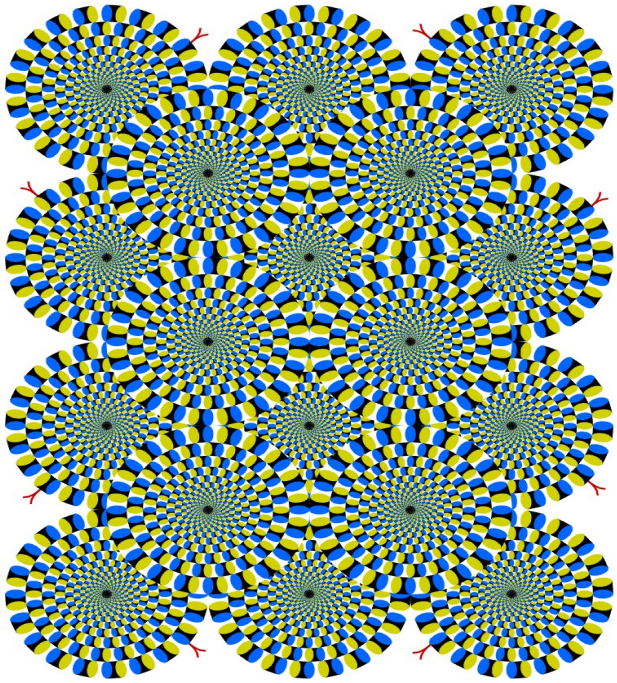
9 Procesamiento de imagen & Percepción



Ver es creer.
Proverbio

Página opuesta: Serpientes rotatorias (Una ilusión visual)
Creado por Akiyoshi Kitaoka (www.ritsumei.ac.jp/~akitaoka/index-e.html)

El robot tiene una pequeña cámara digital que puede ser utilizada para sacar fotos. Una foto tomada por una cámara digital se representa como una *imagen*. Como pudieron ver en el capítulo previo, las imágenes pueden ser dibujadas y se las puede mover dentro de una ventana de gráficos como si fueran cualquier otro objeto gráfico (como una línea, un círculo, etc.). También vimos en el capítulo 5 cómo una imagen tomada de la cámara del robot puede ser visualizada en sus monitores en una ventana aparte. En este capítulo aprenderemos cómo realizar computación sobre imágenes. Aprenderemos cómo se representan las imágenes y cómo podemos crearlas a través de la computación y procesarlas de maneras muy distintas. La representación de imágenes que usaremos es la misma que la que utilizan la mayoría de las cámaras digitales y teléfonos celulares, y la misma representación puede ser usada para mostrarlas en una página Web. También aprenderemos cómo una imagen tomada por la cámara del robot puede ser útil como el ojo de la cámara hacia este mundo usando algunas técnicas de compresión de imágenes. La compresión de imágenes es el primer paso hacia la percepción visual. Un robot equipado con aunque fuera las capacidades de percepción visual más rudimentarias puede ser diseñado para llevar a cabo comportamientos más importantes.

¿Qué es una imagen?

En Myro pueden emitir un comando para que el robot saque una foto y la muestre en el monitor de la computadora usando los comandos:

```
pic = takePicture()
show(pic)
```

La foto de la siguiente página muestra un ejemplo de una imagen tomada con la cámara del robot. Una imagen se compone de muchos pequeños elementos o *píxeles*. En una imagen color, cada píxel contiene información del color conformada por la cantidad de valores de rojo, verde y azul (también llamada RGB). Cada uno de estos valores puede estar dentro del rango [0..255] y por lo tanto se requieren 3 bytes o 24 bits para alojar la información contenida en un píxel. Un píxel de color rojo puro tendrá los valores RGB (255, 0, 0). Una imagen en blanco y negro, por otro lado, sólo contiene el nivel de gris en un píxel que puede ser representado en un solo byte (8 bits) como un número que va desde el 0..255 donde 0 es el negro y 255 es el blanco. La imagen entera es sólo una formación doble dimensional de píxeles. Por ejemplo, las imágenes obtenidas con el Scribbler tienen 256x192 (WxH) o un total de 49,152 píxeles. Dado que cada píxel requiere de 3 bytes de datos, esta imagen tiene un tamaño de 147,456 bytes.

Todas las cámaras digitales se venden especificando el número de megapixeles. Por ejemplo, la cámara que se muestra en la figura posee a 6.3 megapíxeles. Esto hace referencia al tamaño de la imagen más

grande que puede sacar. Cuantos más píxeles en una imagen, mejor será la resolución de la imagen al imprimirla. Con una imagen de 6.3 megapíxeles serán capaces de crear impresiones de buena calidad en un tamaño de 13x12 pulgadas, e incluso más. En comparación, un rollo fotográfico tiene apenas 4000x3000 o 12 millones de píxeles. Las cámaras digitales fácilmente sobrepasan esta resolución y esto es por que, en la última década hemos visto un rápido declive de la fotografía de rollo. Para mostrar imágenes nítidas en una computadora se necesita menos de la mitad de la resolución ofrecida por la cámara que se muestra aquí.

La cámara del Scribbler, con una imagen de 147,456 bytes es de solamente 0.14 megapixels. Aunque sea una resolución baja, es suficiente para realizar la percepción visual del robot.

Para alojar y transferir electrónicamente imágenes (web, e-mail, etc.) de manera rápida y conveniente, los datos en una imagen se pueden comprimir. Varios formatos están disponibles para alojar imágenes electrónicamente: JPEG, GIF, PNG, etc. JPEG es el formato más común usado por cámaras digitales, incluyendo la del Scribbler. El JPEG permite una excelente compresión, sumada a un rango más amplio de colores en comparación con el formato GIF, lo cual lo hace más útil para la mayoría de las aplicaciones de imágenes. Myro soporta formatos JPEG y GIF. Cuando tengamos la intención de procesar una imagen, siempre usaremos el formato JPEG. Usaremos el formato GIF para crear imágenes animadas.

Cuestiones Básicas de Imágenes Myro

Después de sacar una foto con el Scribbler como hicimos arriba, pueden obtener información acerca del tamaño de la imagen con las funciones `getWidth()` y `getHeight()`:

```
picWidth = getWidth(pic)
picHeight = getHeight(pic)
print "Image WxH is", picWidth, "x", picHeight, “pixels.”
```

Si desean guardar la imagen para un uso futuro, pueden usar el comando Myro:

```
savePicture(pic, "OfficeScene.jpg")
```

El archivo `OfficeScene.jpg` será guardado en la carpeta actual. La extensión .jpg le indica al comando que guarde la imagen como JPEG. Si desean guardarla como una imagen GIF, tendrán que usar la extensión. gif, como se muestra abajo:

```
savePicture(pic, "OfficeScene.gif")
```

Más adelante, podrán cargar la foto del disco con la función `makePicture()`:

```
mySavedPicture = makePicture("OfficeScene.jpg")
show(mySavedPicture)
```

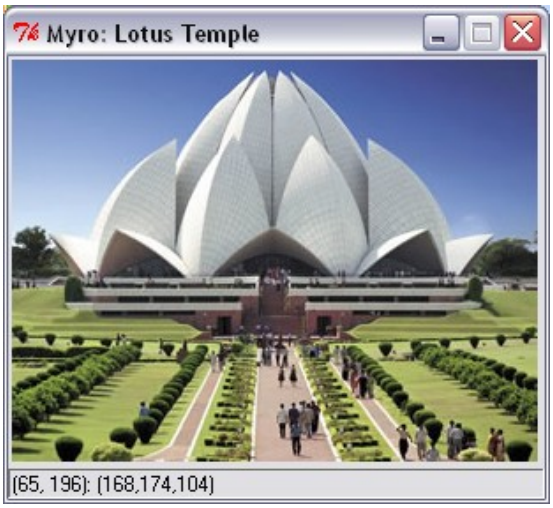
Una linda combinación de comandos que permite navegar y seleccionar la imagen a cargar es la siguiente:

```
mySavedPicture = makePicture(pickAFile())
show(mySavedPicture)
```

El comando `pickAFile` provee una caja de diálogo navegable, igual a la que se puede ver cuando abren y seleccionan archivos en cualquier otra aplicación de computadora. Pueden navegar hacia cualquier carpeta y seleccionar un archivo para abrir como imagen. De hecho, pueden usar el comando `makePicture` para cargar cualquier archivo de imagen JPEG sin importar si fue creado de su propia cámara o la han bajado de la web. Abajo les mostramos cómo cargar una imagen y mostrarla:

```
lotusTemple = makePicture(pickAFile())
show(lotusTemple, “Lotus Temple”)
```

Si mueven el mouse y hacen click en cualquier parte de la foto, también pueden obtener las coordenadas x- e y- del píxel que han seleccionado, junto con sus valores RGB. Esto se muestra en la foto a la derecha. El mouse se ha cliqueado en el píxel ubicado en (65, 196) y sus valores RGB eran (168,174,104). ¿Pueden adivinar dónde es esta ubicación en la foto? Dicho sea de paso, el comando show toma un segundo parámetro opcional que es una cadena de caracteres que se transforma en el título de la ventana de la imagen. Una ventaja de ser capaces de cargar y visualizar cualquier imagen es que también podemos aprender a procesar o manipular estas imágenes en la computadora. Volveremos al procesamiento de imágenes más adelante en el capítulo.



Un robot explorador

Si no necesitan una imagen a color, pueden decirle a Myro que capture una imagen en escala de grises dándole a la función `takePicture()` el parámetro `"gray"`.

```
grayPic = takePicture("gray")
show(grayPic)
```

Habrán notado que sacar la foto en grises tomó menos tiempo que sacar la foto en colores. Como explicamos anteriormente, una foto en escala de grises sólo utiliza un byte por píxel, en lugar de tres. Dado que las imágenes en escala de grises pueden ser transferidas del robot a su computadora de manera más rápida que las de color, pueden ser útiles si desean que se actualicen rápidamente. Por ejemplo, pueden usar la función `joyStick()` combinada con un loop que saca la foto y la muestra para trasformar el robot en un explorador piloteado, similar a los rovers de Marte.

```
joyStick()
for i in range(25):
  pic = takePicture("gray")
  show(pic)
```

El código de arriba abrirá una ventana de joy stick para que puedan controlar al robot y luego tomar y mostrar 25 fotos, una después de la otra. Mientras que se están tomando y mostrando las fotos como en una película, pueden usar el joystick para manejar al robot, usando las imágenes para guiarlo. Por supuesto que si

sacaran el parámetro "gray" de la función takePicture(), obtendrían fotos en color en lugar de en escala de grises, pero tardarían mucho más en transferirse del robot a su computadora, y dificultarían el control del robot.

Películas GIF animadas

La función savePicture() también les permite realizar un GIF animado, que es un tipo especial de foto que en un navegador Web mostrará varias fotos, una después de la otra, en una animación. Para guardar un GIF animado, deben darle a la función savePicture() una lista de fotos (en lugar de una sola) y un nombre de archivo que termine en ".gif". Este es un ejemplo:

```
pic1 = takePicture()
turnLeft(0.5,0.25)
pic2 = takePicture()
turnLeft(0.5,0.25)
pic3 = takePicture()
turnLeft(0.5,0.25)
pic4 = takePicture()

listOfPictures = [pic1, pic2, pic3, pic4]

savePicture(listOfPictures, "turningMovie.gif")
```

La mejor manera de visualizar un archivo GIF animado es usando un navegador Web. En su navegador favorito, usen el menú FILE->Open File, y luego elijan el archivo turningMovie.gif. El navegador Web mostrará todos los cuadros en la película y se detendrá en el último. Para volver a ver la película, presionen el botón de "Reload". También pueden usar un loop para hacer una película más larga con más imágenes:

```
pictureList = []  #Comenzamos con una lista vacía
for i in range(15):
  pic = takePicture()
  pictureList = pictureList + [pic]   #Agregamos una nueva foto
  turnLeft(0.5,0.1)

savePicture(pictureList,"rotatingMovie.gif")
```

Los comandos de arriba crean una película GIF animada de 15 fotos tomadas mientras que el robot rotaba en su lugar. Esta es una buena manera de capturar una escena completa alrededor del robot.

Hacer imágenes

Dado que una imagen es sólo un conjunto de píxeles, también es posible crear o componer las propias imágenes coloreando cada píxel en forma individual. Myro provee comandos simples para llenar o examinar un valor de color o de grises en píxeles individuales. Pueden usar cómputos para determinar qué llenar en cada píxel. Para empezar, pueden crear una imagen en blanco de la siguiente manera:

```
W = H = 100
newPic = makePicture(W, H, black)
show(newPic)
```

La imagen de 100x100 píxeles comienza con todos sus píxeles coloreados de negro puro (por ejemplo RGB = (0,0,0)). Si preferirían que todos los píxeles tuvieran un valor diferente, pueden especificar sus valores RGB:

```
newPic = makePicture(W, H, makeColor(R,G,B))
```

También, si alguna vez lo necesitan, también pueden establecer los píxeles en cualquier color, digamos que blanco, usado el loop:

```
for x in range(W)
  for y in range(H):
    pixel = getPixel(newPic, x, y)
    setColor(pixel, white)
repaint(newPic)
```

El comando `getPixel` devuelve el píxel en las ubicaciones `x-` e `y-` especificadas en la imagen. `setColor` establece el color del píxel dado a cualquier color que se especifique. Arriba estamos usando el color blanco predeterminado. Pueden crear un color nuevo especificando sus valores RGB en el comando:

```
myRed = makeColor(255, 0, 0)
```

Para seleccionar visualmente un color y sus valores RGB correspondientes, pueden usar el comando:

```
myColor = pickAColor()
```

Se mostrará una paleta de colores de la cual podrán seleccionar un color a elección. La paleta también les muestra los valores RGB del color elegido. Luego de elegir un color y presionar OK, el valor de `myColor` será el color seleccionado.

El comando `repaint` actualiza la imagen expuesta para que puedan ver los cambios que han hecho. En el loop de ejemplo de arriba, lo estamos usando luego de que todos los píxeles se han modificado. Si desean ver los cambios a medida que se realizan, pueden incluir el comando repaint dentro del loop:

```
for x in range(W)
  for y in range(H):
    pixel = getPixel(newPic, x, y)
    setColor(pixel, white)
    repaint(newPic)
```

Podrán visualizar cada píxel que se está modificando. Sin embargo, también notarán que esto lleva una cantidad considerable de tiempo aún en la pequeña imagen que estamos creando. Por eso es una buena idea la de actualizar una vez que se hayan modificado todos los píxeles.

Sumado al comando `setColor`, Myro tiene también el comando `setRGB` que puede ser utilizado para establecer el color de un píxel. Este comando usa valores RGB en lugar de un color.

```
setRGB(pixel, (255, 0, 0))
```

También hay un comando correspondiente para obtener los valores RGB de un píxel dado:

```
r, g, b = getRGB(pixel)
```

El comando `getRGB` devuelve el triple (R,G,B) que puede ser asignado a una variable individual como se muestra arriba. Además, dado un píxel, pueden obtener los valores RGB individuales usando los comandos `getRed`, `getGreen`, y `getBlue`. Estos se describen con más detalle al final de este capítulo y se ilustran con ejemplos abajo.

Muchas situaciones de creación y procesamiento de imágenes requieren el procesamiento de cada píxel de la imagen. En los loops de arriba, estamos usando las variables x- e y- para alcanzar a cada píxel en la imagen. Una forma alternativa de lograr el mismo resultado es usando la siguiente forma del loop:

```
for pixel in getPixels(newPic):
  setColor(pixel, gray)
repaint(newPic)
```

Como en la función `timeRemaining` usada en los capítulos anteriores, `getPixels` devuelve el siguiente píxel a ser procesado cada vez alrededor del loop, garantizando de esta manera que todos los píxeles de la imagen estén procesados. Solamente para observar cómo el loop de arriba funciona, quizás deseen probar y poner repaint dentro del loop. La diferencia entre los dos métodos es que el primer loop obtiene un píxel con unas coordenadas `x-` e `y-` dadas, mientras que el último les da un píxel a la vez, sin preocuparse por sus valores `x-` e `y-`.

Sombras de Gris

Usando los comandos de acceso y modificación de píxeles de imágenes se pueden escribir cómputos para crear imágenes interesantes y creativas. Para introducirlos en las técnicas básicas de creación y procesamiento de imágenes, crearemos algunas imágenes completamente dentro del espectro de la escala de grises. En una imagen JPEG, todas las sombras de gris tienen iguales valores RGB. La sombra más oscura de gris es (0,0,0) o negro y la más clara es (255,255,255) o blanca. En lugar de preocuparnos por los triples valores RGB, podemos pensar solamente en un valor en el rango de 0..255 para representar un espectro de grises. Podemos escribir una simple función que devolverá el tono RGB de gris de la siguiente manera:

```
def gray(v):
   # Retorna un color gris RGB para  v
   return makeColor(v, v, v)
```

Generemos una imagen que muestre toda la escala de tonos de gris usando la siguiente regla:

pixel at x, y is colored using the grayscale x+y

Es decir, un píxel de la imagen en 0, 0 obtendrá un tono 0+0=0 o negro, y un píxel en 50,50 obtendrá el tono

50+50=100 lo cual será un gris intermedio. Podemos lograr esto usando el siguiente loop:

```
def main():
   MAX = 255
   W = 100
   H = 100
   # Creamos una imagen en blanco
   myPic = makePicture(W, H)
   show(myPic, “Escalas de grises”)
   # Llenamos cada  pixel con  x+y escala de gris
   for x in range(W):
      for y in range(H):
         setPixel(myPic, x, y, gray((x+y)%(MAX+1))
```

Hemos usado la variable `MAX` arriba para representar el valor máximo del rango de la escala de grises. En la última línea, tomamos el restante `(x+y)%(MAX+1)` para asegurarnos de que el valor de gris estará dentro del rango 0..255. La imagen generada por el comando de arriba se muestra a la derecha. Puede parecer raro al principio que hemos usado las ubicaciones x- e y- de un píxel para computar sus valores dentro de la escala de grises o su brillo. Dado que `x-` e `y-` son números y como el valor de gris en sí es un número está bien hacer esto. Al reconocer que los valores `x-` e `y-` (y su suma) pueden ser más grandes que el rango del espectro de la escala de grises, tomamos el recordatorio de envolver los valores.

Realizar la siguiente tarea: Escriban un programa completo e intenten esto. Una vez que tienen el programa, pruébenlo en una imagen de un tamaño de 123x123 (¿por qué?). Luego, de nuevo pruébenlo en una imagen mucho más grande, digamos de 500x500.

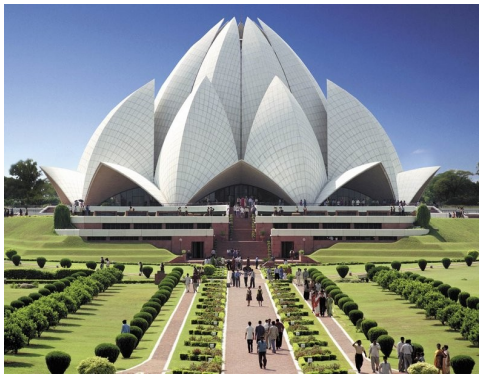
El rango de grises básicamente representa los niveles de brillo. Podemos divertirnos más si cambiamos el rango, digamos a 0..50. Entonces podemos modificar la función de gris como se muestra abajo para mapear proporcionalmente cualquier valor dentro del rango 0..50 al rango 0..255:

```
MAX = 50
def gray(v):
   # returns an rgb gray color for v
   brightness = v*255/MAX
   return makeColor(brightness, brightness, brightness)
```

Realizar la siguiente actividad: Reescribir el programa para usar la definición de arriba de `gray` y crear una imagen de 500x500. Modifiquen el valor de MAX a 25 y generen una imagen de 100x100 (mostrada arriba). Observen que si establecen a MAX en 255, se revertirá a la versión previa de la función.

Realizar la siguiente actividad: A continuación utilicen la regla:

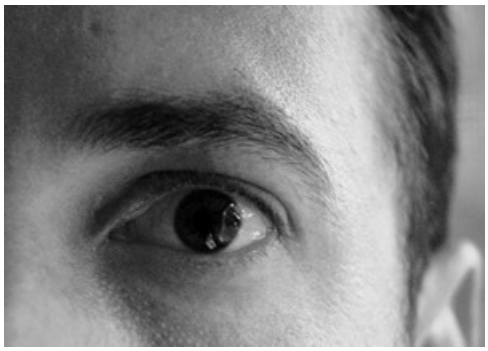
*pixel at x, y is colored using the grayscale x*y (píxel en x se colorea usando la escala de grises x*y)*



Recuerden tomar lo restante como arriba. Cambien `MAX` a 250 y generen una imagen de 500x500. Prueben otros tamaños y valores de `MAX`.

Pueden experimentar con otras reglas, transformaciones geométricas y funciones trigonométricas. Vean los ejercicios al final de este capítulo para más sugerencias.

Procesamiento de imágenes



Uno no siempre debe comenzar con una imagen en blanco para crear una nueva. Pueden tomar imágenes ya existentes y transformarlas de formas interesantes. Los principios básicos siguen siendo los mismos: acceden a un píxel individual y a su valor de color y lo transforman de la manera que quieran. En esta sección haremos algunas transformaciones de imágenes simples para ilustrarlo. Todas las transformaciones de imágenes que aprenderemos requerirán de la misma transformación en todos los valores de píxeles de una imagen dada. También pueden, por supuesto, optar por seleccionar regiones específicas si lo desean.

Usaremos las imágenes que se muestran aquí para aprender acerca de la transformación de imágenes. Siéntanse libres de seleccionar una o más imágenes de su propia cámara digital. En la mayoría de los ejemplos hemos usado transformaciones en imágenes en escala de gris. Siéntanse libres de usar imágenes en color para experimentar con ellas. También notarán que cuanto más grande sea el tamaño de una imagen, más tardará en transformarse. Esta es una consecuencia directa de la cantidad de cómputos que deben hacerse. Por ejemplo, para transformar una imagen de 500x500 estarán operando 250,000 píxeles. Si cada transformación de píxel involucra 10 operaciones, ¡estarán especificando 25 millones de operaciones! Lo que esto demore dependerá de la computadora que estén usando. Volveremos sobre esto con más detalle más adelante en el capítulo. Por ahora, pruebe restringirse a imágenes pequeñas, digamos que no más grandes que 300x300 píxeles.

Como mencionamos antes, aún las cámaras digitales más rudimentarias en estos días proveen varios megapíxeles de resolución de imagen. Por lo tanto, lo primero que debemos aprender es cómo achicar una imagen dada a un tamaño más chico.

Achicar & Agrandar

Imagen original (300x213)

Luego de achicarla (150x106, F=2)

Escribamos un programa que tome una imagen y la achique por un factor, digamos, F. Por ejemplo, si la imagen original es de 3000x3000 píxeles y la achicamos por un factor de 10, terminaríamos con una imagen de 300x300 píxeles. Esta transformación se logra seleccionando el valor de cada píxel de la nueva imagen basado en la imagen vieja. Podemos usar la siguiente regla para nuestra transformación:

*New pixel at x, y is a copy of the old pixel at x*F, y*F (Nuevo píxel en x, y es una copia del viejo píxel en x*F, y*F*

El programa de abajo le pide al usuario que elija una imagen, la muestre y luego le pide que especifique el factor por el cual quiere achicarla (que debe ser más grande que 1).

```
def main():
   # read an image and display it
   oldPic = makePicture(pickAFile())
   show(myPic, "Before")

   X = getWidth(oldPic)
   Y = getHeight(oldPic)
   # Input the shrink factor and computer size of new image
   F = int(ask("Enter the shrink factor."))
   newx = X/F
   newy = Y/F
   # create the new image
   newPic = makePicture(newx, newy)
   for x in range(1,newx):
      for y in range(1,newy):
         setPixel(newPic, x, y, getPixel(myPic, x*F, y*F))
   show(newPic, "After")
```

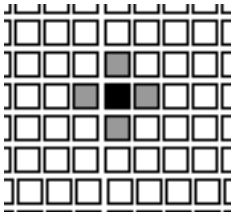
Realizar la siguiente actividad: Seleccionar una o más imágenes y asegúrense de que estén en formato JPEG y de que sean imágenes de alta resolución (para nuestro propósito, de 400x400 píxeles o más). Hagan correr el programa sobre cada imagen y observen el output por distintos valores de F. Modifiquen el

programa para usar el comando `savePicture` para guardar las imágenes en un archivo para usar en los ejemplos de abajo.

El agrandamiento se realiza de manera complementaria usando la regla:

New pixel at x, y is a copy of the old pixel at x/F, y/F (el nuevo píxel en x, y es una copia del píxel viejo en x/F, y/F)

El tamaño de la nueva imagen será F veces el tamaño de la imagen original. Cuando agrandan una imagen de una más pequeña, en realidad están generando datos donde no los había. Por lo tanto, es probable que la imagen resultante sea más borrosa que la original. Esto es similar a los reproductores de DVD que proclaman convertir una película a una definición más alta que la de la señal estándar. Si observan detenidamente, podrán ver defectos en las imágenes convertidas.



Vecindad de 4 píxeles

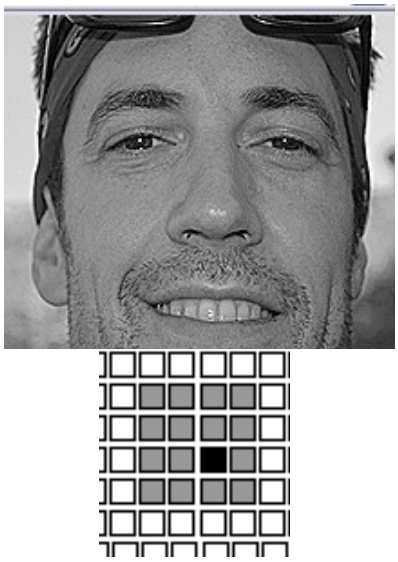
Imagen borrosa

Borroso & Nítido

Pueden hacer más borrosa una imagen tomando el promedio de los valores de color de los píxeles vecinos. El tamaño de la vecindad determina cuán borrosa es la imagen. Abajo, mostramos cómo hacer borroso cada píxel usando una vecindad de 4 (ver imagen arriba):

```
n1 = getPixel(oldPic, x, y)
n2 = getPixel(oldPic, x, y-1)
n3 = getPixel(oldPic, x-1, y)
n4 = getPixel(oldPic, x+1, y)
n5 = getPixel(oldPic, x, y+1)
v = (getRed(n1)+...+getRed(n5))/5
setPixel(newPic, x, y, gray(v))
```

Observen arriba que `n1` es el píxel original. Pueden usar diferentes tamaños y formas de vecindades para crear efectos borrosos más elaborados. La siguiente página muestra los efectos de usar una vecindad más grande, y aún cuadrada.



vecindad de 15-píxeles

En lugar de promediar los valores de los píxeles, si le restan los valores de píxeles terminan haciendo más nítida la imagen. Es decir,

```
v = 5*getRed(n1) -
      (getRed(n2)+…+getRed(n5))
setPixel(newPic, x, y, gray(v))
```

Imagen borrosa

Realizar la siguiente actividad: Escriban un programa completo que use la fórmula para promediar la vecindad para hacer más borrosas y nítidas las imágenes. Dependiendo del tamaño y la forma de la vecindad que elijan, los rangos del índice del loop variarán. Por ejemplo, el loop para la vecindad de 4- píxeles aparecerá de la siguiente manera:

```
for x in range(1,X-1):
   for y in range(1,Y-1):
      # Average pixel values...
```

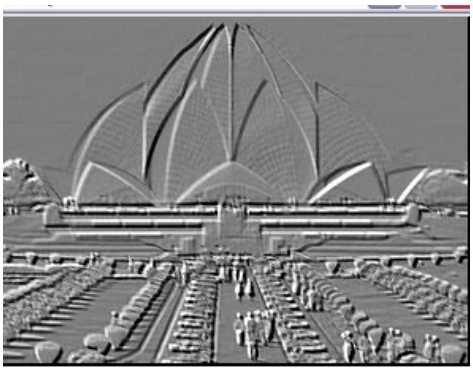
A medida que la vecindad crece, tendrán que ajustar los valores de iniciar y detener de las variables de x e y en el loop de arriba.

Negativo & Repujado

La creación del negativo de una imagen se obtiene invirtiendo los valores de los píxeles. Para valores en escala de grises esto equivale a restar de 255. Para píxeles de color deben restar de 255 para cada uno de los valores RGB. La regla para escala de grises es:

New pixel at x,y is 255-value of old pixel at x,y (Nuevo píxel enx, y es 255- valor del viejo píxel en x, y)

Negativo



Repujado

Pueden especificar la transformación completa usando el loop:

```
for x in range(X):
    for y in range(Y):
        pixel = getPixel(oldPic, x, y)
        v = MAX – getRed(pixel)
        setPixel(newPic, x, y, gray(v))
```

De manera similar, pueden crear un efecto de repujado o de relieve restando del valor actual de los píxeles, el valor (o una fracción del mismo) de un píxel a dos píxeles de distancia:

```
pixel1 = getPixel(oldPic, x, y)
pixel2 = getPixel(oldPic, x+2, y+2)
v = getRed(pixel1) – getRed(pixel2)
setPixel(newPic, x, y, gray(v))
```

o

```
v = getRed(pixel1) + (MAX/2 – getRed(pixel2))
```

Realizar la siguiente actividad: Implementen las dos reglas mostradas arriba. Prueben crear un negativo de una imagen color, así como el de una imagen en escala de grises. Prueben distintas fracciones para el repujado para ver el efecto.

Seguramente, mientras leían y experimentaban con las técnicas de arriba, se preguntaban sobre el tipo de procesamiento de imágenes realizado por algunos de los programas de software populares de edición de fotos. Quizás ya estén familiarizados o han escuchado de Adobe Photoshop, Fireworks, GIMP, etc. La mayoría de estos programas usan técnicas similares a las que hemos descrito arriba y proveen más facilidades y opciones para la manipulación de imágenes.

Comprensión de Imágenes & Visión del Robot

La imagen que se muestra a la derecha fue tomada de un artículo de diario de un diario local en el oeste de New York. Lean el texto que la acompaña y vean si pueden identificar a cada individuo en la foto. Luego de leer el texto, sin duda, confirmarán que la persona en el centro de la fila de adelante es el DT Kevin Starr. Sería muy difícil identificar a las personas si no estuvieran presentes los textos. De todas maneras, podrían responder a preguntas como cuántas personas hay en la foto. Este es un ejemplo del problema de *comprensión de imagen*: Dada una imagen, cómo pueden determinar, computacionalmente, que hay siete personas en la imagen, que tres de ellos están arrodillados y cuatro parados, etc. Tampoco tendrían problemas para dibujar una caja en el rostro de cada persona en la imagen. En esta sección les introducimos algunas técnicas básicas para comprensión de imágenes y para usarlas para la creación de un sistema de percepción visual para el robot Scribbler. Ya están familiarizados con los comandos básicos de procesamiento de imágenes. Los mismos comandos serán usados para la comprensión de imágenes.

"LOS MEJORES – El equipo de volley de los All-WNY rodea al DT del año, Kevin Starr de Frontier. En la hilera superior, desde la izquierda: Brian Heffernan de Eden, Tom Kait de Lake Shore, Greg Gegenfurtner de Sweet Home yGeorge Beiter de Kenmore West. Fila de abajo: desde la izquierda, están David Nagowski de Eden y Rich Bland de Orchard Park."

Para que un robot perciba su entorno visualmente debe comenzar por tomar una foto. Una vez obtenida la imagen, la percepción de los contenidos de la imagen se realiza computacionalmente. En lugar de tomar la tarea de comprensión de imagen mostrada arriba,

comenzaremos de manera sencilla: ¿cómo reconoce una pelota el Scribbler? Una vez que la reconoce, ¿puede seguirla a donde sea que vaya?

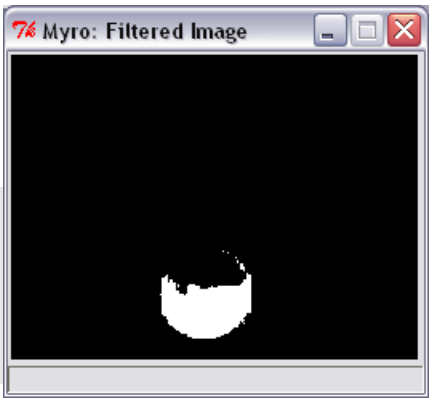
El segundo problema se hace sencillo una vez que sabemos como reconocer una pelota. Podemos escribir un programa de control simple así:

```
while timeTemaining(T):
    # take picture and locate the ball
    …
    if <ball is on the left of visual field>:
        turnLeft(cruiseSpeed)
    elif <ball is in the middle of visual field>:
        forward(cruiseSpeed)
    else:
        turnRight(cruiseSpeed)
```

El principal problema es localizar la pelota en la imagen. Dados los comandos de manipulación de imagen que aprendieron hasta ahora, piensen cómo se podría lograr esto.

Los principios básicos en la comprensión de imágenes se basan en comenzar en el nivel del píxel. Hagamos que las cosas sean un poco más concretas. Miren la imagen que se muestra arriba. Fue tomada por un robot Scribbler.

Para nuestros propósitos hemos exagerado la escena al elegir una pelota de color brillante (rosa). Cuando hagan click en el mouse en cualquier parte de la imagen les dará los valores RGB de ese píxel. La foto muestra los valores de un píxel en el centro de la pelota. Observen que el píxel rojo es 253 y el valor del píxel verde es de 66. Dicho sea de paso, el verdadero rosa tiene valores RGB (255,175,175). Podemos usar esta información como forma de identificar la pelota. Aquí hay un loop de transformación de imagen simple que convierte todos los píxeles en la imagen en píxeles blancos o negros. Si un píxel tiene un alto valor de rojo y un valor de verde más bajo, será convertido al blanco, si no, en negro.




```
for pixel in getPixels(p):
   r, g, b = getRGB(pixel)
   if r > 200 and g < 100:
      setRGB(pixel, (255, 255, 255))
   else:
      setRGB(pixel, (0, 0, 0))
```

Aquí se muestra la imagen resultante. Como pueden ver, con solo seleccionar los valores apropiados de colores individuales pueden filtrar las regiones indeseadas y focalizar únicamente en los colores que les interesan. Aunque la pelota entera no ha sido filtrada, es suficiente para usar la imagen para identificar su ubicación. Al ajustar los valores de umbral de RGB pueden refinar más este filtro. Se requerirá más procesamiento para identificar la ubicación de la pelota en el campo visual (que es la imagen).

Estamos interesados en localizar la pelota a la izquierda, derecha o en el centro de la imagen. Una forma de lograrlo es ubicar la coordenada x- en el punto central de la pelota. Una forma de hacerlo es tomar el promedio de las ubicaciones x- de todos los píxeles blancos de la imagen:

```
def locateBall(picture):
    tot_x = 0
    count = 0
    for pixel in getPixels(p):
        r, g, b = getRGB(pixel)
        if r > 200 and g < 100:
            tot_x = tot_x + getX(pixel)
            count = count + 1
    return tot_x/count
```

De forma alternativa, también pueden promediar las ubicaciones x- de todos los píxeles rosas sin usar el filtro. En la imagen mostrada, el promedio de valor de x- devuelto era 123. Dado que la imagen tiene 256 píxeles de ancho, 123 es casi justo en el centro. Abajo, mostramos las ubicaciones obtenidas en cada una de las imágenes mostradas.

ubicación = 41

ubicación = 123

ubicación = 215

Dado que el ancho de la imagen es 256 píxeles, pueden dividir el campo visual en tres regiones: izquierda (0..85), centro (86..170), y derecha (171, 255). El control del robot para seguir a la pelota puede definirse como se muestra abajo:

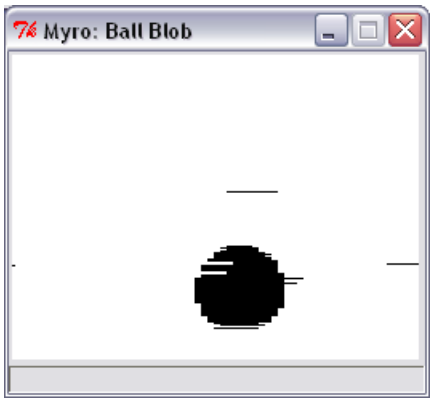
```
while timeTemaining(T):
    # take picture and locate the ball
    pic = takePicture()
    ballLocation = locateBall(pic)

    if ballLocation <= 85:
        turnLeft(cruiseSpeed)
    elif ballLocation <= 170:
        forward(cruiseSpeed)
    else:
        turnRight(cruiseSpeed)
```

Realizar la siguiente actividad: Implementar el programa de arriba para reconocer un objeto dado en la foto. El objeto puede ser una caja, una pelota, un cesto de basura, etc. Experimenten con distintos objetos de color. ¿Pueden usar estas técnicas para reconocer cualquier objeto? ¿digamos un animal? ¿una persona? etc.

Blobs

Estableciendo un blob en la pelota.



Resultado de `takePicture("blob")`

El tipo de filtro de umbral usado arriba es comúnmente llamado *blob filtering (filtrado blob)*. Todos los píxeles que satisfacen las propiedades del color seleccionado se identifican y el resto se elimina. Esta es una operación tan común en el procesamiento de imágenes que muchas cámaras y hardware de cámaras integran el filtrado blob como primitivo. El Scribbler también tiene esa capacidad. Usando esto, no necesitan preocuparse por identificar valores de color específicos para el umbral de filtro. Todo lo que deben hacer es tomar la foto y mostrarla. Luego, usando el mouse, definir la región rectangular que contiene su blob deseado. Hagan esto, saquen una foto, visualícenla y luego hagan click en el mouse en la esquina superior izquierda de la pelota en la imagen y arrástrenla a la esquina inferior derecha antes de soltarla. Deben asegurarse de que estén conectados al robot, porque la región blob seleccionada se transfiere de vuelta al robot. Una vez que esto esté definido, podrán usar el filtro blob predefinido sacando fotos y usando el comando:

```
pic = takePicture("blob")
```

Ni bien la foto es tomada, el detector blob de la cámara entra en funcionamiento y devuelve la imagen resultante (mostrada arriba). Ahora pueden usar esta imagen como antes para controlar el robot. Tendrán un promedio de la ubicación de todos los píxeles negros.

Si no, pueden usar la función `getBlob()` de la siguiente manera:

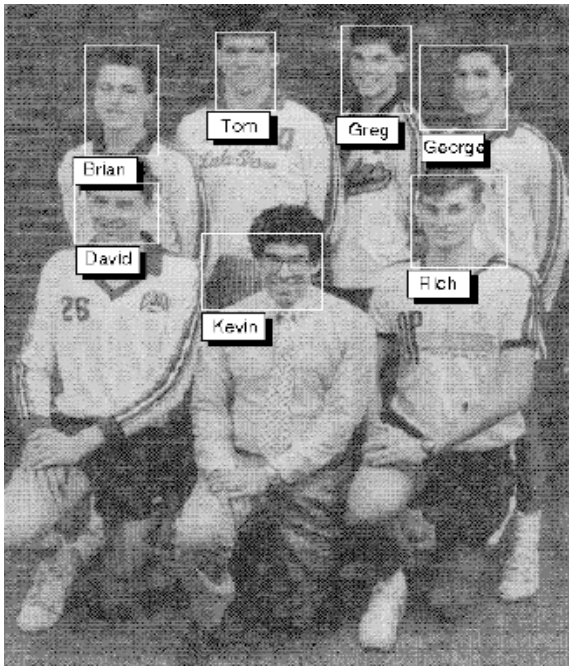
```
onPixels, avgX, avgY = getBlob()
```

La función devuelve dos valores: El número de los píxeles “encendidos” (por ejemplo, los píxeles en la imagen que estaban en el blob detectado), y la ubicación promedio x- e y- de los píxeles blob. Esta es una forma mucho mejor y más rápida de realizar reconocimiento de blob y notarán que el robot rastrea la pelota mucho mejor.

Realizar la siguiente actividad: Rehacer el ejemplo anterior de la pelota y aplique las características de blob itroducidas aquí.

Cálculos futuros sobre una imagen pueden resultar en la identificación de bordes de objetos. Esto se llama ***detección de borde***. Estos pueden ser usados para hacer reconocimiento de objetos: ¿Hay un auto en la foto? ¿Un rostro? etc. El

reconocimiento de objetos requiere de la habilidad para identificar características que son típicas de un objeto. El dominio del reconocimiento de imágenes y visión computacional es rica en técnicas y aplicaciones como éstas. A la derecha, mostramos los resultados de reconocer rostros e identificar individuos en la foto. A medida que los investigadores avanzan en estas técnicas y a medida que se incrementan las capacidades computacionales de las cámaras digitales, estamos comenzando a ver muchas de estas incorporadas incluso a las cámaras digitales más básicas. Tales técnicas pueden ser usadas para focalizar automáticamente en el sujeto principal y también para ajustar la exposición para obtener una foto perfecta, aún para fotógrafos amateurs.



Resumen

En este capítulo, han visto varias facilidades Myro que pueden usar usadas para la creación, el procesamiento y la comprensión de imágenes. La mayoría de las técnicas de creación, procesamiento y comprensión de imágenes involucran la manipulación de la imagen en el nivel del píxel. Sin embargo, muchos efectos y aplicaciones interesantes pueden ser desarrollados usando estas ideas simples. Muchas de estas características se encuentran cada vez más en dispositivos cotidianos, como cámaras digitales, dispositivos de identificación biométrica, y aun sistemas de distribución automática de correo. Combinadas con comportamientos del robot estas facilidades pueden ser usadas para modelar comportamientos más inteligentes.

Revisión Myro

`getHeight(<picture>)`
`getWidth(<picture>)`
Devuelve el alto y el ancho del objeto `<picture>` (en píxeles).

`getPixel(<picture>, x, y)`
Devuelve el pixel en `x,y` en `<picture>`.

`getPixels(<picture>)`
Cuando es usado en un loop, devuelve un píxel a la vez de `<picture>`.

`getRGB(pixel)`
`getRed(<pixel>)`
`getGreen(<pixel>)`

`getBlue(<pixel>)`
Devuelve los valores RGB del `<pixel>`.

`makeColor(<red>, <green>, <blue>)`
Crea un objeto color con los valores `<red>`, `<green>`, y `<blue>` dados (todos están en el rango [0..255]).

`makePicture(<file>)`
`makePicture(<width>, <height>)`
`makePicture(<width>, <height>, <color>)`
Crea un objeto imagen ya sea leyendo una imagen de un `<file>`, o del `<width>` y `<height>` dados. Si no se especifica `<color>`, la imagen creada tiene un fondo blanco.

`pickAColor()`
Crea una ventana de diálogo interactiva para seleccionar visualmente un color. Devuelve el objeto color correspondiente al color seleccionado.

`pickAFile()`
Crea una ventana de diálogo interactiva que le permite al usuario navegar a una carpeta y seleccionar un archivo para abrir. Nota: No puede usado para crear archivos nuevos.

`repaint()`
`repaint(<picture>)`
Actualiza la <picture> mostrada.

`savePicture(<picture>, <file>)`
`savePicture(<picture list>, <gif file>)`
Guarda `<picture>` en el arvhico especificado (un GIF o JPEG como determine la extensión del `<file>`: `.gif` o `.jpg`). `<picture list>` se guarda en un archivo GIF animado.

`setColor(<pixel>, <color>)`
`setRed(<pixel>, <value>)`
`setGreen(<pixel), <value>)`
`setBlue(<Pixel>, <value>)`
Establece el color de `<pixel>` al `<color>` o `<value>` especificados.

`show(<picture>)`
`show(<picture>, <name>)`
Muestra la `<picture>` en la pantalla en una ventana llamada `<name>` (string).

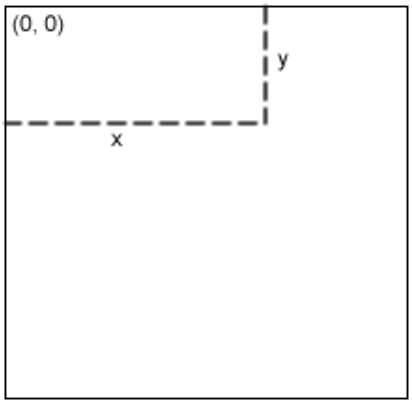
`takePicture()`
`takePicture("gray")`
`takePicture("blob")`
Saca una foto con la cámara del Scribbler. Por defecto es una foto color, o en escala de grises (`"gray"`), o una imagen filtrada basada en el blob (`"blob"`). Ver texto del capítulo para ejemplos.

Revisión Python

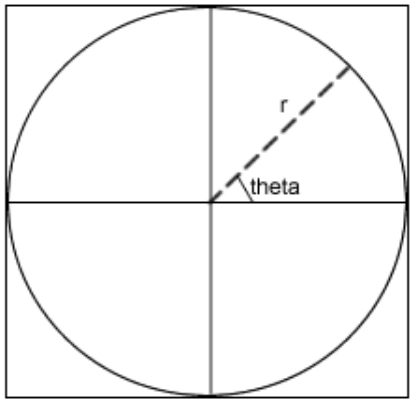
No se introdujeron características de Python en este capítulo.

Ejercicios

1. En lugar de visualizar cada píxel en una imagen como valores x- e y- en el sistema cartesiano de coordenadas, imaginen que está en un sistema de coordenadas polar con su origen en el centro de la imagen (Ver la imagen abajo).



Cartesian Coordinates



Polar Coordinates

Escriban una función llamada polar(x, y) que devuelve las coordenadas polares (r, theta) de un punto dado. Usando esta función escriban una transformación de la siguiente manera:

```
for x in range (1, W):
   for y in range(1, H):
      r, theta = polar(x, y)
      setPixel(myPic, x, y,
               gray((((r+theta)%16)-8)*MAX/16+MAX/2))
```

Crear una imagen de 400x400 usando la transformación de arriba. Comiencen con una imagen negra.

2. Basados en el ejercicio de arriba, prueben transformar una imagen (como un retrato). Por ejemplo, usen la regla: nuevo x, y es una copia del $\sqrt{r * \frac{W}{2}}$, *theta* de la imagen original.

3. Escriban una transformación que gire la imagen en el sentido de las agujas del reloj (o contra las agujas del reloj) en 90 grados. Pruebenlo en una imagen cuadrada a elección.

4. Escriban una transformación de imagen que promedie dos imágenes (del mismo tamaño) en una sola imagen.

5. Escriban un programa que una dos o más imágenes juntas para crear un panorama.

6. Usen las técnicas de detección de blob descriptas en este capítulo para reconocer latas de Coca Cola y Pepsi. Escriban un programa para salir a la caza de una Coca o Pepsi.

7. Pueden expandir el ejercicio de arriba para reconocer pequeñas figuritas de juguete de personajes de dibujos animados de Disney como Mickey, Minnie, Daisy, Goofy, Pluto, y Donald?