JiScript User's Guide

Java has several features that make it attractive for mathematical illustration. First, it was designed with powerful graphical capabilities that make the production of high-quality illustrations possible. In addition, Java enables us to create dynamic illustrations, such as animations, that may be used to demonstrate or explore a mathematical idea more effectively. Finally, Java programs may be easily distributed over the Internet as applets running in web browsers.

This guide describes `JiScript`, a Python module that allows programmers to write Python code to create illustrations easily in Java. In particular, `JiScript` uses the same set of graphics commands as Bill Casselman's `PiScript`, a Python module for creating figures in PostScript. My intent is to allow programmers to create illustrations in either PostScript or Java with a single graphics command set and without having to master much Java.

It is, at least in theory, possible to accomplish anything with `JiScript` that can be accomplished with pure Java. This may not be readily apparent, however, as my intention is to sweep Java under the rug as much as possible and give programmers the pleasure and efficiency of developing purely in Python. My experience is that `JiScript` programs may be developed much more quickly than the equivalent programs in Java. I trust that experienced Java users can determine how to add capabilities they may find missing.

`JiScript` programs will be run with `Jython`, an implementation of Python written in Java and freely available on the Internet. In essence, we will program in Python but have access to all the Java classes. Instructions for installing both `JiScript` and `Jython` are given in an appendix to this guide.

The `JiScript` home page is

`http://merganser.math.gvsu.edu/david/jiscript`

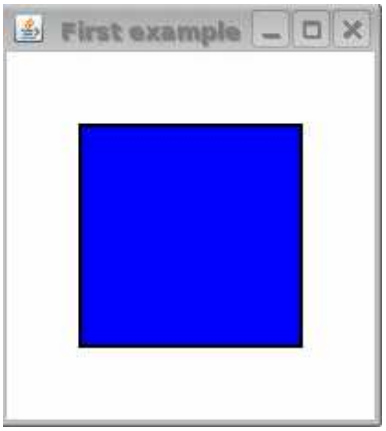
while the PiScript home page is

`http://www.math.ubc.ca/~cass/piscript.`

This manual provides an introduction to using `JiScript` and attempts to explain its novel features assuming that the reader is familiar with `PiScript`. An appendix provides a list of all the graphics commands and how they may be used.

1: Two simple examples

Let's begin with a simple example that causes the following window to appear when run.



Here is the program that does it. Place this code in a file `Box.jy` and run it with the command `jython Box.jy`.

```
from jiscript.JiModule import *

def draw():
    beginpage()
    newpath()
    box(40, 40, 120, 120)
    fill(0, 0, 1)
    setlinewidth(2)
    stroke()
    endpage()

openframe(200, 200, draw, 'First Example')
```

Quite a bit of this code should look familiar to `PiScript` users. In particular, the drawing commands are precisely those that would, when included in a `PiScript` program, create a PostScript file displaying the blue box. Notice that the functions `init()` and `finish()` are not needed. One other important difference is that the drawing commands are in a function, here called `draw`. More about this will said momentarily.

The first two lines of this program

```
from jiscript.JiModule import *
```

import the `JiModule` module, which is required to make sense of the drawing commands. As we'll see in this manual, `JiScript` comes in two flavors. We begin by introducing the `JiModule` module, which will demonstrate most of the basic concepts behind `JiScript`. Later, the `JiScript` module, and how it extends the capabilities of `JiModule`, will be described.

The final line of the program

```
openframe(200, 200, draw, 'First Example')
```

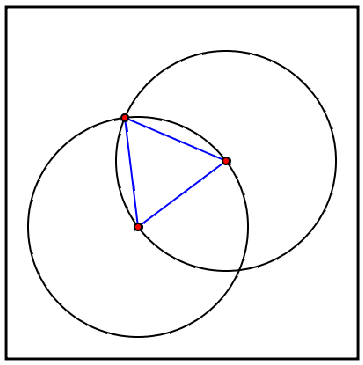
causes the window to be displayed on the screen. More specifically, you will notice two components in the window: a "frame," consisting of the title bar and the gray border, and a "panel," the white region on which the blue rectangle is drawn. You may wish to think of this like an artist's canvas inside a frame.

What does `openframe` do? First, it specifies the width and height, in pixels, of the frame and places a panel inside the frame. The third argument, `draw`, gives the name of the function containing the drawing commands for drawing on the panel. Finally, an optional title is given for the frame's title bar.

You will generally want `openframe` to be the last line of your program as anything that comes after will likely be ignored by `JiScript`.

When drawing on a panel, the default coordinate system has the origin at the lower left corner of the panel with the units in the horizontal and vertical directions in pixels. Just as in `PiScript`, this coordinate system may be modified using, say, `translate`, `rotate`, and the like.

This next example, which gives an illustration of the first proposition in *The Elements*, illustrates how more complex figures can be created from the same template. Aside from using more graphics commands, this program follows the same outline.



```
from jiscript.JiModule import *
import math

def draw():
    beginpage()

    gsave()
    center()                    # set up coordinate system
    scale(50)
    scalelinewidth(2)

    p0 = [-1, -1]               # define endpoints
    p1 = [1, 0.5]
    dx = p1[0] - p0[0]
    dy = p1[1] - p0[1]
    dist = math.sqrt(dx*dx + dy*dy)

    newpath()                   # draw circles
    circle(p0, dist)
    stroke()

    newpath()
    circle(p1, dist)
    stroke()

    gsave()                     # set up coordinate system
    newpath()                   # to draw triangle
    translate(p0)
    setrad()
    rotate(math.atan2(dy, dx))
```

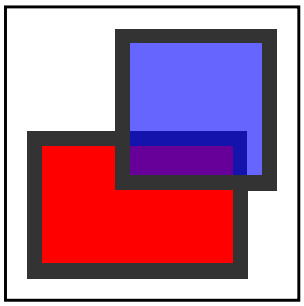
```
    gsave()
    moveto(0, 0)                # draw triangle
    for i in range(3):
        lineto(dist, 0)
        translate(dist, 0)
        rotate(2.0*math.pi/3)
    closepath()
    stroke([0, 0, 1])
    grestore()

    for i in range(3):          # draw points
        newpath()
        circle(0, 0, 0.08)
        fill([1, 0, 0])
        stroke()
        translate(dist, 0)
        rotate(2.0*math.pi/3)

    grestore()
    grestore()
    endpage()
openframe(400, 400, draw, 'Euclid I')
```

2: Some novel graphics features

Since Java provides a few capabilities that are not available in PostScript, `JiScript` provides some new features, which we'll meet in the next few sections.

Transparency: In PostScript, and hence `PiScript`, colors are opaque. This means that painting with a color will cover anything underneath. Java, however, allows us to define colors with a specified transparency by adding a fourth component to the color.

Consider the following figure



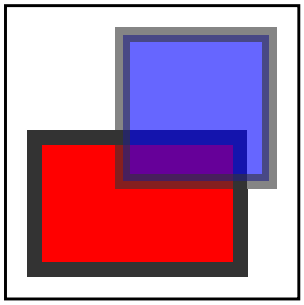
which is created by this program.

```
from jiscript.JiModule import *

def draw():
    beginpage()
```

```
    setlinewidth(10)
    newpath()
    box(20, 20, 140, 90)
    fill(1,0,0)
    stroke(0.2)

    newpath()
    box(80, 80, 100, 100)
    fill(0, 0, 1, 0.6)          # <---- 60\% opaque blue
    stroke(0.2)

    endpage()

openframe(200, 200, draw, 'Transparency')
```

Think of this fourth component, usually called the color's *alpha* value, as specifying the color's opacity. If this component is 0, the color is entirely transparent, while if it's 1, it is entirely opaque.

A second way of working with transparent colors, which may be profitably used, is through the `setcomposite()` function. For instance, `setcomposite(0.6)` has the effect of multiplying a color's alpha value by 0.6. The following illustrates this effect:



```
from jiscript.JiModule import *

def draw():
    beginpage()

    setlinewidth(10)
    newpath()
    box(20, 20, 140, 90)
    fill(1,0,0)
    stroke(0.2)

    setcomposite(0.6)           # <---- setcomposite here
    newpath()
    box(80, 80, 100, 100)
    fill(0, 0, 1)               # this blue will have alpha = 0.6
    stroke(0.2)                 # as will this gray

    endpage()

openframe(200, 200, draw, 'Transparency')
```

You may wish to take a moment to explain all of the features you see in the illustration above. The alpha value defined by `setcomposite` becomes part of the graphics state and may be manipulated in the usual way with `gsave` and `grestore`.

Writing image files: We may write PNG and JPEG image files using the `writePNG()` and `writeJPEG()` functions. After defining the drawing method, you may simply use

```
writePNG(300, 300, draw, ‘‘image.png’’, 1.5)
```

to create a PNG file, called `image.png`, containing the illustration. The first arguments give the dimensions of a panel on which we'll draw while the final argument is a scaling factor. The file `image.png` will therefore consist of a 450×450 grid of pixels. If you would prefer to see the illustration in addition to write it to a file, you may use

```
openframe(300, 300, draw)
writePNG(‘‘image.png’’, 1.5)
```

In the same way, JPEG files may be written using

```
writeJPEG(300, 300, draw, ‘‘image.jpg’’, 1.5, 0.8)
```

where the final argument is a *quality* parameter, which controls how much the JPEG algorithm compresses the image. This quality parameter varies between 0, small file size but poor quality, and 1, large file size but high quality.

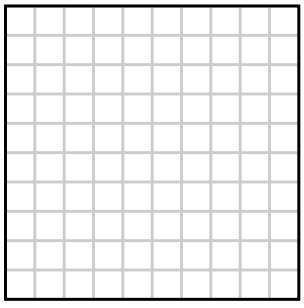
We may also use

```
openframe(300, 300, draw)
writeJPEG(‘‘image.jpg’’, 1.5, 0.8)
```

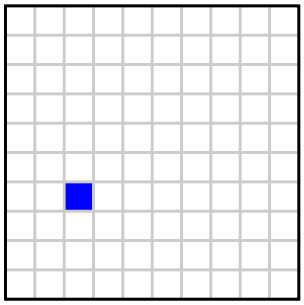
3: Drawing with pixels

Since JiScript illustrations are typically viewed on a computer screen, some important issues that arise as these will necessarily be relatively low-resolution images. By comparison, the PostScript image files produced by PiScript may be rendered to the resolution of the output device.

Our image is rendered on a grid of pixels.

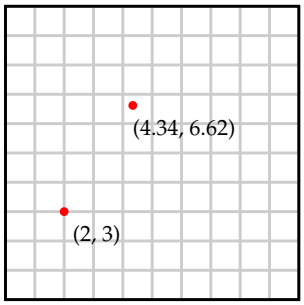


When an image is created, the region occupied by a pixel is colored a uniform color. This means that pixel coordinates are discrete: our actions may only have the effect of coloring, say, pixel $(2, 3)$.

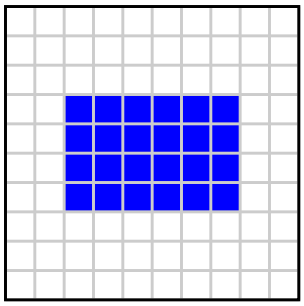


Notice here that we will refer to pixels as ordered pairs of integers. As we create mathematical illustrations, however, we will frequently refer to points whose coordinates are floating point numbers rather than integers. For example, we may wish to graph $y = \cos x$, which may involve plotting the point $(\pi, -1)$. The means by which floating point coordinates are converted into integer pixel coordinates is worth some investigation.

Let us first be more explicit about the default coordinate system. Earlier, we mentioned that the origin of the coordinate system is the lower left hand corner of the image and the units on the horizontal and vertical axes are pixels. To be more precise, the integer coordinates are located *between* pixels. Consider, for instance, the points shown below.

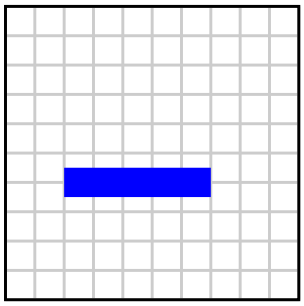


This choice makes sense when drawing rectangles whose vertices have integer coordinates. For example, shown below is the result of filling the rectangle whose lower left corner is at $(2, 3)$ and whose width is 6 and height is 4.

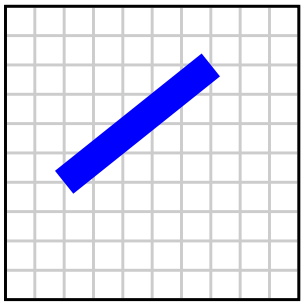


The pixels that are lighted cause the region inside the rectangle to be illuminated.

The situation is more complicated, however, when we wish to draw lines. For example, suppose that we wish to stroke the horizontal line from $(3, 4)$ to $(7, 4)$ with a line width of one pixel. To do this, we would ideally see this:

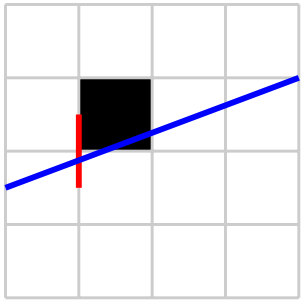


However, this cannot be represented by pixels; stroking the line requires that we fill a rectangular region that covers half the pixels below the line and half above the line. This issue also arises when we wish to stroke a line that is not horizontal.

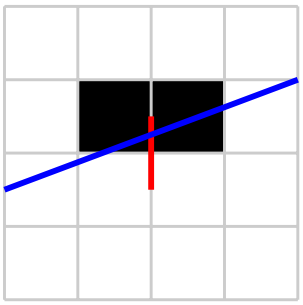


How do we determine which pixels should be lighted? To answer this question, we will now describe Bresenham's algorithm, which is commonly used when rendering a line in pixels. To begin, suppose that the endpoints of the line we wish to draw have integer coordinates (x_0, y_0) and (x_1, y_1). For convenience, we will assume that $x_0 < x_1$. Moreover, we will assume that the slope $m = (y_1 - y_0)/(x_1 - x_0)$ satisfies $0 \leq m \leq 1$. It should be clear how to modify the algorithm should the slope not lie in this range.

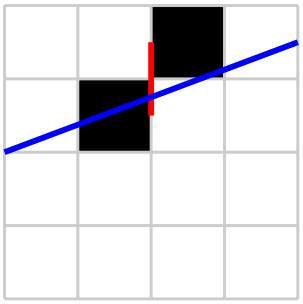
Our strategy for determining which pixels to color is this: the pixel $(\hat{x}, \hat{y})$ is colored if the line intersects the segment where $x = \hat{x}$ and $\hat{y} - 1/2 \leq y < \hat{y} + 1/2$.



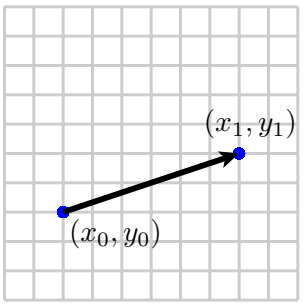
To begin the algorithm, we color the pixel (x_0, y_0) and move to the right. At every step of the algorithm, we will assume that we have just colored the pixel $(\hat{x}, \hat{y})$. We will now move to the next column of pixels and determine which pixel to color Since the slope is assumed to between 0 and $1/2$, the only possible pixels to color next are $(\hat{x}+1, \hat{y})$ or $(\hat{x}+1, \hat{y}+1)$. To determine which pixel is colored, we consider the point $(\hat{x} + 1, \hat{y} + 1/2)$. If this point is above the line, we will color $(\hat{x} + 1, \hat{y})$.



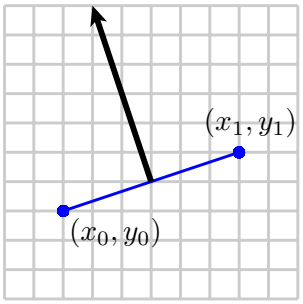
If this point is on or below the line, we will color $(\hat{x}+1, \hat{y}+1)$.



We will now develop an efficient way to determine which side of the line a given point lies on. Denote the vector describing the difference of the endpoints as $\langle \Delta x, \Delta y \rangle = \langle x_1 - x_0, y_1 - y_0 \rangle$. Because of our assumptions, we have $\Delta y \geq 0$ and $\Delta x > 0$.



Rotating this vector by $90°$ produces a vector $\langle -\Delta y, \Delta x \rangle$ that is normal to the line.



For reasons that will be clear momentarily, we will double this vector to obtain the normal $\mathbf{n} = -\langle 2\Delta y, 2\Delta x \rangle$. Remember that the line may be described as the zero set of the function

$$h(x, y) = \mathbf{n} \cdot \langle x, y \rangle - \mathbf{n} \cdot \langle x_0, y_0 \rangle$$
$$h(x, y) = -2\Delta y\, x + 2\Delta x\, y + C$$

where C is a constant. Moreover, we have

$$h(x,y)\begin{cases} > 0 & \text{if } (x,y) \text{ lies above the line} \\ = 0 & \text{if } (x,y) \text{ lies on the line} \\ < 0 & \text{if } (x,y) \text{ lies below the line.} \end{cases}$$

Notice also that

$$h(x+s,y) - h(x,y) = -2\Delta y\, s$$
$$h(x,y+r) - h(x,y) = 2\Delta x\, r$$

Using our function $h(x,y)$, it is a simple matter to determine whether our test point lies above or below the line. At every step of the algorithm, we shall let $h(\hat{x},\hat{y}+1/2) = r$. Initially, $(\hat{x},\hat{y}) = (x_0,y_0)$, which lies on the line. We therefore have

$$r = h(\hat{x},\hat{y}+1/2) = h(x_0,y_0+1/2) = h(x_0,y+0) + \Delta x = \Delta x.$$

When we step horizontally to our test point, we have

$$h(\hat{x}+1,\hat{y}+1/2) = h(\hat{x},\hat{y}+1/2) - 2\Delta y = r - 2\Delta y.$$

Therefore, we set $r \leftarrow r - 2\Delta y$.

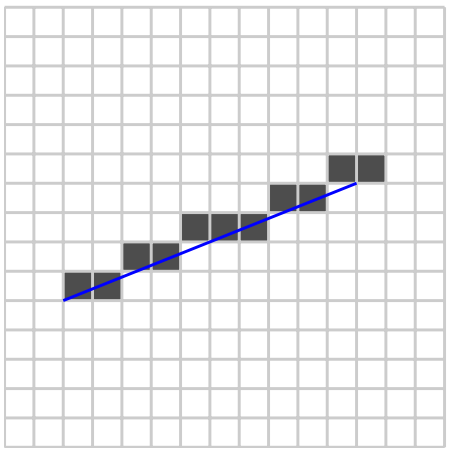
If $r = h(\hat{x}+1,\hat{y}+1/2) > 0$, the test point lies below the line. This means that we next color the pixel $(\hat{x}+1,\hat{y})$. We therefore set $\hat{x} \leftarrow \hat{x}+1$ and proceed.

However, if $r = h(\hat{x}+1,\hat{y}+1/2) \geq 0$, the test point lies above or on the line so the next pixel we color is $(\hat{x}+1,\hat{y}+1)$. In this case, we set $\hat{x} \leftarrow \hat{x}+1$, $\hat{y} \leftarrow \hat{y}+1$, $r \leftarrow r+2\Delta x$.

The algorithm now reads like this:

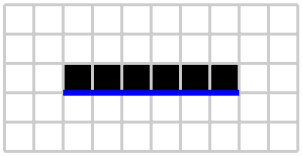
initialize $\hat{x} \leftarrow x_0$, $\hat{y} \leftarrow y_0$, $r \leftarrow \Delta y$
color pixel $(\hat{x},\hat{y})$
while $\hat{x} < x_1$
  $\hat{x} \leftarrow \hat{x}+1$
  $r \leftarrow r - 2\Delta y$
  if $r \geq 0$, **then** $\hat{y} \leftarrow \hat{y}+1$, $r \leftarrow r+2\Delta x$
  color pixel $(\hat{x},\hat{y})$

Here is a result of the algorithm.



Notice that each step of the algorithm may be performed using integer arithmetic, which causes the algorithm to be very efficient. This explains why obtained the normal vector $\mathbf{n}$ by scaling the vector $\langle -\Delta y, \Delta x \rangle$ by a factor of 2. This efficiency is important since lines are drawn so frequently.

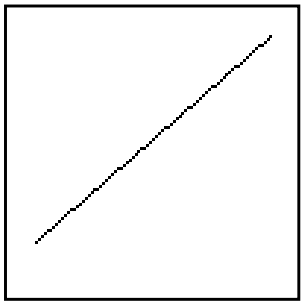
We began this discussion by asking how a horizontal draw is drawn since it seems to overlap half a pixel above and below the line. Bresenham's algorithm provides the answer: a horizontal line will be rendered by coloring the pixels above the line:



Finally, here is one small warning about the images in this section. The coordinate system Java uses places the origin at the upper left of the frame and the vertical coordinate increases as we move down the frame. JiScript modifies this coordinate system to make it more natural for mathematicians to use and for consistency with PiScript. Therefore, the lines drawn in this section, as seen by JiScript, have a positive slope but, as seen by Java, which does the actual rendering, these lines have a negative slope. The illustrations for the discussion of Bresenham's algorithm were created with PiScript to illustrate the algorithm. If drawn with JiScript, they would differ by one pixel in the vertical direction.

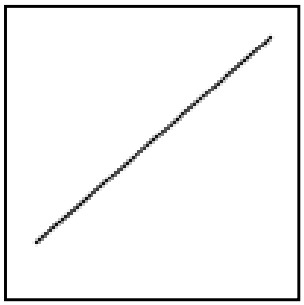
4: Antialiasing

In the last section, we looked at lots of pictures of lines while we were learning how to draw them. Those images were greatly magnified so that individual pixels were apparent. Let's look at a line draw at a more reasonable scale. Here, one pixel corresponds to one PostScript point, a scale roughly equal to that of an image viewed on a computer screen.



In spite of the fact that individual pixels are no longer visible, the line still appears jagged. (Now that you're familiar with Bresenham's algorithm, you may have a deeper appreciation for the pattern in the jaggedness.)

Antialiasing is a technique that helps create the illusion of smoothness. Here is what we see when we draw the same line using antialiasing.

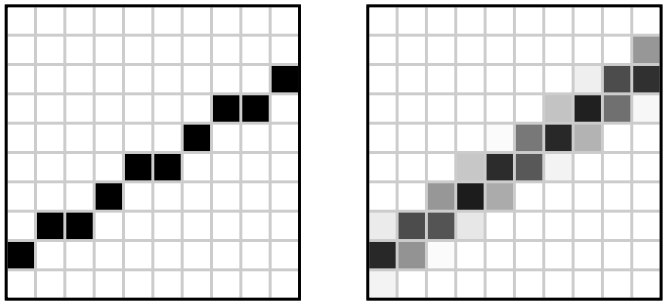


In JiScript, antialiasing can be switched on and off with either

```
antialiasing(True)
antialiasing(False)
```

By default, antialiasing is switched on in JiScript; this means that if you were to draw the line above, you would see the second, smoother version.

To understand how antialiasing works, let's take a closer look at a line drawn once without antialiasing and again with antialiasing.



`antialiasing(False)` `antialiasing(True)`

Notice that some of the pixels left uncolored when the line is drawn without antialiasing appear to be colored in various shades of gray. The amount of gray used for a particular pixel is determined by how far the pixel is away from the line. When seen at a more reasonable resolution, the jagged edges are softened and the eye believes that the line is relatively smooth.

In the previous paragraph, I said that the pixels *appear* to be colored gray. Actually, the color assigned to a pixel in this example is black; the gray that we see gray is produced by modifying the alpha value (the opacity) of the color assigned to the pixel. This sometimes produces visible effects when all the elements in a figure are rendered with antialiasing since colors that are underneath others may show through.

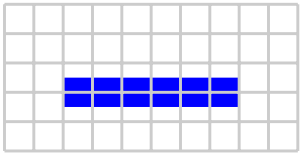
Due to the computer power available in the mid 1990's, Pixar created the original *Toy Story* at a resolution about equal to that of a standard laptop of today. When shown in a movie theater, a single pixel occupied a region about $1/4$" square. While this seems like a surprisingly low resolution, Pixar was confident that their antialiasing techniques would produce attractive images at that resolution.

Most often, images will be improved using antialiasing. However, we'll see a few examples where there is some benefit to turning it off when drawing a few elements in a figure. Also, more computation is required to render a figure with antialiasing. This is usually not an issue for the kinds of images

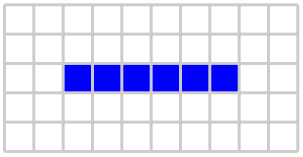
we will create here, but it may be in an animation in which individual frames require the rendering of many elements.

5: Purestroke

Now that we're familiar with antialiasing, there is a final issue raised by the low resolution of our images. Imagine that we wish to draw a horizontal line one pixel wide and that the coordinates of the endpoints are integers. Ideally, we are thinking of a line like this:

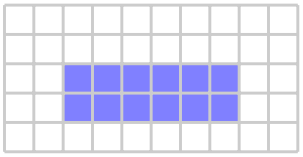


We can now imagine two ways to draw this line, neither of which is completely satisfying. First, Bresenham's algorithm would produce the following.



This gives us a horizontal line one pixel wide as we requested. However, the line has been shifted up by half a pixel. If we draw lines in this way, the upper and lower edges of a rectangle will be handled inconsistently: the lower edge will lie inside the rectangle while the upper edge lies outside. In addition, when creating a mathematical illustration, the accuracy of the coordinates we create is often important; we may not want the rendering algorithm to move them around after we have computed them.

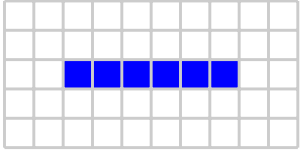
Antialiasing gives us a second option. Since our ideal line covers half-pixels in two rows, we may color both rows of pixels with an alpha of 0.5.



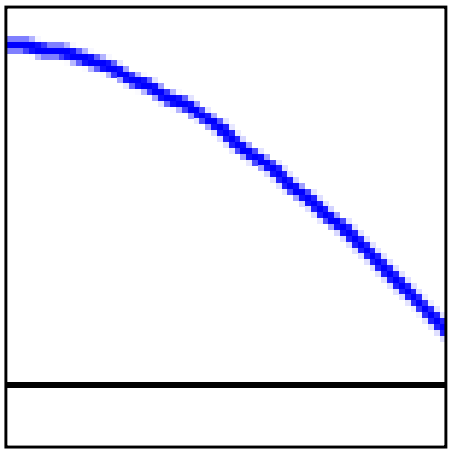
This has the benefit of giving us the line we requested, but its thickness has doubled and it appears lighter.

Both options have advantages and disadvantages, and both are available to us.

Java's line rendering algorithm takes the first approach and shifts the coordinates of the endpoints so they lie in the center of pixels. In other words, whether we are using antialiasing or not, we will see our horizontal line shifted up by half a pixel.



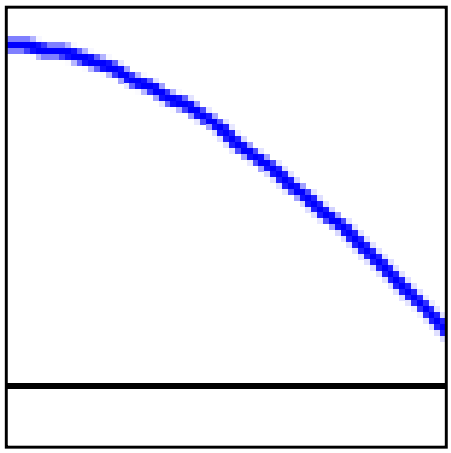
The benefit is that horizontal and vertical lines are rendered with the correct thickness and color. However, problems arise when we are drawing mathematical objects. For instance, the graph of a function is usually drawn by connecting a sequence of points on the graph by straight lines. If the endpoints of these straight lines are shifted by the rendering algorithm, the graph may not appear smooth. The following graph, drawn with antialiasing, illustrates the point.



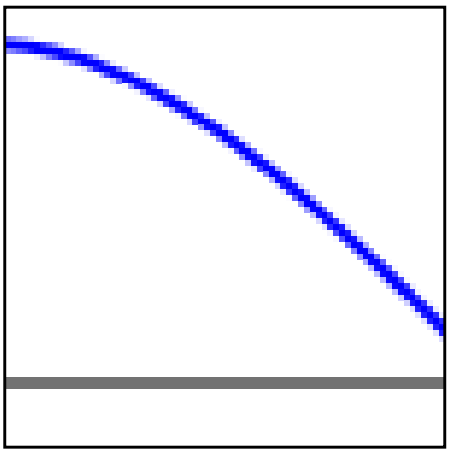
Fortunately, we may change this behavior using the `purestroke` option.

```
purestroke(True)
purestroke(False)
```

If we set `purestroke` to `True`, then the endpoint of a line segment is *not* modified to lie in the center of a pixel; the coordinates that we specify are the ones used. Below on the right, the same graph is rendered in this way. The effect is subtle but real: the graph now appears smoother.



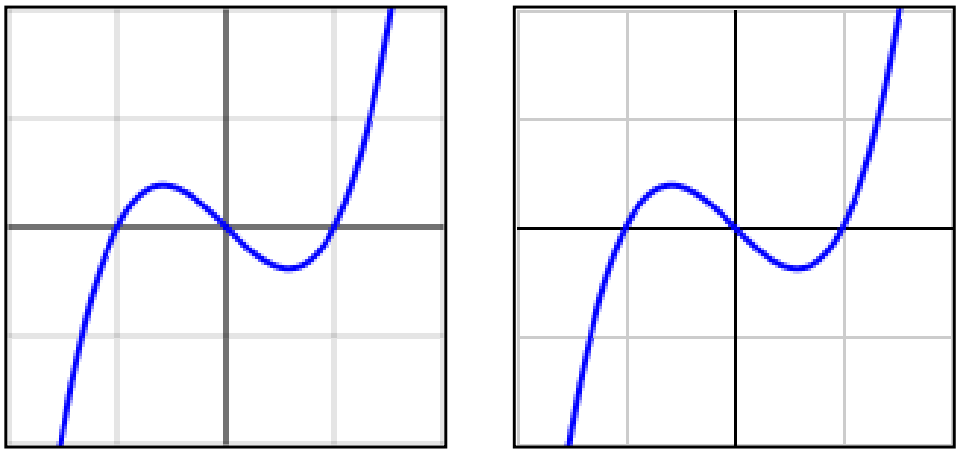
`purestroke(False)`



`purestroke(True)`

Notice that when `purestroke` is set to `True`, the axis appears lighter and wider since the coordinates of the axes now lie between pixels implying that the ideal axis covers two rows of half-pixels. Since `antialiasing` is also set to `True`, the line is rendered by coloring two rows of pixels black with an alpha value of 0.5

By default, JiScript sets `purestroke` to `True`. As the previous example shows, you may wish to change its value within an illustration so that horizontal and vertical lines appear sharper. Shown below is the graph along with a grid and axes. On the left, `purestroke` is always set to `True`. On the right, `purestroke` is `False` when the grid and axes are rendered, but switched to `True` for the graph so that it appears smooth.



Here is the code for the right illustration.

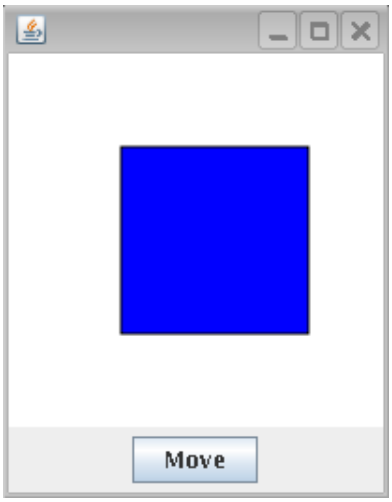
```
from jiscript.JiModule import *

def draw():
    beginpage()

    gsave()
    center()
    scale(75)

    #  set purestroke to False for the grid and axes
    purestroke(False)
    newpath()
    for i in range(-2, 3):
        moveto(i, -2)
        lineto(i, 2)
        moveto(-2, i)
        lineto(2, i)
    stroke(0.8)         # stroke the grid lines

    newpath()
    moveto(-2, 0)
    lineto(2, 0)
    moveto(0, -2)
    lineto(0, 2)
    stroke()            # stroke the axes
```

```
    #  set purestroke to True for the graph
    purestroke(True)
    newpath()
    setlinewidth(2)
    graph(lambda x: x**3 - x, -2, 2)
    stroke([0, 0, 1])   # stroke the graph

    grestore()
    endpage()

openframe([300, 300, draw])
```

6: Using buttons

One of Java's more attractive features is the ease with which interactive illustrations may be created and manipulated by viewers. In this section, we'll begin introducing `JiScript`'s interactive features by describing how buttons may be added to figures.

Let's return to our original, lowly box and add a button that, when pressed, moves the box to the right by 5 pixels.




```
from jiscript.JiModule import *

x = 40
def draw():
    beginpage()

    newpath()
    box(x, 40, 120, 120)
    fill(0, 0, 1)
    setlinewidth(2)
    stroke()

    endpage()
```

```
def move():
    global x
    x += 5
    refresh()
addbutton('Move', move)
openframe(200, 200, draw)
```

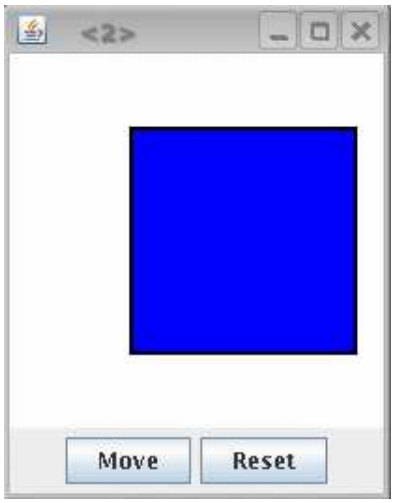
Much of this looks familiar: we import the `JiModule` module, define our drawing function `draw()`, and display the results with `openframe`.

There are, however, a few novel features. First, we introduce a variable, called `x`, which will hold the x-coordinate of the left edge of the box.

Also, we define a function `move`, which will be called when the button is pressed. Notice that the variable `x` needs to be declared `global` since it is modified in the function. Also, it is important to note the command `refresh()`, which causes the illustration to be redrawn. Forgetting this will likely cause you to stare at the screen and wonder why nothing changed.

Finally, we use the command `addbutton` to place a button in the bottom of the figure labeled with the text "Move" and that calls the function `move()` when pressed.

That's all there is to it. You may notice that the box walks off the right edge of the window after a few presses of the button. We will therefore add a second button that resets the box to its original configuration.

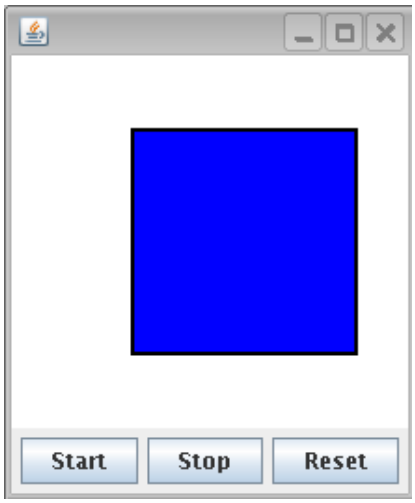


The program is the same except for the definition of a new function `reset` and a new button to call it.

```
def reset():
    global x
    x = 40
    refresh()
addbutton("Move", move)
addbutton("Reset", reset)
openframe(200, 200, draw)
```

7: Animations

If you're like me, you found that pressing the button got awfully tiring. Instead of manually pressing the button, we might like to make an animation in which the box continually moves to the right, one frame after another.



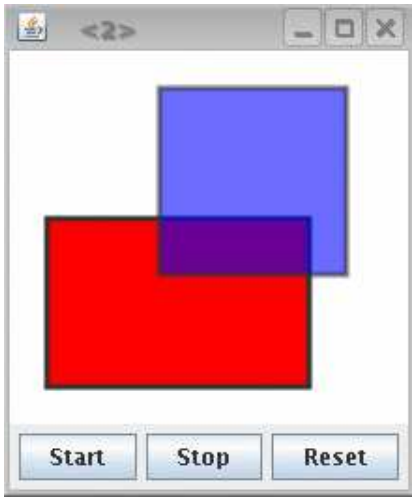

```
from jiscript.JiModule import *

x = 40
def draw():
    beginpage()

    newpath()
    box(x, 40, 120, 120)
    fill(0, 0, 1)
    setlinewidth(2)
    stroke()

    endpage()

def move():             # Define a function to generate each frame
    global x
    x += 1
    refresh()
settimer(25, move)      # Define a timer to create frames every 25 milliseconds

def begin():            # This function starts the timer
    starttimer()

def pause():            # This one pauses it
    stoptimer()

def reset():
    global x
    stoptimer()
    x = 40
    refresh()
```

```
addbutton('Start', begin)
addbutton('Stop', pause)
addbutton('Reset', reset)

openframe(200, 200, draw)
```

The new features here include the command `settimer(25, move)`, which creates a timer to call the function `move()` every 25 milliseconds once started. Observe that the `move()` function is the same function that was triggered by the button press in the previous program. You may think of the timer as pressing a button every 25 milliseconds.

We have also added three buttons to control the timer by starting, stopping and resetting it.

The following example demonstrates how to use transparency to fade an element into an illustration.



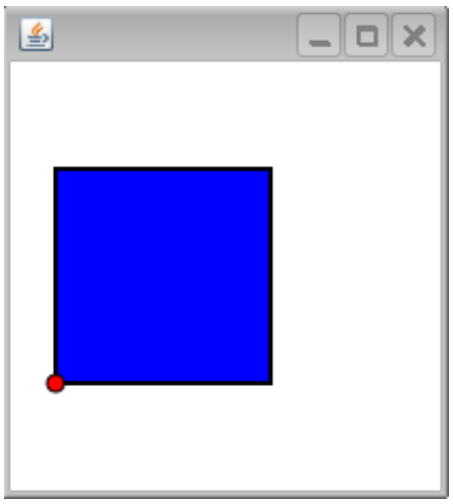

```
from jiscript.JiModule import *

alpha = 0
def draw():
    beginpage()

    setlinewidth(3)
    newpath()
    box(20, 20, 140, 90)
    fill(1,0,0)
    stroke(0.2)

    setcomposite(alpha)
    newpath()
    box(80, 80, 100, 100)
    fill(0, 0, 1)
    stroke(0.2)

    endpage()
```

```
def increment():
    global alpha
    alpha += 0.01
    if alpha >= 1.0:
        alpha = 1
        stoptimer()
    refresh()

settimer(25, increment)

def begin():
    starttimer()
def pause():
    stoptimer()
def reset():
    global alpha
    stoptimer()
    alpha = 0
    refresh()

addbutton("Start", begin)
addbutton("Stop", pause)
addbutton("Reset", reset)
openframe(200, 200, draw)
```

The variable `alpha`, which controls the opacity of the blue rectangle, is initially set to 0 so that blue rectangle is entirely transparent. When the animation is started, however, the value of `alpha` increases and the blue rectangle begins to appear until `alpha` reaches the value 1.

8: Moveable points

It is sometimes helpful to have points that the viewer may move around the figure while some elements in the figure respond appropriately. For instance, in the diagram below is a point that, when clicked and dragged, will move while remaining the lower left corner of the rectangle.



```
from jiscript.JiModule import *

def draw():
    beginpage()
    center()
    scale(50)
    newpath()
    box(movingpoint.x, -1, 2, 2)
    setlinewidth(2)
    fill(0, 0, 1)
    stroke()
    newpath()
    placemoveable(movingpoint)
    endpage()
def move(point, x, y):
    point.setpoint(x, -1)
movingpoint = Moveablepoint(-1, -1, move)
addmoveable(movingpoint)
openframe(200, 200, draw)
```

The use of `Moveablepoints` is a little trickier than the earlier interactive features we have seen so let's take this program apart rather carefully beginning near the end. With the command

```
movingpoint = Moveablepoint(-1, -1, move)
```

we define a `Moveablepoint` initially positioned at the point $(-1, -1)$. When a viewer clicks and drags this point, the function `move` will be called. We then register this `Moveablepoint` with the figure using

```
addmoveable(movingpoint)
```

This lets the figure know it should pay attention for attempts to move the point.

The `move` function is also a little different than what we've seen.

```
def move(point, x, y):
    point.setpoint(x, -1)
```

There are three arguments. The first, called `point`, will be the point that is being moved. We must include this argument as a convenience when dealing with many `Moveablepoints`, as we'll see in later examples. The final two arguments are the coordinates to which the viewer has requested the point be moved. Notice that we only use the `x` coordinate in setting the new coordinates of the point.

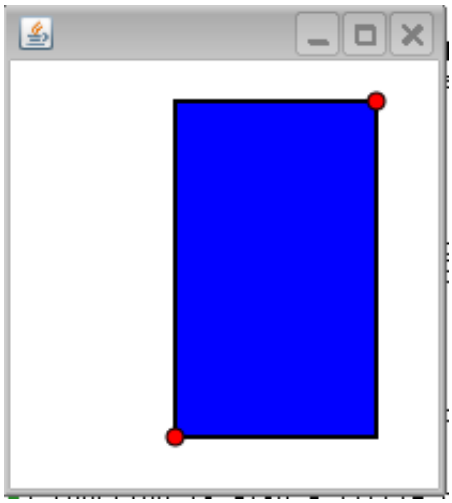
Finally, we specify when to draw the point using the `placemoveable` function.

```
def draw():
    ....
    newpath()
    placemoveable(movingpoint)
    endpage()
```

It is possible, of course, that we modify the coordinate transform many times during the `draw` method. The `Moveablepoint` is always placed, using its prescribed coordinates, in the current coordinate system. Furthermore, the coordinates `x, y` in the `move` function should be interpreted relative to the current coordinate transform when the point was placed.

To summarize, there are four steps necessary to use a `Moveablepoint`: define a function to move the point, define the point by specifying its initial coordinates and moving function, register the point with the `addmoveable()` function, and use `placemoveable` to draw the point.

Of course, we may have any number of `Moveablepoints` in an illustration.



```
from jiscript.JiModule import *

def draw():
    beginpage()

    center()
    scale(50)
    newpath()
    dx = upperright.x - lowerleft.x
    dy = upperright.y - lowerleft.y
    box(lowerleft.x, lowerleft.y, dx, dy)
    setlinewidth(2)
    fill(0, 0, 1)
    stroke()

    newpath()
    placemoveable(lowerleft)
    placemoveable(upperright)

    endpage()

def move(point, x, y):
    point.setpoint(x, y)
```

```
lowerleft = Moveablepoint(-1, -1, move)
upperright = Moveablepoint(1, 1, move)
addmoveable(upperright)
addmoveable(lowerleft)

openframe(200, 200, draw)
```

Notice how the same function `move` is used to move both points. When an attempt is made to move either point, this function is called with the argument `point` set to the point being moved.

`Moveablepoints` have several attributes that may be used to customize their appearance.

`setstyle`	``circle'',``square'',``diamond''
`setfillcolor`	a color
`setstrokecolor`	a color
`setsize`	a float, interpreted in pixels
`setstrokesize`	a float
`setfilled`	`True`, `False`
`setstroked`	`True`, `False`

For example, to have a blue `Moveablepoint` shaped like a diamond, you may use:

```
bluepoint = Moveablepoint(-1, 2, move)
bluepoint.setfillcolor( [0,0,1] )
bluepoint.setstyle(``diamond'')
```

9: Other mouse events

Suppose that you have drawn a collection of circles and you would like the viewer to be able to select a few of the circles by clicking in them. In this case, you would like to know when the viewer clicks the mouse and at what location.

You may respond to the following seven different types of mouse action:

enter:	the mouse enters the panel
exit:	the mouse exits the panel
click:	the mouse is quickly pressed and released
press:	the mouse is pressed
release:	the mouse is released after being pressed
drag:	the mouse is moved after being pressed
move:	the mouse is moved without being pressed

Using these events is relatively straightforward. To begin, we will define a function to be called when one of these events occurs. For instance, if we want to print the coordinates of the location where the mouse is clicked we may define:

```
def click(x, y):
    print x, y
```

Then register this function with

```
onclick(click)
```

When used in this way, the coordinates x and y are interpreted with respect to the default coordinate transform. If however, you wish to receive the coordinates with respect to a coordinate transform defined in the drawing process, you may define a `Mark`, like this.

```
mark = Mark()
```

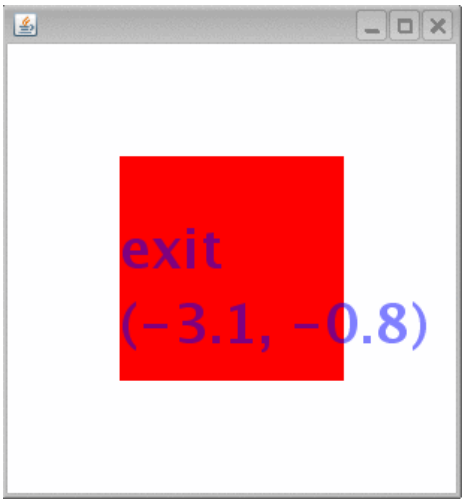
and use it when you register the function to be called when an event happens:

```
onclick(click, mark)
```

Finally, after the desired coordinate transform is defined in the `draw` method, use the `placemark` function like this:

```
    placemark(mark)
```

Here is an example that illustrates how to put everything together. When an event occurs, the type of event and the point at which it occurs is printed in the window.




```
from jiscript.JiModule import *

s = ''
px, py = 0, 0
def draw():
    beginpage()

    center()
    scale(50)
    placemark(mark)      # placemark occurs here

    gsave()
    box(-1.5, -1.5, 3, 3)
    fill(1, 0, 0)
    grestore()
```

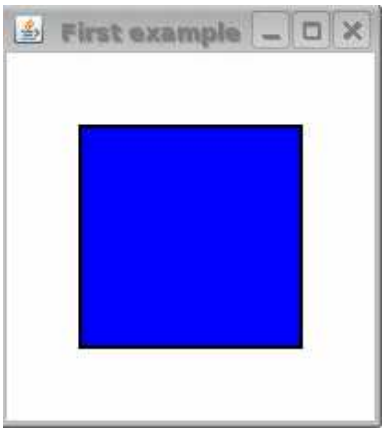
```
    newpath()
    setcolor(0, 0, 1, 0.5)
    setfont(36)
    moveto(-1.5, 0)
    show(s)
    if len(s) > 0:
        newpath()
        moveto(-1.5, -1)
        pt = '(%.1f, %.1f)' % (px, py)
        show(pt)
    endpage()
def click(x, y):
    global s, px, py
    px, py = x, y
    s = 'click'
def enter(x, y):
    global s, px, py
    px, py = x, y
    s = 'enter'
def exit(x, y):
    global s, px, py
    px, py = x, y
    s = 'exit'
def press(x, y):
    global s, px, py
    px, py = x, y
    s = 'press'
def release(x, y):
    global s, px, py
    px, py = x, y
    s = 'release'
def drag(x, y):
    global s, px, py
    px, py = x, y
    s = 'drag'
def move(x, y):
    global s, px, py
    px, py = x, y
    s = 'move'
mark = Mark()
onclick(click, mark)
onenter(enter, mark)
onexit(exit, mark)
onpress(press, mark)
onrelease(release, mark)
ondrag(drag, mark)
onmove(move, mark)
```

```
openframe(300, 300, draw)
```

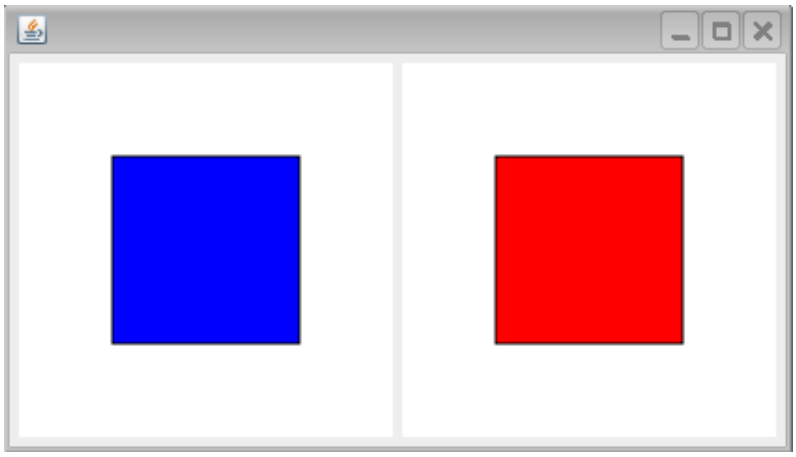
10: The JiScript module

Up to this point, we have been using the `JiModule` module. As you can see, `JiModule` allows us to put together interactive illustrations with relative ease. In this section, we will introduce the `JiScript` module, which gives more flexibility than `JiModule` with a small loss in ease of use.

Before we begin, recall the first example in which we drew a blue box:



When displayed, there is a frame, the gray border, that contains a panel on which we draw. Up to this point, we have drawn on only one panel. Frequently, however, we would like to draw on two or more panels. The `JiScript` module gives us this ability, as demonstrated by the following program.



Let's first look at the code that produces the two boxes.

```
from JiScript import *
```

```
def left(panel):
    panel.beginpage()
    panel.center()
    panel.scale(50)
    panel.newpath()
    panel.box(-1, -1, 2, 2)
    panel.fill(0, 0, 1)
    panel.stroke()
    panel.endpage()

def right(panel):
    panel.beginpage()
    panel.center()
    panel.scale(50)
    panel.newpath()
    panel.box(-1, -1, 2, 2)
    panel.fill(1, 0, 0)
    panel.stroke()
    panel.endpage()

openframe([ [ 200, 200, left], [200, 200, right] ])
```

A few things are happening here that deserve careful attention. First, we import `JiScript` rather than `JiModule`. The functions `left` and `right` are the drawing functions for the left and right panels. Notice that these functions now have an argument, called `panel`, and that the graphics commands are functions associated with `panel`. Aside from these two differences, the graphics commands are familiar.

Finally, `openframe` receives a list whose entries are lists describing the panels' dimensions and drawing methods. When `openframe` receives a list like this, it places the panels in the frame from left to right in the order in which received.

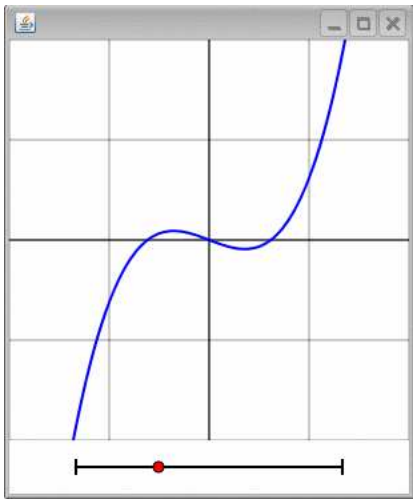
To understand what's happening here, let's look under the hood for a moment. The `JiScript` package contains a module called `JiPanel`, which provides a panel on which to draw. When we have used `JiModule` previously, there is a single `JiPanel` defined inside `JiModule` and all of the drawing commands go to it. Now that we have multiple panels, we need to make sure the drawing commands go to the correct panel. This is why the drawing functions `left` and `right` now require an argument specifying which panel receives the drawing commands.

In the example above, the `openframe` function creates the `JiPanels` we will use. However, we will see that it is sometimes convenient to create the `JiPanels` explicitly. We can do that like this:

```
leftpanel = JiPanel(200, 200, left)
openframe([ leftpanel, [200, 200, right] ])
```

Why would we want to define a `JiPanel` explicitly? Remember that when we used a `Moveablepoint` using the `JiModule` module, we registered the point so that our program would listen for attempts to move the point. We were really registering the `Moveablepoint` with `JiModule`'s `JiPanel`. If we have multiple panels, we will need to indicate which panel should listen for attempts to move the point.

To demonstrate, let's study the following example



The slider on the bottom controls a parameter c, which is allowed to vary from -1 to 1. The panel above shows the graph of the function $x^3 + cx$. Here's the code:

```
from JiScript import *

def plot(p):            # the drawing function for the top panel
    p.beginpage()
    p.center()
    p.scale(75)

    p.newpath()
    for i in range(-2, 3):      # draw the grid
        p.moveto(-2, i)
        p.lineto(2, i)
        p.moveto(i, -2)
        p.lineto(i, 2)
    p.stroke(0.6)

    p.newpath()                 # draw the axes
    p.moveto(-2, 0)
    p.lineto(2, 0)
    p.moveto(0, -2)
    p.lineto(0, 2)
    p.stroke()

    p.newpath()                 # plot the graph
    p.graph(lambda x: x**3 + parameter * x, -2, 2)
    p.setlinewidth(2)
    p.stroke([0,0,1])

    p.endpage()

def slider(p):          # the drawing function for the slider below
    p.beginpage()
    p.center()
    p.scale(100)
```

```
    p.setlinewidth(2)
    p.newpath()
    p.moveto(-1, 0)
    p.lineto(1, 0)
    tick = 0.05
    p.moveto(1, tick)
    p.lineto(1, -tick)
    p.moveto(-1, tick)
    p.lineto(-1, -tick)
    p.stroke()
    p.placemoveable(slidingpoint)
    p.endpage()
def slide(point, x, y): # the function that moves the slider
    global parameter
    if x < -1: x = -1
    elif x > 1: x = 1
    point.setpoint(x, 0)
    parameter = x
    refresh()
parameter = -1
slidingpoint = Moveablepoint(parameter, 0, slide)
bottom = JiPanel(300, 40, slider)
bottom.addmoveable(slidingpoint)
openframe({'center': [300, 300, plot], 'south': bottom})
```

As before, we have two drawing functions, one for each panel. The argument is abbreviated to `p` to save keystrokes. There is a `Moveablepoint` called `slidingpoint`, which is defined near the end of the code. Notice that this point is registered with `bottom`, the `JiPanel` that holds the slider. The drawing function `slider` includes the `placemoveable` command.

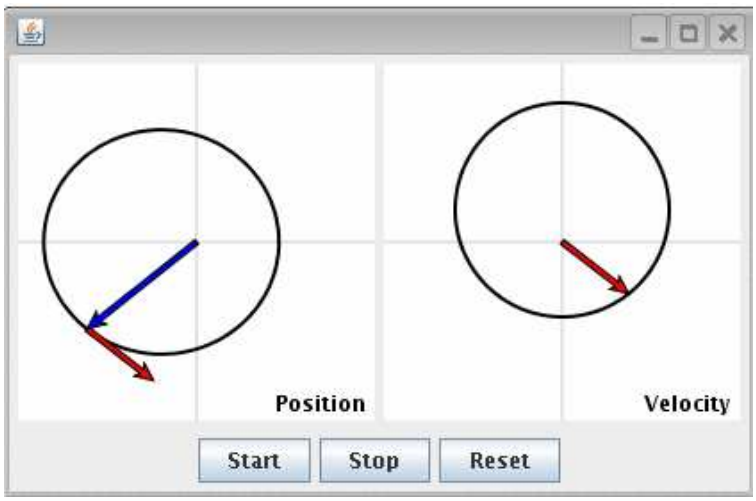
You may notice that the layout of the two panels is different. When we gave `openframe` a list, it simply laid the panels down from left to right. Here, we give `openframe` a dictionary with keys "center" and "south." The panel associated to "center" is placed in the center of the frame, while the panel associated to "south" is in the southern part of the frame. Other possible keys are "north," "east," and "west." These keys may also be abbreviated by their first letter.

Besides `Moveablepoints`, you will want to employ this strategy when listening for mouse events in a panel. For instance, if you would like to listen for mouse clicks in a panel, define the panel and register with the panel the function to be called when the mouse is clicked:

```
leftpanel = JiPanel(200, 200, draw)
leftpanel.onclick(click, mark)
```

We may call also use a `JiPanel`'s functions `writePNG` or `writeJPEG` to save the graphical contents of that panel as an image file.

Here is a final example, demonstrating Hamilton's theorem relating the velocity and position of, say, a planet moving under the gravitational influence of the sun, that illustrates how buttons and timers work as before.




```
from JiScript import *
import math

def r(theta):
    return 1.0/(1+eccentricity*math.cos(theta))

def position(p):
    p.beginpage()
    ...
    p.endpage()

def velocity(p):
    p.beginpage()
    ...
    p.endpage()

def begin():
    starttimer()

def pause():
    stoptimer()

def reset():
    global theta
    stoptimer()
    theta = 0
    refresh()

def update():
    global theta
    p = [r(theta)*math.cos(theta),
         r(theta)*math.sin(theta)]
    v = [-math.sin(theta), math.cos(theta)+eccentricity]
    dt = 0.025
    theta = math.atan2(p[1] + dt*v[1], p[0] + dt*v[0])
    refresh()
```

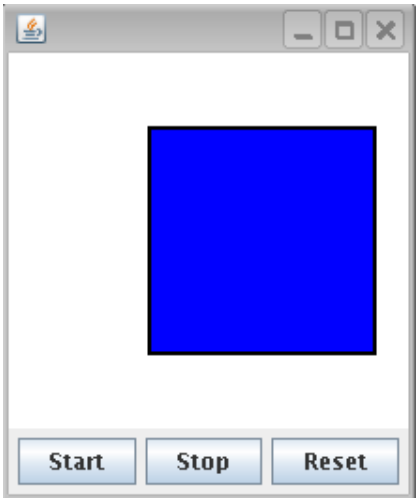
```
eccentricity = 0.3
theta = 1.5
settimer(25, update)
addbutton("Start", begin)
addbutton("Stop", pause)
addbutton("Reset", reset)
openframe([ [200, 200, position], [200, 200, velocity] ])
```

11: Creating applets

So far, we have used `JiScript` to create new windows that pop up on our screen. To distribute an illustration or animation widely, we may wish to post it on the web as an applet that may be loaded into a web browser. This involves a slight extension, which we'll now describe.

Please note that programs written with `JiModule` will not work as applets; we must use `JiScript`.

We will demonstrate by converting our earlier program, `BoxAnimation.jy`, to an applet. Remember that this simply displayed a blue box and moved it across the window when the timer was started.



We begin by modifying the program to import `JiScript` rather than `JiModule`. In this example, this means that our drawing function needs the panel passed as an argument.

```
from JiScript import *

x = 40
def draw(p):
    p.beginpage()
    p.newpath()
    p.box(x, 40, 120, 120)
    p.fill(0, 0, 1)
    p.setlinewidth(2)
    p.stroke()
    p.endpage()
```

```
def move():
    global x
    x += 1
    refresh()

def reset():
    global x
    stoptimer()
    x = 40
    refresh()

def begin():
    starttimer()

def pause():
    stoptimer()

settimer(25, move)
addbutton('Start', begin)
addbutton('Stop', pause)
addbutton('Reset', reset)
openframe(200, 200, draw)
```

When this program is in a separate Python module, perhaps called `BoxAnimation.jy`, it is simple to test and to make changes. When you are ready to create an applet, comment out the `openframe` command since the applet will provide it's own frame.

The construct an applet, we need to define a second module, which we'll call `BoxAnimationApplet.jy`, and a class called `BoxAnimationApplet` inside it. Due to Java's restrictions, the name of the module must agree with the name of the class, and the class must extend the Java class `javax.swing.JApplet`.

```
from JiScript import *
from javax.swing import *
import BoxAnimation

class BoxAnimationApplet(JApplet):
    def __init__(self):
        layoutapplet(200, 200, BoxAnimation.draw, self)
```

You should note that the `layoutapplet` function has the same syntax as the `openframe` command, except that `self` is given as a final argument.

To test the applet, you may wish to include the line

```
testapplet(BoxAnimationApplet())
```

at the end of the module. You may then run the applet with

```
jython BoxAnimationApplet.jy
```

If you see two frames, it is probably because you did not remove the `openframe` command from `BoxAnimation.jy`.

When this works satisfactorily, remove the `testapplet` command, and you are ready to make a jar file containing your applet.

```
jythonc -j boxanimation.jar -c BoxAnimationApplet.jy
```

This process usually takes a few seconds and it will annoy you if you must do it many times. This is why I suggest testing the program as an application first. Regardless, you should be left with a file called `boxanimation.jar`.

The final ingredient is to construct an HTML file that will load the applet into a browser. Here's an example:

```
<applet
  code=BoxAnimationApplet
  width=200
  height=240
  archive=boxanimation.jar>
</applet>
```

If this is in a file `boxanimation.html`, you may test your applet by loading that file into a web browser or by using the Java tool `appletviewer` by typing

```
appletviewer boxanimation.html
```

If you do not remove the `testapplet` command from the applet file, you will get a most vexing error message.

Applets with a more complicated layout work as before. For example, here is the file `KeplerApplet.jy` used to create an applet from our `KeplerAnimation` module.

```
from JiScript import *
from javax.swing import *
import KeplerAnimation
class KeplerApplet(JApplet):
    def __init__(self):
        layoutapplet([[200, 200, KeplerAnimation.position],
                      [200, 200, KeplerAnimation.velocity]], self)
```

12: Free advice!

Here is a collection of suggestions starting with the more technical.

- In dynamic illustrations, don't forget to `refresh`.
- In a dynamic illustration that requires a lot of calculation, put as much of the calculation in the function that calls `refresh` rather than in the function that does the drawing. The drawing function will be called at times you might not expect, such as when the window is raised to the front of the desktop. The best strategy is therefore to separate calculations from drawing.

- Those with some experience with graphics in Java may want access to the `JiPanel`'s graphics object. It is available with the `getgraphics()` command.
- Provide reset buttons. It's easy to mess up illustrations, sometimes due to the programmer failing to foresee some user actions, so give the user a way back to the original state.
- Use visual cues rather than text. Experience shows that relationships between elements in a figure that are expressed with visual cues, such as color and shape, are more easily detected than when text is used.
- In a series of illustrations, use color consistently and with purpose. For instance, color points that move red and points that follow another point's motion gray.
- As figures become more complicated, use muted colors or grays for peripheral elements in an illustration. For instance, when creating a figure that illustrates a construction, it can be effective to have elements left over from earlier parts of the construction fade into the background. One way to achieve this is with a timer that modifies the alpha values of the colors you are using.
- Provides less choice for the user rather than more. There are many reasons that you might want to create a dynamic illustration, from communicating a mathematical idea to a new audience to exploring a new concept for your own understanding. Generally speaking, an illustration should have a focus. Make sure that the user's choices are so few that he is unable to avoid the point you are trying to make. Simple animations are often effective in helping achieve this goal.

Appendix A: Installing Jython and JiScript

To install `Jython`, you will first need to install a version of the Java Developer's Kit, available at `http://java.sun.com/javase/downloads/index.jsp` and add the `bin` directory to your executable path.

`Jython` may be downloaded from `http://www.jython.org`; installation instructions are also available from that page. This can be as simple as typing

```
java -classpath . jython-21
```

in a command window with the current directory set to the one in which you extracted the `jython` files.

Finally, you will need to get the `JiScript` files, available in a zip file from

```
http://merganser.math.gvsu.edu/david/jiscript
```

and extract them into a directory. You then need to edit the `jython` registry so that the `python.path` variable points to this directory. On my installation, `JiScript` is installed in

```
/home/david/jiscript
```

and `jython` is in

```
/home/david/jython2.2.1
```

I edited one line in the file

```
/home/david/jython2.2.1/registry
```

to read

```
python.path = /home/david/
```

Note that you will probably need to remove the comment ("#") before `python.path`.