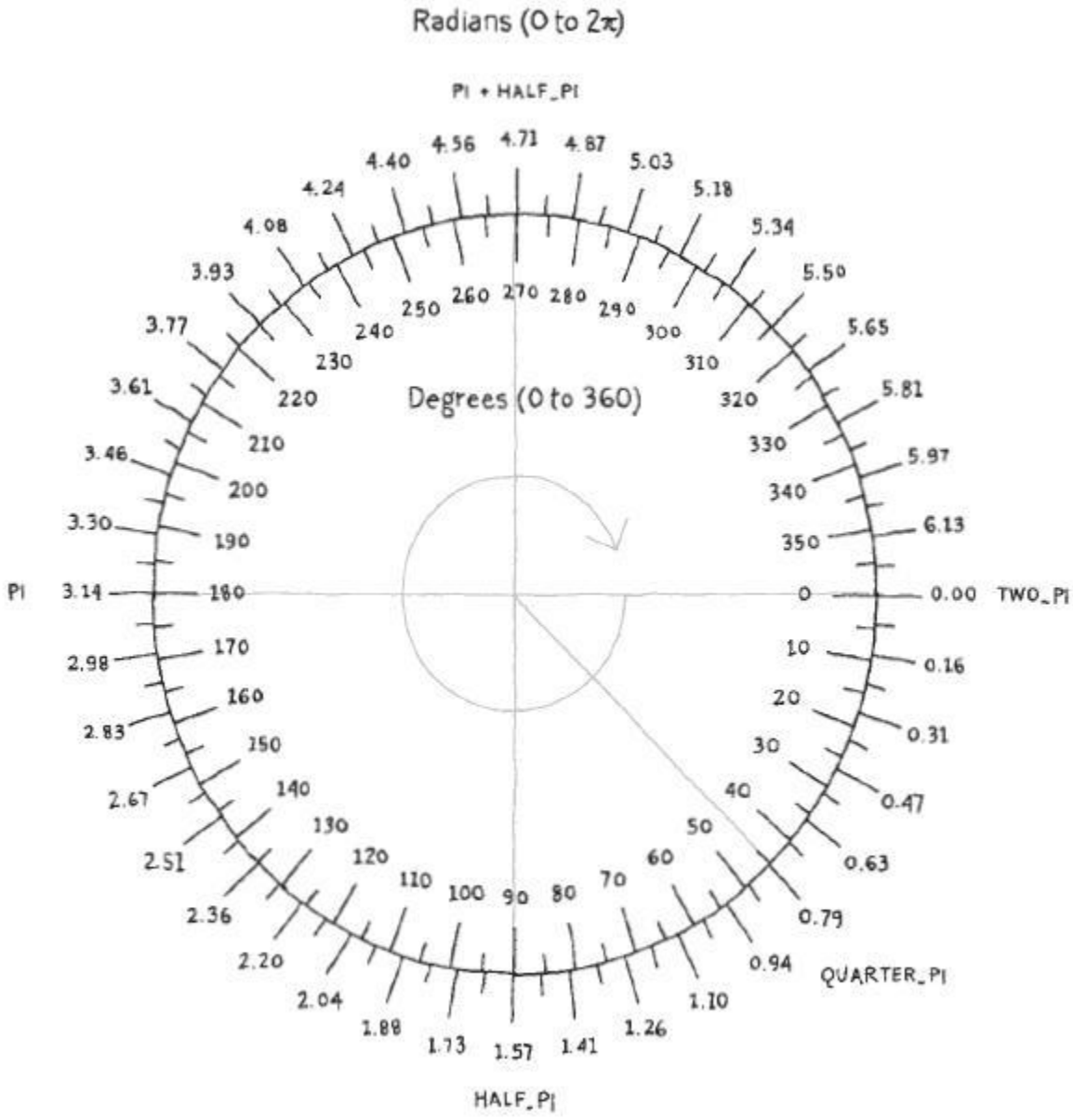


Figura 3-2 mediciones en grados y radianes.

Ejemplo 3-8: Dibujar con grados

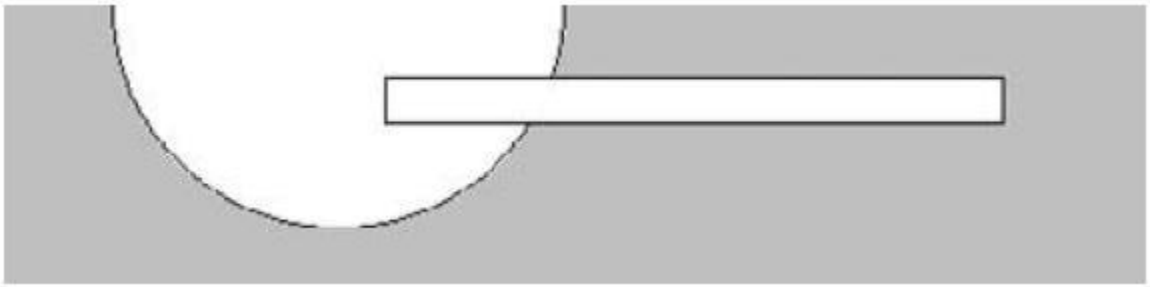
Si prefieres usar la medición en grados, puedes convertirlos a radianes con la función *radians()*. Esta función toma un ángulo en grados y los cambia a su valor correspondiente en radianes. El siguiente ejemplo es del mismo Ejemplo 3-7, pero usa la función radians() para definir el valor del comienzo y el final en grados:

```
size (480,120);
arc( 90,60,80,80,0, radians (90));
arc ( 190,60,80,80, radians(180), radians(450));
arc ( 390,60,80,80, radians(45), radians(225));
```

Orden de dibujo

Cuando el programa está corriendo, el computador empezará en la parte de arriba y leerá cada línea de código hasta que llegue a la última línea y luego parará. Si quieres dibujar una forma en la parte de arriba sobre todas las demás formas, necesitas seguir el orden descrito anteriormente.

Ejemplo 3-9: Controlar el orden de los dibujos



```
size (480,120);
ellipse (140,0,190,190);
// El rectángulo se dibuja en la parte de arriba de la elipse
// Porque viene después en el código
rect (160,30,260,20);
```

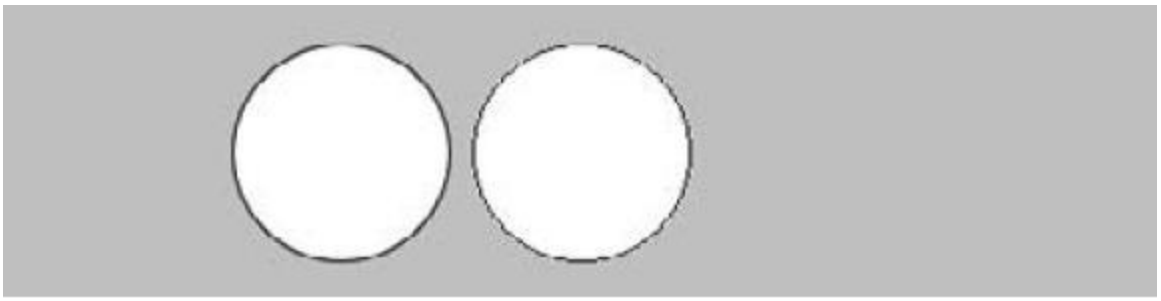
Puedes pensar como si estuvieras pintando con una brocha o haciendo un collage. El último elemento que agregas es lo que está visible en la parte de arriba.

Propiedades de las formas

La más básica y útil de las propiedades es el grosor del trazo y el antialiasing, también llamado smoothing (alisamiento).

Ejemplo 3-11: Dibujar líneas lisas

La función *smooth()* alisa los bordes de las líneas dibujadas en la pantalla mezclando los bordes con el valor del pixel más cercano. Si el alisamiento ya está prendido, a la inversa, la función nosmooth()será apagada:

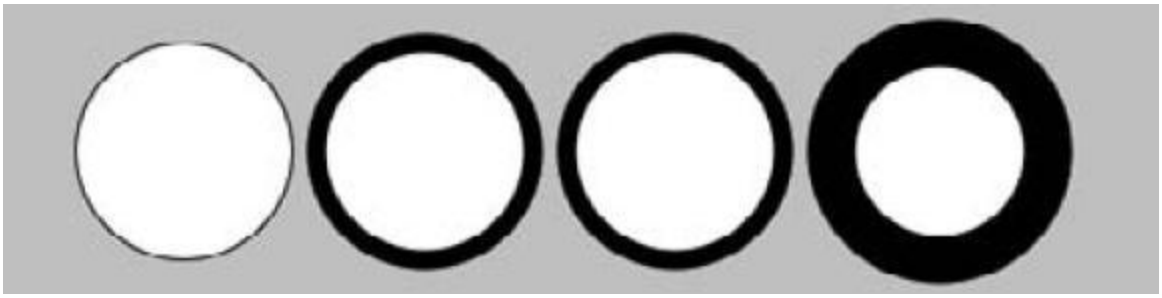


```
size (480,120);
smooth (); // prende el alisado
ellipse (140,60,90,90);
nosmooth (); // apaga el alisado
ellipse (240,60,90,90);
```

Nota: *Algunas de las implementaciones de Processing (como la versión para JavaScript) siempre serán formas lisas; otras podrían no admitir el alisado del todo. En algunas situaciones, no es posible activar y desactivar el alisado entre el mismo dibujo a través de draw(). Ver la referencia smooth() para más detalles.*

Ejemplo 3-12: Establecer el grosor del trazo

El grosor del trazo establecido es un solo pixel, pero esto puede ser cambiado con la función *strokeweight*(). El solo parámetro *strokeweight*() establece el grosor de las líneas dibujadas:



```
size (480,120);
smooth ();
ellipse (75,60,90,90);
strokeWeight (8); // Grosor del trazo de 8 pixeles
ellipse (175,60,90,90);
ellipse (279,60,90,90);
Strokeweight(20); // Grosor del trazo de 20 pixeles
ellipse (389,60,90,90);
```

Ejemplo 3-13: Establecer los atributos del trazo

La función *StrokeJoin()* cambia la forma en que las líneas son unidas (como luce la esquina), y la función *StrokeCap()* cambia como las líneas son dibujadas desde el principio hasta el final:

```
size (480,120);
smooth ();
strokeweight (12);
strokejoin (ROUND); // alrededor de la esquina del trazo
rect (40,25,70,70);
strokejoin (BEVEL); // bisel de la esquina del trazo
rect (140,25,70,70);
strokecap (SQUARE); // cuadrar los finales de línea
line (270,25,340,95);
strokecap (ROUND); // alrededor de los finales de línea
line (350,25,420,95);
```

La colocación de formas como *rect()* y *ellipse()* son controladas con las funciones *rectMode()* y *ellipseMode()*. Compruebe la referencia (Help -> Reference) para ver los ejemplos de cómo ubicar rectángulos desde su centro (mejor que desde la esquina superior izquierda), o para dibujar *ellipses* desde su esquina superior izquierda como los rectángulos.

Cuando ninguno de estos atributos están establecidos, todas las formas dibujadas después son afectadas. Por ejemplo, en el ejemplo 3-12, notamos como el segundo y tercer círculo tienen el mismo grosor de trazo, aunque el grosor sea establecido sólo una vez antes de ser dibujados ambos.

Color

Hasta ahora todas las forma se han llenado de blanco con líneas negras, y el fondo de la ventana de display ha tenido una luz gris. Para cambiarlos, usa las funciones *background(), fill() y stroke()*. Los valores de estos parámetros están en el rango de 0 a 255, cuando 255 es blanco, 128 es gris medio, y 0 es negro. La figura 3-3 muestra el mapa de como los valores de 0 a 255 tienen diferentes niveles.

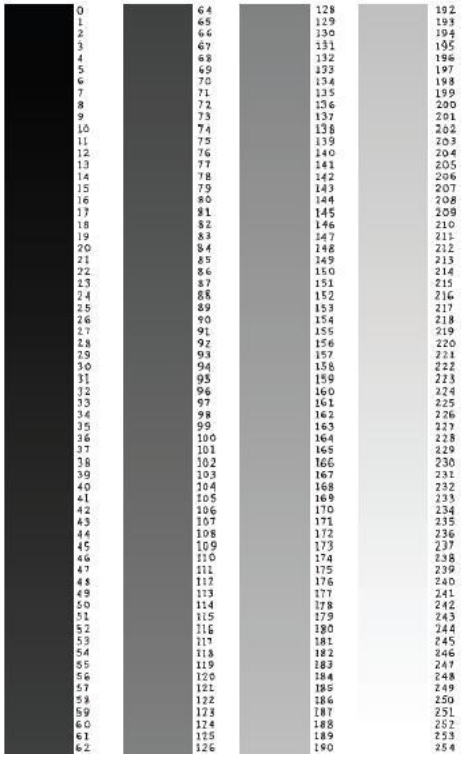


Figura 3.3: Escala de grises de 0 a 255

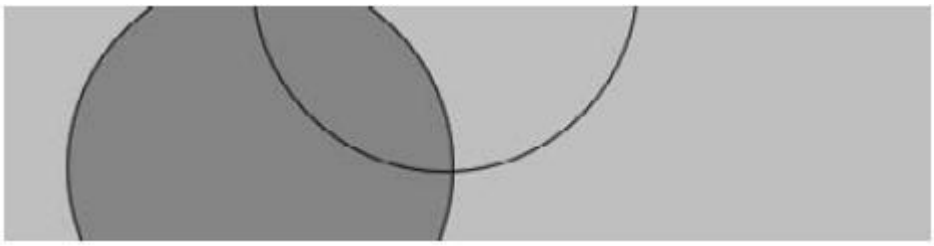
Ejemplo 3-14: Pintar con grises

Este ejemplo muestra 3 valores diferentes del gris en un fondo negro:

```
size (480, 120);
smooth ();
background (0);                 // Black
fill (204);                     // Light gray
ellipse (132, 82, 200, 200);  // Light gray circle
fill (153) ;                    // Medium gray
ellipse (228, -16, 200,200 ); // Medium gray circle
fill (102);                     // Dark gray
ellipse (268, 118, 200, 200); // Dark gray circle
```

Ejemplo 3-15: Controlar el relleno y el trazo

Puedes inhabilitar el trazado así que habrá trazo sin nostroke() y puedes inhabilitar el relleno de una forma con noFill():

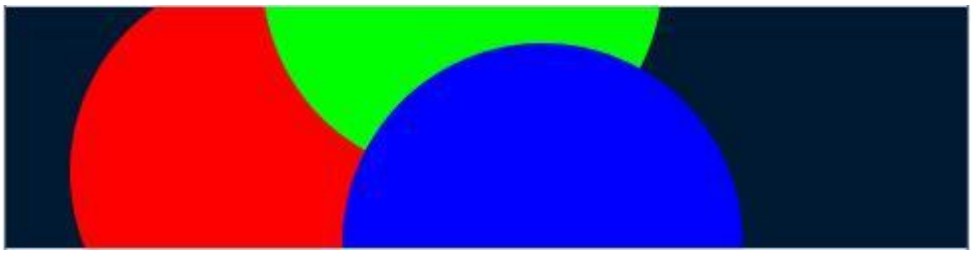


```
size (480, 120);
smooth ();
fill (153);                    // Medium gray
ellipse (132, 82, 200, 200);   // Gray circle
noFill();                      // Turn off fill
ellipse (228, -16, 200, 200);  // Outline circle
noStroke ();                   // Turn off stroke
ellipse (268, 118, 200, 200);  // Doesn't draw!
```

Sea cuidadoso de no inhabilitar el relleno y el trazo al mismo tiempo, como lo hemos hecho en el ejemplo anterior. Porque nada quedará dibujado en la pantalla.

Ejemplo 3-16: dibujar con color

Para ir más allá en los valores de la escala de grises, usa 3 parámetros para especificar los componentes rojo, verde y azul de un color. Porque este libro está impreso en blanco y negro, aquí sólo podrás ver los valores grises. Corre el código en Processing para revelar los colores:



```
size (480, 120);
noStroke ();
smooth ();
blackground (0, 26, 51);         // Dark blue color
fill (255, 0, 0);                // Red color
ellipse (132, 82, 200, 200);      // Red circle
fill (0, 255, 0);                //Green color
ellipse (228, -16, 200, 200);   //Green circle
fill (0, 0 , 255);               // Blue color
ellipse (268, 118, 200, 200);   // Blue color
```

Este es referido como un color RGB, el cual viene de la definición del color de las pantallas de los computadores. Los tres valores standard rojo, verde y azul; ya su rango que va desde 0 a 255. La forma en que los valores del gris lo hacen usando el color RGB no es muy intuitiva, así que para escoger los colores, usa Tools -> color selector (herramientas-> seleccionar el color), el cual te muestra una paleta de colores similar a la que se encuentra en cualquier software (ver la figura 3-4). Selecciona un color, y luego usa los valores R, G y B como parámetros para las funciones *background(), fill() o stroke()*.

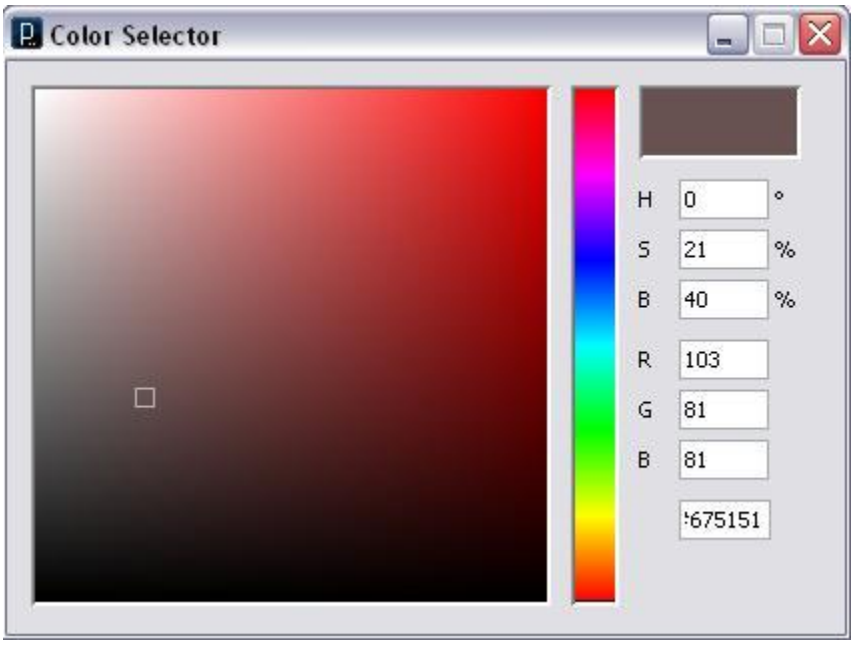


Figura 3-4 Selección de color en Processing

Ejemplo 3-17: Establecer la transparencia

Agregando un cuarto parámetro adicional a *fill()* o *Stroke()*, puedes controlar la transparencia. Este cuarto parámetro es conocido como el valor *alpha*, y también se usa dentro del rango de 0 a 255 para establecer la cantidad de transparencia. El valor 0 define el color como transparente total (no se mostrará), el valor 255 es totalmente opaco, y los valores entre estos extremos causan que los colores se mezclen en la pantalla.

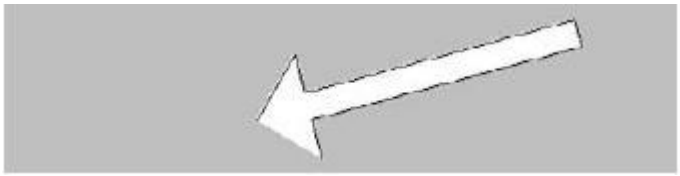
```
size (480, 120);
noStroke();
smooth ();
background (204,226,225);    // Light blue color
fill ( 255, 0, 0, 160);      // Red color
ellipse (132, 82, 200, 200) ;// Red circle
fill (0, 255, 0, 160);       // Green color
ellipse (228, -16, 200, 200);// Green circle
fill (0, 0, 255, 160);       // Blue color
ellipse (268, 118, 200, 200);// Blue circle
```

Formas personalizadas

No estás limitado a usar las figuras geométricas básicas, también puedes definir nuevas formas conectando una serie de puntos.

Ejemplo 3-18: Dibujar una flecha

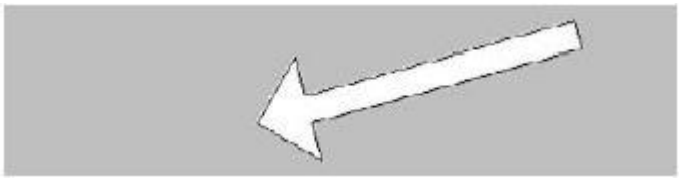
La función *begindShape()* señala el comienzo de una nueva forma. La función vertex() es usada para definir cada par de las coordenadas X y Y para la forma. Finalmente, *endShape()* es utilizado para señalar que la forma a sido terminada.



```
size (480,120);
beginShape ();
vertex (180, 82);
vertex (207, 36);
vertex (214, 63);
vertex (407, 11);
vertex (412, 30);
vertex (219, 82);
vertex (226, 109);
endShape();
```

Ejemplo 3-19: Cerrar la brecha

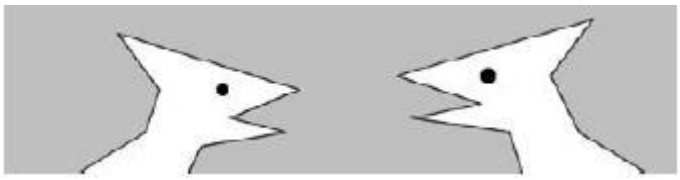
Cuando se ejecuta el ejemplo 3-18, puedes ver que el primer y el último punto no están conectados. Para hacer esto, agrega la palabra CLOSE (cerrar) como un parámetro en *endShape()* así:



```
size (480, 120);
beginShape();
vertex (180, 82);
vertex (207,36);
vertex (214, 63);
vertex (407, 11);
vertex (412, 30);
vertex (219, 82);
vertex (226, 109);
endShape (CLOSE);
```

Ejemplo 3-20: crear algunas criaturas

El poder de definir formas con *vertex()* es la habilidad para hace formas con contornos más complejos. Processing puede dibujar miles y miles de líneas al mismo tiempo para rellenar la pantalla con formas fantásticas que surgen de tu imaginación. El siguiente es un modesto pero más complejo ejemplo:



```
size (480, 120);
smooth ();

//Left creature
beginShape();
vertex (50, 120);
vertex (100, 90);
vertex (110, 60);
vertex (80, 20);
vertex (210, 60);
vertex (160, 80);
vertex (200, 90);
vertex (140, 100);
vertex (130, 120);
endShape();
fill(0);
ellipse (155, 60, 8, 8);

// Right creature
fill (255);
beginShape();
vertex (370, 120);
vertex (360, 90);
vertex (290, 80);
vertex (340, 70);
```

```
vertex (280, 50);
vertex (420, 10);
vertex (390, 50);
vertex (410, 90);
vertex (460, 120);
endShape(0;
fill (0);
ellipse (345, 50, 10, 10);
```

Comentarios

Los ejemplos de este capítulo usan doble slash // al final de una línea para agregar comentarios en el código. Los comentarios hacen parte del programa y son ignorados cuando el programa está corriendo. Son útiles para hacer notas explicando que hay en el código. Si otro está leyendo tu código, los comentarios son especialmente importantes para ayudar a entender tu proceso de pensamiento.

Los comentarios son especialmente importantes para diferentes opciones, como cuando estamos intentando escoger el color correcto. Así que, por ejemplo, podría estar intentando encontrar el rojo correcto para una elipse:

```
size (200, 200);
fill (165, 57, 57);
ellipse (100, 100, 80, 80);
```

Ahora supongo que quiero intentar con otro rojo diferente, pero no quiero perder el viejo. Puedo copiar y pegar la línea, hacer un cambio, y luego "comentar" el viejo:

```
size (200, 200);
//fill (165, 57, 57);
fill (144, 39, 39);
ellipse (100, 100, 80, 80);
```

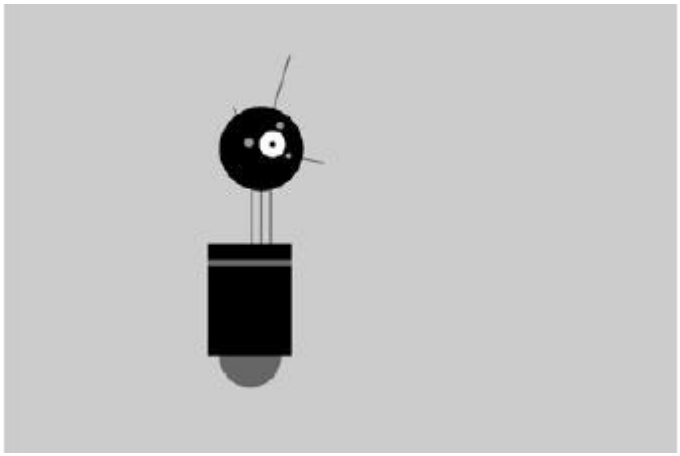
Colocando // al comienzo de la línea se desactiva temporalmente. O puedo remover el // y colocarlo en frente de la otra línea si quiero intentarlo de nuevo:

```
size (200, 200);
fill (165, 57, 57);
//fill (144, 39, 39);
ellipse (100, 100, 80, 80);
```

Nota: Como un atajo, puedes usar Ctrl-/(Cmd-/ en Mac) para agregar o quitar comentarios desde la línea corriente o un bloque de texto seleccionado. También puedes comentar varias líneas al mismo tiempo con notación alternativa para comentar introducida en el apéndice A.

Mientras se trabaja con los bocetos de Processing, podrás encontrar la creación de docenas de ideas: usando comentarios para hacer notas o para inhabilitar código puede ayudarte hacer un seguimiento de múltiples opciones.

Robot 1: Dibujo



Este es P5, el robot de Processing. Hay 8 programas diferentes para dibujar y animarlo en el libro - cada uno explora una idea de programación diferente. El diseño de P5 fue inspirado por Sputnik l (1957) Shakey del Instituto de investigación de Standford (1966-1972), El zumbido de combate en David Lynch's Dune (1984), y HAL 9000 desde 2001: Una odisea en el espacio (1968), entre otros robots favoritos.

Para dibujar este robot se introdujeron los conceptos vistos en este capítulo. Se establecieron los parámetros de las funciones *fill()* y *stroke()* en valores grises. Las funciones *line()*, *ellipse()*,y *rect()* definen las formas que crean el cuello, la antena, el cuerpo y la cabeza del robot. Para familiarizarse mas con las funciones, corra el programa y cambie los valores para rediseñar el robot:

```
size (720, 480);
smooth ();
strokeWeight(2);
backround (204);
ellipseMode (RADIUS);

// Neck
stroke (102);                // Set stroke to gray
line (266, 257, 266, 162);  // Left
line (276, 257, 276, 162);  // Middle
line (286, 257, 286, 162);  // Right

// antennae
line (276, 155, 246, 112);  // Small
line (276, 155, 306, 56);   // Tall
line (276, 155, 342, 170);  // Medium
```

```
// Body
noStroke();                 // Disable stroke
fill (102);                 // Set fill to gray
ellipse (264, 377, 33, 33); // Antigravity orb
fill (0);                   // Set fill to black
rect (219, 257, 90, 120);   // Main body
fill (102);                 // Set fill to gray
rect (219, 274, 90, 6);     // Gray stripe

// Head
fill (0);                   // Set fill to black
ellipse (276, 155, 45, 45); // Head
fill (255);                 // Set fill to white
ellipse (288, 150, 14, 14); // Large eye
fill (0);                   // Set fill to black
ellipse (288, 150, 3, 3);   // Pupil
fill (153);                 // Set fill to light gray
ellipse (263, 148, 5, 5);   // Small eye 1
ellipse (296, 130, 4, 4);   // Small eye 2
ellipse (305, 162, 3, 3);   // Small eye 3
```

CAPITULO 4

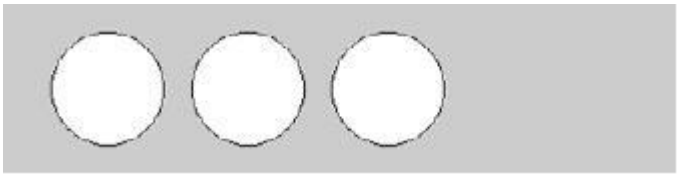
4 Variables

Una variable guarda un valor en su memoria y este puede ser usado luego en un programa. La variable puede ser usada varias veces dentro de un programa, y el valor es cambiado fácilmente mientras el programa está corriendo.

La razón principal por la que usamos las variables es para evitar la repetición de líneas en el código. Si estás escribiendo el mismo número más de una vez, considera marcarlo en una variable para hacer tu código más general y fácil de actualizar.

Ejemplo 4-1: Reutilizar el mismo valor

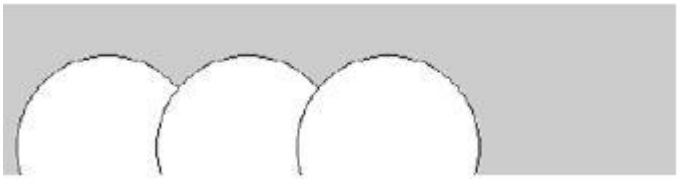
Por ejemplo, cuando haces la coordenada Y, y el diámetro para los dos círculos en este ejemplo en variables, los mismos valores son usados para cada elipse:



```
size (480, 120);
smooth ();
int y = 60;
int d = 80;
ellipse (75, y, d, d);  // Left
ellipse (175, y, d, d); // Middle
ellipse (275, y, d, d); // Right
```

Ejemplo 4-2: Cambio de valores

Cambiando simplemente las variables Y y d se alteran las 3 elipses:



```
sizes (480, 120);
smooth ();
int y = 100;
int d = 130;
ellipse (75, y, d, d);  // Left
ellipse (175, y, d, d); // Middle
ellipse (275, y, d, d); // Right
```

Sin las variables, necesitarías cambiar la coordenada Y usada en el código tres veces y el diámetro seis veces. Cuando comparamos los ejemplos 4-1 y 4-2, notamos como el botón de las tres líneas es el mismo, y sólo el de las dos líneas del medio tiene diferentes variables. Las variables te permiten separar las líneas del código que cambian desde las líneas que no lo hacen, lo cual hace el programa más fácil de modificar. Por ejemplo, si colocas variables que controlan los colores y el tamaño de las formas en un lugar, luego puedes rápidamente explorar diferentes opciones visuales concentrándose en unas cuantas líneas de código.

Haciendo variables

Cuando haces tus propias variables, determina el *nombre,* el *tipo de datos,* y el *valor.* Tú decides cómo llamar a la variable. Escoge un nombre que te informe acerca de lo que guarda la variable, pero que sea consistente y no demasiado detallado. Por ejemplo, el nombre de la variable "radius" será más claro que "r" cuando mires el código después.

El rango de valores que puede ser almacenado entre una variable es definido por su tipo de datos. Por ejemplo, el *típico número entero de datos* puede guardar números sin decimales (números enteros). En código, *entero* es abreviado a *int*. Hay tipos de datos para guardar cada tipo de datos: enteros, de puntos flotantes (decimales) números, caracteres, palabras, imágenes, fuentes, etcétera.

Las variables deben ser *declaradas* primero, lo que deja a un lado el espacio en la memoria del computador para guardar información. Cuando declaramos una variable, se necesita especificar su tipo de dato (como int), lo cual indica que tipo de información será guardada. Después de establecer el nombre y el tipo de dato, este puede ser asignado a un valor para la variable:

```
int x;  // Declare x es una variable entera
x = 12; // Asigna una valor a la variable x
```

Este código hace lo mismo, pero es más corto:

```
int x = 12; // Declara que x es una variable entera y asigna un valor
```

El nombre del tipo de dato está incluido en la línea de código que declara a la variable, pero no se vuelve a escribir. Cada vez que se escribe el tipo de datos en frente del nombre de la variable, el computador piensa que estás tratando de declarar una nueva variable. No puedes tener dos variables con el mismo nombre en la misma parte del programa (Ver el apéndice D), así el programa tendría un error:

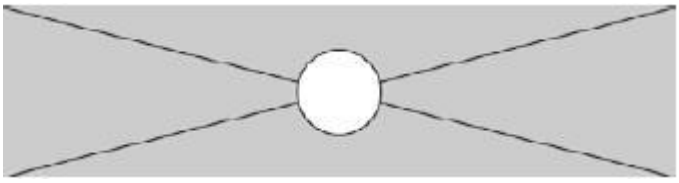
```
int x;       // Declara x como una variable entera
int x = 12; // ERROR! no puedes tener dos variables llamadas x
```

Las variables de processing

Processing tiene una serie de variables especiales para almacenar información acerca del programa mientras este corre. Por ejemplo, el *largo (width)* y el *alto (height)*. Estos valores son establecidos por la función size(). Estos pueden ser usados para dibujar elementos relativos al tamaño de la ventana, incluso si la línea size() cambia.

Ejemplo 4-3: ajustar el tamaño, ver lo que sigue

En este ejemplo, cambiamos los parámetros a size(), para ver como funciona:



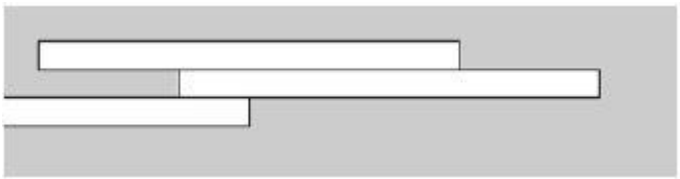
```
size (480, 120);
smooth ();
line (0, 0, width, height);  // Line from (0,0) to (480, 120)
line (width, 0, 0, height);  // Line from (480, 0) to (0,120)
ellipse (width/2, height/2, 60, 60);
```

Otra variable especial le hace seguimiento al estado del valor del mouse y el teclado y mucho más. Estos son discutidos en el capítulo 5

Un poco de matemáticas

La gente a menudo asume que las matemáticas y la programación son lo mismo. Aunque el conocimiento de matemáticas puede ser muy útil para cierto tipo de código, la aritmética básica cubre las partes más importantes.

Ejemplo 4-4: Aritmética básica



```
size (480, 120);
 int x = 25;
 int h = 20;
 int y = 25;
  rect (x, y, 300, h);    // Top
   x = x + 100;
  rect (x, y + h, 300, h);// Middle
   x = x - 250;
  rect (x, y + h*2, 300, h); // Bottom
```

En código, los símbolos como +, -, y * son llamados *operadores.* Cuando son colocados entre dos valores, crean una *expresión.* Por ejemplo, 5 + 9 y 1024 - 512 son expresiones. Los operadores para las operaciones matemáticas básicas son:

+ Adición
- Sustracción
** Multiplicación*
/ División
= Asignación

Processing tiene reglas establecidas para definir cuáles operadores tiene prioridad sobre los demás, es decir cuales cálculos se hacen primero, segundo, tercero, y así sucesivamente. Estas reglas se definen en el orden en el cuál el código está corriendo. Un poco de conocimiento acerca del largo camino para la comprensión de cómo funciona una línea corta de código como este:

```
int x = 4+4*5;  // Assign 24 to x
```

La expresión 4 * 5 es evaluada primero porque la multiplicación tiene mayor prioridad. Segundo, 4 es agregado al producto de 4 * 5 para dar 24. Por último, el operador asignación (el símbolo de igual) tiene la menor prioridad, el valor 24 es asignado a la variable x. Esto es clarificado con un paréntesis, pero el resultado es el mismo:

```
int x = 4+(4*5); // Assign 24 to x
```

Si quieres forzar para adición sea primero, sólo tienes que mover el paréntesis. Porque el paréntesis tiene la mayor prioridad que la multiplicación, el orden es cambiado y el cálculo es afectado:

```
int x = (4+4)*5; // Assign 40 to x
```

Un acrónimo para este orden es enseñado en clases de matemáticas: PEMDAS, el cual se encuentra en paréntesis, exponentes, multiplicaciones, divisiones, adiciones y sustracciones, donde el paréntesis tiene la mayor prioridad y la sustracción la menor. El orden completo de operaciones se encuentra en el apéndice C.

Algunos cálculos son usados más frecuentemente en programación que los métodos abreviados que se han desarrollado; siempre es bueno guardar

algunas pulsaciones de tecla. Por ejemplo, le puedes agregar a una variable, o sustraer de ella, con un sólo operador:

```
x+=10; // Esto es igual a  x = x + 10
y-=15; // Esto es igual a  y = y - 15
```

Es también común agregar o sustraer 1 de una variable, así que los atajos existen para esto también. Los operadores ++ y -- hacen esto:

```
x++; // Esto es igual a x = x + 1
y--; // Esto es igual a y = y - 1
```

Más atajos pueden ser encontrados en la referencia.

Repetición

A medida que escribes más programas, notarás que los patrones se producen cuando las líneas de código se repiten. Pero con leves variaciones. Una estructura de código es llamada *for loop* para hacer posible correr una línea de código más de una vez para condensar este tipo de repetición en menos líneas. Esto hace más modular tu programa y más fácil de cambiar.

Ejemplo 4-5: Hacer lo mismo una y otra vez

Este ejemplo tiene el tipo de patrón que puede ser simplificado para *for* loop:

```
size (480, 120);
smooth ();
strokeWeight (8);
line (20, 40, 80, 80);
line (80, 40, 140, 80)
line (140, 40, 200, 80);
line (200, 40, 260, 80);
line (260, 40, 320, 80);
line (320, 40, 380, 80);
line (380, 40, 440, 80);
```

Ejemplo 4-6: Usa un *for* loop

La misma cosa puede ser hecha para el for, y menos código:

```
size (480, 120);
smooth ();
```

```
strokeWeight (8);
for (int i = 20; i < 400; i += 60) {
line (i, 40, i + 60, 80);
}
```

El *for* loop es diferente, en varias formas del código que hemos escrito hasta ahora. Notarás los tirantes, los caracteres {y}. El código entre los tirantes es llamado *bloque.* Este es el código que será repetido en cada iteración del *for* loop.

Dentro de los paréntesis hay tres declaraciones, separadas por punto y coma, que trabajan juntos para controlar cuántas veces corre el código dentro del bloque. De izquierda a derecha, estas declaraciones se refieren a la *inicialización* (init), el *examen,* y la *actualización*:

```
for (init; test; update) {
statements
}
```

Típicamente el *init* declara una nueva variable para usar dentro de *for* loop y asignar un valor. El nombre de la variable i es usado frecuentemente, pero no tiene nada de especial. El *examen* evalúa el valor de esta variable, y la *actualización* cambia el valor de la variable. La figura 4-1 muestra el orden en el cuál estos corren y como controlan la declaración del código dentro del bloque.

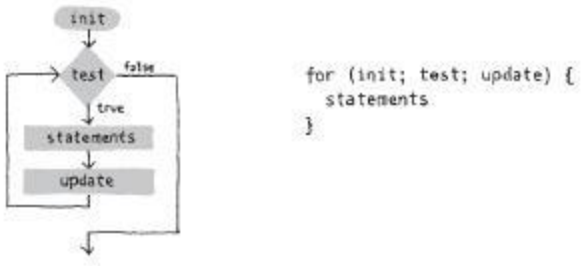


Figura 4-1. Diagrama de flujo de un for loop.

La declaración de *examen* requiere más explicación. Siempre es una *expresión relacional* que compara dos valores con un *operador relacional.* En este ejemplo, la expresión es "i < 400" y el operador es el símbolo < (menor que). Los operadores relacionales más comunes son:

```
>  Mayor que
<  Menor que
>= mayor o igual que
<= Menor o igual que
== Igual que
!= Diferente de
```

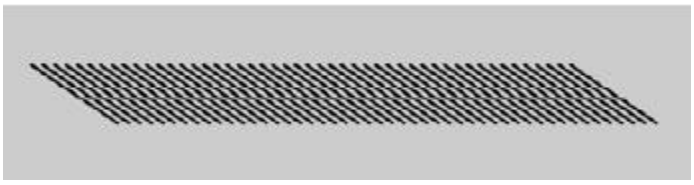
La expresión relacional siempre evalúa a *verdadero o falso.* Por ejemplo, la expresión 5 > 3 es *verdadera.* Podemos preguntar, "es cinco

mayor que tres?" Porque la respuesta es "sí". Entonces decimos que la expresión es *verdadera.*

Para la expresión 5 < 3, preguntamos, "es cinco menor que tres?" Porque la respuesta es "no", entonces decimos que la expresión es *falsa.* Cuando la evaluación es *verdadera,* el código dentro del bloque está corriendo, y cuando es *falsa,* el código dentro del bloque no está corriendo y el *for* loop termina.

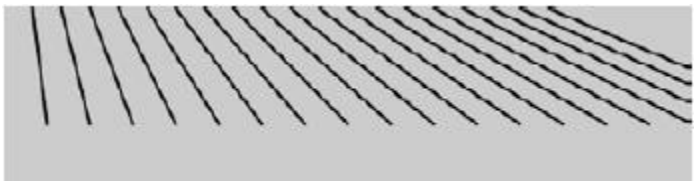
Ejemplo 4-7: Flexiona tus músculos for loop's

El máximo poder de trabajar con un *for* loop es la habilidad para hacerle cambios al código rápidamente. Porque el código dentro del bloque normalmente se ejecuta en múltiples ocasiones, un cambio al código es ampliado cuando el código está corriendo. Modificando ligeramente el ejemplo 4-6, podemos crear un rango de patrones diferente:



```
size (480, 120);
smooth ();
strokeWeight (2);
 for (int i = 20; i < 400; i +=8) {
line (i, 40, i + 60, 80);
}
```

Ejemplo 4-8: Abanico de líneas



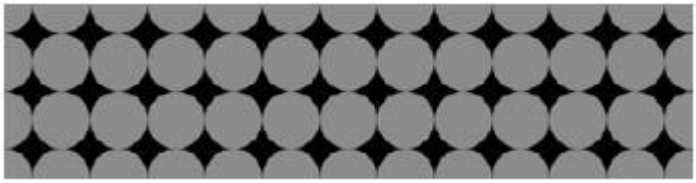
```
size (480, 120);
smooth ();
strokeWeight (2);
 for (int i =20; i <400; i +=20) {
line (i, 0, i + i/2, 80);
}
```

Ejemplo 4-9: Retorcer las línea

```
size (480, 120);
smooth ();
 for (int i =20; i <400; i +=20) {
line (i, 0, 0 + i/2, 80);
line (i + i/2, 80, i*1.2, 120);
}
```

Ejemplo 4-10: Incrustar un *for* loop a otro

Cuando un *for* loop es incrustado dentro de otro, el número de repeticiones es multiplicado. Primero, miremos un corto ejemplo, y luego lo descomponemos en el ejemplo 4-11:



```
size (480, 120);
background (0);
smooth ();
noStroke ();
for (int y =0; y <= height; y += 40) {
  for (int x =0; x <= width; x += 40) {
    fill (255, 140);
  ellipse (x, y, 40, 40);
 }
}
```

Ejemplo 4-11: Filas y columnas

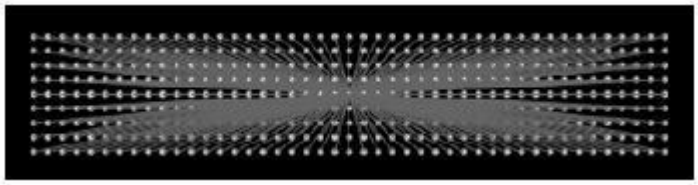
En este ejemplo, el *for* loop es adyacente, en lugar de uno incrustado dentro de otro. El resultado muestra que un *for* loop está dibujando una columna de 4 círculos y el otro está dibujando una fila de 13 círculos:

```
size (480, 120);
background (0);
smooth ();
noStroke ();
for (int y = 0; y < height+45; y += 40) {
  fill (255, 140);
  ellipse (0, y, 40, 40);
}
for (int x = 0; x < width+45; x += 40) {
  fill (255, 140);
  ellipse (x, 0, 40, 40);
}
```

Cuando uno de estos *for* loops es colocado dentro de otro, como en el ejemplo 4-10, las cuatro repeticiones del primer loop son agravadas con las trece del segundo en orden para correr dentro del bloque incrustado 52 veces (4 x 13=52).

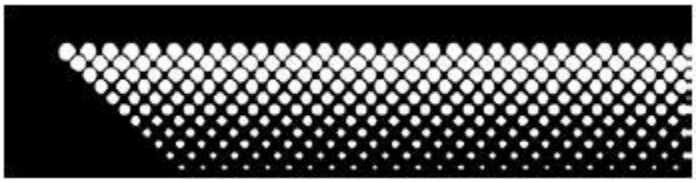
El ejemplo 4-10 es una buena base para explorar varios tipos de patrones de repetición visual. Los siguientes ejemplos muestran un par de formas que pueden ser extendidas, pero este es sólo una pequeña muestra de lo que es posible. En el ejemplo 4-12, el código dibuja una línea desde cada punto en la red del centro a la pantalla. En el ejemplo 4-13, se reducen los puntos suspensivos con cada nueva fila y son removidos hacia la derecha agregando la coordenada y a la coordenada x.

Ejemplo 4-12: pasadores y líneas



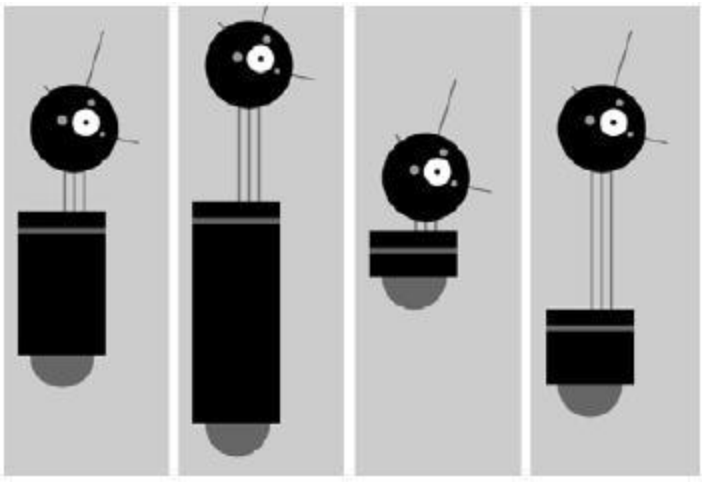
```
size (480, 120);
background (0);
smooth ();
fill (255);
stroke (102);
for (int y = 20; y <= height-20; y += 10) {
  for (int x = 20; x <= width-20; x += 20) {
    ellipse (x, y, 4, 4);
    // Draw a line to the center of the display
    line (x, y, 240, 60);
   }
 }
```

Ejemplo 4-13: Puntos de medios tonos



```
size (480, 120);
background (0);
smooth ();
for (int y = 32; y <= height; y += 8) {
  for (int x = 12; x <= width; x += 15) {
    ellipse (x + y, y, 16 - y/10.0, 16 - y/10.0);
   }
 }
```

Robot 2: Variables



Las variables introducidas en este programa hacen que el código se vea diferente del robot 1 (ver "robot 1: Dibujar" en el capítulo 3), pero ahora es más fácil de modificar, porque los números dependen el uno del otro estando en una sola ubicación. Por ejemplo, el cuello puede ser dibujado en la variable *body-height.* El grupo de variables en la parte de arriba del código, controlan los aspectos del robot que queremos cambiar: ubicación, altura del cuerpo, y altura del cuello. Puedes ver el rango de posibles variaciones en la figura; de izquierda a derecha, aquí están los valores que les corresponden:

y = 390	y = 460	y = 310	y = 420
bodyHeight = 180	bodyHeight = 260	bodyHeight = 80	bodyHeight = 110
neckHeight = 40	neckHeight = 95	neckHeight = 10	neckHeight = 140

Cuando alteras tu propio código para usar variables en lugar de números, planea el cambio cuidadosamente, luego haz las modificaciones en pasos cortos. Por ejemplo, cuando este programa fue escrito, cada variable fue creada una a la vez para minimizar la complejidad de transición. Después que una variable ha sido agregada y el código se

ejecuta para asegurarse de que estaba trabajando, la siguiente variable se agrega:

```
int x = 60;           // x-coordinate
int y - 420;          // y-coordinate
int bodyHeight = 110; // Body Height
int neckHeight - 140; // Neck Height
int radius = 45;      // Head Radius
int ny = y - bodyHeight - neckHeight - radius; // Neck Y

size (170, 480);
smooth ();
strokeWeight (2);
background (204);
ellipseMode (RADIUS);

//Neck

stroke (102);
line (x+2, y=bodyHeight, x+2, ny);
line (x+12, y-bodyHeight, x+12, ny);
line (x+22, y-bodyHeight, X+22, ny);

// Antennae

line (x+12, ny, x-18, ny-43);
line (x+12, ny, x+42, ny-99);
line (x+12, ny, x+78, ny+15);

// Body

noStroke ();
fill (102);
ellipse (x, y-33, 33, 33);
fill (0);
rect (x-45, y-bodyHeight, 90, bodyHeight-33);
fill (102);
rect (x-45, y-bodyHeight+17, 90, 6);

// Head

fill (0);
ellipse (x+12, ny, radius, radius);
fill (255);
ellipse (x+24, ny-6, 3, 3);
fill (0);
ellipse (x+24,ny-6, 3, 3);
fill (153);
ellipse (x, ny-8, 5, 5);
ellipse (x+30, ny-26, 4, 4);
ellipse (x+41, ny+6, 3, 3);
```

CAPÍTULO 5

5 Respuesta

El código que responde a la entrada del mouse, teclado y otros dispositivos que tienen que funcionar continuamente. Para hacer que esto pase, coloca las líneas que actualizan dentro de una función de Processing llamada draw().

Ejemplo 5-1: La función draw()

Para ver cómo funciona draw(), corra este ejemplo:

```
void draw () {
  // Displays the frame count to the console
  println("I'm drawing");
  println(frameCount);
}
```

Verás lo siguiente:

```
I'm drawing
1
I'm drawing
2
I'm drawing
3
```

El código dentro del bloque draw() corre desde encima hasta el fondo, luego repite hasta que dejas el programa haciendo click en el botón de Stop o cerrando la ventana. Cada vuelta a través de draw() es llamado *frame.* (Por defecto la velocidad de cada frame es 60 frames por segundo, pero esto puede ser cambiado. Ver el ejemplo 7-2 para más información.) En el ejemplo previo, La funcion println() escribio el texto "I'm drawing" seguido de un recuento de fotogramas según su contabilidad por la variable especial *framecount (recuento)* (1,2,3...). El texto aparece en la consola, en el área negra al fondo de la ventana del editor de Processing.

Ejemplo 5-2: La función setup()

Para complementar la función looping draw(). Processing tiene una función llamada setup() que corre solo una vez cuando el programa comienza:

```
void setup () {
  println(I'm starting");
}

void draw (0 {
  println("I'm running");
```

```
}
```

Cuando este código está corriendo, lo siguiente aparecerá escrito en la consola:

```
I'm starting
I'm running
I'm running
I'm running
```

El texto "I'm running" continúa escribiéndose en la consola hasta que el programa sea parado.

En un programa típico, el código dentro de *setup()* es usado para definir los valores del comienzo. La primera línea siempre es la función *size()*, a menudo seguido por un código para establecer los colores del relleno y el trazo del comienzo, o tal vez para cargar imágenes y fuentes. (Si no incluyes la función *size()*, la ventana de display será de 100x100 pixeles.)

Ahora sabes cómo usar *setup()* y *draw()*, pero esta no es toda la historia. Hay una ubicación más para poner código - también puedes poner variables afuera de *setup()* y *draw()*. Si creas una variable dentro de *setup()*, no puedes usarla dentro de *draw()*, así que necesitas colocar esas variables en otro lugar. Algunas variables son llamadas variables *globales*, porque pueden ser usadas en cualquier parte ("globalmente") en el programa. Esto es más claro cuando tenemos una lista de orden en la cual el código está corriendo:

1. Variables declaradas fuera de *setup()* y *draw()* son creadas.

2. El código dentro de *setup()* corre una vez.

3. El código dentro de *draw()* corre continuamente.

Ejemplo 5-3: *setup()*, conoce a *draw()*

El siguiente ejemplo los muestra juntos:

```
int x = 280;
int y = -100;
int diameter = 380;

void setup() {
  size (480, 120);
  smooth (0);
  fill (102);
}

void draw () {
  background (204);
  ellipse (x, y, diameter, diameter);
```

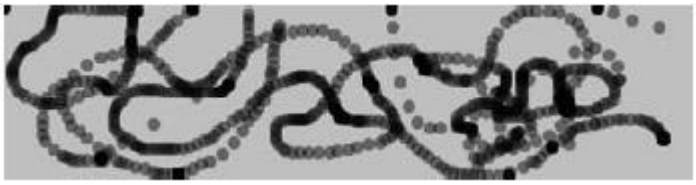
```
}
```

Seguir

Ahora que tenemos el código corriendo continuamente, podemos rastrear la posición del mouse y usar estos números para mover elementos en la pantalla.

Ejemplo 5-4: Rastrear el Mouse

La variable mouseX guarda la coordenada x, y la variable mouseY guarda la coordenada y:



```
void setup() {
  size (480, 120);
  fill (0, 102);
  smooth ();
  noStroke();
}

void draw(){
  ellipse(mouseX, mouseY, 9, 9);
}
```

En este ejemplo, cada vez que el código en el bloque *draw()* está corriendo, un nuevo círculo es dibujado en la ventana. Esta imagen fue hecha moviéndose alrededor del mouse para controlar la ubicación del círculo. Porque el relleno es establecido para ser parcialmente transparente, las áreas densas y negras muestran donde el mouse pasa más tiempo y donde se mueve más lento. Los círculos que son espaciados aparte muestran cuando el mouse se estaba moviendo más rápido.

Ejemplo 5-5: El punto te sigue

En este ejemplo, un nuevo círculo es agregado a la ventana cada vez que el código *draw()* está corriendo. Para refrescar la pantalla y sólo mostrar el círculo más nuevo, coloca la función *background()* al comienzo de *draw()* antes de que la forma sea dibujada:

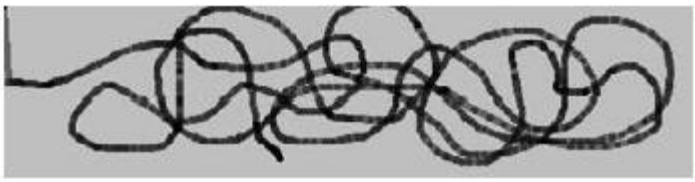
```
void setup() {
```

```
  size(480, 120);
  fill(0, 102);
  smooth();
  noStroke();
}

void draw() {
  background(204);
  ellipse(mouseX, mouseY, 9, 9);
}
```

La función background() limpia toda la ventana, así que tienes que estar seguro de colocar siempre antes otras funciones dentro de draw(); de otra manera, las formas dibujadas serán borradas.

Ejemplo 5-6: dibujar continuamente

Las variables pmouseX y pmouseY guardan la posición del mouse en el cuadro anterior. Como mouseX y mouseY son variables especiales, son actualizadas cada vez que *draw()* corre. Combinadas, pueden ser usadas para dibujar líneas continuamente conectando la ubicación actual y la más reciente:



```
void setup() {
  size(480, 120);
  strokeWeight(4);
  smooth();
  stroke(0, 102);
}

void draw() {
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Ejemplo 5-7: Ajustar el grosor sobre la marcha

Las variables pmouseX y pmouseY pueden también ser usadas para calcular la velocidad del mouse. Esto es hecho midiendo la distancia entre la actual y la más reciente ubicación del mouse. Si el mouse se mueve lentamente, la distancia es pequeña, pero si el mouse empieza a moverse rápidamente, la distancia crece. Una función llamada *dist()* simplifica este cálculo, como se muestra en el siguiente ejemplo. Aquí, la velocidad del mouse es usada para establecer el grosor de la línea dibujada:

```
void setup() {
  size(480, 120);
  smooth();
  stroke(0, 102);
}

void draw() {
  float weight = dist(mouseX, mouseY, pmouseX, pmouseY);
  strokeWeight(weight);
  line(mouseX, mouseY, pmouseX, pmouseY);
}
```

Ejemplo 5-8: Facilitando lo que hace

En el ejemplo 5-7, los valores del mouse son convertidos directamente en posiciones en la pantalla. Pero a veces quieres que esos valores sigan al mouse libremente - para atrasar detrás y crear movimientos más fluidos. Esta técnica es llamada *easing (facilitando)*. En facilitando hay dos valores: el valor actual y el valor para avanzar (ver la figura 5-1). En cada paso del programa, el valor actual se mueve un poco más cerca al valor objetivo:

```
float x;
float easing = 0.01;
float diameter = 12;

void setup() {
 size(220, 120);
 smooth();
}

void draw() {
 float targetX = mouseX;
 x += (targetX - x) * easing;
 ellipse(x, 40, 12, 12);
 println(targetX + " : " + x);
}
```

El valor de la variable x siempre se acerca al de targetX. La velocidad en la cual se atrapa con targetX es establecida con la variable *easing*, un número entre 0 y 1. Un valor pequeño para facilitar la causa de más de un retardo de un valor mayor. Con un valor facilitador de 1, no hay retardo. Cuando corres el ejemplo 5-8, los valores actuales son mostrados en la consola a través de la función *println()*. Cuando se mueve el mouse, notarás como los números

están aparte, pero cuando el mouse para de moverse, el valor x se acerca a targetX.

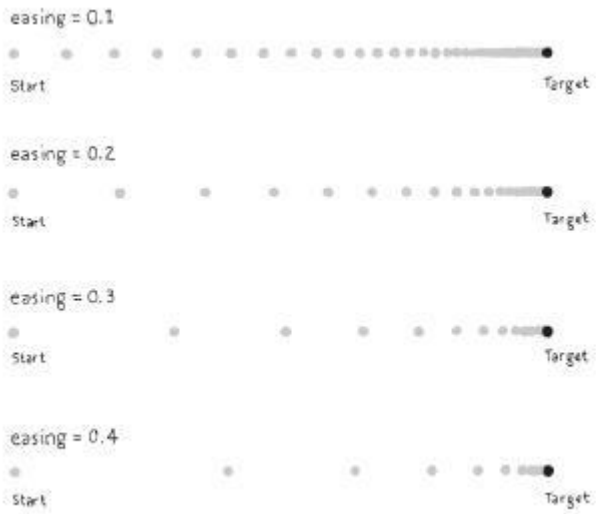


Figura 5-1. Easing.

Todo el trabajo en este ejemplo pasa en la línea que comienza en x +=. Ahí, la diferencia entre el objetivo y el valor actual es calculado, luego es multiplicado por la variable facilitadora y agregada a x para traerlo cerca al objetivo.

Ejemplo 5-9: Líneas suaves facilitadoras

En este ejemplo, la técnica facilitadora es aplicada al ejemplo 5-7. En comparación, las líneas son más suaves:

```
float x;
float y;
float px;
float py;
float easing = 0.05;

void setup() {
 size(480, 120);
 smooth();
 stroke(0, 102);
}

void draw() {
```

```
 float targetX = mouseX;
 x += (targetX - x) * easing;
 float targetY = mouseY;
 y += (targetY - y) * easing;
 float weight = dist(x, y, px, py);
 strokeWeight(weight);
 line(x, y, px, py);
 py = y;
 px = x;
}
```

Mapa

Cuando los números son usados para dibujar en la pantalla, es útil convertir los valores de una serie de números a otra.

Ejemplo 5-10: Asignar valores a una serie

La variable *mouseX* usualmente está entre 0 y el ancho de la ventana pero tal vez quieras volver a asignar estos valores a una serie diferente de coordenadas. Puedes hacer esto haciendo cálculos como dividir *mouseX* por un número para reducir el rango y luego agregar o sustraer un número para desplazarlo a la derecha o a la izquierda:

```
void setup() {
 size(240, 120);
 strokeWeight(12);
 smooth();
}

void draw() {
 background(204);
 stroke(255);
 line(120, 60, mouseX, mouseY); // White line
 stroke(0);
 float mx = mouseX/2 + 60;
 line(120, 60, mx, mouseY); // Black line
}
```

La función *map()* es más una forma general para hacer este tipo de cambios. Convierte una variable de una serie de números a otra. El primer parámetro es la variable para ser convertida, el segundo y tercer parámetros son los valores altos y bajos de esa variable, y el cuarto y quinto son los valores altos y bajos deseados. La función *map()* esconde la matemática detrás de la conversión.

Ejemplo 5-11: Mapa con la función *map()*

Este ejemplo reescribe el ejemplo 5-10 usando *map()*:

```
void setup() {
 size(240, 120);
 strokeWeight(12);
 smooth();
}
void draw() {
background(204);
stroke(255);
line(120, 60, mouseX, mouseY); // White line
stroke(0);
float mx = map(mouseX, 0, width, 60, 180);
line(120, 60, mx, mouseY); // Black line
}
```

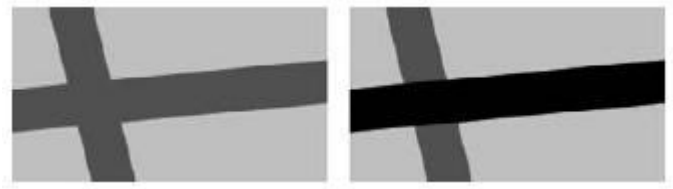
La función *map()* hace que el código sea fácil de leer, porque los valores máximos y mínimos son escritos claramente como parámetros. En este ejemplo, los valores de mouseX que están entre 0 y el ancho son convertidos a un número desde 60 (cuando mouseX es 0) hasta 180 (cuando mouseX es el ancho). Encontrarás útil la función *map()* en varios ejemplos a través de este libro.

Click

Además de la ubicación del mouse, Processing también comprueba si el botón del mouse está siendo presionado. La variable *mousepressed* tiene un valor diferente cuando el botón del mouse es presionado y cuando no. La variable *mousepressed* es un tipo de datos llamado *boolean*, lo cual significa que tiene sólo dos posibles valores: *verdadero y falso.* El valor de *mousepressed* es *verdadero* cuando el botón es presionado.

Ejemplo 5-12: Haga click en el Mouse

La variable *mousepressed* es usada junto con la declaración *if* para determinar cuándo una línea de código correrá y cuando no. Intenta con este ejemplo antes de explicar con más detalle:



```
void setup() {
 size(240, 120);
 smooth();
 strokeWeight(30);
}
```

```
void draw() {
 background(204);
 stroke(102);
 line(40, 0, 70, height);

  if (mousePressed == true) {
   stroke(0);
}
 line(0, 70, width, 50);
}
```

En este programa, el código dentro del bloque *if* corre solamente cuando el botón del mouse es presionado. Cuando un botón no es presionado, este código es ignorado. Como se discutió con la "Repetición" de *for loop* en el capítulo 4, *if* también tiene un *test* que es evaluado con *verdadero o falso*:

```
if (test) {
  statements
}
```

Cuando el examen es *verdadero*, el código dentro del bloque está corriendo; cuando el examen es *falso*, el código dentro del bloque no está corriendo. El computador determina si el examen es *verdadero o falso* evaluando la expresión dentro del paréntesis. (Si te gustaría refrescar la memoria, la discusión de expresiones relacionales está en el Ejemplo 4-6.)

El símbolo == compara los valores en la izquierda y la derecha del examen si son equivalentes. Este símbolo == es diferente del operador de asignación, el símbolo = sólo. El símbolo == pregunta "son estas cosas iguales?" y el símbolo = establece el valor de una variable.

Nota: *Es un error común, incluso para programadores con experiencia, escribir = en el código cuando quiere escribir ==. El software Processing no siempre te advertirá cuando hagas eso, así que debes ser cuidadoso.*

Alternativamente, el examen de *draw()* en el ejemplo 5-12 puede ser escrito así:

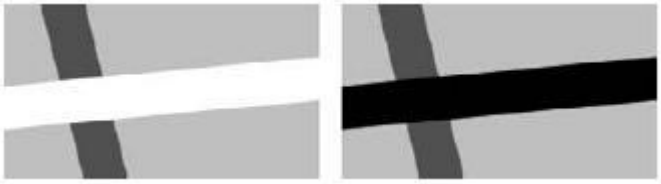
```
if (mousePressed) { }
```

Las variables Boolean, incluyendo mousePressed, no necesitan la comparación explícita con el operador ==, porque sólo pueden ser *verdaderas o falsas*.

Ejemplo 5-13: Detecta cuando no se haga click

Un sólo bloque de *if* te da la opción de correr algún código o saltarlo. Puedes extender un bloque de *if* con un bloque *else*, lo que permite a tu programa escoger entre dos opciones. El código dentro del

bloque *else* corre cuando el valor del bloque *if* da resultado *falso*. Por ejemplo, el color del trazo para un programa puede ser blanco cuando el botón del mouse no es presionado, y puede cambiar cuando el botón es presionado:



```
void setup() {
 size(240, 120);
 smooth();
 strokeWeight(30);
}

void draw() {
 background(204);
 stroke(102);
 line(40, 0, 70, height);

  if (mousePressed) {
   stroke(0);
} else {
   stroke(255);
}
line(0, 70, width, 50);
}
```

Ejemplo 5-14: Múltiples botones del Mouse

Processing también tiene tracks en los cuales el botón es presionado si tienes más de un botón en tu mouse. La variable mouseButton puede ser uno de tres valores: IZQUIERDO, CENTRO o DERECHO. Para examinar cuál botón fue presionado, se necesita el operador ==, como se muestra aquí:

```
void setup() {
 size(120, 120);
 smooth();
 strokeWeight(30);
}
```

```
void draw() {
 background(204);
 stroke(102);
 line(40, 0, 70, height);

if (mousePressed) {
      if (mouseButton == LEFT) {
      stroke(255);
} else {
      stroke(0);
   }

 line(0, 70, width, 50);
  }
}
```

Un programa puede tener muchas más estructuras *if y else* (ver las figuras 5-2) que estas que encontramos en estos cortos ejemplos. Que pueden ser encadenados junto a una larga serie con cada prueba para cada cosa diferente, y el bloque *if* puede ser integrado dentro de otro bloque *if* para hacer más compleja la decisión.

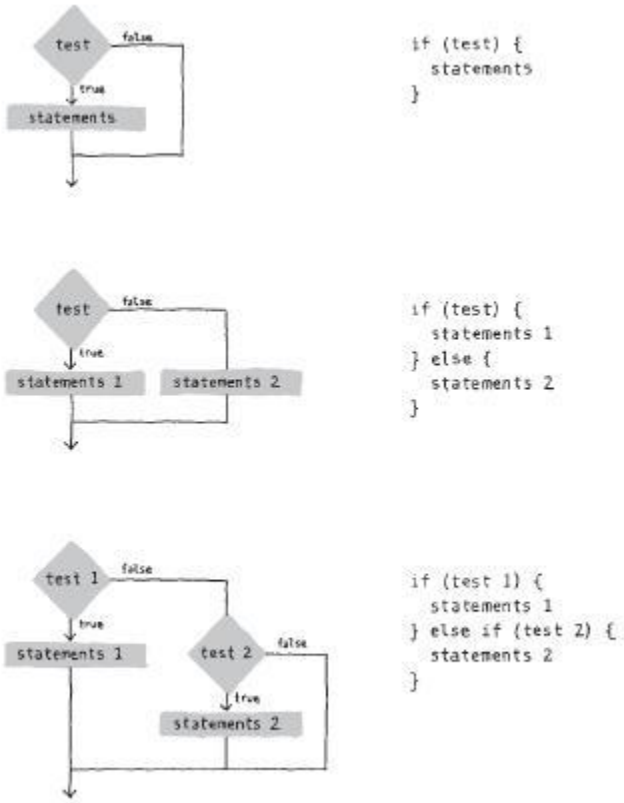


Figura 5-2. La estructura if y else hacen decisiones acerca de cuál bloque del código para ejecutar.

Ubicación

Una estructura *if* puede ser usada con los valores mouseX y mouseY para determinar la ubicación del cursor dentro de la ventana.

Ejemplo 5-15: Encontrar el cursor

Por ejemplo, este ejemplo prueba si el cursor está en el lado izquierdo o en el derecho de una línea y luego mueve la línea hacia el cursor:

```
float x;
int offset = 10;

void setup() {
 size(240, 120);
 smooth();
 x = width/2;
}

void draw() {
      background(204);
            if (mouseX > x) {
            x += 0.5;
            offset = -10;
}
      if (mouseX < x) {
      x -= 0.5;
      offset = 10;
}
 line(x, 0, x, height);
 line(mouseX, mouseY, mouseX + offset, mouseY - 10);
 line(mouseX, mouseY, mouseX + offset, mouseY + 10);
 line(mouseX, mouseY, mouseX + offset*3, mouseY);
}
```

Para escribir programas que tengan interfaces gráficas de usuario (botones, casillas de verificación, barras de desplazamiento, etc), necesitamos escribir un código que sepa cuando el cursor está dentro y cerrado en el área de la pantalla. Los siguientes dos ejemplos nos muestra como mirar si el cursor está dentro de un círculo y un rectángulo. El código está escrito de una forma modular con variables, así que puede ser usado para mirar cualquier círculo y rectángulo cambiando los valores.

Ejemplo 5-16: Los límites de un círculo

Para el examen del círculo, usamos la función dist() para establecer la distancia desde el centro del círculo hasta el cursor, luego examinamos para ver si la distancia es menor que el radio del círculo (ver la figura 5-3). Si es así, sabemos que estamos adentro. En este ejemplo, cuando el cursor está dentro del área del círculo, si tamaño incrementa:

```
int x = 120;
int y = 60;
int radius = 12;

void setup() {
  size(240, 120);
  smooth();
  ellipseMode(RADIUS);
}

void draw() {
  background(204);
  float d = dist(mouseX, mouseY, x, y);
  if (d < radius) {
    radius++;
    fill(0);
  } else {
    fill(255);
  }
  ellipse(x, y, radius, radius);
}
```

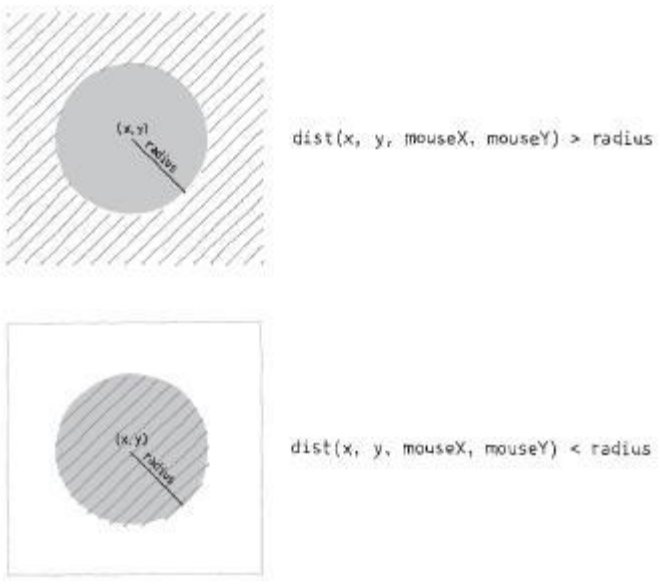


Figura 5-3. Círculo de prueba de vuelo.

Ejemplo 5-17: Los límites de un rectángulo

Usamos otra prueba para ver si el cursor está dentro del rectángulo. Hacemos cuatro exámenes separados para mirar si el cursor está en el lado correcto de cada borde del rectángulo, luego comparamos cada examen y si todos son *verdaderos*, sabemos que el cursor está dentro. Este es ilustrado en la figura 5-4. Cada paso es simple, pero se ve complicado cuando están todos juntos:

```
int x = 80;
int y = 30;
int w = 80;
int h = 60;

void setup () {
  size(240, 120);
}

void draw() {
  background(204);
  if ((mouseX > x) && (mouseX < x+w) &&
      (mouseY > y) && (mouseY < y+h)) {
    fill(0);
  } else {
```

```
    fill(255);
  }
  rect(x, y, w, h);
}
```

El examen en la declaración de *if* es un poco más complicado como hemos visto. Cuatro exámenes individuales (e.g..mouseX > x) son combinados con el operador lógico AND, el símbolo &&, para asegurar que cada expresión relacional en la secuencia es *verdadera*. Si alguna de ellas es f*alsa*, todo el examen es *falso* y el color del relleno no será establecido a negro. Esto es explicado más detalladamente en la entrada para &&.

Tipo

Processing rastrea cuando alguna tecla del teclado es presionada, así como la última tecla presionada. Como la variable mousePressed, la variable keyPressed es *verdadera* cuando alguna tecla es presionada, y *falsa* cuando ninguna tecla es presionada.

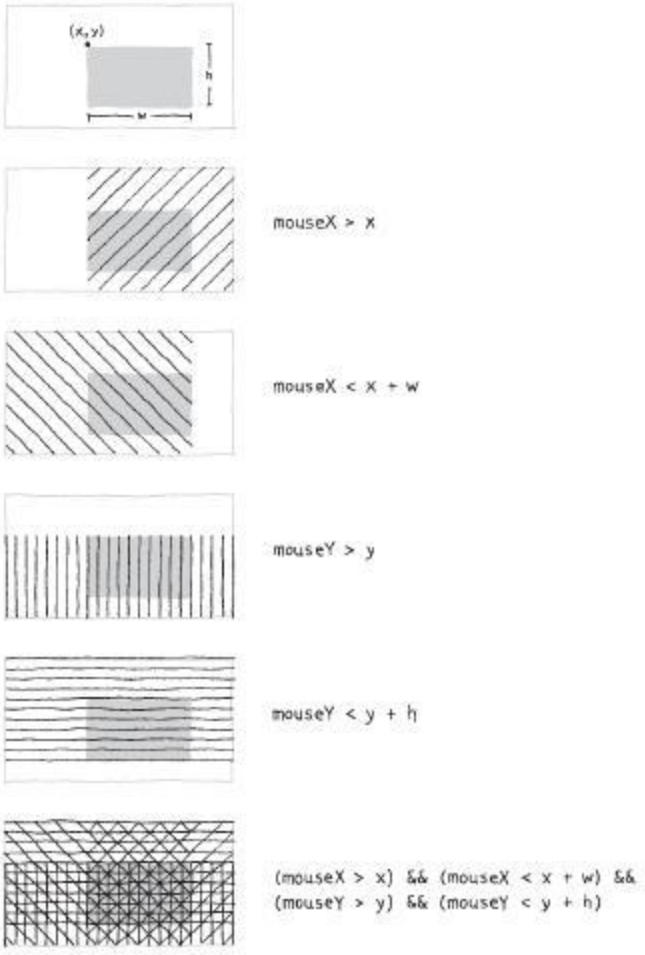
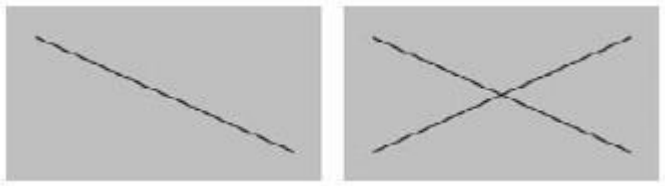


Figura 5-4. Rectángulo de prueba de vuelo.

Ejemplo 5-18: Pulsar una tecla

En este ejemplo, la segunda línea es dibujada solamente cuando una tecla es presionada:



```
void setup() {
  size(240, 120);
  smooth();
}

void draw() {
  background(204);
  line(20, 20, 220, 100);
    if (keyPressed) {
      line(220, 20, 20, 100);
  }
}
```

La variable *key* guarda la tecla que ha sido presionada recientemente. El tipo de datos para *key* es *char*, el cual es la forma corta de "carácter" pero usualmente pronunciada como la primera sílaba de "charcoal" (carbón vegetal). Una variable *char* puede guardar cualquier carácter, el cual incluye letras del alfabeto, números, y símbolos. Diferente de un valor *string* (ver el ejemplo 6-8), el cual es distinguido por sus dobles comillas, el tipo de datos de *char* es especificado por una sola coma. Así es como una variable char es declarada y asignada:

```
char c = 'A'; // Declares and assigns 'A' to the variable c
```

y estos ejemplos causarán error:

```
char c = "A"; // Error! can't assign a string to a char
char h = A;   // Error! Missing the single quotes from 'A'
```

A diferencia de la variable *boolean*, *keyPressed* la cual los vuelve *falsos* cada vez que una tecla es liberada. La variable *key* guarda su valor hasta que la siguiente tecla sea presionada. El siguiente ejemplo usa el valor de *key* para dibujar un carácter en la pantalla. Cada vez que una nueva tecla es presionada, el valor se actualiza y dibuja un nuevo carácter, pero cuando los presionas, nada está siendo dibujado.

Ejemplo 5-19: Dibujar algunas letras

Este ejemplo introduce la función *textSize()* para establecer el tamaño de las letras, la función *textAlign()* para centrar el texto en su coordenada x, y la función *text()* para dibujar las letras. Estas funciones son discutidas con más detalle en las páginas 84-85.


```
void setup() {
  size(120, 120);
  textSize(64);
  textAlign(CENTER);
}

void draw() {
  background(0);
  text(key, 60, 80);
}
```

Usando una estructura *if*, podemos examinar para ver si una tecla en específico es presionada y escoger para dibujar algo en la pantalla como respuesta.

Ejemplo 5-20: Comprobar las teclas específicas

En este ejemplo, Examinamos la H o la N para ser escritas. Usamos el operador de comparación, el símbolo ==, para ver si el valor *key* es igual a los caracteres que estamos buscando:

```
void setup() {
 size(120, 120);
 smooth();
}

void draw() {
 background(204);
  if (keyPressed) {
   if ((key == 'h') || (key == 'H')) {
    line(30, 60, 90, 60);
```

```
  }

  if ((key == 'n') || (key == 'N')) {
     line(30, 20, 90, 100);
   }
 }

line(30, 20, 30, 100);
line(90, 20, 90, 100);
}
```

Cuando vemos que H o N son presionadas, necesitamos comprobar que tanto para las letras minúsculas como para las mayúsculas, si alguien presiona la tecla de mayúsculas o tiene activado el bloqueo. Combinamos los dos exámenes con una lógica OR, el símbolo ||. Si trasladamos la segunda declaración de *if* en este ejemplo a un lenguaje sencillo, dice, "si la tecla 'h' es presionada OR la tecla 'H' es presionada." Diferente del ejemplo 5-17 con la lógica AND (el símbolo &&). Solamente una de estas expresiones necesita ser *verdadera* para que todo el *test* sea *verdadero*.

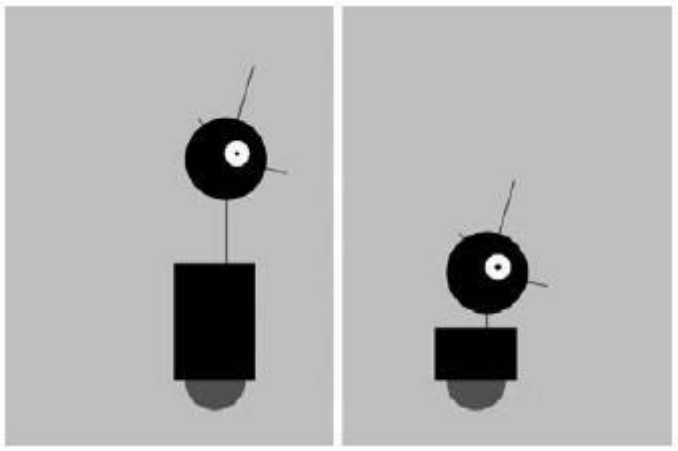
Algunas teclas son más difíciles de detectar, porque no están atadas a una letra en particular. Teclas como Shift, Alt, y la tecla de flecha están codificadas y requieren un paso extra para figurar si son presionadas. Primero, necesitamos comprobar el código con la variable *keyCode* para ver cuál es la tecla. Los valores de *keyCode* usados más frecuentemente son ALT, CONTROL, y SHIFT, así como las teclas de flechas, UP, DOWN, LEFT y RIGHT.

Ejemplo 5-21: Moverse con las teclas de flecha

El siguiente ejemplo muestra cómo usar las teclas de flecha izquierda o derecha para mover un rectángulo:

```
int x = 215;

void setup() {
 size(480, 120);
}

void draw() {
 if (keyPressed && (key == CODED)) { // If it's a coded key
    if (keyCode == LEFT) {           // If it's the left arrow
    x--;
    } else if (keyCode == RIGHT) {   // If it's the right arrow
    x++;
   }
  }
rect(x, 45, 50, 50);
}
```

Robot 3: Respuesta



Este programa usa las variables introducidas en Robot 2 (ver "Robot 2: variables" en el capítulo 4) y hace que sea posible cambiarlas mientras el programa está corriendo así que las figuras responden al mouse. El código dentro del bloque *draw()* corre varias veces cada segundo. En cada fotograma, las variables definidas en el programa cambian en respuesta a las variables mouseX y mousePressed.

El valor de mouseX controla la posición del robot con una técnica muy fácil, de modo que sus movimientos sean menos instantáneos y por lo tanto se sienta más natural. Cuando el botón del mouse sea presionado, el valor de neckHeight y bodyHeight cambia para hacer más corto al robot.

```
float x = 60;           // X-coordinate
float y = 440;          // Y-coordinate
int radius = 45;        // Head Radius
int bodyHeight = 160;   // Body Height
int neckHeight = 70;    // Neck Height

float easing = 0.02;

void setup() {
 size(360, 480);
 smooth();
 strokeWeight(2);
 ellipseMode(RADIUS);
}

void draw() {

int targetX = mouseX;
x += (targetX - x) * easing;

if (mousePressed) {
 neckHeight = 16;
```

```
 bodyHeight = 90;
} else {
 neckHeight = 70;
 bodyHeight = 160;
}

float ny = y - bodyHeight - neckHeight - radius;

background(204);

// Neck
stroke(102);
line(x+12, y-bodyHeight, x+12, ny);

// Antennae
line(x+12, ny, x-18, ny-43);
line(x+12, ny, x+42, ny-99);
line(x+12, ny, x+78, ny+15);

// Body
noStroke();
fill(102);
ellipse(x, y-33, 33, 33);
fill(0);
rect(x-45, y-bodyHeight, 90, bodyHeight-33);

// Head
fill(0);
ellipse(x+12, ny, radius, radius);
fill(255);
ellipse(x+24, ny-6, 14, 14);
fill(0);
ellipse(x+24, ny-6, 3, 3);
}
```

CAPITULO 6

6 Media

Processing es capaz de dibujar más que simples líneas y formas. Es hora de aprender cómo cargar imágenes raster, archivos de vector, y fondos en nuestros programas para extender las posibilidades visuales de fotografía, diagramas detallados, y diversos tipos de letra.

Processing utiliza una carpeta llamada *data* para guardar ciertos archivos, así nunca tendrás que pensar acerca de su ubicación cuando estés haciendo un boceto que va a correr en el escritorio, en la Web, o en un dispositivo móvil. Hemos colocado algunos archivos media online para que los uses en los ejemplos de este capítulo: *http://www.processing.org/learning/books/media.zip*

Descarga este archivo, descomprímelo en el escritorio (o donde creas más conveniente). Y haz una nota mental de su ubicación.

Nota: *Para descomprimir en Mac OS X, haz doble click sobre el archivo, y este creará una carpeta llamada media. En Windows, haz doble click en el archivo media.zip, el cual te abrirá una nueva ventana. En esa ventana, arrastra la carpeta media hacia el escritorio.*

Crea un nuevo dibujo, y selecciona Add File desde el menú de dibujo. Encuentra el *lunar.jpg* desde la carpeta media que descomprimiste y selecciónalo. Si todo estuvo bien, en el área de mensaje leerás "1 archivo agregado al dibujado".

Para comprobar el archivo, selecciona Show Sketch Folder en el menú de dibujo. Debes ver una carpeta llamada *data*, con una copia de *lunar.jpg* dentro. Cuando agregas un archivo al dibujo, la carpeta *data*, automáticamente será creada. En lugar de usar el comando Add File del menú, puedes hacer lo mismo arrastrando los archivos hacia el área del editor de la ventana de Processing. Los archivos serán copiado a la carpeta *data* de la misma forma (y la carpeta *data* será creada si no existe).

También puedes crear la carpeta *data* fuera de Processing y copiar los archivos tu mismo. No puedes tener el mensaje diciendo que los archivos han sido agregados, pero este es un método útil cuando estás trabajando con un gran número de archivos.

Nota: En Windows y Mac OS X, las extensiones están escondidas por defecto. Es una buena idea cambiar la opción de modo que siempre puedas ver todo el nombre de tus archivos. En Mac OS X, selecciona las preferencias desde el menú Finder, y luego asegúrate que "Mostrar todas las extensiones de los archivos" este marcado en la ficha de opciones avanzadas. En Windows, busca en las "Opciones de carpeta", y establece la opción ahí.

Imágenes

Hay tres pasos a seguir antes de poder dibujar una imagen en la pantalla:

1. Agrega la imagen a la carpeta de dibujos *data* (las instrucciones se dieron previamente).

2. Crear una variable *PImage* para guardar la imagen.

3. Cargar la imagen en una variable con *loadimage()*.

Ejemplo 6-1: Cargar una imagen

Después de que los tres pasos se hayan realizado, podrás dibujar la imagen en la pantalla con la función *image()*. El primer parámetro de *image()* específica para dibujar la imagen; el segundo y tercero establece las coordenadas X y Y.

```
PImage img;

void setup() {
 size(480, 120);
 img = loadImage("lunar.jpg");
  }

void draw() {
 image(img, 0, 0);
}
```

El cuarto y quinto parámetro son opcionales y establecen la anchura y la altura para dibujar la imagen. Si el cuarto y quinto parámetros no son usados, la imagen es dibujada en el tamaño en el que fue creado.

Estos próximos ejemplos muestran cómo trabajar con más de una imagen en el mismo programa y cómo cambiar de tamaño una imagen.

Ejemplo 6-2: Cargar más imágenes

Para este ejemplo, necesitarás agregar el archivo *capsule.jpg* (encuéntralo en la carpeta *media* que ya has descargado) a tu dibujo usando uno de los métodos descritos anteriormente.

```
PImage img1;
PImage img2;

void setup() {
 size(480, 120);
 img1 = loadImage("lunar.jpg");
 img2 = loadImage("capsule.jpg");
}

void draw() {
 image(img1, -120, 0);
 image(img1, 130, 0, 240, 120);
 image(img2, 300, 0, 240, 120);
}
```

Ejemplo 6-3: Pasando el Mouse alrededor con las imágenes

Cuando los valores de *mouseX* y *mouseY* son usados como parte de los cuarto y quinto parámetros de *image()*, el tamaño de la imagen cambia con el movimiento del mouse:

```
PImage img;

void setup() {
 size(480, 120);
 img = loadImage("lunar.jpg");
}

void draw() {
 background(0);
 image(img, 0, 0, mouseX * 2, mouseY * 2);
}
```

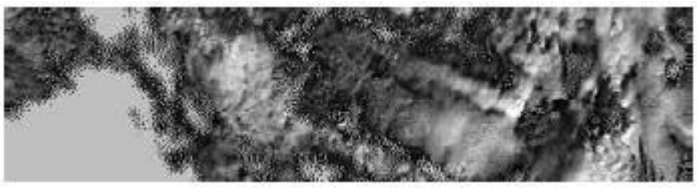
Nota: *Cuando una imagen se muestra más larga o más pequeña de su tamaño actual, puede distorsionarse. Sea cuidadoso para preparar sus imágenes en los tamaños que serán usados. Cuando el tamaño del display de una imagen es cambiado con la función image(), la imagen actual en la unidad de disco duro no cambia.*

Processing puede cargar y mostrar imágenes raster en formatos JPEG, PNG y GIF. (Formas de vector en el formato SVG pueden ser mostradas en

una forma diferente, como describiremos en "formas" más adelante en este capítulo.) Puedes convertir imágenes de los formatos JPEG, PNG y GIF usando programas como GIMP y Photoshop. La mayoría de las cámaras digitales salvan las imágenes en JPEG, pero usualmente necesitamos reducirlas a un tamaño que fue usado antes con Processing. Una típica cámara digital crea una imagen que es varias veces más larga que el área de dibujo de la mayoría de los dibujos de Processing, así que el cambio de tamaño a varias imágenes que fueron agregadas antes a los dibujos de la carpeta *data*, corren más eficientemente, y pueden salvar espacio en el disco.

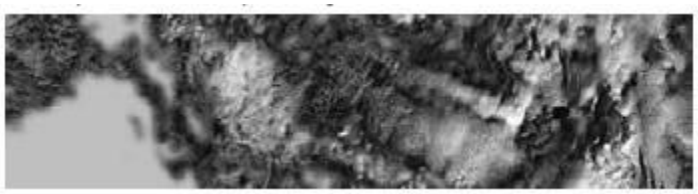
Las imágenes GIF y PNG apoyan la transparencia, lo cual significa que los pixeles pueden ser invisible o parcialmente visibles (Recordemos la discusión de los valores alfa de *color()* en las páginas 26-29). Las imágenes GIF tienen 1-bit de transparencia, lo cual significa que los píxeles son totalmente blancos o totalmente transparentes. Las imágenes PNG tienen 8-bit de transparencia, lo cual significa que cada pixel puede tener un nivel variable de opacidad. El siguiente ejemplo muestra la diferencia, usando los archivos *clouds gif* y *clouds.png* que se encuentran en la carpeta *media* que descargaste. Asegúrate de agregarlos al dibujo antes de intentar cada ejemplo.

Ejemplo 6-4: Transparencia con un GIF



```
PImage img;

void setup() {
  size(480, 120);
  img = loadImage("clouds.gif");
}
void draw() {
  background(255);
  image(img, 0, 0);
  image(img, 0, mouseY * -1);
}
```

Ejemplo 6-5: Transparencia con un PNG



```
PImage img;

void setup() {
 size(480, 120);
 img = loadImage("clouds.png");
}

void draw() {
 background(204);
 image(img, 0, 0);
 image(img, 0, mouseY * -1);
}
```

Nota: *Recuerda incluir la extensión del archivo .gif, .jpg, o .png cuando cargues la imagen. También, asegúrate de que el nombre de la imagen esté escrito exactamente como aparece en el archivo, incluyendo todas las letras. Y si te falta alguna, lee la nota que fue incluida anteriormente en este capítulo acerca de cómo asegurarse que la extensión del archivo esté visible en Mac OS X y Windows.*

Fuentes

Processing puede mostrar texto en varias fuentes distintas a las establecidas. Antes de mostrar el texto en otro tipo de letra, necesitas convertir una de las fuentes de tu computador en formato VLW, el cual guarda cada letra como una pequeña imagen. Para hacer esto, selecciona Crear Fuente del menú de herramientas para abrir la caja de diálogo (Figura 6-1). Especifique la fuente a la que quieres convertir, así como también el tamaño y si quieres que sea suave (anti-aliased).

Figura 6-1. Crear herramienta de fuente.

Nota: *Haz la selección de la fuente cuidadosamente, considerando lo siguiente: Crear una fuente del tamaño que quieras para usarla en tu dibujo (o más grande), pero recuerda que los tamaños grandes incrementan el tamaño de la fuente del archivo. Seleccionar la opción de los caracteres solamente si vas a usar caracteres no romanos, como textos japoneses o chinos, porque esto también incrementa significativamente el tamaño de archivo.*

Cuando das click en el botón OK para crear una herramienta de fuente, la fuente VLW es creada y colocada en la carpeta *data* del dibujo. Ahora es posible cargar una fuente y agregar palabras al dibujo. Esta parte es similar a trabajar con imágenes, pero hay un paso extra:

1. Agregar la fuente a la carpeta *data* del dibujo (instrucciones dadas anteriormente).

2. Crear una variable PFont para guardar la fuente.

3. Cargar la fuente en la variable con loadFont().

4. Usar el comando textFont() para establecer la fuente actual.

Ejemplo 6-6: Dibujar con fuentes

Ahora puedes dibujar estas letras en la pantalla con la función *text()*, y puedes cambiar el tamaño con *textSize()*. Para este ejemplo, necesitarás usarlo para crear la herramienta de fuente para crear una

fuente VLW (y modificar la línea *loadFont()* para usarlo, o puedes usar *AndaleMono-36.vlw* desde la carpeta *media*:

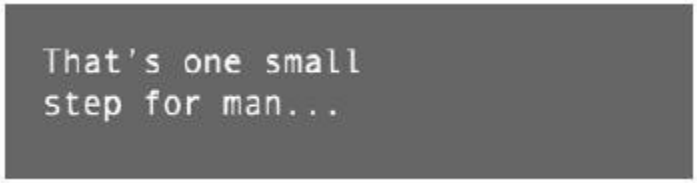

```
PFont font;

void setup() {
 size(480, 120);
 smooth();
 font = loadFont("AndaleMono-36.vlw");
 textFont(font);
}

void draw() {
 background(102);
 textSize(36);
 text("That's one small step for man...", 25, 60);
 textSize(18);
 text("That's one small step for man...", 27, 90);
}
```

El primer parámetro para *text()* es el carácter para dibujar en la pantalla. (Nótese que los caracteres están encerrados entre paréntesis.) El segundo y tercer parámetros establecen la ubicación vertical y horizontal. La ubicación es relativa a la base del texto (ver figura 6-2).

Figura 6-2. Topografía de las coordenadas.

Ejemplo 6-7: Dibujar en una caja de texto

También puedes establecer una caja de texto para dibujar dentro de ella, agregando cuatro o cinco parámetros que especifiquen la anchura y la altura de la caja:



```
PFont font;

void setup() {
 size(480, 120);
 font = loadFont("AndaleMono-24.vlw");
 textFont(font);
}

void draw() {
 background(102);
 text("That's one small step for man...", 26, 30, 240, 100);
}
```

Ejemplo 6-8: Guardar el texto en un *String*

En el previo ejemplo, las palabras dentro de la función *text()* empiezan a hacerse más difíciles para leer. Podemos guardar estas palabras en una variable para hacer el código más modular. El tipo de datos *String* es usado para guardar datos de texto. Aquí hay una nueva versión del previo ejemplo que usa un *String*:

```
PFont font;

String quote = "That's one small step for man...";

void setup() {
 size(480, 120);
 font = loadFont("AndaleMono-24.vlw");
 textFont(font);
}

void draw() {
 background(102);
 text(quote, 26, 30, 240, 100);
}
```

Hay una serie de funciones adicionales que afectan como son mostradas las letras en la pantalla. Estas son explicadas, con ejemplos, en la categoría topografía de la referencia de Processing.

Formas

Si haces un vector de formas en un programa como Inkscape o illustrator, puedes cargarlos en Processing directamente. Esto es útil para formas que no deseas crear con las funciones de dibujo de Processing. Con imágenes, necesitas agregarlas a tu dibujo antes de que sean cargadas.

Hay tres pasos para dibujar y cargar un archivo SVG:

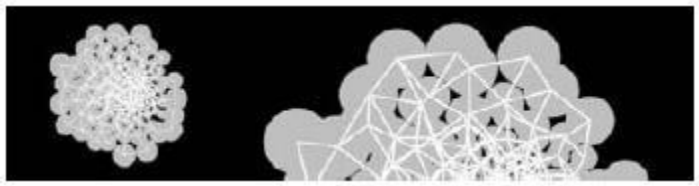
1. Agregar un archivo SVG a la carpeta *data* del dibujo.

2. Crear una variable *PShape* para guardar el archivo del vector.

3. Cargar el archivo del vector en la variable con *loadShape()*.

Ejemplo 6-9: Dibujar con formas

Después de seguir estos pasos, puedes dibujar la imagen en la pantalla con la función *shape()*:



```
PShape network;

void setup() {
 size(480, 120);
 smooth();
 network = loadShape("network.svg");
}

void draw() {
 background(0);
 shape(network, 30, 10);
 shape(network, 180, 10, 280, 280);
}
```

Los parámetros para *shape()* son similares a los de image(). El primer parámetro le dice a *shape()* en cual SVG puede dibujar y el siguiente par de parámetros establecen la posición. Un cuarto y quinto parámetro opcional establece la anchura y la altura.

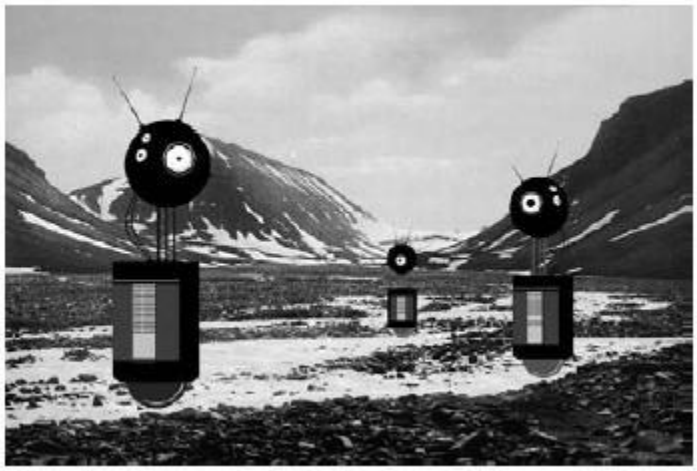
Ejemplo 6-10: Formas de escala

A diferencia de las imágenes raster, las formas de vector pueden escalar a cualquier tamaño sin perder resolución. En este ejemplo, la forma es escalada basada en la variable *mouseX*, y la función *shapeMode()* es usada para dibujar la forma desde el centro, en lugar de tener la posición predeterminada, la esquina superior izquierda:

```
PShape network;

void setup() {
 size(240, 120);
 smooth();
 shapeMode(CENTER);
 network = loadShape("network.svg");
}

void draw() {
 background(0);
 float diameter = map(mouseX, 0, width, 10, 800);
 shape(network, 120, 60, diameter, diameter);
}
```

Nota: *Hay limitaciones para el tipo de archivo SVG que quieras cargar. Processing no es compatible con todas las características de SVG. Ver la referencia de Processing para PShape para más detalles.*

Robot 4: Media



A diferencia de los robots creados desde líneas y rectángulos, dibujados en Processing en los previos capítulos, estos robots fueron creados con un vector dibujando en el programa. Para algunas formas, es a menudo más fácil apuntar y hacer click en una herramienta de software como Inkscape o Illustrator que definir las formas con coordenadas de código.

Existe un trade-off para seleccionar una técnica de creación de una imagen sobre otra. Cuando las formas están definidas en Processing, existe más flexibilidad para modificarlos mientras el programa está corriendo. Si las formas están definidas en otra parte y luego son cargadas en Processing, los cambios son limitados a la posición, el ángulo, y el tamaño. Cuando cargas cada robot desde un archivo SVG,

como muestra el ejemplo, la variación de las características en Robot 2 (ver "Robot 2: Variables" en el capítulo 4) son imposibles.

Las imágenes pueden ser cargadas en un programa para traer visuales creados en otros programas o capturados con una cámara. Con esta imagen en el fondo, nuestros robots están robots están explorando formas de vida en Noruega en el alba del siglo 20.

Los archivos SVG y PNG son usados en este ejemplo pueden ser descargados desde *http://www.processing.org/learning/books/media.zip.*

```
PShape bot1;
PShape bot2;
PShape bot3;
PImage landscape;

float easing = 0.05;
float offset = 0;

void setup() {
 size(720, 480);
 bot1 = loadShape("robot1.svg");
 bot2 = loadShape("robot2.svg");
 bot3 = loadShape("robot3.svg");
 landscape = loadImage("alpine.png");
 smooth();
}

void draw() {
 // Set the background to the "landscape" image; this image
 // must be the same width and height as the program
 background(landscape);


// Set the left/right offset and apply easing to make
// the transition smooth
float targetOffset = map(mouseY, 0, height, -40, 40);
offset += (targetOffset - offset) * easing;

// Draw the left robot
shape(bot1, 85 + offset, 65);

// Draw the right robot smaller and give it a smaller offset
float smallerOffset = offset * 0.7;
shape(bot2, 510 + smallerOffset, 140, 78, 248);

// Draw the smallest robot, give it a smaller offset
smallerOffset *= -0.5;
shape(bot3, 410 + smallerOffset, 225, 39, 124);
}
```

CAPITULO 7

7 Movimiento

Como él dibujo a un libro, la animación en la pantalla es creada dibujando una imagen, luego dibujando ligeramente otra imagen diferente, luego otra, y así sucesivamente. La ilusión de que el movimiento es fluido es creada por la *persistencia de la visión*. Cuando un conjunto de imágenes similares es mostrada rápidamente, nuestros cerebros transforman estas imágenes en movimiento.

Ejemplo 7-1: Ver la velocidad del cuadro

Para crear movimientos suaves, Processing intenta correr el código dentro de draw() a 60 frames cada segundo. Para confirmar la velocidad del cuadro, corre este programa y mire los valores que se muestran en la consola. La variable *frameRate* rastrea la velocidad del programa.

```
void draw() {
  println(frameRate);
}
```

Ejemplo 7-2: Establecer la velocidad del cuadro

La función *frameRate()* cambia la velocidad en la cual el programa corre. Para ver el resultado, descomentar diferentes versiones de *frameRate()* en este ejemplo:

```
void setup() {
  frameRate(30);   // Thirty frames each second
  //frameRate(12); // Twelve frames each second
  //frameRate(2);  // Two frames each second
  //frameRate(0.5);// One frame every two seconds
}

void draw() {
  println(frameRate);
}
```

Nota: *Processing intenta correr el código a 60 frames cada segundo, pero toma alrededor de 1/60th de segundo correr el método draw(), luego la velocidad del cuadro disminuirá. La función frameRate()especifica que solamente la mayor y la velocidad actual del cuadro para cualquier programa depende del código que esté corriendo el computador.*

Velocidad y dirección

Para crear ejemplos de un movimiento fluido, usamos un tipo de datos llamado *float*. Este tipo de variable guarda números con decimales, lo

cual nos da mejor resolución para trabajar con movimiento. Por ejemplo, cuando usamos ints, Los más lento que puedes mover cada frame es un pixel al tiempo (1,2,3,4...), pero con *floats*, puedes moverlos tan lento como tú quieras (1.01, 1.01, 1.02, 1.03,...).

Ejemplo 7-3: Mover una forma

El siguiente ejemplo mueve una forma de izquierda a derecha actualizando la variable x:

```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
 size(240, 120);
 smooth();
 ellipseMode(RADIUS);
}

void draw() {
 background(0);
 x += speed; // Increase the value of x
 arc(x, 60, radius, radius, 0.52, 5.76);
}
```

Cuando corres este código, notarás que las formas se mueven frente a la derecha de la pantalla cuando el valor de la variable x es más grande que el del ancho de la ventana. El valor de x continúa incrementándose, pero la forma ya no es visible.

Ejemplo 7-4: Envolver:

Existen varias alternativas para este comportamiento, el cual puedes escoger de acuerdo a tu preferencia. Primero, extenderemos el código para mostrar cómo mover de nuevo la forma hacia la orilla izquierda después que desaparece de la derecha. En este caso, dibuje la pantalla como un cilindro aplanado con la forma moviéndose alrededor del exterior para regresar a su punto de comienzo:

```
int radius = 40;
float x = -radius;
float speed = 0.5;

void setup() {
 size(240, 120);
 smooth();
 ellipseMode(RADIUS);
}

void draw() {
 background(0);
 x += speed;                        // Increase the value of x
   if (x > width+radius) {          // If the shape is off screen,
   x = -radius;                     // move to the left edge
  }
arc(x, 60, radius, radius, 0.52, 5.76);
}
```

En cada vuelta a través de *draw()*, las pruebas de código se hacen para ver si el valor de x ha incrementado más allá del ancho de la pantalla (más el radio de la forma). Si es así, establecemos el valor de x a un valor negativo, de manera que si continúa incrementándose, entrará en la pantalla desde la izquierda. Ver la figura 7-1 para un diagrama de cómo funciona.

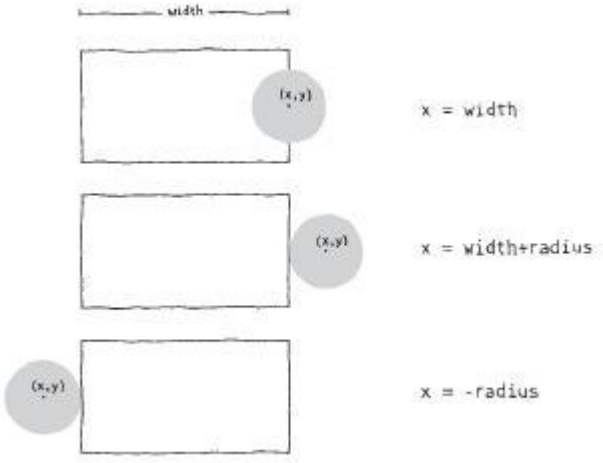


Figura 7-1. Probando para el borde izquierdo de la pantalla.

Ejemplo 7-5: Rebotar en la pared

En este ejemplo, extendemos el ejemplo 7-3 para tener los cambios de dirección de la forma cuando golpea un borde, en lugar de envolver alrededor de lado izquierdo. Para hacer que esto pase, agregamos una nueva variable para guardar la dirección de la forma. Un valor de dirección de 1 mueve la forma hacia la derecha, y un valor de -1 mueve la forma hacia la izquierda:

```
int radius = 40;
float x = 110;
float speed = 0.5;
int direction = 1;

void setup() {
 size(240, 120);
 smooth();
 ellipseMode(RADIUS);
}

void draw() {
 background(0);
 x += speed * direction;
  if ((x > width-radius) || (x < radius)) {
  direction = -direction;                    // Flip direction
 }

if (direction == 1) {
  arc(x, 60, radius, radius, 0.52, 5.76);  // Face right
 } else {
  arc(x, 60, radius, radius, 3.67, 8.9);   // Face left
 }
}
```

Cuando la forma alcanza una orilla, este código mueve de un tirón la dirección de la forma cambiando el signo de la variable *direction*. Por ejemplo, si la variable *direction* es positiva cuando la forma toque la orilla, el código la mueve a negativa.

Interpolación

A veces quieres animar una forma para ir de un punto de la pantalla hacia otro. Con unas cuantas líneas de código, puedes establecer la posición de inicio y la posición final, luego calcular las posiciones del medio (tween) en cada frame.

Ejemplo 7-6: Calcular las posiciones del medio

Para hacer de este ejemplo un código modular, hemos creado un grupo de variables en la parte de arriba. Corre este código unas cuantas veces y cambia los valores para ver como este código puede mover una forma desde una ubicación hacia otra en una gama de velocidades. Cambia la variable *step* para alterar la velocidad:

```
int startX = 20;          // Initial x-coordinate
int stopX = 160;          // Final x-coordinate
int startY = 30;          // Initial y-coordinate
int stopY = 80;           // Final y-coordinate
float x = startX;         // Current x-coordinate
float y = startY;         // Current y-coordinate
float step = 0.005;       // Size of each step (0.0 to 1.0)
float pct = 0.0;          // Percentage traveled (0.0 to 1.0)

void setup() {
 size(240, 120);
 smooth();
}

void draw() {
background(0);
 if (pct < 1.0) {
  x = startX + ((stopX-startX) * pct);
  y = startY + ((stopY-startX) * pct);
  pct += step;
 }
ellipse(x, y, 20, 20);
}
```

Azar

A diferencia del movimiento suave y lineal, común en los gráficos de computador, el movimiento en el mundo físico suele ser complicado. Por ejemplo, piensa en una hoja flotando en el agua, o una hormiga arrastrándose por un terreno áspero. Podemos simular las impredecibles cualidades del mundo generando números al azar. La función *random()* calcula estos valores; podemos establecer un rango para ajustar la cantidad de desorden en una programa.

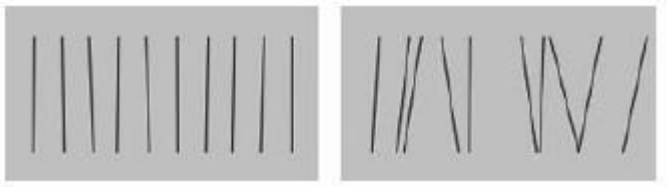
Ejemplo 7-7: Generar valores al azar

El siguiente corto ejemplo muestra valores al azar en la consola, con un rango limitado por la posición del mouse. La función *random()* siempre regresa a un valor del punto flotante, así que asegúrate en el lado izquierdo del operador de asignación (=) sea un *float* como lo es aquí:

```
void draw() {
  float r = random (0, mouseX);
  println(r);
}
```

Ejemplo 7-8: Dibujar al azar

El siguiente ejemplo se basa en el ejemplo 7-7; usa los valores de *random()* para cambiar la posición de las líneas en la pantalla. Cuando el mouse está en la parte izquierda de la pantalla, el cambio es pequeño; si se mueve hacia la derecha, el valor de *random()* incrementa y el movimiento se vuelve más exagerado. Porque la función *random()* está dentro de *for loop*, un nuevo valor random es calculado para cada punto de cada línea:



```
void setup() {
 size(240, 120);
 smooth();
}

void draw() {
 background(204);
  for (int x = 20; x < width; x += 20) {
   float mx = mouseX / 10;
   float offsetA = random(-mx, mx);
   float offsetB = random(-mx, mx);
   line(x + offsetA, 20, x - offsetB, 100);
  }
}
```

Ejemplo 7-9: Mover formas al azar

Cuando se utiliza random() para mover formas alrededor de la pantalla, los valores al azar pueden generar formas más naturales en apariencia. En el siguiente ejemplo, la posición del círculo es modificada por valores al azar en cada dibujo a través de draw(). Porque la función background() no es usada, las ubicaciones anteriores se remontan:

```
float speed = 2.5;
int diameter = 20;
float x;
float y;

void setup() {
 size(240, 120);
 smooth();
 x = width/2;
 y = height/2;
}

void draw() {
 x += random(-speed, speed);
 y += random(-speed, speed);
 ellipse(x, y, diameter, diameter);
}
```

si ves este ejemplo el tiempo suficiente, podrás ver el círculo dejando la ventana y volviendo. Esto se deja al azar, pero podemos agregarle algunas estructuras *if* o usar la función *constrain()* para que el círculo no deje la pantalla. La función c*onstrain*() limita un valor en un rango específico, en el cual puede ser usado para guardar un X y Y dentro de los límites de la pantalla. Reemplazando *draw*() en el código anterior con lo siguiente te asegurarás de que la elipse permanecerá en la pantalla:

```
void draw() {
 x += random(-speed, speed);
 y += random(-speed, speed);
 x = constrain(x, 0, width);
 y = constrain(y, 0, height);
 ellipse(x, y, diameter, diameter);
}
```

Nota: *La función randomSeed() puede ser usada para forzar a random() a producir la misma secuencia de números cada vez que el programa esté corriendo. Esto se describe más adelante en la referencia de Processing.*

Temporizadores

Cada programa de Processing cuenta la cantidad de tiempo que ha pasado desde el comienzo. Lo cuenta en milisegundos (miles de un segundo), así que en vez de contar 1 segundo, el cuenta 1000; en vez de contar 5

segundos, son 5000; y después de un minuto, son 60000. Podemos usar este contador para desencadenar animaciones en tiempos específicos. La función millis() regresa este valor del contador.

Ejemplo 7-10: El pasó del tiempo

Puedes ver el paso del tiempo cuando tu programa esté corriendo:

```
void draw() {
  int timer = millis();
  println(timer);
}
```

Ejemplo 7-11: Desencadenando eventos programados

Cuando se combina con un bloque *if*, los valores de *millis()* pueden ser usados para secuenciar animaciones y eventos dentro del programa. Por ejemplo, después de que han transcurrido dos segundos, el código dentro del bloque *if* puede desencadenar un cambio, en este ejemplo, las variables llamadas *time1* y *time2* determinan cuando cambiar el valor de la variable x:

```
int time1 = 2000;
int time2 = 4000;
float x = 0;

void setup() {
 size(480, 120);
 smooth();
}

void draw() {
 int currentTime = millis();
 background(204);

  if (currentTime > time2) {
  x -= 0.5;
 } else if (currentTime > time1) {
  x += 2;
 }
ellipse(x, 60, 90, 90);
}
```

Circular

Si eres un as de la trigonometría, ya sabrás lo impresionante que son las funciones seno y coseno. Si no lo eres, esperamos que durante los próximos ejemplos se dispare tu interés. No discutiremos los detalles de la matemática aquí, pero te mostraremos algunas aplicaciones para generar movimientos fluidos.

La figura 7-2 muestra una visualización del valor de la onda seno y cómo se relaciona con los ángulos. En la parte superior e inferior de la onda, notarás cómo la tasa de cambio (el cambio en los ejes verticales) se pone más lento, pero, interrumpe la dirección. Esta es la calidad de la curva que genera un movimiento interesante.

Las funciones *sin()* y *cos()* en Processing regresar valores entre -1 y 1 para el seno o para el coseno del ángulo especificado. Como *arc()*, los ángulos deben ser dados en valores radianes (ver los ejemplos 3-7 y 3-8 para recordar cómo trabajan los radianes). Para ser útil para dibujar, los valores *float* regresados por *sin()* y *cos()* usualmente son multiplicados por un valor más grande.

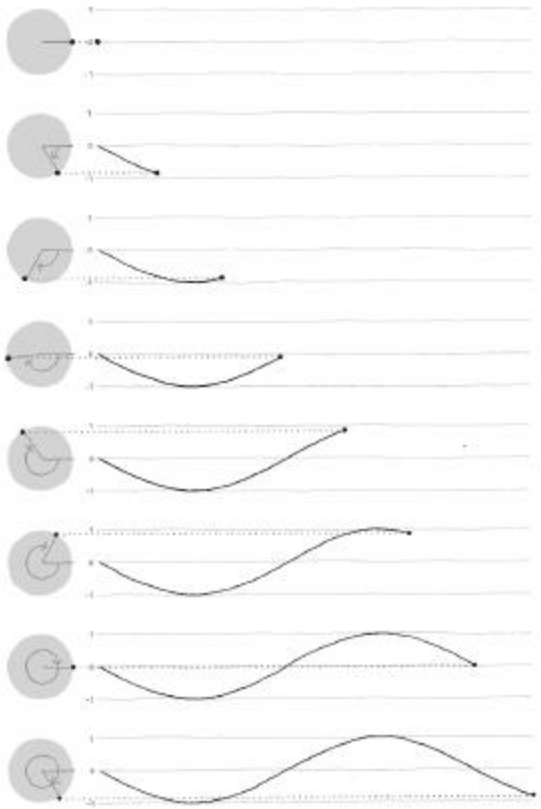


Figura 7-2. Valores del seno y el coseno.

Ejemplo 7-12: Valores para la onda seno

Este ejemplo muestra cómo los valores para sin() ciclan desde -1 hasta 1 a medida que el ángulo incrementa. Con la función map(), la variable sinval es convertida desde este rango de valores de 0 a 255. Este nuevo valor es usado para establecer el color del fondo de la ventana:

```
float angle = 0.0;

void draw() {
  float sinval = sin(angle);
```

```
  println(sinval);
  float gray = map(sinval, -1, 1, 0, 255);
  background(gray);
  angel += 0.1;
}
```

Ejemplo 7-13: Movimiento de la onda seno

Este ejemplo muestra como estos valores pueden ser convertidos en movimiento:

```
float angle = 0.0;
float offset = 60;
float scalar = 40;
float speed = 0.05;

void setup() {
  size(240, 120);
  smooth();
}

void draw() {
  background(0);
  float y1 = offset + sin(angle) * scalar;
  float y2 = offset + sin(angle + 0.4) * scalar;
  float y3 = offset + sin(angle + 0.8) * scalar;

  ellipse(80, y1, 40, 40);
  ellipse(120, y2, 40, 40);
  ellipse(160, y3, 40, 40);
  angle += speed;
}
```

Ejemplo 7-14: Movimiento circular

Este ejemplo muestra cómo estos valores pueden ser convertidos en movimiento:

```
float angle = 0.0;
float offset = 60;
float scalar = 30;
float speed = 0.05;

void setup() {
 size(120, 120);
 smooth();
}

void draw() {
 float x = offset + cos(angle) * scalar;
 float y = offset + sin(angle) * scalar;
 ellipse( x, y, 40, 40);
 angle += speed;
}
```

Ejemplo 7-14: Movimiento circular

Cuando las funciones *sin()* y *cos()* se usan juntas, pueden producir movimiento circulares. El valor de *cos()* proviene de las coordenadas x, y el valor de sin() de las coordenadas y. Ambos son multiplicados por una variable llamadas scalar para cambiar el radio del movimiento y se suman con una valor de desplazamiento para establecer el centro del movimiento circular:

```
float angle = 0.0;
float offset = 60;
float scalar = 30;
float speed = 0.05;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 40, 40);
  angle += speed;
}
```

Ejemplo 7-15: Espirales

Un leve cambio hace incrementar el valor de *scalar* cada frame produce un espiral, en lugar de un círculo:

```
float angle = 0.0;
float offset = 60;
float scalar = 2;
float speed = 0.05;

void setup() {
  size(120, 120);
  fill(0);
  smooth();
}

void draw() {
  float x = offset + cos(angle) * scalar;
  float y = offset + sin(angle) * scalar;
  ellipse( x, y, 2, 2);
  angle += speed;
  scalar += speed;
}
```

Trasladar, Rotar, Escalar

Cambiar las coordenadas de la pantalla es una técnica alternativa para crear movimiento. Por ejemplo, puedes mover una forma 50 pixeles a la derecha, o puedes mover la ubicación de la coordenada (0.0) 50 píxeles a la izquierda - el resultado visual en la pantalla es el mismo. Modificando la coordenada que tiene el sistema por defecto, podemos crear diferentes *transformaciones* incluyendo translación, rotación y escalación. La Figura 7-3 demuestra esto gráficamente.

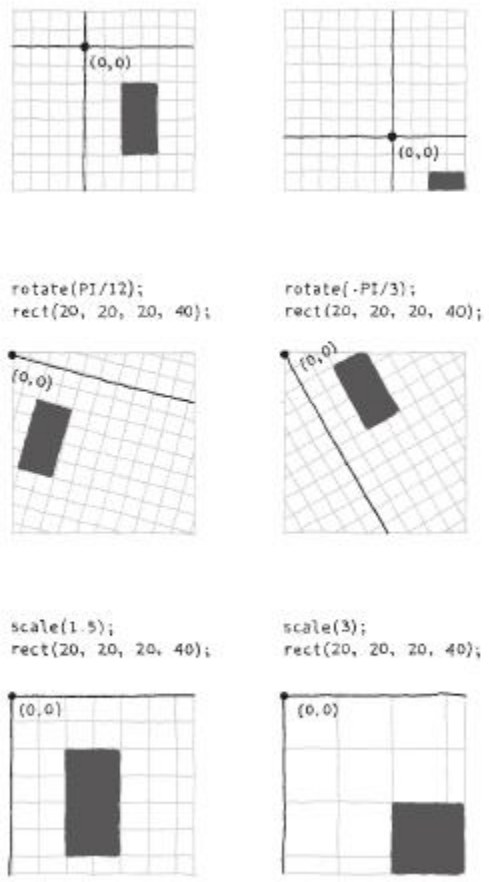


Figura 7-3. Translación, rotación y escalación de las coordenadas.

Trabajar con transformaciones puede ser difícil, pero la función *traslate()* es la más sencilla, así que empezaremos con ella. Esta función puede cambiar la coordenada del sistema a la izquierda, derecha, arriba y abajo con sus dos parámetros.

Ejemplo 7-16: Trasladando la ubicación

En este ejemplo, notarás que cada rectángulo está dibujado en las coordenadas (0,0), pero son movimos alrededor en la pantalla, porque son afectados por *traslate()*:

```
void setup() {
  size(120, 120);
}
```

```
void draw() {
  traslate(mouseX, mouseY);
  rect(0, 0, 30, 30);
}
```

La función *traslate()* establece las coordenadas (0,0) de la pantalla a la ubicación del mouse. En la siguiente línea de texto, el *rect()* es dibujado en un nuevo (0,0) de hecho es dibujado en la ubicación del mouse.

Ejemplo 7-17: Múltiples translaciones

Después de hacer una transformación, se aplican a todas las funciones de dibujo subsiguientes. Notarás que pasa cuando un segundo comando de *traslate* es agregado a un control de un segundo rectángulo:

```
void setup() {
  size(120, 120);
}

void draw() {
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
  translate(35, 10);
  rect(0, 0, 15, 15);
}
```

Ejemplo 7-18: transformaciones de aislamiento

Para aislar los efectos de una transformación de manera que no afecten los comandos luego, usa las funciones *pushMatrix()* y *popMatrix()*. Cuando la función pushMatrix() esté corriendo, ella salvará una copia de la coordenada actual del sistema y luego restaura ese sistema después de *popMatrix()*:

```
void setup() {
  size(120, 120);
}
```

```
void draw() {
  pushMatrix();
  translate(mouseX, mouseY);
  rect(0, 0, 30, 30);
  popMatrix();
  translate(35, 10);
  rect(0, 0, 15, 15);
}
```

En este ejemplo, el rectángulo más pequeño siempre dibuja en la parte superior izquierda porque *translate(mouseX, mouseY)* es cancelado por el *popMatrix()*.

Nota: *Las funciones pushMatrix() y popMatrix() son usadas siempre en pares. Para cada pushMatrix(), necesitas tener una pareja popMatrix().*

Ejemplo 7-19: Rotación

La función *rotate()* rota las coordenadas del sistema. Tiene un parámetro, el cual es el ángulo (en radianes) para rotar. Siempre rota relativamente a (0,0), sabiendo que se rota alrededor del origen. Para girar una forma alrededor de su centro, primero usa *translate()* para moverlo de su ubicación a dónde te gustaría poner la forma, luego llama a *rotate()*, y luego dibuja la forma con su centro en las coordenadas (0,0):

```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  translate(mouseX, mouseY);
  rotate(angle);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Ejemplo 7-20: Combinando transformaciones

Cuando *translate()* y *rotate()* se combinan, El orden en el cual aparecen afecta los resultados. EL siguiente ejemplo es idéntico al 7-

19, excepto que *translate()* y *rotate()* están invertidos. La forma ahora rota alrededor de la esquina superior izquierda del la ventana del display, con la distancia desde la esquina establecida por *translate()*:

```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  rotate(angle);
  translate(mouseX, mouseY);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Nota: *También puedes usar las funciones rectMode(), ellipseMode(), imageMode() y shapeMode() para que sea más fácil dibujar de las formas desde el centro.*

Ejemplo 7-21: Escalando

La función *scale()* estira las coordenadas sobre la pantalla. Así como *rotate()*, esta transforma desde el origen. Por lo tanto así como se hace con *rotate()*, para escalar una forma desde su centro, transladarlo a su ubicación, escalarlo, y luego dibujar en el centro de la coordenada(0,0):

```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}
```

```
void draw() {
  translate(mouseX, mouseY);
  scale(sin(angle) + 2);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Ejemplo 7-22: Mantener los dibujos con trazos regulares

A partir de las líneas gruesas en el ejemplo 7-21, puedes ver como la función *scale()* afecta el grosor del trazo. Para mantener un trazo con un grosor consistente como el de las formas escaladas, divide el grosor deseado por el valor escalado:

```
float angle = 0.0;

void setup() {
  size(120, 120);
  smooth();
}

void draw() {
  translate(mouseX, mouseY);
  float scalar = sin(angle) + 2;
  scale(scalar);
  strokeWeight(1.0 / scalar);
  rect(-15, -15, 30, 30);
  angle += 0.1;
}
```

Ejemplo 7-23: Un brazo articulado

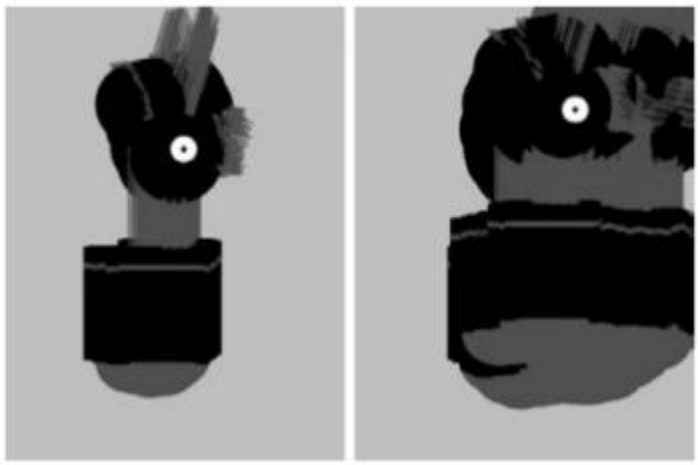
En este última parte y más largo ejemplo de transformación, hemos puesto una serie las funciones *translate()* y *rotate()* juntas para crear un brazo vinculado que se doble de ida y vuelta. Además cada *translate()* mueve la posición de las líneas, y cada *rotate()* agrega a la rotación anterior para doblarse más:

```
float angle = 0.0;
float angleDirection = 1;
float speed = 0.005;

void setup() {
  size(120, 120);
  smooth();
```

```
}

void draw() {
  background(204);
  translate(20, 25);  // Move to start position
  rotate(angle);
  strokeWeight(12);
  line(0, 0, 40, 0);
  translate(40, 0);  // Move to next joint
  rotate(angle * 2.0);
  strokeWeight(6);
  line(0,0, 30, 0);
  translate(30, 0);  // Move to next joint
  rotate(angle * 2.5);
  strokeWeight(3);
  line(0, 0, 20, 0);


  angle += speed * angleDirection;
  if ((angle > QUARTER_PI) || (angle < 0)) {
     angleDirection *= -1;
   }
}
```

Aquí, no usamos un *pushMatrix()* o *popMatrix()*, porque queremos propagar las transformaciones - para cada transformación para construir el último. El sistema de coordenadas es automáticamente reseteado por defecto cuando *draw()* comienza cada frame.

Robot 5: Movimiento



En este ejemplo, las técnicas para el movimiento al azar y circular son aplicadas al robot. El *background()* es removido para hacer más fácil ver la posición del robot y el cambio del cuerpo.

En cada frame, es agregado un número al azar entre -4 y 4 para la coordenada x, y un número al azar entre -1 y 1 para la coordenada y.

Esto causa que el robot se mueva desde la izquierda a la derecha y de arriba hacia abajo. Los números son calculados desde las función *sin()* que cambia la altura de su cuello para que oscile entre 50 y 100 píxeles de altura:

```
float x = 180;          // X - coordinate
float y = 400;          // Y - coordinate
float bodyHeight = 153; // Body height
float neckHeight = 56;  // Neck height
float radius = 45;      // Head radius
float angle = 0.0;      // Angle for motion

void setup() {
  size(360, 480);
  smooth();
  ellipseMode(RADIUS);
  background(204);
}

void draw() {
  // Change position by a small random amount
  x += random(-4, 4);
  y += random(-1, 1);

  // Change height of neck
  neckHeight = 80 + sin(angle) * 30;
  angle += 0.05;

  // Adjust the height of the head
  float ny = y - bodyHeight - neckHeight - radius;

  // Neck
  stroke(102);
  line(x+2, y-bodyHeight, x+2, ny);
  line(x+12, y-bodyHeight, x+12, ny);
  line(x+22, y-bodyHeight, x+22, ny);

// Antennae
 line(x+12, ny, x-18, ny-43);
 line(x+12, ny, x+42, ny-99);
 line(x+12, ny, x+78, ny+15);

 // Body
 noStroke();
 fill(102);
 ellipse(x, y-33, 33, 33);
 fill(0);
 rect(x=45, y-bodyHeight+17, 90, 6);
 fill(102);
 rect(x-45, y-bodyHeight+17, 90, 6);

 // Head
```

```
 fill(0);
 ellipse(x+12, ny, radius, radius);
 fill(255);
 ellipse(x+12, ny, radius, radius);
 fill(255);
 ellipse(x+24, ny-6, 14, 14);
 fill(0);
 ellipse(x+24, ny-6, 3, 3);
}
```

CAPITULO 8

8 Funciones

Las funciones son los ladrillos para los programas de Processing. Estos han aparecido en todos los ejemplos que hemos presentado. Por ejemplo, usamos frecuentemente la función *size()*, la función *line()*, y la función *fill()*. Este capítulo muestra cómo escribir nuevas funciones para extender las capacidades incorporadas de Processing más allá de sus características.

El poder de las funciones es la modularidad. Las funciones son unidades de software independiente que son usadas para construir programas más complejos- como los ladrillos LEGO. Donde cada tipo de ladrillo tiene un propósito específico, y hacer un modelo complejo requiere juntar diferentes partes. Así, con las funciones, el verdadero poder de estos ladrillos es la habilidad para construir diferentes funciones desde el mismo conjunto de elementos. El mismo grupo de LEGOS que hace una nave espacial puede ser re-usado para construir un camión, un rascacielos, y muchos otros objetos.

Las funciones son de gran ayuda si quieres dibujar una forma más compleja como un árbol sobre otro. La función para dibujar la forma del árbol podría ser hecha por los comandos de construcción de Processing, cómo *line()*, que crea la forma. Después que el código para dibujar el árbol es escrito, no necesitas pensar de nuevo acerca de los detalles del dibujo del árbol - simplemente escribes *tree()* (o cómo quieras llamar la función) para dibujar la forma. Las funciones permiten hacer una secuencia compleja de declaraciones para ser abstraídas, así que puedes enfocarte en tu mayor objetivo (los comandos *line()* que definen la forma del árbol). Una vez la función es definida, el código de adentro necesita ser repetido otra vez.

Funciones Básicas

Un computador corre un programa una línea a la vez. Cuando una función está corriendo, el computador salta hasta donde está definida la función y corre el código allí, luego salta de nuevo a donde lo dejó.

Ejemplo 8-1: Tirar los dados

Este comportamiento es dibujado con la función *rollDice()* escrita para este ejemplo. Cuando un programa empieza, corre el código en *setup()* y luego para. El programa toma un desvío y corre el código dentro de *rollDice()* cada vez que aparece:

```
void setup() {
println("Ready to roll!");
rollDice(20);
rollDice(20);
rollDice(6);
println("Finished.");
}
void rollDice(int numSides) {
int d = 1 + int(random(numSides));
println("Rolling... " + d);
}
```

Las dos líneas de código en *rollDice()* seleccionan un número al azar entre 1 y el número de lados en el dado, e imprime ese número a la consola. Cómo los números son al azar, podrás ver números diferentes cada vez que el programa esté corriendo:

```
Ready to roll!
Rolling... 20
Rolling... 11
Rolling... 1
Finished.
```

Cada vez que la función *rollDice()* está corriendo dentro de *setup()*, el código dentro de la función corre desde arriba hasta abajo, luego el programa continúa en la siguiente línea dentro de *setup()*.

La función *random()* (descrita en la página 97) regresa un número desde 0 hasta (pero no incluyendo) el número especificado. Así que *random(6)* regresa un número entre 0 y 5.99999.... Porque *random()* regresa un valor *float*, también usamos *int()* para convertirlo en un entero. Así que *int(random(6))* regresará a 0, 1, 2, 3, 4 o 5. Luego agregaremos 1 de manera que el número regresado esté entre 1 y 6 (como un troquel). Como en otros casos en este libro, contando desde 0 hace más fácil de usar los resultados de *random()* con otros cálculos.

Ejemplo 8-2: Otra forma de enrollar

Si un programa equivalente se escribiera sin la función *rollDice()*, podría verse así:

```
void setup() {
 println("Ready to roll!");
 int d1 = 1 + int(random(20));
 println("Rolling... " + d1);
 int d2 = 1 + int(random(20));
 println("Rolling... " + d2);
 int d3 = 1 + int(random(6));
 println("Rolling... " + d3);
 println("Finished.");
}
```

La función *rollDice()* en el ejemplo 8-1 hace que el código sea más fácil de leer y manejar. El programa es más claro, porque el nombre de la función claramente tiene un propósito. En este ejemplo, veremos la función *random()* en *setup()*, pero su uso no es tan obvio. El número de lados en la matriz es claro, con una función: Cuando el código dice *rollDice(6)*, es obvio que está simulando el rollo de un dado de seis caras. También, el ejemplo 8-1 es más fácil de mantener, porque la información no es repetida. La frase **Rolling...** es repetida tres veces aquí. Si quieres cambiar ese texto por algo más, necesitarías actualizar el programa en tres lugares, mejor que hacer una sola edición dentro de la función *rollDice()*. Además de como se ve en el ejemplo 8-5, una función también puede hacer que un programa sea más corto (Por lo tanto más fácil de manejar y leer), lo cual ayuda a reducir el número de errores.

Hacer una Función

En esta sección, dibujaremos un búho para explicar los pasos para hacer una función.

Ejemplo 8-13: Dibujar un Búho

Primero dibujaremos un búho sin usar una función:

```
void setup() {
 size(480, 120);
 smooth();
}

void draw() {
 background(204);
 translate(110, 110);
 stroke(0);
 strokeWeight(70);
 line(0, -35, 0, -65); // Body
 noStroke();
 fill(255);
 ellipse(-17.5, -65, 35, 35); // Left eye dome
 ellipse(17.5, -65, 35, 35); // Right eye dome
 arc(0, -65, 70, 70, 0, PI); // Chin
 fill(0);
 ellipse(-14, -65, 8, 8); // Left eye
 ellipse(14, -65, 8, 8); // Right eye
 quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
}
```

Notarás que *translate()* es usado para mover el original (0,0) a 110 píxeles encima y 110 píxeles abajo. Luego el búho es dibujado relativamente a (0,0), con estas coordenadas a veces positivas y negativas ya que están centradas en el nuevo punto 0,0.Ver figura 8-1.

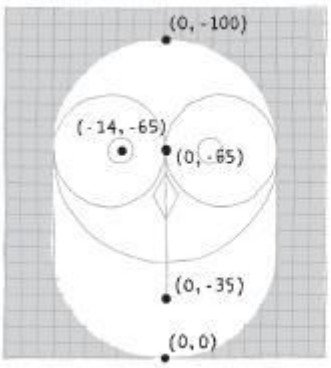


Figura 8-1. Las coordenadas del búho.

Ejemplo 8-4: Dos es compañía

El código presentado en el ejemplo 8-3 es razonable si hay sólo un búho, pero cuando dibujamos el segundo, la longitud es casi duplicada:

```
void setup() {
 size(480, 120);
 smooth();
}

void draw() {
 background(204);
 // Left owl
 translate(110, 110);
 stroke(0);
 strokeWeight(70);
 line(0, -35, 0, -65); // Body
 noStroke();
 fill(255);
 ellipse(-17.5, -65, 35, 35); // Left eye dome
 ellipse(17.5, -65, 35, 35);  // Right eye dome
 arc(0, -65, 70, 70, 0, PI);  // Chin
 fill(0);
ellipse(-14, -65, 8, 8);               // Left eye
ellipse(14, -65, 8, 8);                // Right eye
```

```
quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak

// Right owl
translate(70, 0);
stroke(0);
strokeWeight(70);
line(0, -35, 0, -65); // Body
noStroke();
fill(255);
ellipse(-17.5, -65, 35, 35); // Left eye dome
ellipse(17.5, -65, 35, 35); // Right eye dome
arc(0, -65, 70, 70, 0, PI); // Chin
fill(0);
ellipse(-14, -65, 8, 8); // Left eye
ellipse(14, -65, 8, 8); // Right eye
quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
}
```

El programa creció desde 21 líneas hasta 34, porque el código para dibujar el primer búho fue cortado y pegado en el programa y *translate()* fue insertado para moverlo 70 píxeles hacia la derecha. Esta es una tediosa e ineficiente manera para dibujar un segundo búho, sin mencionar el dolor de cabeza de agregar un tercer búho con este método. Pero duplicar el código es innecesario, porque este el tipo de situación donde una función puede venir al rescate.

Ejemplo 8-5: Una función Búho

En este ejemplo, una función es introducida para dibujar dos búhos con el mismo código. Si hacemos que el código dibuje el búho en la pantalla en una nueva función, el código necesita aparecer sólo una vez en el programa:

```
void setup() {
 size(480, 120);
 smooth();
}

void draw() {
 background(204);
 owl(110, 110);
 owl(180, 110);
 }

void owl(int x, int y) {
```

```
 pushMatrix();
 translate(x, y);
 stroke(0);
 strokeWeight(70);
 line(0, -35, 0, -65); // Body
 noStroke();
 fill(255);
 ellipse(-17.5, -65, 35, 35); // Left eye dome
 ellipse(17.5, -65, 35, 35); // Right eye dome
 arc(0, -65, 70, 70, 0, PI); // Chin
 fill(0);
 ellipse(-14, -65, 8, 8); // Left eye
 ellipse(14, -65, 8, 8); // Right eye
 quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
 popMatrix();
}
```

Puedes ver desde las ilustraciones que en este ejemplo y en el ejemplo 8-4 se tienen el mismo resultado, pero este ejemplo es más corto, porque el código para dibujar el búho aparece sólo una vez, dentro de la bien llamada función *owl()*. Este código corre dos veces, porque es llamado dos veces dentro de *draw()*. EL búho es dibujado en dos diferentes ubicaciones porque los parámetros pasados dentro de la función establecen las coordenadas X y Y.

Los parámetros son una parte importante de una función, porque ellos proveen la flexibilidad. Vimos en otro ejemplo en la función *rollDice()*; el parámetro llamado *numSides, que* hacen posible simular un dado de 6 lados, un dado de 20 lados, o un dado con cualquier número de lados. Esta es como muchas otras funciones de Processing. Por ejemplo, los parámetros de la función *line()* hacen posible dibujar una línea desde cualquier pixel en la pantalla a cualquier otro pixel. Sin los parámetros, la función podría ser capaz de dibujar una línea desde un punto fijo a otro.

Cada parámetro tiene un tipo de datos (así como *int* o *float*), porque cada parámetro es una variable que es creada cada vez que las funciones corren. Cuando este ejemplo esté corriendo, por primera vez la función Búho es llamada, el valor del parámetro X es 110, y Y también es 110. En el segundo uso de la función, el valor de X es 180 y Y de nuevo es 110. Cada valor es pasado en la función y luego donde el nombre de la variable aparezca dentro de la función, este es reemplazado con el valor entrante.

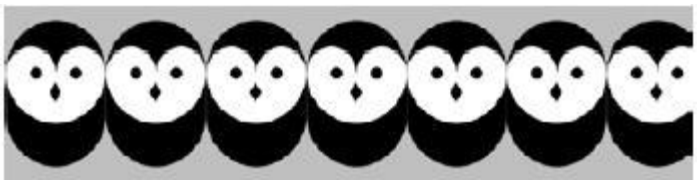
Asegúrate que los valores pasados en una función coincidan con los tipos de datos de los parámetros. Por ejemplo, si el siguiente aparece dentro de setup() por ejemplo 8-5:

```
owl(110.5, 120.2);
```

Esto podría crear un error, porque el tipo de datos para los parámetros X y Y es *int*, y los valores 110.5 y 120.2 son valores *float*.

Ejemplo 8-6: Aumentando la población sobrante

Ahora que tenemos una función básica para dibujar el búho en cualquier ubicación, podemos dibujar muchos búhos eficientemente colocando la función dentro de un *for loop* y cambiando el primer parámetro cada momento a través del loop:



```
void setup() {
 size(480, 120);
 smooth();
}

void draw() {
 background(204);
  for (int x = 35; x < width + 70; x += 70) {
  owl(x, 110);
  }
}

// Insert owl() function from Example 8-5
```

Es posible seguir agregando más y más parámetros a la función para cambiar diferentes aspectos de cómo es dibujado en búho. Los valores podrían pasar por un cambio de color del búho, rotación, escala, o el diámetro de sus ojos.

Ejemplo 8-7: Búhos de diferentes tamaños

En este ejemplo, hemos agregado dos parámetros para cambiar el valor gris y el tamaño de cada búho:

```
void setup() {
```

```
 size(480, 120);
 smooth();
}

void draw() {
 background(204);
 randomSeed(0);
  for (int i = 35; i < width + 40; i += 40) {
  int gray = int(random(0, 102));
  float scalar = random(0.25, 1.0);
  owl(i, 110, gray, scalar);
  }
}

void owl(int x, int y, int g, float s) {
 pushMatrix();
 translate(x, y);
 scale(s); // Set the size
 stroke(g); // Set the gray value
 strokeWeight(70);
 line(0, -35, 0, -65); // Body
 noStroke();
 fill(255-g);
 ellipse(-17.5, -65, 35, 35); // Left eye dome
 ellipse(17.5, -65, 35, 35); // Right eye dome
 arc(0, -65, 70, 70, 0, PI); // Chin
 fill(g);
 ellipse(-14, -65, 8, 8); // Left eye
 ellipse(14, -65, 8, 8); // Right eye
 quad(0, -58, 4, -51, 0, -44, -4, -51); // Beak
 popMatrix();
}
```

Valores de retorno

Las funciones pueden hacer cálculos y luego regresar un valor al programa. Ya hemos usado funciones de este tipo, incluyendo *random()* y *sin()*. Notarás que cuando este tipo de funciones aparecen, el valor de retorno es usualmente asignado a una variable:

```
float r = random(1, 10);
```

En este caso, *random()* regresar un valor entre 1 y 10, el cual es luego asignado a la variable r.

Una función que regresar un valor también es usada frecuentemente como un parámetro en otra función. Por ejemplo:

```
point(random(width), random(height));
```

En este caso, los valores desde *random()* no son asignados a una variable - son pasados como parámetros para *point()* y usados para posicionar el punto entre la ventana.

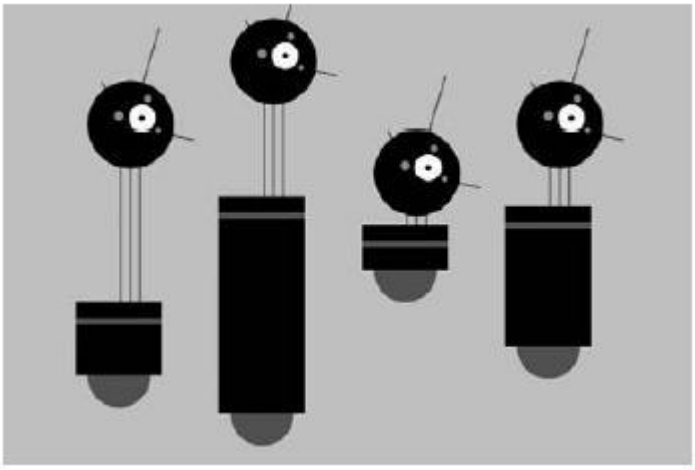
Ejemplo 8-8: Regresar un valor

Para hacer que una función regrese un valor, reemplace la palabra clave *void* con el tipo de datos al que quieras que la función regrese. En tu función, específica los datos para que sean pasados de nuevo con la palabra clave *return*. Por ejemplo, esto incluye una función llamada *calculateMars()* que calcula el peso de una persona o un objeto o nuestro planeta vecino:

```
void setup() {
 float yourWeight = 132;
 float marsWeight = calculateMars(yourWeight);
 println(marsWeight);
}

float calculateMars(float w) {
float newWeight = w * 0.38;
return newWeight;
}
```

Notarás que el tipos de datos *float* devuelve un valor de punto flotante antes de que el nombre de la función sea mostrado, y la última línea del bloque, la cual regresa la variable *newWeight*. En la segunda línea de *setup()*, es asignado el valor a la variable *marsWeigh*t. (Para ver tu propio peso en Marte, cambia el valor de tu variable *Weight* por la de tu peso.)

Robot 6: Funciones



En contraste con Robot2 (ver "Robot 2: Variables" en el capítulo 4), este ejemplo usa una función para dibujar cuatro variaciones en el robot dentro del mismo programa. Porque la función *drawRobot()* aparece

cuatro veces dentro de *draw()*, el código dentro del bloque *drawRobot()* corre cuatro veces, cada vez con un conjunto de parámetros diferentes para cambiar la posición y la altura del cuerpo del robot.
Notarás lo que eran las variables globales en Robot 2 ahora han sido aisladas dentro de la función *drawRobot()*. Porque estas variables aplican sólo para dibujar el robot, pertenecen a las llaves de adentro que definen el bloque de la función *drawRobot()*. Porque el valor de la variable *radius* no cambia, no necesita tener un parámetro. En cambio, es definida al principio de *drawRobot()*:

```
void setup() {
 size(720, 480);
 smooth();
 strokeWeight(2);
 ellipseMode(RADIUS);
}

void draw() {
background(204);
Functions 127
drawRobot(120, 420, 110, 140);
drawRobot(270, 460, 260, 95);
drawRobot(420, 310, 80, 10);
drawRobot(570, 390, 180, 40);
}

void drawRobot(int x, int y, int bodyHeight, int neckHeight) {
int radius = 45;
int ny = y - bodyHeight - neckHeight - radius;

// Neck
stroke(102);
line(x+2, y-bodyHeight, x+2, ny);
line(x+12, y-bodyHeight, x+12, ny);
line(x+22, y-bodyHeight, x+22, ny);

// Antennae
line(x+12, ny, x-18, ny-43);
line(x+12, ny, x+42, ny-99);
line(x+12, ny, x+78, ny+15);

// Body
noStroke();
fill(102);
ellipse(x, y-33, 33, 33);
fill(0);
rect(x-45, y-bodyHeight, 90, bodyHeight-33);
fill(102);
rect(x-45, y-bodyHeight+17, 90, 6);

// Head
fill(0);
```

```
ellipse(x+12, ny, radius, radius);
fill(255);
ellipse(x+24, ny-6, 14, 14);
fill(0);
ellipse(x+24, ny-6, 3, 3);
fill(153);
ellipse(x, ny-8, 5, 5);
ellipse(x+30, ny-26, 4, 4);
ellipse(x+41, ny+6, 3, 3);
}
```

CAPÍTULO 9

9 OBJETOS

La Programación orientada a objetos (OPP) es una forma diferente de pensar tus programas. Aunque el término "Programación orientada a objetos" puede sonar intimidante, estas son buenas noticias: desde el capítulo 6 hemos estado trabajando con objetos, cuando has usado *PImage, PFont, String* y *Psaphe.* A diferencia de los tipos de datos primitivos como *boolean, int,* y *float,* que pueden almacenar un solo valor, un objeto puede almacenar muchos. Pero esto es solo una parte de la historia. Los objetos están pensados como un grupo de variables con funciones relacionadas. Como ya sabemos, se trabaja con variables y funciones, trabajar con objetos simplifica combinar lo que ya tienes aprendido en un paquete más comprensible.

Trabajar con objetos es importante, porque ellos permiten colocar grandes ideas en pequeños bloques de código. Esta forma de trabajo es una mirada al mundo natural donde, por ejemplo, los órganos están hechos de tejido, los tejidos están hechos de células y así sucesivamente. Así mismo, como tu código tiende a complicarse más, debes pensar en términos con estructuras más pequeñas que formarán unas más complicadas. Es fácil escribir y dar solidez, a piezas de código pequeños, entendiendo que trabajan en conjunto con piezas grandes de código que hacen todo al mismo tiempo.

Un objeto en software es una colección de variables y funciones relacionadas entre sí. En el contexto de objetos, una variable es llamada *field* (o una instancia de variable) y una función es llamada un *method. Fields* y *methods* trabajan igualmente que las variables y las funciones vistas en capítulos anteriores. Pero nosotros usaremos los nuevos términos enfatizando que son una parte de un objeto. Es decir, un objeto combina datos relacionados (fields) con acciones y comportamientos relacionados (methods). La idea es agrupar datos relacionados con métodos relacionados que actúen en esos datos.

Ejemplo, tomemos de modelo un radio, pensemos cuáles parámetros podemos ajustar (en la radio) y cuáles acciones pueden afectar esos datos:

field: volumen, frecuencia, banda(FM,AM),power(on,off)
Methods:setVolumen,setFrecuencia,setBanda.

Modelando el simple mecanismo de un dispositivo es fácil comparar el modelado de un organismo como una hormiga o una persona. No es posible reducir tal complejidad de organismos a unos pocos *fields* y *methods,* pero si es posible crear un modelo para una simulación interesante. El videojuego *the Sims* es un claro ejemplo. Este juego es jugado simulando las actividades que las personas tienen diariamente.
Los personajes tienen la suficiente personalidad para ser jugados, siendo un juego adictivo, pero no más. De hecho, ellos tienen un

cuarto atributo que les da personalidad: son aseados, extrovertidos, activos, traviesos, y amables. Cuando el conocimiento del modelo, es posible tomarlo de complejos organismos simplificados, podríamos comenzar un programa de una hormiga con muy pocos fields y methods:

Fields: type (worker, soldier), weight, length
Methods: walk, pinch, releasePheromones, eat

Si hacemos una lista de unas hormigas con *fields* y *methods,* podemos enfocarnos en elegir diferentes aspectos de ese modelo de hormiga. No hay una manera correcta de crear un modelo, con tal de que tomes lo apropiado para crear tu modelo y logres la meta.

CLASES Y OBJETOS

Después de que creas un objeto, debes definir una clase. Una clase es la especificación para un objeto. Usando analogías arquitectónicas. Una clase puede ser el proyecto para una casa, y el objeto es como la casa en sí misma. Cada casa puede estar hecha desde el proyecto y tener algunas variaciones, y el proyecto es sólo la especificación, no una estructura construida. Por ejemplo, una casa puede ser azul y otra roja; una casa puede venir con luces artificiales y la otra sin ellas. Igualmente con objetos, la clase define el tipo de objetos y comportamientos, pero cada objeto (casa) puede tener una clase individual (proyecto), tiene variables (color, luces) esto puede establecer diferentes valores. Usando unos términos más técnicos, cada objeto es una instancia de una clase y cada instancia tiene su propio conjunto de fields y methods.

Definiendo una clase.

Antes de escribir una clase, es recomendable hacer un pequeño planeamiento, pensar sobre qué fields y que methods deberías tener en tu clase. Hacer una lluvia de ideas e imaginar todas las posibles opciones y luego priorizar sobre cuales elegirás para tener un mejor trabajo. Puedes hacer cambios durante el proceso de programación, pero es importante tener un buen comienzo.

Para tus fields, escoge nombres claros y decide qué tipo de datos para cada uno. El fields dentro de una clase puede tomar cualquier tipo de valor. Una clase puede tener simultáneamente muchos boleanos, floats, imagenes, strings. Lo que se te antoje. Hay que tener en mente que la razón por la cual hacemos una clase es porque tendremos muchos elementos relacionados. Para tus methods, elige nombres claros y elige los valores que regresen (cualquiera). Los methods están usando los fields para cambiar sus valores y ejecutar cambios basados en los valores dentro de los fields.

Para nuestra primera clase. Nos basaremos en el ejemplo 7-9 de este libro. Nosotros empezaremos por una lista de los fields desde este ejemplo:

```
float x
float y
int diamater
float speed
```

En el siguiente paso, veremos que methods podrían ser usados para la clase. Buscaremos la función *draw()* desde el ejemplo para adaptarla. Veremos dos componentes primarios. La posición de la forma es actualizada y dibujada en la pantalla. Vamos a crear dos métodos para una clase, una para cada tarea:

```
void move()
void display()
```

Ninguno de estos métodos devolverá un valor, por la tanto hay que tener un retorno de tipo void. Cuando estemos escribiendo líneas basadas en la lista de los fields y los métodos. Es recomendable seguir los siguientes pasos:

1. crear el bloque
2. Agregar los fields.
3. Escribir el constructor (lo explicaremos en breve) que asignará los valores a los fields.
4. Agregar el método.

Primero, creamos un bloque:

```
class JitterBug {
}
```

Notar que la palabra *class* esta en minúsculas y el nombre *JitterBug* está en mayúsculas. No es necesario nombrar la clase con una letra mayúscula, pero esto es una convención (recomendable) usada para indicar que esto es una clase. (La palabra class, sin embargo, debe estar en minúsculas porque es una regla de el lenguaje de programación.)

Segundo, agregamos los fields. Cuando hagamos esto, debemos decidir cuales fields deseamos tener dentro de los valores asignados al constructor, en especial, el método es usado para este propósito. Consejo, para querer diferenciar cada valor de cada clase, podemos definir cuando ellos están declarados. Para la clase *JitterBug*, hemos decidido que los valores para X, Y y Diameter pasarán, pero serán declarados de la siguiente manera;

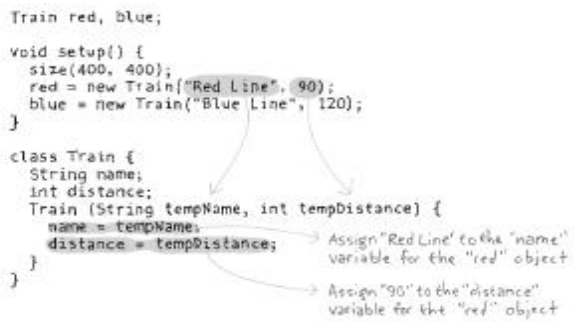
```
class JitterBug {
      float x;
      float y;
      int diameter;
      float speed = 0.5;
}
```

Tercero, agregamos el constructor. El constructor siempre tiene el mismo nombre que la clase. El propósito del constructor es asignar los valores iníciales cuando los fields en el objeto son creados. (Una instancia para la clase (figura 9-1)). Los bloques de código dentro del constructor corren una vez el objeto es creado. Discutido anteriormente, pasamos por tres parámetros al constructor cuando el objeto es inicializado. Cada uno de los valores anteriormente asignados a las variables temporales solo existe cuando el código dentro del constructor es leído. Y para clarificar esto, nosotros agregamos el nombre *temp* a cada una de esas variables, pero esto puede nombrarse con algunos términos que prefieras. Solo está usando el asignamiento de valor cuando los fields están en una parte de la clase. Nótese que el constructor nunca retorna un valor y por lo tanto no tiene un *void* u otro tipo de dato antes. Después de que adherimos el constructor, la clase se puede ver de este modo:

```
class JitterBug {

 float x;
 float y;
 int diameter;
 float speed = 0.5;

  JitterBug(float tempX, float tempY, int tempDiameter) {
    x = tempX;
    y = tempY;
    diameter = tempDiameter;
   }
}
```

El último paso es agregar el método. Esta parte es sencilla: como escribiendo las funciones, pero esto aquí estará contenido en una clase. También, el código tendrá un espacio. Cada línea dentro del texto se sangra un poco para mostrar cuales están dentro del bloque. Con el constructor y los métodos, el código se separa de nuevo para mostrarlo con claridad y jerarquía:

```
class JitterBug {

 float x;
 float y;
 int diameter;
 float speed = 2.5;

JitterBug(float tempX, float tempY, int tempDiameter) {
 x = tempX;
 y = tempY;
 diameter = tempDiameter;
}

void move() {
 x += random(-speed, speed);
 y += random(-speed, speed);
}

void display() {
 ellipse(x, y, diameter, diameter);
 }

}
```



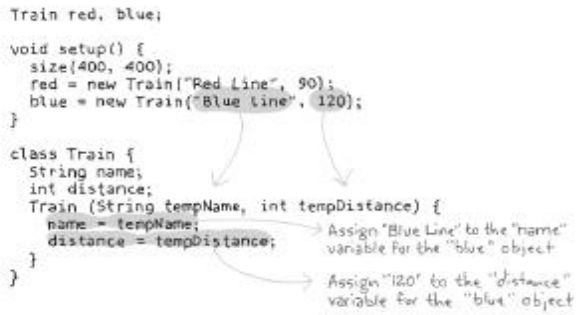


Figura 9-1 Pasando los valores al constructor.

Ejemplo 9-1. Haciendo un objeto.

Ahora que tienes definida una clase, para usarla en el programa debes definir un objeto desde la clase. Aquí están dos pasos para crear un objeto:

1. Declara una variable para el objeto.
2. Inicializar el objeto con la palabra clave *New*

Empezaremos mostrando cómo trabaja esto en un sketch de Processing y luego continuaremos con explicar cada parte a profundidad:

```
JitterBug bug; // Declare object

void setup() {
 size(480, 120);
 smooth();
 // Create object and pass in parameters
 bug = new JitterBug(width/2, height/2, 20);
}

void draw() {
 bug.move();
 bug.display();
}

// Put a copy of the JitterBug class here
```

Cada clase es un tipo de dato y cada objeto es una variable. Declarar variables para objetos es similar a las variables para los datos primitivos como boolean, int, y float. El objeto es declarado por declarar el tipo de dato seguido por el nombre de la variable:

```
JitterBug bug;
```

EL segundo paso es inicializar el objeto con la palabra clave *new.* Esto crea espacios de memoria para el objeto y los fields. El nombre del constructor es escrito a la derecha de la palabra clave *New*, seguida por los parámetros en el constructor, tendremos:

```
JitterBug bug = new JitterBug(200.0, 250.0, 30);
```

Los tres números dentro del paréntesis son los parámetros que son pasados en el constructor de clase *JitterBug*. El número de estos parámetros y sus tipos de datos debe coincidir con los del constructor.

Ejemplo 9-2: Haciendo múltiples objetos

En el ejemplo 9-1, vemos algo nuevo: el periodo (dot) que es usado para acceder a los métodos de los objetos dentro de *draw()*. El operador dot es usado para unir el nombre del objeto con sus campos y métodos. Esto se aclara en este ejemplo, donde dos objetos son hechos de la misma clase, el comando *jit.move()* se refiere al método para el *move()*que pertenece al objeto llamado *jit*, y *bug.move()* que se refieren al método *move()* que pertenece al objeto llamado *bug*:

```
JitterBug jit;
JitterBug bug;

void setup() {
 size(480, 120);
 smooth();
 jit = new JitterBug(width * 0.33, height/2, 50);
 bug = new JitterBug(width * 0.66, height/2, 10);
 }

void draw() {
 jit.move();
 jit.display();
 bug.move();
 bug.display();
}

// Put a copy of the JitterBug class here
```

Ahora que la clase existe como su propio módulo de código, cualquier cambio modificará los objetos hechos. Por ejemplo, podrías agregar un campo a la clase *JitterBug* que controla el color, u otra que determina su tamaño. Estos valores pueden ser pasados usando el constructor o asignados usando métodos adicionales, tales como *setColor()* o *setSize()*. Y como es una unidad autónoma, también puedes usar la clase JitterBug u otro dibujo.

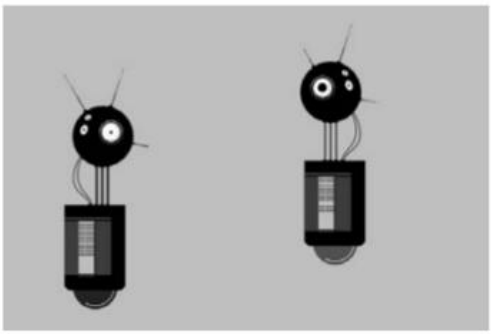
Es buen tiempo para aprender acerca de la pestaña de la función medio ambiente de Processing (figura 9-2). Las pestañas permiten difundir el código a través de más de un archivo. Esto hace que el código más grande sea más fácil de editar y en general, más manejable. Usualmente una nueva pestaña es creada para cada clase, lo cual refuerza la modularidad del trabajo con clases y hace el código más fácil de encontrar.

Para crear una nueva pestaña, haz click en la flecha que está al lado derecho de la barra de pestañas. Cuando selecciones Nueva Pestaña del, se te solicitará nombrar la pestaña dentro de la ventana de mensaje. Usando esta técnica, modificando el código de este ejemplo para intentar hacer una nueva pestaña para la clase *JitterBug*.

NOTA: *Cada pestaña se muestra como un archivo separado .pde dentro de la carpeta de dibujos.*

Figura 9-2. Código que puede ser divido dentro de diferentes pestañas para hacer más fácil su manejo.

Robot 7: Objetos



Un objeto de un software combina métodos (funciones) y campos (variables) dentro de una unidad. La clase de *Robot.* En este ejemplo define todos los objetos del robot que será creado a partir de este. Cada objeto de *Robot* tiene su propio conjunto de campos para guardar una posición y la ilustración que dibujara en la pantalla. Cada uno tiene métodos para actualizar la posición y mostrar la ilustración.

Los parámetros para *bot1* y *bot2* en setup() definen las coordenadas de X y Y y el archivo *.svg* que será usado para representar el robot. Los parámetros *tempX* y *tempY* son pasados en el constructor y asignados a los campos *xpos* y *ypos*. El parámetro *svgName* es usado para cargar la ilustración relacionada. Los objetos (*bot1* y *bot2*) dibujan en su propia ubicación y con una ilustración diferente porque tienen valores únicos pasados en los objetos a través de sus constructores:

```
Robot bot1;
Robot bot2;

void setup() {
 size(720, 480);
 bot1 = new Robot("robot1.svg", 90, 80);
 bot2 = new Robot("robot2.svg", 440, 30);
 smooth();
}

void draw() {
 background(204);

 // Update and display first robot
 bot1.update();
 bot1.display();

 // Update and display second robot
 bot2.update();
 bot2.display();
}

class Robot {
 float xpos;
 float ypos;
 float angle;
 PShape botShape;
 float yoffset = 0.0;

// Set initial values in constructor
Robot(String svgName, float tempX, float tempY) {
 botShape = loadShape(svgName);
 xpos = tempX;
 ypos = tempY;
 angle = random(0, TWO_PI);
}
```

```
// Update the fields
void update() {
 angle += 0.05;
 yoffset = sin(angle) * 20;
}

// Draw the robot to the screen
void display() {
 shape(botShape, xpos, ypos + yoffset);
 }
}
```

CAPIYULO 10

10 Matrices

Hemos introducido nuevas ideas de programación en cada capítulo (variables, funciones, objetos) y ahora hemos llegado al último paso - matrices!
Una *matriz* es una lista de variables que comparten un nombre en común. Las matrices son útiles porque hacen posible trabajar con más variables sin crear un nuevo nombre para cada una. Esto hace el código más corto, más fácil de leer, y más conveniente para actualizar.

Ejemplo 10-1: Muchas variables

Para ver lo que queremos decir, referirse al ejemplo 7-3. Este código trabaja bien si movemos sólo una forma alrededor, pero si queremos mover dos? Necesitamos hacer una nueva variable X y actualizarla dentro de *draw()*:

```
float x1 = -20;
float x2 = 20;

void setup() {
 size(240, 120);
 smooth();
 noStroke();
}

void draw() {
 background(0);
 x1 += 0.5;
 x2 += 0.5;
 arc(x1, 30, 40, 40, 0.52, 5.76);
 arc(x2, 90, 40, 40, 0.52, 5.76);
}
```

Ejemplo 10-2: Muchas más variables

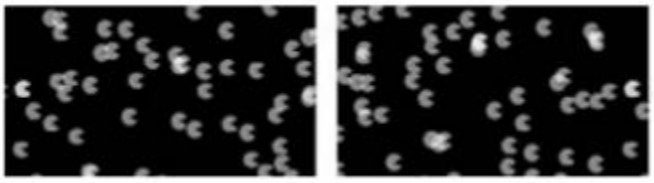
El código para el previo ejemplo sigue siendo manejable, pero si queremos tener cinco círculos? Necesitamos agregar tres variables más a las dos que ya tenemos:

```
float x1 = -10;
float x2 = 10;
float x3 = 35;
float x4 = 18;
float x5 = 30;

void setup() {
 size(240, 120);
 smooth();
 noStroke();
}

void draw() {
 background(0);
 x1 += 0.5;
 x2 += 0.5;
 x3 += 0.5;
 x4 += 0.5;
 x5 += 0.5;
 arc(x1, 20, 20, 20, 0.52, 5.76);
 arc(x2, 40, 20, 20, 0.52, 5.76);
 arc(x3, 60, 20, 20, 0.52, 5.76);
 arc(x4, 80, 20, 20, 0.52, 5.76);
 arc(x5, 100, 20, 20, 0.52, 5.76);
}
```

Este código comienza a salirse de control.

Ejemplo 10-3: Matrices, no variables

Imagina que podría pasar si quisieras tener 3000 círculos. Esto podría significar que tendrías que crear 3000 variables individuales, luego actualizar cada una separadamente. Podrías hacer un seguimiento de esas tantas variables? Te gustaría? Por ejemplo, usamos una matriz:



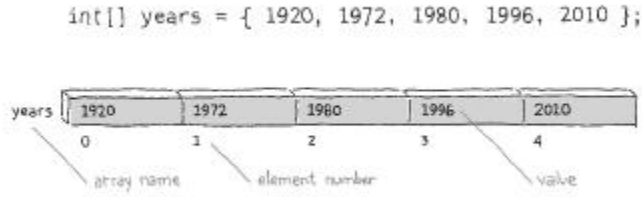
```
float[] x = new float[3000];

void setup() {
```

```
  size(240, 120);
  smooth();
  noStroke();
  fill(255, 200);
  for (int i = 0; i < x.length; i++) {
  x[i] = random(-1000, 200);
 }
}

void draw() {
background(0);
  for (int i = 0; i < x.length; i++) {
  x[i] += 0.5;
  float y = i * 0.4;
  arc(x[i], y, 12, 12, 0.52, 5.76);
 }
}
```

Gastaremos el resto de este capítulo hablando acerca de los detalles que hacen posible este ejemplo.

Hacer una matriz

Cada item en una matriz es llamado un *elemento*, y cada uno tiene un valor *index* para marcar su posición dentro de la matriz. Como las coordenadas en la pantalla, los valores index para una matriz empiezan contándose desde 0. Por ejemplo, el primer elemento en una matriz tiene el valor index 0, el segundo elemento en la matriz tiene un valor index 1, y así sucesivamente. Si hay 20 valores en la matriz, el valor index del último elemento es 19. La figura 10-1 muestra la estructura conceptual de una matriz.



La figura 10-1. Una matriz es una lista de una o más variables que comparten el mismo nombre.

Usar matrices es similar que trabajar con variables solas; se sigue el mismo patrón. Cómo ya sabes, puedes hacer una variable entera llamada X con este código:

```
int x;
```

Para hacer una matriz, Sólo se tiene que colocar entre paréntesis después del tipo de datos:

```
int[] x;
```

La belleza de crear una matriz es la habilidad para hacer 2,1 o 100,000 valores de variables con sólo una línea de código. Por ejemplo, la siguiente línea de código crea una matriz de 2,000 variables enteras:

```
int[] x = new int[2000];
```

Puedes hacer matrices de todos los tipos de datos de Processing: *boolean*, *float*, *String*, *PShape* etc, así como cualquier clase definida por el usuario. Por ejemplo, el siguiente código crea una matriz de 32 variables *PImage*:

```
PImage[] images = new PImage[32];
```

Para hacer una matriz, empieza con el nombre del tipo de datos, seguido por los paréntesis. El nombre que selecciones para la matriz es el siguiente, seguido por el operador de asignación (el símbolo de igual), seguido por la palabra clave *new*, seguido por el nombre del tipo de datos otra vez, con el número de elementos para crear entre paréntesis. Este patrón trabaja para todos los tipos de datos de matriz.

NOTA: *Cada matriz puede guardar sólo un tipo de datos (boolean, int, float, PImage, etc). No puedes mezclar y coincidir con diferentes tipos de datos dentro de una matriz. Si necesitas hacer esto, trabaja con objetos.*

Antes de salir adelante nosotros mismos, vamos a reducir la velocidad y hablar más detalladamente acerca del trabajo con matrices. Como haciendo un *objeto*, ahí hay tres pasos para trabajar con una matriz:

1. Declara la matriz y define el tipo de datos.

2. Crea la matriz con la palabra clave *new* y define la longitud.

3. Asigna valores a cada elemento.

Cada paso puede suceder en su propia línea, o todos los pasos pueden ser comprimidos juntos. Cada uno de los siguientes tres ejemplos muestra una técnica diferente para crear una matriz llamada X que guarda dos enteros, 12 y 2. Pon mucha atención a lo que pasa antes de *setup()* y a lo que pasa dentro de *setup()*.

Ejemplo 10-4: Declara y asigna una matriz

Primero declararemos una variable fuera de *setup()* y luego crearemos y asignaremos los valores adentro. El syntax x[0] se refiere al primer elemento de la matriz y x[1] al segundo:

```
int[] x; // Declare the array

void setup() {
 size(200, 200);
 x = new int[2]; // Create the array
 x[0] = 12; // Assign the first value
 x[1] = 2; // Assign the second value
}
```

Ejemplo 10-5: Asignación del conjunto compacto

Aquí hay un ejemplo un poco más compacto, en el cuál la matriz es declarada y creada en la misma línea, luego los valores son asignados dentro de *setup()*:

```
int[] x = new int[2]; // Declare and create the array

void setup() {
 size(200, 200);
 x[0] = 12; // Assign the first value
 x[1] = 2; // Assign the second value
}
```

Ejemplo 10-6: Asignar una matriz de una sola vez

También puedes asignar valores a la matriz cuando es creada, si es parte de la declaración:

```
int[] x = { 12, 2 }; // Declare, create, and assign

void setup() {
 size(200, 200);
}
```

NOTA: *Evite crear matrices dentro de draw(), por qué crear una nueva matriz en cada frame podría ralentizar la velocidad.*

Ejemplo 10-7: Revisar el primer ejemplo

Como un ejemplo completo de cómo usar una matriz, hemos grabado el ejemplo 10-1 aquí. Aunque no hemos visto todavía los beneficios revelados en el ejemplo 10-3, vemos algunos detalles importantes de cómo trabajan las matrices:

```
float[] x = {-20, 20};

void setup() {
 size(240, 120);
 smooth();
 noStroke();
}
void draw() {
 background(0);
 x[0] += 0.5; // Increase the first element
 x[1] += 0.5; // Increase the second element
 arc(x[0], 30, 40, 40, 0.52, 5.76);
 arc(x[1], 90, 40, 40, 0.52, 5.76);
}
```

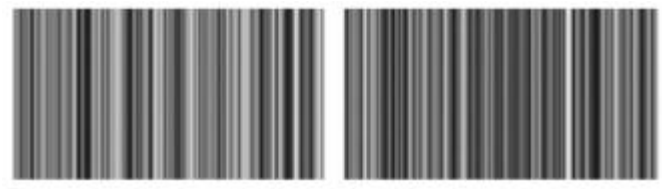
Repetición y matrices

El *for* loop, introducido en "Repetition" en el capítulo 4, hace más fácil trabajar con matrices largas mientras se mantiene conciso el código. La idea es escribir un loop para moverse a través de cada elemento de la matriz uno por uno. Para hacer eso, necesitas saber la longitud de la matriz. La variable *longitud* es asociada con cada matriz que guarda el número de elementos. Usamos el nombre de una matriz con el punto operador (un periodo) para acceder a este valor. Por ejemplo:

```
int[] x = new int[2]; // Declare and create the array
println(x.length); // Prints 2 to the Console
int[] y = new int[1972]; // Declare and create the array
println(y.length); // Prints 1972 to the Console
```

Ejemplo 10-8: Rellena una matriz en un *for* loop

Un *for* loop puede ser usado para rellenar una matriz con valores, o para leer valores atrás. En este ejemplo, la matriz es rellenada primero con números al azar dentro de *setup()*, y luego estos números son usados para establecer valores de grosor dentro de *draw()*. Cada vez que el programa está corriendo, es establecido un nuevo conjunto de números al azar que es puesto en la matriz:



```
float[] gray;

void setup() {
  size(240, 120);
  gray = new float[width];
  for (int i = 0; i < gray.length; i++) {
  gray[i] = random(0, 255);
 }
}

void draw() {
  for (int i = 0; i < gray.length; i++) {
  stroke(gray[i]);
  line(i, 0, i, height);
 }
}
```

Ejemplo 10-9: Seguir los movimientos del Mouse

En este ejemplo, hay dos matrices para guardar la posición del mouse- una para la coordenada X y una para la coordenada Y. Estas matrices guardan la ubicación del mouse para los 60 frames anteriores. Con cada nuevo frame, los valores de las viejas coordenadas X y Y son removidos y reemplazados con los valores actuales de mouseX y mouseY. Los nuevos valores son agregados a la primera posición de la matriz, pero antes de que esto pase, cada valor en la matriz es movido una posición a la derecha (de atrás hacia adelante) para darles espacio a los nuevos números. Este ejemplo visualiza esta acción.

También, en cada frame, las 60 coordenadas son usadas para dibujar una serie de elipses en la pantalla:

```
int num = 60;
int x[] = new int[num];
int y[] = new int[num];

void setup() {
 size(240, 120);
 smooth();
 noStroke();
}

void draw() {
 background(0);
 // Copy array values from back to front

for (int i = x.length-1; i > 0; i--) {
 x[i] = x[i-1];
 y[i] = y[i-1];
}

 x[0] = mouseX; // Set the first element
 y[0] = mouseY; // Set the first element
  for (int i = 0; i < x.length; i++) {
  fill(i * 4);
  ellipse(x[i], y[i], 40, 40);
 }
}
```

NOTA: *La técnica para guardar un desplazamiento de números búfer en una matriz mostrada en este ejemplo y la figura 10-2 es menos eficiente que una técnica alternativa que usa el operador % (módulo). Esto es explicado en los ejemplos* **->** *Basics* **->** *Input* **->** *StoringInput ejemplo incluidos con Processing.*

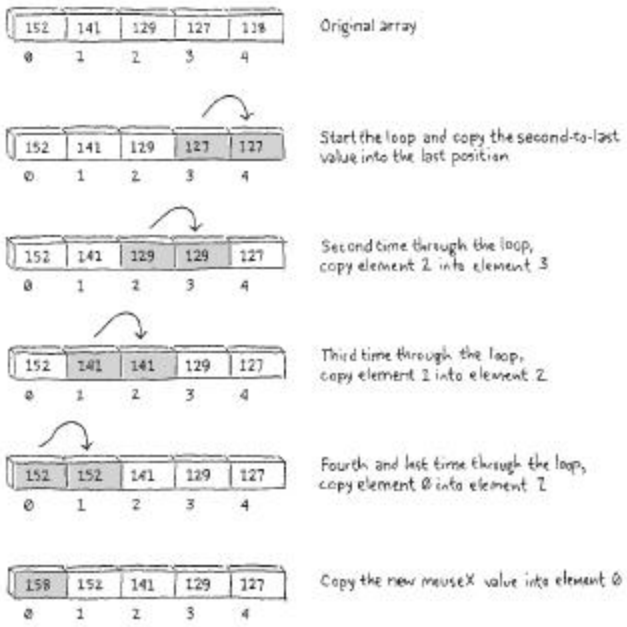
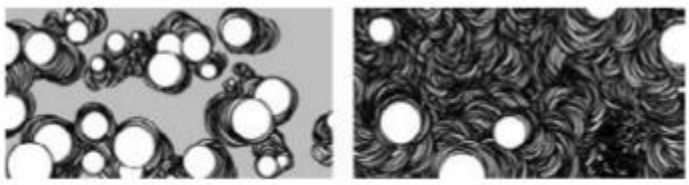


Figura 10-2. Cambio de los valores en una matriz un lugar hacia la derecha.

Matrices de los objetos

Los dos cortos ejemplos de esta sección reúnen todos los conceptos importantes de programación de este libro: variables, condicionales, funciones, objetos y matrices. Hacer una matriz de objetos es lo mismo que marcar las matrices introducidas en las páginas anteriores, pero hay una consideración adicional: porque cada elemento de una matriz es un objeto, primero debe ser creado con la palabra clave *new* (como cualquier otro objeto) antes de asignarlo a la matriz. Con una clase definida así como *JitterBug* (ver el capítulo 9), esto significa usar *new* para establecer cada elemento antes de que sea asignado a la matriz. O, para una construcción en la clase Processing así como *PImage*, lo que significa usar la función *loadImage()* para crear el objeto antes de que sea asignado.

Ejemplo 10-10: Manejar varios objetos

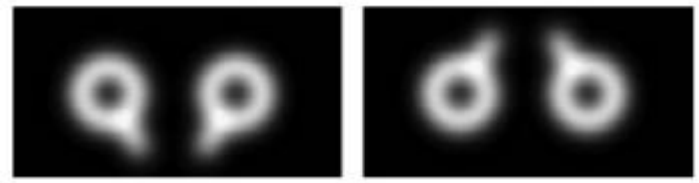


```
 JitterBug[] bugs = new JitterBug[33];

void setup() {
 size(240, 120);
 smooth();
 for (int i = 0; i < bugs.length; i++) {
  float x = random(width);
  float y = random(height);
  int r = i + 2;
  bugs[i] = new JitterBug(x, y, r);
 }
}

void draw() {
 for (int i = 0; i < bugs.length; i++) {
  bugs[i].move();
  bugs[i].display();
 }
}
// Copy JitterBug class here
```

El ejemplo final de matriz carga una secuencia de imágenes y guarda cada una como un elemento dentro de una matriz de objetos *PImage*.

Ejemplo 10-11: Secuencia de imágenes

Para correr este ejemplo, debes obtener las imágenes de la carpeta *media.zip* como describimos en el capítulo 6, las imágenes son nombradas secuencialmente (*frame-0001.png*, *frame-002.png*, y así sucesivamente), lo cual hace posible crear el nombre de cada archivo dentro de un *for* loop, como hemos visto en la línea dieciocho del programa:



```
int numFrames = 12; // The number of frames
PImage[] images = new PImage[numFrames]; // Make the array
int currentFrame = 1;

void setup() {
 size(240, 120);
 for (int i = 1; i < images.length; i++) {
 String imageName = "frame-" + nf(i, 4) + ".png";
 images[i] = loadImage(imageName); // Load each image
}
 frameRate(24);
}
```

```
void draw() {
 image(images[currentFrame], 0, 0);
 currentFrame++; // Next frame
 if (currentFrame >= images.length) {
  currentFrame = 1; // Return to first frame
 }
}
```

El formato de la función *nf()* se da en números así que *nf(1,4)* regresa al string "0001" y *nf(11,4)* regresa "0011". Estos valores son concatenados con el comienzo del nombre del archivo ("frame-") y el final (".png") para crear el nombre del archivo completo como una variable *String*. Los archivos son cargados en la matriz de la siguiente línea. Las imágenes son mostradas en la pantalla una a la vez en *draw()*. Cuando la última imagen en la matriz es mostrada, el programa regresa al principio de la matriz y muestra las imágenes de nuevo en secuencia.

Robot 8: Matrices

Las matrices hacen que sea más fácil para un programa trabajar con varios elementos. En este ejemplo, una matriz de objetos de *Robot* es declarada en la parte superior. Luego la matriz es asignada dentro de *setup()*, y cada objeto de *Robot* es creado dentro de *for* loop. En *draw()*, otro *for* loop es usado para actualizar y mostrar cada elemento de la matriz *bots*.

El *for* loop y una matriz hacen una combinación poderosa. Notarás las sutiles diferencias entre el código para este ejemplo y Robot 7 (ver "Robot 7: Objetos" en el capítulo 9) en contraste a los extremos cambios en el resultado visual. Una vez que una matriz es creada y un *for* loop puesto en un lugar, es más fácil trabajar con 3 elementos así como con 3000.

La decisión para cargar el archivo SVG dentro de *setup()* más bien que en la clase de *Robot* en la que el cambio desde Robot 7 es mayor. Esta

vez fue hecha así, porque que el archivo es cargado sólo una vez, en vez de muchas veces. Como hay muchos elementos en la matriz (en este caso, 20 veces). Este cambio hace que el código comience más rápido porque cargar un archivo toma tiempo, y se usa menos memoria porque el archivo es guardado una vez. Cada elemento de la matriz bot referencia el mismo archivo.

```
Robot[] bots; // Declare array of Robot objects

void setup() {
 size(720, 480);
 PShape robotShape = loadShape("robot1.svg");

// Create the array of Robot objects
bots = new Robot[20];

// Create each object
for (int i = 0; i < bots.length; i++) {

// Create a random x-coordinate
float x = random(-40, width-40);

// Assign the y-coordinate based on the order
float y = map(i, 0, bots.length, -100, height-200);
bots[i] = new Robot(robotShape, x, y);
}
smooth();
}

void draw() {
background(204);

// Update and display each bot in the array
for (int i = 0; i < bots.length; i++) {
 bots[i].update();
 bots[i].display();
}
}
class Robot {
 float xpos;
 float ypos;
 float angle;
 PShape botShape;
 float yoffset = 0.0;
// Set initial values in constructor
Robot(PShape shape, float tempX, float tempY) {
botShape = shape;
xpos = tempX;
ypos = tempY;
angle = random(0, TWO_PI);
}
```

CAPIYULO 11

11 ampliar

Este libro se enfoca en el uso de Processing para gráficos interactivos, porque es el núcleo de lo que hace Processing. Como sea, el software puede hacer mucho más y a menudo es parte de proyectos que se mueven más allá de una simple pantalla de computador. Por ejemplo, Processing ha sido usado para controlar máquinas, exportar imágenes para películas en alta definición, y exportar modelos para impresiones en 3D.

En las últimas décadas, Processing ha sido usado para videos musicales para Radiohead y R.E.M., para hacer ilustraciones para publicaciones como *Nature* y *the New York Times*, para sacar esculturas para exhibiciones en galerías, para controlar un video en una pared de 120x12- pies, para tejer suéteres, y mucho más. Processing tiene esta flexibilidad por su sistema de bibliotecas.

Una *biblioteca* de Processing es una colección de código que amplía el software más allá de su núcleo de funciones y clases. Las bibliotecas han sido importantes para el crecimiento del proyecto, porque permiten a los desarrolladores agregar nuevas funciones con rapidez. Como pequeños proyectos de auto contenidos, las bibliotecas son más fáciles de manejar que si estas funciones fueran integradas al software principal.

En adición a las bibliotecas incluidas con Processing (estas son llamadas bibliotecas *core*). Hay alrededor de 100 bibliotecas *contribuidas* que son vinculadas desde la website de Processing. Todas las bibliotecas son listadas online en http://processing.org/reference/libraries/.

Para usar una biblioteca, selecciona importar biblioteca desde el menú de dibujo. Escogiendo una biblioteca podrás agregar una línea de código que indica que la biblioteca será usada con el dibujo actual. Por ejemplo, cuando la biblioteca OpenGL es agregada, esta línea de código es agregada en la parte superior del dibujo:

```
import processing.opengl.*;
```

Antes de que una biblioteca contribuida sea importada a través del menú de dibujo, debe ser descargada desde el website y colocada dentro de la carpeta de *bibliotecas* en tu computador. Tu carpeta de *bibliotecas* está ubicada en tu libro de dibujos. Puedes encontrar la ubicación de tu libro de dibujos abriendo las preferencias. Ubica la biblioteca descargada en una carpeta dentro de tu libro de dibujos llamado *bibliotecas*. Si esta carpeta no existe todavía, créala.

Como mencionamos, ahí hay más de 100 bibliotecas Processing, por lo que evidentemente no todas pueden ser discutidas aquí. Hemos

seleccionado unas cuantas que pensamos son las más útiles y divertidas para introducirlas en este capítulo.

3D

Hay dos formas para dibujar en 3D con Processing; ambas requieren agregar un tercer parámetro a la función *size()* para cambiar la forma en que los gráficos dibujados. Por defecto, Processing dibuja usando un procesador en 2D que es muy preciso, pero lento. Este es el procesador JAVA2D. A veces más rápido pero con menos calidad esta su versión P2D, el procesador 2D de Processing. También puedes cambiar el procesador para Processing en 3D, uno llamado P3D, p OpenGL, para seguir tus programas para dibujar en una dimensión adicional, el z-axis (ver figura 11-1).

Procesador con Processing 3D así:

```
size(800, 600, P3D);
```

Y con OpenGL así:

```
size(800, 600, OPENGL);
```

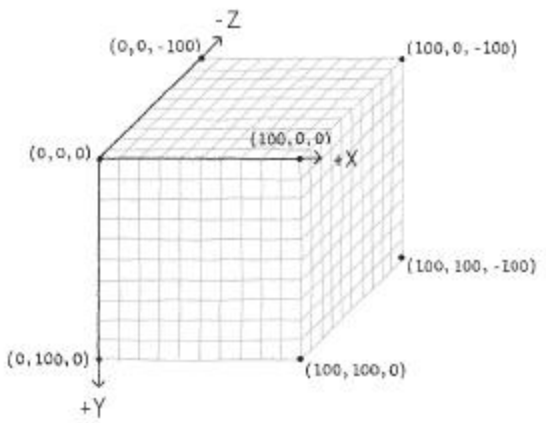


Figura 11-1. Coordenadas del sistema de Processing 3D

El procesador P3D está construido, pero el procesador OpenGL es una biblioteca y requiere la declaración de *importación* dentro del código, como se mostró en la parte superior del ejemplo 11-1. El procesador OpenGL hace uso de hardware con gráficas más rápidas que están disponibles en la mayoría de las máquinas que se venden hoy en día.

NOTA: *No se garantiza que el procesador OpenGL sea más rápido en todas las situaciones; ver la referencia de size() para más detalles.*

Muchas de las funciones introducidas en este libro tienen variaciones para trabajar en 3D. Por ejemplo, en las funciones de dibujo básico *point()*, *line()* y *vertex()* simplemente se agregan los parámetros Z a

los parámetro X y Y que fueron cubiertos anteriormente. Las transformaciones *translate()*, *rotate()* y *scale()* también operan en 3D.

Ejemplo 11-1: Un demo en 3D

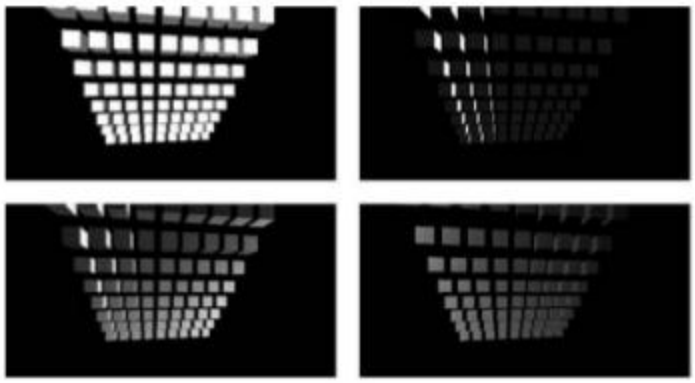
Más funciones en 3D son cubiertas en la referencia de Processing, pero aquí hay un ejemplo para empezar:

```
import processing.opengl.*;

void setup() {
 size(440, 220, OPENGL);
 noStroke();
 fill(255, 190);
}

void draw() {
 background(0);
 translate(width/2, height/2, 0);
 rotateX(mouseX / 200.0);
 rotateY(mouseY / 100.0);
 int dim = 18;
 for (int i = -height/2; i < height/2; i += dim*1.2) {
  for (int j = -height/2; j < height/2; j += dim*1.2) {
   beginShape();
   vertex(i, j, 0);
   vertex(i+dim, j, 0);
   vertex(i+dim, j+dim, -dim);
   vertex(i, j+dim, -dim);
   endShape();
  }
 }
}
```

Cuando empiezas a trabajar en 3D, nuevas funciones estarán disponibles para explorar. Es posible cambiar la cámara, la iluminación y propiedades de los materiales, para dibujar formas en 3D como esferas y cubos.

Ejemplo 11-2: Iluminación

Este ejemplo se construye sobre el ejemplo 11-1 pero reemplazando los rectángulos con cubos y agregando unos cuantos tipos de luces. Intenta comentar o descomentar diferentes luces para ver cómo trabaja cada una por sí misma y en combinación con otras:



```
import processing.opengl.*;

void setup() {
 size(420, 220, OPENGL);
 noStroke();
 fill(255);
}

void draw() {
 lights();
 //ambientLight(102, 102, 102);
 //directionalLight(255, 255, 255, // Color
 // -1, 0, 0); // Direction XYZ
 //pointLight(255, 255, 255, // Color
 // mouseX, 110, 50); // Position
 //spotLight(255, 255, 255, // Color
 // mouseX, 0, 200, // Position
 // 0, 0, -1, // Direction XYZ
 // PI, 2); // Concentration
rotateY(PI/24);
background(0);

translate(width/2, height/2, -20);
int dim = 18;
for (int i = -height/2; i < height/2; i += dim*1.4) {
for (int j = -height/2; j < height/2; j += dim*1.4) {
pushMatrix();
translate(i, j, -j);
box(dim, dim, dim);
popMatrix();
}
}
}
```

Hay cuatro tipos de luces in Processing: spot, point, directional y ambient. Las luces spot irradian en una forma cónica; tiene una dirección, ubicación, y color. Las luces point irradian desde un punto como una bombilla de cualquier color. Las luces direccionales se proyectan en una dirección para crear luces fuertes y oscuras. Las

luces ambiente crean una luz uniforme de cualquier color sobre toda la escena y son usadas siempre con otras luces. La función *lights()* necesita una iluminación por defecto establecida con una luz ambient y directional. Las luces necesitan ser reseteadas cada vez a través de *draw()*, así que deberían aparecer en la parte superior de *draw()* para asegurar resultados consistentes.

Trabajar en 3D introduce la idea de una "cámara" que es puesta en la escena tridimensional siendo construida. Como una cámara del mundo real, se asigna el espacio 3D en la bandera del plano 2D de la pantalla. Moviendo la cámara cambia la forma en que Processing mapea las coordenadas 3D de tu dibujo en una pantalla 2D.

Ejemplo 11-3: La cámara de Processing

Por defecto, Processing crea una cámara que apunta al centro de la pantalla, por lo tanto las formas de distancia desde el centro se ven en perspectiva. La función *camera()* ofrece el control sobre la ubicación de la cámara, la ubicación en la que se ha señalado, y la orientación (arriba, abajo, inclinado). En el siguiente ejemplo, el mouse es usado para mover la ubicación hacia donde la cámara está apuntando:

```
import processing.opengl.*;

void setup() {
 size(420, 220, OPENGL);
 noStroke();
}

void draw() {
 lights();
 background(0);
 float camZ = (height/2.0) / tan(PI*60.0 / 360.0);
 camera(mouseX, mouseY, camZ, // Camera location
 width/2.0, height/2.0, 0, // Camera target
 0, 1, 0); // Camera orientation
 translate(width/2, height/2, -20);
 int dim = 18;
 for (int i = -height/2; i < height/2; i += dim*1.4) {
  for (int j = -height/2; j < height/2; j += dim*1.4) {
  pushMatrix();
  translate(i, j, -j);
  box(dim, dim, dim);
  popMatrix();
```

```
   }
  }
}
```

Esta sección ha presentado la punta del iceberg desde la capacidad 3D. En adición a la funcionalidad básica mencionada aquí, hay muchas bibliotecas de Processing que ayudan a generar formas en 3D, cargando y exportando formas en 3D, y proporcionando un control más avanzado de la cámara.

Exportación de la Imagen

La imágenes animadas creadas por un programa de Processing pueden ser convertidas en una secuencia de archivos con la función *saveFrame()*. Cuando *saveFrame()* aparece al final de *draw()*, salva una secuencia numerada de imágenes del programa del salida en formato TIFF llamadas *screen*-0001.tif, *screen*-0002.tif, y así sucesivamente, a la carpeta de dibujos. Estos archivos pueden ser importados a un programa de video o animación y salvados como un archivo de película. También puedes especificar tu propio nombre de archivo y formato de archivo de imagen con una línea de código así:

```
saveFrame("output-####.png");
```

NOTA: *Cuando estés usando saveFrame() dentro de draw(), un nuevo archivo es guardado en cada frame – así que ten cuidado, como este puede llenar rápidamente tu carpeta de dibujos con miles de archivos.*

Usa el símbolo # (almohadilla) para mostrar donde aparecerán los números en el nombre del archivo. Ellos son reemplazados con los números del frame actual cuando los archivos son guardados. También puedes especificar una subcarpeta para guardar las imágenes dentro, lo cual es útil cuando estás trabajando con varias imágenes frames:

```
saveFrame("frames/output-####.png");
```

Ejemplo 11-4: Guardando imágenes

Este ejemplo muestra cómo guardar imágenes mediante el almacenamiento de suficientes frames para una segunda animación. Si el programa corre a 30 frames por segundo y luego sale después de 60 frames:


```
float x = 0;

void setup() {
 size(720, 480);
 smooth();
 noFill();
 strokeCap(SQUARE);
 frameRate(30);
}

void draw() {
 background(204);
 translate(x, 0);
 for (int y = 40; y < 280; y += 20) {
  line(-260, y, 0, y + 200);
  line(0, y + 200, 260, y);
 }

if (frameCount < 60) {
 saveFrame("frames/SaveExample-####.tif");
 } else {
 exit();
 }
x += 2.5;
}
```

Processing escribirá una imagen basada en la extensión del archivo que usas (.png, .jpg, o .tif se construyen y algunas plataformas pueden ayudar a otros). Una imagen .tif es guardada sin comprimir, lo cual es más rápido pero toma mucho espacio en el disco duro. Ambas .png y .jpg crearán archivos pequeños, pero usualmente están comprimidos, lo requiere más tiempo para guardar, haciendo que el dibujo corra más lento.

Si tu salida es de gráficos vectoriales, puedes escribir la salida para archivos PDF para mejor resolución. La biblioteca para exportar PDF hace posibles escribir archivos PDF directamente desde un dibujo. Estos archivos gráficos vectoriales pueden ser escalados a cualquier tamaño sin perder resolución, lo cual los hace ideales para imprimir - desde posters y banderas hasta un libro entero.

Ejemplo 11-5: Dibujar a un PDF

Este ejemplo se basa en el ejemplo 11-4 para dibujar más galones de diferentes pesos, pero remueve el movimiento. Crea un archivo PDF llamado Ex-11-5.pdf para los parámetros tres y cuatro de *size()*:

```
import processing.pdf.*;

void setup() {
 size(600, 800, PDF, "Ex-11-5.pdf");
 noFill();
```

```
 strokeCap(SQUARE);
}

void draw() {
 background(255);
  for (int y = 100; y < height - 300; y+=20) {
  float r = random(0, 102);
  strokeWeight(r / 10);
  beginShape();
  vertex(100, y);
  vertex(width/2, y + 200);
  vertex(width-100, y);
  endShape();
 }
 exit();
}
```

La geometría no está dibujada en la pantalla; está escrito directamente en el archivo PDF, el cual es guardado en la carpeta de dibujos. Este código en este ejemplo corre una vez y luego sale al final de *draw()*. El resultado saliente es mostrado en la figura 11-2.

Hay más ejemplos de exportaciones PDF incluidos con el software Processing. Mire en la sección de exportación PDF de los ejemplos de Processing para ver más técnicas.

Figura 11-2. PDF exportado desde el ejemplo 11-5.

Hola Arduino

Arduino es un prototipo de plataforma electrónica con una serie de tablas micro controladoras y el software del programa. Processing y Arduino comparten una larga historia juntos; son hermanos de proyectos con varias ideas y metas similares, aunque se dirigen a dominios separados. Porque comparten el mismo editor y desarrollador de programación y una sintaxis similar, es fácil moverse entre ellos y transferir conocimiento acerca de uno en el otro.

En esta sección, nos enfocamos leer datos en Processing desde un tablero de Arduino y luego visualizar esos datos en la pantalla. Esto hace posible el uso de nuevas entradas en los programas de Processing y el seguimiento de los programas de Arduino para ver las entradas de sus sensores como gráficos. Estas nuevas entradas pueden ser cualquier cosa que conceda a un tablero de Arduino. Estos dispositivos van desde una distancia sensorial a una brújula o una red de malla de sensores de temperatura.

Esta sección asume que tienes un tablero de Arduino y que tienes los conocimientos básicos para usarlo. Si no es así, puedes aprender más online en http://www.arduino.cc y en el excelente libro *Getting Started with Arduino* de Massimo Banzi (O'Reilly). Unas vez hayas cubierto lo básico, puedes aprender más acerca del envío de datos entre Processing y Arduino en otro sobresaliente libro, *Making Things Talk* de Tom Igoe (O'Reilly).

Los datos pueden ser transferidos entre un dibujo en Processing y un tablero de Arduino con ayuda de la biblioteca serial de Processing. *Serial* es un formato de datos que envía un byte a la vez. En el mundo del Arduino, un byte es un tipo de datos que puede guardar valores entre 0 y 255; trabaja como un *int*, pero con un rango mucho más pequeño. Los números más grandes son enviados rompiéndolos en una lista de bytes y volviéndolos a montar luego.

En los siguientes ejemplos, nos enfocamos en el lado de relación de Processing y en mantener simple el código de Arduino. Visualizamos los datos que vienen desde el tablero de Arduino un byte a la vez. Con las técnicas cubiertas en este libro y los cientos ejemplos online de Arduino, esperamos que esto sea suficiente para que comiences.

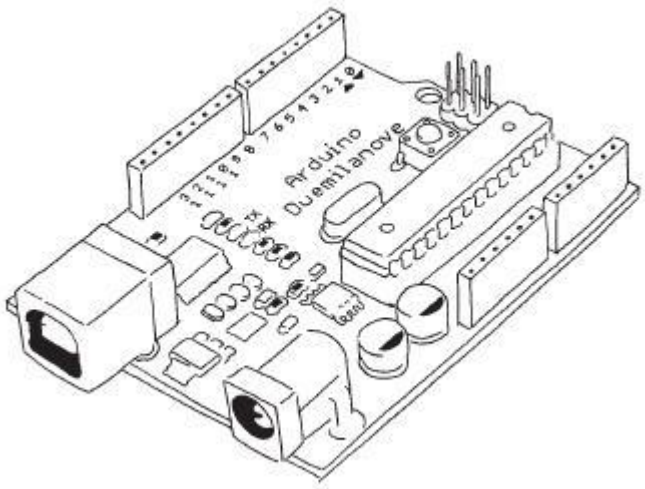


Figura 11-3. Un tablero de Arduino Duemilanove.

Ejemplo 11-6: Leer un Sensor

El siguiente código de Arduino es usado con los siguientes tres ejemplos de Processing:

```
// Note: This is code for an Arduino board, not Processing

int sensorPin = 0;    // Select input pin
int val = 0;

void setup() {
 Serial.begin(9600); // Open serial port
}

void loop() {
 val = analogRead(sensorPin) / 4;    // Read value from sensor
 Serial.print(val, BYTE);            // Print variable to serial port
 delay(100);                         // Wait 100 milliseconds
}
```

Existen dos detalles importantes acerca de este ejemplo de Arduino. Primero, requiere que se adjunte un sensor en la entrada análoga en el pin 0 del tablero de Arduino. Debes usar un sensor de luz (también llamado una foto celda, célula fotoeléctrica, o resistor de luz dependiente) u otro resistor análogo así como un termistor (resistor sensitivo de temperatura), un sensor de flexión, o un sensor de presión (resistor sensitivo de fuerza). El diagrama del circuito y el dibujo de la tablilla con los componentes se muestran en la figura 11-4. Luego, notarás que el valor es regresado por la función *analogRead()* y es dividido en 4 antes de que sea asignado a *val*. Los valores de *analogRead()* están entre 0 y 1023, así que dividimos por 4

para convertirlos a un rango de 0 a 255 de forma que los datos puedan ser enviados en un sólo byte.

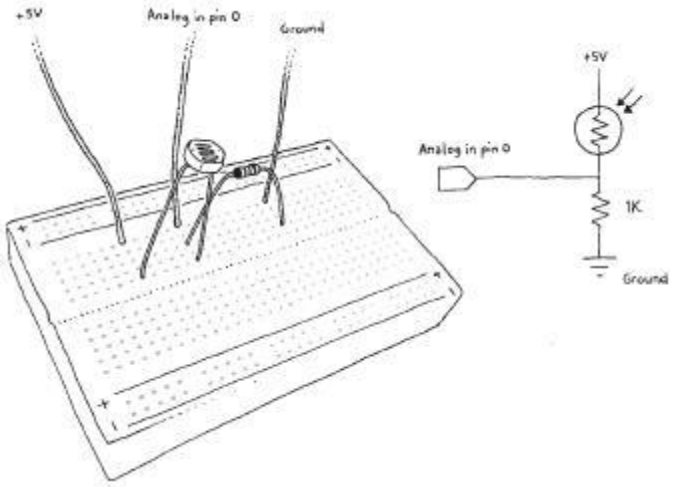


Figura 11-4. Adjuntando una luz a un pin analogo 0.

Ejemplo 11-7: Leer los datos del Puerto Serial

La primera visualización del ejemplo muestra cómo leer los datos seriales desde el tablero de Arduino y cómo convertir datos en los valores que se ajustan a las dimensiones de la pantalla:

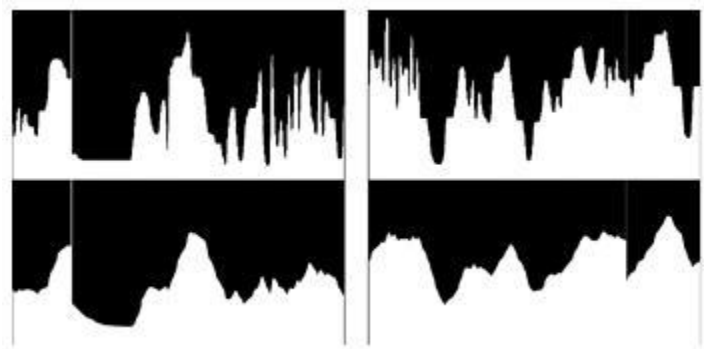
```
import processing.serial.*;

Serial port;     // Create object from Serial class
float val;       // Data received from the serial port

void setup() {
 size(440, 220);
 // IMPORTANT NOTE:
 // The first serial port retrieved by Serial.list()
 // should be your Arduino. If not, uncomment the next
 // line by deleting the // before it. Run the sketch
 // again to see a list of serial ports. Then, change
 // the 0 in between [ and ] to the number of the port
 // that your Arduino is connected to.
 //println(Serial.list());
 String arduinoPort = Serial.list()[0];
 port = new Serial(this, arduinoPort, 9600);
}

void draw() {
 if (port.available() > 0) {          // If data is available,
  val = port.read();                  // read it and store it in val
  val = map(val, 0, 255, 0, height);  // Convert the value
}
 rect(40, val-10, 360, 20);
}
```

La biblioteca serial es importada en la primera línea y el puerto serial es abierto en *setup()*. Puede que sea o no sea fácil que tu dibujo de Processing hable con el tablero de Arduino; eso depende de la configuración del hardware. A menudo hay más de un dispositivo que el de dibujo de Processing tratando de comunicarse con él. Si el código no trabaja la primera vez, lee cuidadosamente el comentario en *setup()* y sigue las instrucciones.

Dentro de *draw()*, el valor es comparado dentro del programa con el método *read()* de los objetos seriales. El programa lee los datos desde el puerto serial solamente cuando un nuevo byte está disponible. El método *available()* controla la vista si un nuevo byte está listo y regresa el número de bytes disponible. Este programa está escrito para que un sólo byte nuevo sea leído cada vez a través de *draw()*. La función *map()* convierte en valor entrante desde su rango inicial de 0 a 255 a un rango de 0 a la altura de la pantalla; en este programa es desde 0 a 220.

Ejemplo 11-8: Visualizar la corriente de datos

Ahora que los datos están llegando a través, vamos a visualizarlos en un formato más interesante. Los valores llegan directamente desde un sensor que a menudo es errático, y es útil para suavizar por un promedio de ellos. Aquí, presentamos la señal sin procesar desde el sensor de luz ilustrado en la figura 11-4, en la mitad superior del ejemplo y la señal suavizada en la mitad inferior



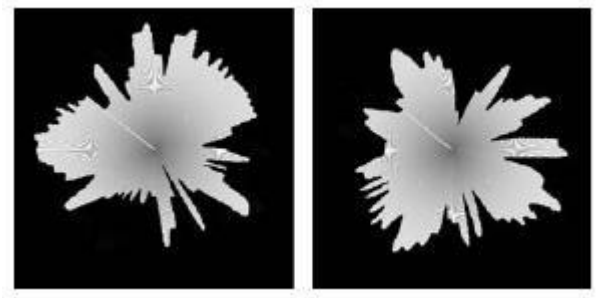
```
import processing.serial.*;

Serial port;   // Create object from Serial class
float val;     // Data received from the serial port
int x;
float easing = 0.05;
float easedVal;

void setup() {
 size(440, 440);
 frameRate(30);
 smooth();
```

```
 String arduinoPort = Serial.list()[0];
 port = new Serial(this, arduinoPort, 9600);
 background(0);
}

void draw() {
 if ( port.available() > 0) {        // If data is available,
  val = port.read();                 // read it and store it in val
  val = map(val, 0, 255, 0, height); // Convert the values
}

float targetVal = val;
easedVal += (targetVal - easedVal) * easing;

stroke(0);
line(x, 0, x, height);               // Black line
stroke(255);
line(x+1, 0, x+1, height);           // White line
line(x, 220, x, val);                    // Raw value
line(x, 440, x, easedVal + 220);         // Averaged value

x++;
if (x > width) {
  x = 0;
 }
}
```

Similar a los ejemplo 5-8 y 5-9, este dibujo usa la técnica facilitadora. Cada nuevo byte del tablero de Arduino es establecido como un valor target, la diferencia entre el valor actual y el valor target es calculada, y el valor actual es movido más cerca al target. Ajustar la variable *easing* para afectar la cantidad de suavizante aplicado a los valores entrantes.

Ejemplo 11-9: Otra forma de mirar los Datos

Este ejemplo está inspirado por los radares de las pantallas. Los valores son leídos en la misma forma desde el tablero de Arduino, pero son visualizados en un patrón circular usando las funciones *sin()* y *cos()* introducidas anteriormente en los ejemplo del 7-12 al 7-15:



```
import processing.serial.*;

Serial port;    // Create object from Serial class
float val;      // Data received from the serial port
float angle;
float radius;

void setup() {
 size(440, 440);
 frameRate(30);
 strokeWeight(2);
 smooth();
 String arduinoPort = Serial.list()[0];
 port = new Serial(this, arduinoPort, 9600);
 background(0);
}

void draw() {
 if ( port.available() > 0) { // If data is available,
  val = port.read(); // read it and store it in val
  // Convert the values to set the radius
  radius = map(val, 0, 255, 0, height * 0.45);
 }

int middleX = width/2;
int middleY = height/2;
float x = middleX + cos(angle) * height/2;
float y = middleY + sin(angle) * height/2;
stroke(0);
line(middleX, middleY, x, y);

x = middleX + cos(angle) * radius;
y = middleY + sin(angle) * radius;
stroke(255);
line(middleX, middleY, x, y);

angle += 0.01;
}
```

La variable *angle* es actualizada continuamente para mover la línea de dibujo al valor actual alrededor del círculo, y la variable *val* escala la longitud de la línea en movimiento para establecer su distancia desde el centro de la pantalla. Después de una vez alrededor del círculo, los valores comienzan a escribir en la parte superior de los datos anteriores.

Estamos emocionados con el potencial del uso de Processing y Arduino juntos para salvar el mundo del software y la electrónica. A diferencia de los ejemplos aquí impresos, la comunicación puede ser bidireccional. Los elementos en la pantalla pueden afectar lo que está pasando en el tablero de Arduino. Esto significa que puedes usar un programa de Processing como una interface entre tu computador y

motores, altavoces, luces, cámaras, sensores, y casi cualquier cosa que pueda ser controlada con una señal eléctrica. De nuevo, más información acerca de Arduino puede ser encontrada en http://www.arduino.cc.

Comunidad

Hemos trabajado duro para hacer fácil la exportación de los programas de Processing de manera que puedas compartirlos con otros. En el segundo capítulo, discutimos cómo compartir tus programas mediante la exportación. Creemos que el intercambio favorece el aprendizaje y la comunidad. Así como modificas tus programas desde este libro y empiezas a escribir tus propios programas a partir de cero, te animamos a que leas esta sección del libro y compartas tu trabajo con otros. En este momento, los grupos en OpenProcessing, Vimeo, Flickr, y la Wiki de Processing, son lugares emocionantes para visitar y contribuir también. En Twitter, las búsquedas de #Processing y Processing.org ceden resultados interesantes. Estas comunidades siempre están moviéndose y corriendo. Mira el sitio principal de Processing (http://www.processing.org) otros links frescos como este:

»» http://www.openprocessing.org

»» http://www.vimeo.com/tag:processing.org

»» http://www.flickr.com/groups/processing/

»» http://www.delicious.com/tag/processing.org/

A Codificación de Consejos

Codificar es un tipo de escritura. Como todos los tipos de escritura, el código tiene unas reglas específicas. Para comparar, mencionaremos rápidamente algunas de las reglas para el inglés que probablemente no sabes acerca de un tiempo, ya que es de una segunda naturaleza. Algunas de las reglas más invisibles son escritas de izquierda a derecha y ponen un espacio entre cada palabra. Otras normas más evidentes son convenciones ortográficas, la capitalización de los nombres de las personas y los lugares, y usando puntuación al final de las oraciones para proporcionar énfasis! Si quebramos una o más de estas reglas cuando escribimos un email a un amigo, el mensaje todavía sigue a través. Por ejemplo, "Hola ben, Qué más" comunica casi igual de bien que, "Hola Ben. Cómo estás hoy?" Como sea, la flexibilidad con las reglas de escritura no transfiere a la programación. Porque tu escritura para comunicarte con un computador, en vez de comunicarte con otra persona, necesita ser más precisa y cuidadosa. Un carácter fuera de lugar es a menudo una diferencia entre un programa que corre y uno que no.

Processing trata de decirte donde has cometido errores y adivinar que es el error. Cuando oprimes el botón de Run, si tienes problemas de gramática (sintaxis) con tu código (los llamamos *Bugs*), luego el Mensaje de Área se torna rojo y Processing trata de destacar la línea de código que se sospecha tiene el problema. La línea de código con el bug es a menudo una línea arriba o abajo de la línea destacada, aunque en algunos casos, no está ni cerca.
El texto en el área de mensaje trata de ser útil y sugiere el problema potencial, pero a veces el mensaje es demasiado críptico para ser entendido. Para un principiante, este error de mensajes puede ser frustrante. Entiende que Processing es una simple pieza de software que está tratando de ser útil, pero tiene un conocimiento limitado de lo que estás tratando de hacer.

Los errores de mensaje largos son imprimidos en la consola más detalladamente, y algunas veces desplazarse a través del texto puede dar indicios. Adicionalmente, Processing puede encontrar solamente un bug a la vez. Si tu programa tiene varios bugs, necesitarás seguir corriendo el programa y arreglarlo uno a la vez.

Por favor lee y vuelve a leer las siguientes sugerencias cuidadosamente para ayudarte a escribir el código limpio.

Funciones y Parámetros

Los programas están compuestos de varias partes pequeñas, las cuales son agrupadas juntas para hacer largas estructuras. Nosotros tenemos un sistema similar en Inglés: las palabras son agrupadas en frases,, las cuales son combinadas para hacer oraciones, las cuales son combinadas para crear párrafos. La idea es la misma en código, pero las partes pequeñas tengan diferentes nombres y se comporten

diferente. Sus *funciones* y *parámetros* son dos partes importantes. Las funciones son los bloques de construcción básicos de un programa de Processing. Los parámetros son valores que definen cómo se comporta la función.

Considera una función como *background()*. Como el nombre lo sugiere, es usada para establecer el color de fondo de la pantalla. La función tiene tres parámetros que definen el color. Estos números definen los componentes rojo, verde y azul del color para definir el color compuesto. Por ejemplo, el siguiente código dibuja un fondo azul:

```
background(51, 102, 153);
```

Mira cuidadosamente esta sola línea de código. Los detalles clave son los paréntesis después del nombre de la función que adjunta los números, las comas entre cada número, y el punto y coma al final de la línea. El punto y coma es usado como un periodo. Esto significa que una declaración a terminado así que el computador puede mirar para el comienzo del siguiente. Todas estas partes necesitan estar ahí para que el código corra. Compara la línea del ejemplo anterior con estas tres versiones quebradas de la misma línea:

```
background 51, 102, 153; // Error! Missing the parentheses
background(51 102, 153); // Error! Missing a comma
background(51, 102, 153) // Error! Missing the semicolon
```

El computador es muy implacable de la más mínima omisión y desviación de lo que está esperando. Si recuerdas estas partes, tendrás menos bugs. Pero si olvidas escribirlos, lo cual todos hacemos, no es un problema. Processing te alertará acerca del problema, y cuando esté arreglado, el programa correrá bien.

Codificando el color

Los códigos del color del medio ambiente de Processing en diferentes partes de cada programa. Palabras que son parte de Processing son dibujadas como azules o naranjas para distinguirlas de las otras partes del programa que inventaste. Las palabras que son únicas en tu programa, así como tus variables y nombres de funciones, son dibujadas en negro. Símbolos básicos como (), [], y > también son negros.

Comentarios

Los comentarios son notas que escribes para tí mismo (u otra gente) dentro del código. Debes usarlos para clarificar lo que el código está haciendo en un lenguaje llano y proporcionar información adicional así como el título y autor del programa. Un comentario comienza con dos barras inclinadas (//) y continúa hasta el final de la línea:

```
// This is a one-line comment
```

Puedes hacer una línea múltiple de comentarios comenzando con /* y terminando con */. Por ejemplo":

```
/* This comment
continues for more
than one line
*/
```

Cuando un comentario es escrito correctamente, el color del texto se tornará gris. Todo el área del comentario se torna gris así que claramente puedes ver dónde comienza y termina.

Mayúsculas y Minúsculas

Processing distingue letras mayúsculas de letras minúsculas y por lo tanto lee "Hello" como una palabra distinta de "hello". Si estás tratando de dibujar un rectángulo con la función *rect()* y escribes *rect()*, el código no correrá. Puedes ver si Processing reconoce tu intento de código mirando el color del texto.

Estilo

Processing es flexible acerca de cuánto espacio es usado para el formato de tu código. A Processing no le importa si escribes:

```
 rect(50, 20, 30, 40);
or:
rect (50,20,30,40);
or:
rect ( 50,20,
30, 40) ;
```

Como sea, está en tu interés hacer que el código sea fácil de leer. Esto se convierte en algo especialmente importante cuando el código crece en longitud. Limpiar el formato hace inmediatamente que la estructura del código sea legible, y el formato descuidado a menudo oscurece los problemas. Logra el hábito de escribir el código limpio. Hay varias formas para dar formato al código bien, y la forma en que escojas guardar tus cosas es una preferencia personal.

Consola

La consola es el botón del área de desarrollo de Processing. Puedes escribir mensajes a la consola con la función *println()*. Por ejemplo, el siguiente código imprime un mensaje con el tiempo actual:

```
println("Hello Processing.");
println("The time is " + hour() + ":" + minute());
```

Un Paso a la Vez

Recomendamos escribir unas cuantas líneas de código a la vez y correr el código frecuentemente para asegurar que los bugs no se acumulen sin su conocimiento. Todos los programas ambiciosos son escritos con una línea a la vez. Rompe tu proyecto en subproyectos más simples y completalos uno a la vez así tendrás varios pequeños sucesos, en lugar de un enjambre de bugs. Si tienes un bug, intenta aislar el área del código donde pienses que el problema miente. Intenta pensar en los errores de fijación como si estuvieras resolviendo un misterio o un rompecabezas. Si te quedas atrancado o frustrado, toma un descanso para aclarar tu mente o pide ayuda a un amigo. Algunas veces, la respuesta está delante de tu nariz pero requiere una segunda opinión para hacerla más clara.

B Tipos de Datos

Existen diferentes categorías de datos. Por ejemplo, piensa acerca de los datos en una tarjeta ID. La tarjeta tiene números para guardar peso, altura, fecha de nacimiento, dirección de la casa, y código postal. Tiene palabras para guardar el nombre de una persona y la ciudad. Existe también datos de imagen (una foto) y a menudo un órgano de decisión de los donantes, el cual tiene una decisión si/no. En Processing, tenemos diferentes tipos de datos para guardar cada tipo de datos. Cada uno de los siguientes tipos es explicado con más detalle en otra parte del libro, pero esto es un glosario.

Name	Description	Range of values
int	Integers (whole numbers)	-2,147,483,648 to 2,147,483,647
float	Floating-point values	-3.40282347E+38 to 3.40282347E+38
boolean	Logical value	true or false
char	Single character	A–z, 0–9, and symbols
String	Sequence of characters	Any letter, word, sentence, and so on
PImage	PNG, JPG, or GIF image	N/A
PFont	VLW font; use the Create Font tool to make	N/A
PShape	SVG file	N/A

Como pauta, un número *float* tiene cerca de cuatro dígitos de precisión después del punto decimal. Si estás contando o tomando pequeños pasos, deberías usar un valor *int* para tomar los pasos, y luego tal vez escalarlos por un float si es necesario cuando los use.

Existen más tipos de datos que los que mencionamos aquí, pero estos son los más útiles para el trabajo que hacemos típicamente en Processing. De hecho, como mencionamos en el capítulo 9, hay infinitos tipos de datos, porque cada clase es un tipo de datos diferente.

C Orden de las Operaciones

Cuando los cálculos matemáticos son realizados en un programa, cada operación toma lugar de acuerdo a un orden preespecificado. Este *orden de las operaciones* asegura que el código corra de la misma forma todo el tiempo. Esto no es distinto que la aritmética o álgebra, pero programando tiene otros operadores que son menos familiares.

En la siguiente tabla, los operadores en la parte superior se corren antes de los de abajo. Por lo tanto, una operación dentro de paréntesis correrá primero y una asignación correrá de último.

Name	Symbol	Examples
Parentheses	()	a * (b + c)
Postfix, Unary	++ -- !	a++ --b !c
Multiplicative	* / %	a * b
Additive	+ -	a + b
Relational	> < <= >=	if (a > b)
Equality	== !=	if (a == b)
Logical AND	&&	if (mousePressed && (a > b))
Logical OR	||	if (mousePressed || (a > b))
Assignment	= += -= *= /= %=	a = 44

D Ámbito de las Variables

Las reglas del ámbito de las variables es definido simplemente: una variable creada dentro de un bloque (código encerrado en tirantes: {and}) existe solamente dentro de este bloque. Esto significa que una variable creada dentro de *setup()* puede ser usada solamente dentro del bloque de *setup()*, igualmente, una variable declarada dentro de *draw()* puede ser usada solamente dentro del bloque de *draw()*. La excepción a esta regla es una variable declarada fuera de *setup() y draw()*. Estas variables pueden ser usadas en *setup()* y en *draw()* (o dentro de cualquier otra función que hayas creado). Piensa en el área de fuera de *setup()* y *draw()* como un bloque de código implícito. A estas variables las llamamos *variables globales*, porque pueden ser usadas en cualquier parte dentro del programa. Llamamos a una variable que es usada solamente dentro de un bloque una *variable local*. Los siguientes son una pareja de ejemplos de código que además explican el concepto. Primero:

```
int i = 12; // Declare global variable i and assign 12

void setup() {
 size(480, 320);
 int i = 24; // Declare local variable i and assign 24
 println(i); // Prints 24 to the console
}
```

```
void draw() {
 println(i); // Prints 12 to the console
}
```

Y segundo:

```
void setup() {
 size(480, 320);
 int i = 24; // Declare local variable i and assign 24
}

void draw() {
 println(i); // ERROR! The variable i is local to setup()
}
```