Review

- Exam 2 Topics
- Recursive Functions Review
- Inheritance Review

Exam 2 Topics

1. String Manipulation
   - len(), strip(), split(), join()
2. Data Structures
   - lists, dictionary, combinations
3. Transformations
   - pushMatrix(), popMatrix(), translate(), rotate(), scale()
4. Image Processing
   - loadPixels(), updatePixels(), getPixel(), setPixel(), image()
5. Functions
   - defining, calling
6. Debugging
7. Recursive Functions
   - Designing: conditional, base case, recursive function call
8. Inheritance
   - subclassing, methods and instance variable overriding

Transformations Review

- pushMatrix()
- popMatrix()
- translate()
- rotate()
- scale()

Three ways to transform the coordinate system:

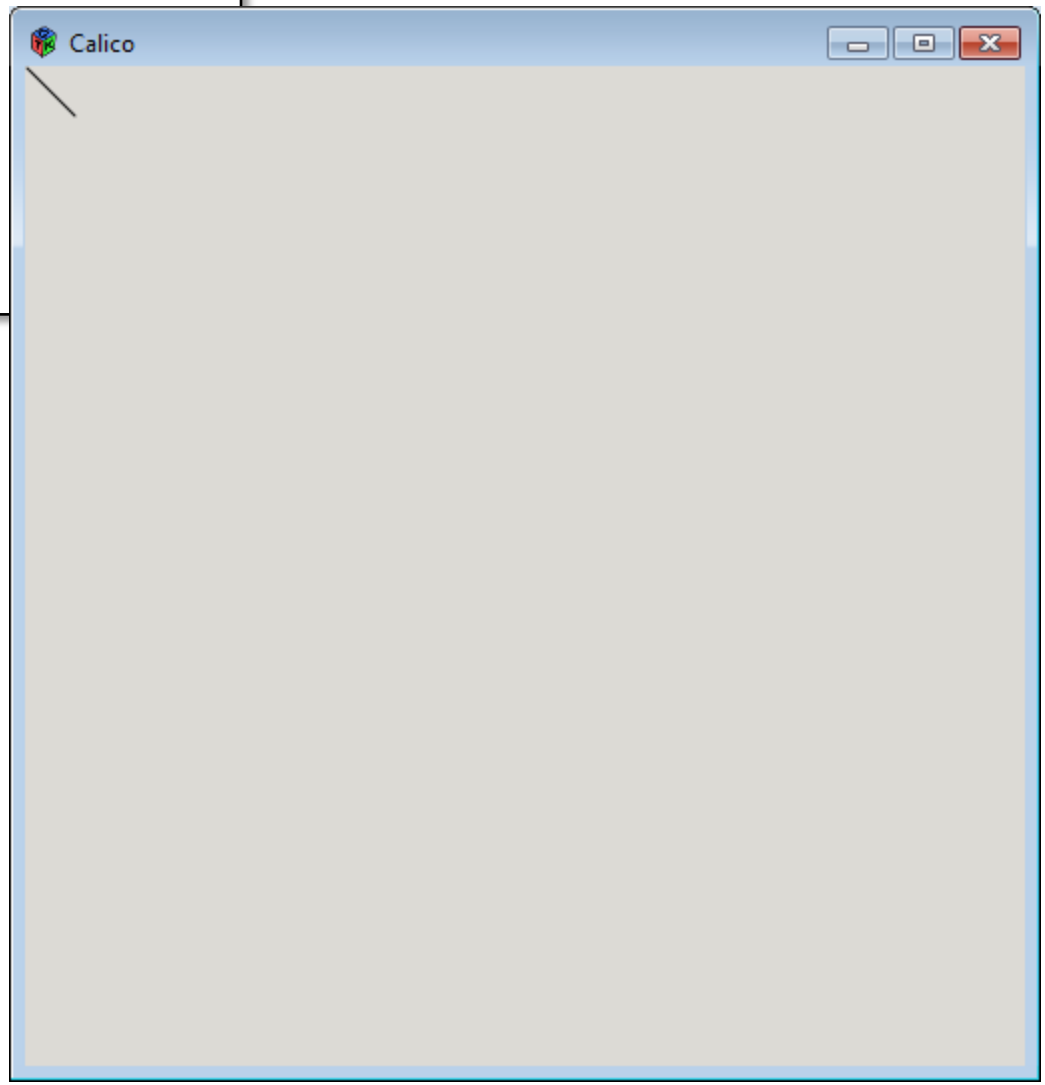
1. **Translate**
   - Move axes left, right, up, down …
2. **Scale**
   - Magnify, zoom in, zoom out, about the origin …
3. **Rotate**
   - Tilt clockwise, counter-clockwise, about the origin …

Scale

- All coordinates are multiplied by an x-scale-factor and a y-scale-factor.
- The size of everything is magnified about the origin (0,0)
- Stroke thickness is also scaled.

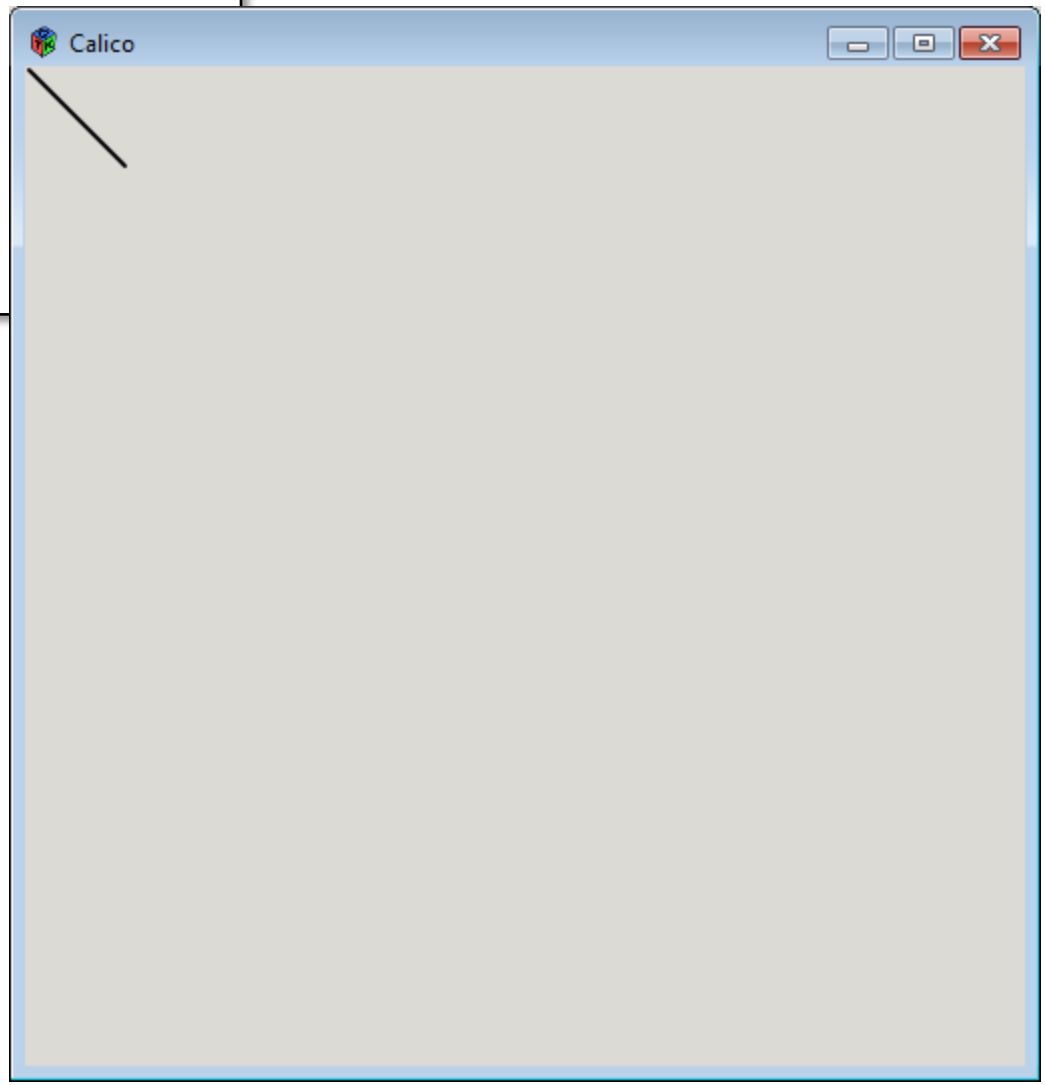
```
scale( factor )
scale( x-factor, y-factor )
```

```
from Processing import *

window(500, 500)

line(1, 1, 25, 25)
```



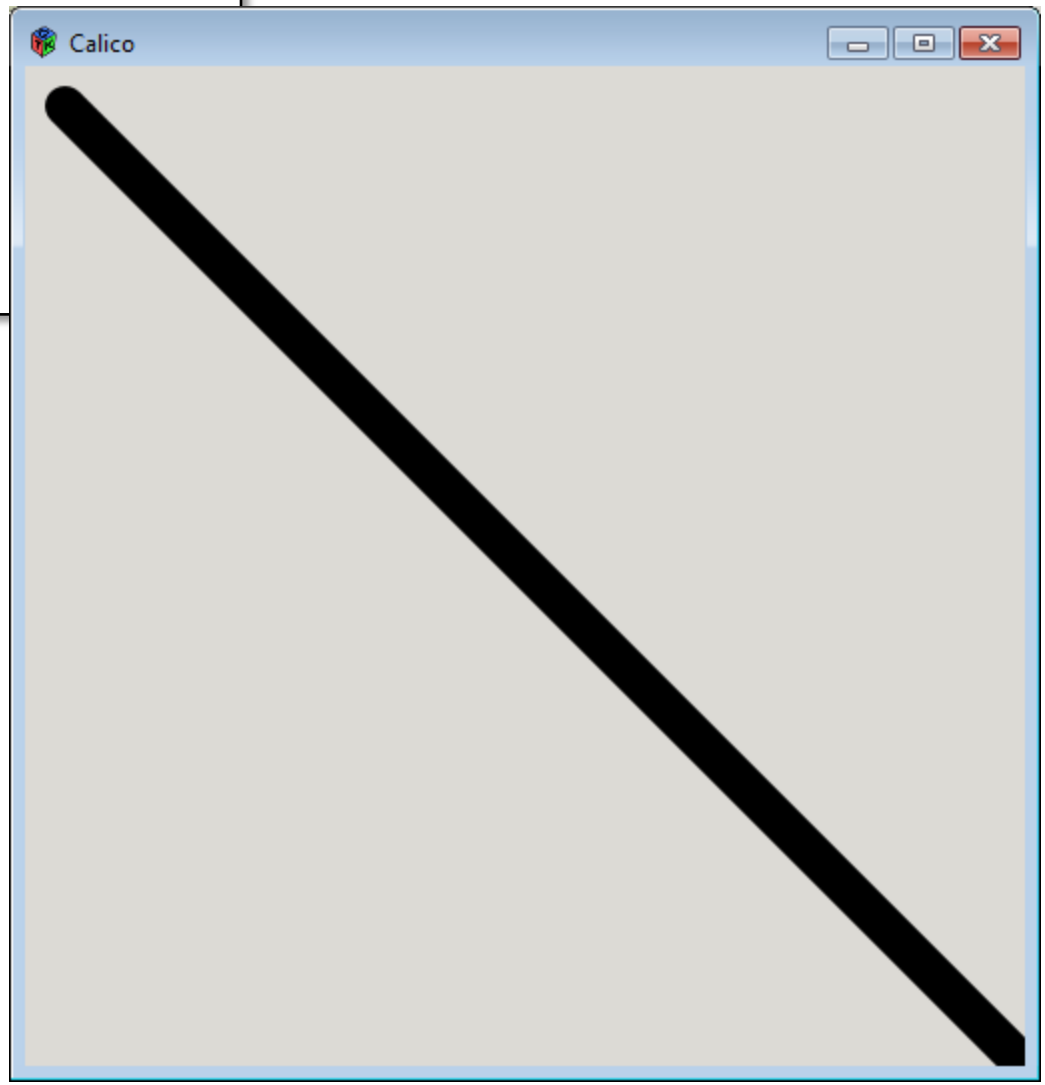
example2.pde

```
from Processing import *

window(500, 500)

scale(2,2)
line(1, 1, 25, 25)
```



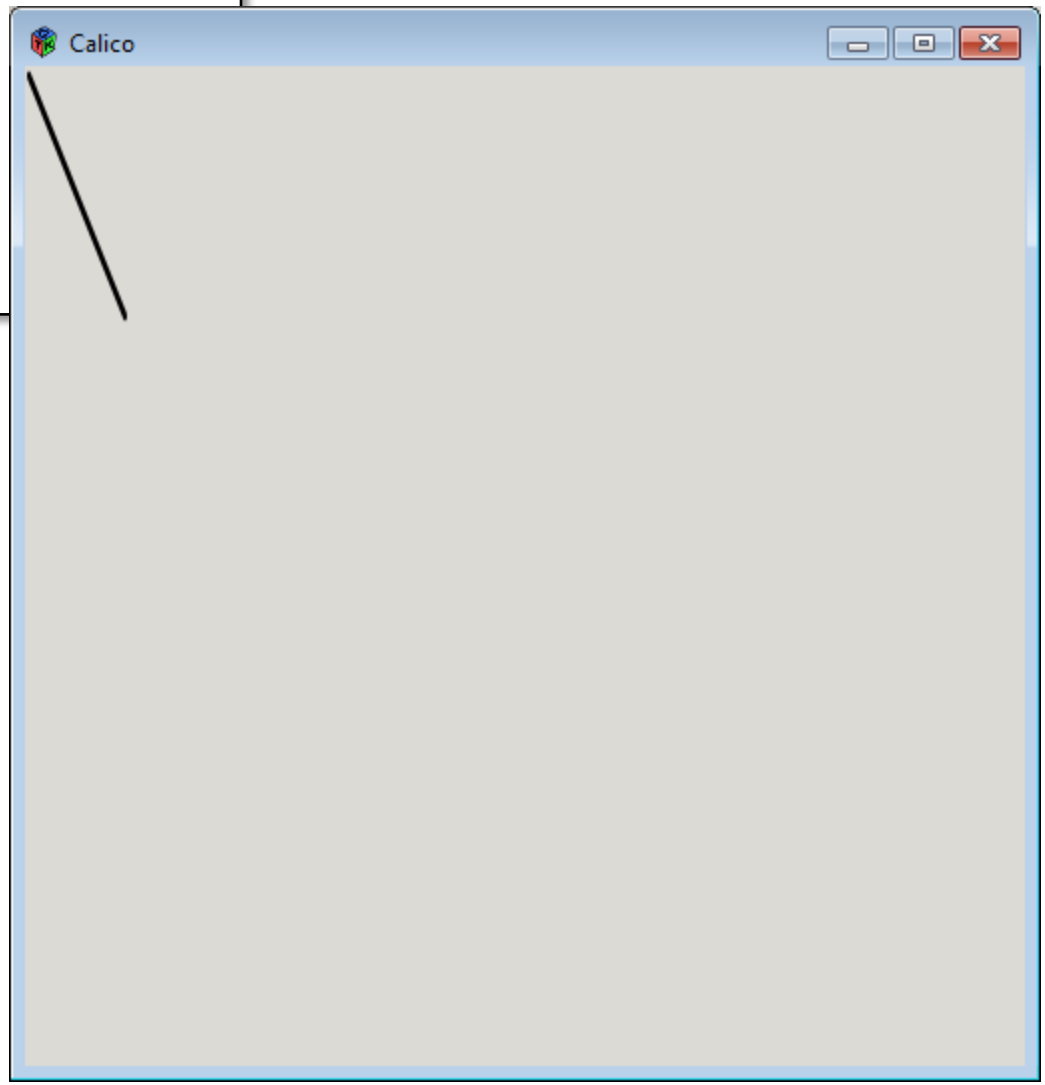
example2.pde

```
from Processing import *

window(500, 500)

scale(20,20)
line(1, 1, 25, 25)
```



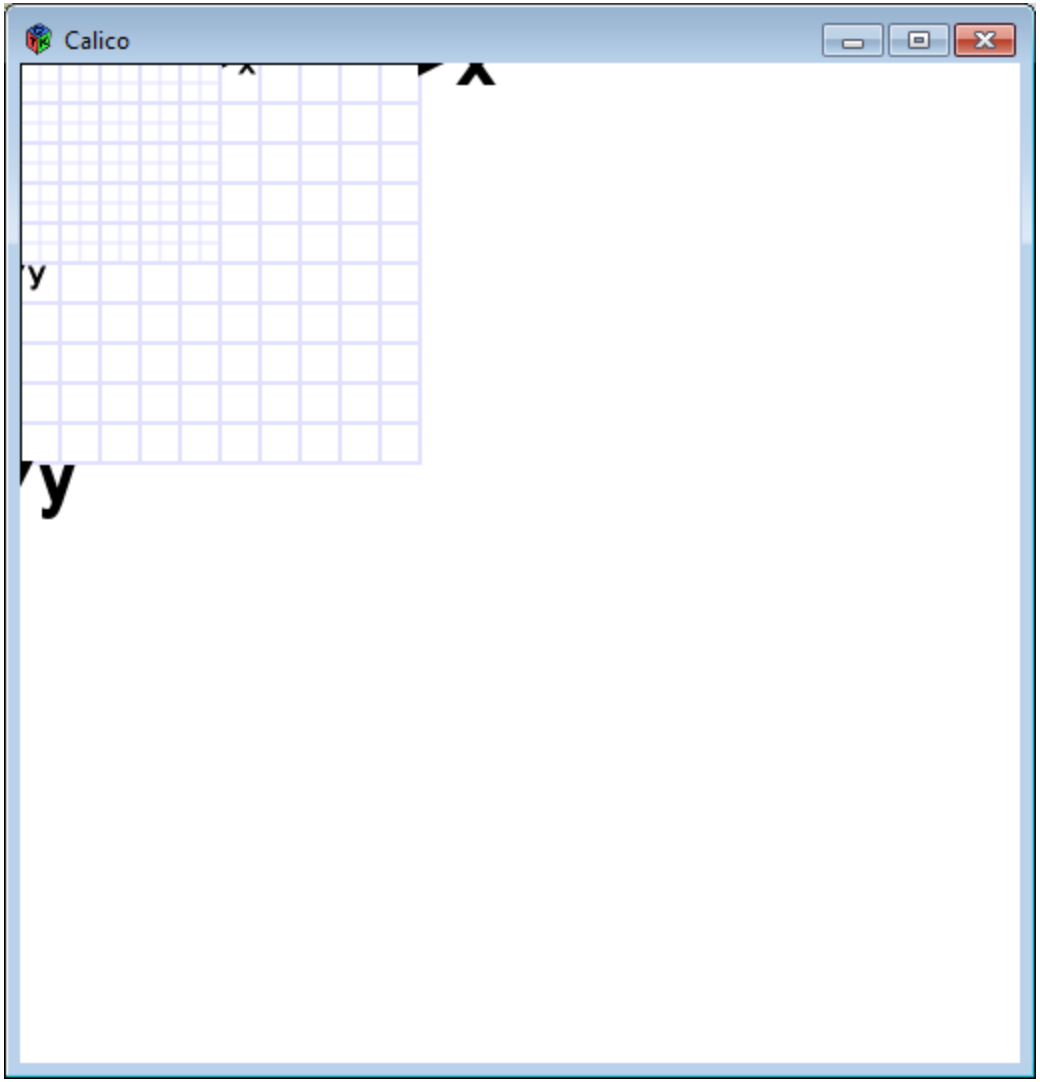
example2.pde

```
from Processing import *

window(500, 500)

scale(2,5)
line(1, 1, 25, 25)
```



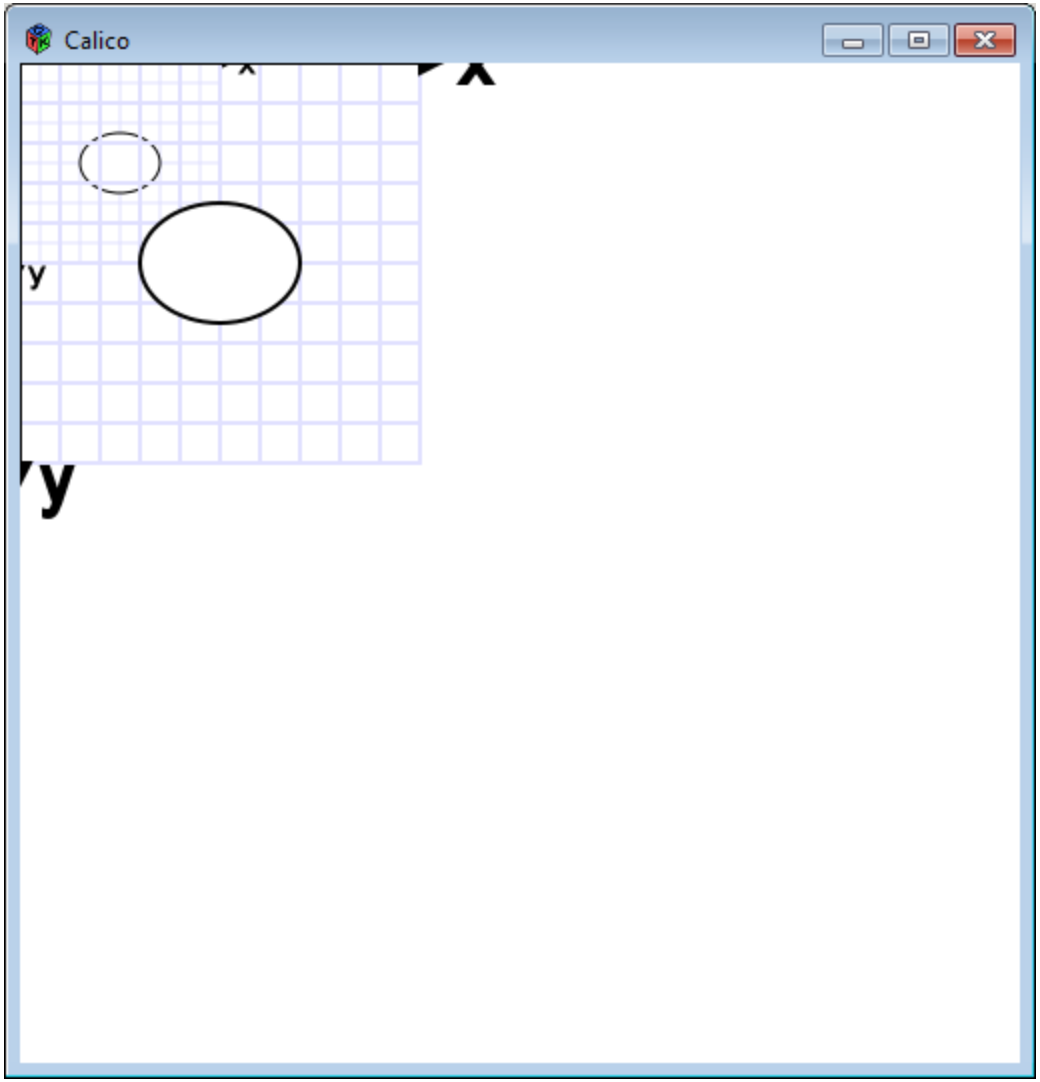
example2.pde

```
from Processing import *
window(500, 500)

background(255)
grid()
scale(2,2)
grid()
```



grid.pde

```
from Processing import *
window(500, 500)

background(255)

grid()
fill(255)
ellipse(50,50,40,30)

scale(2,2)
grid()
fill(255)
ellipse(50,50,40,30)
```



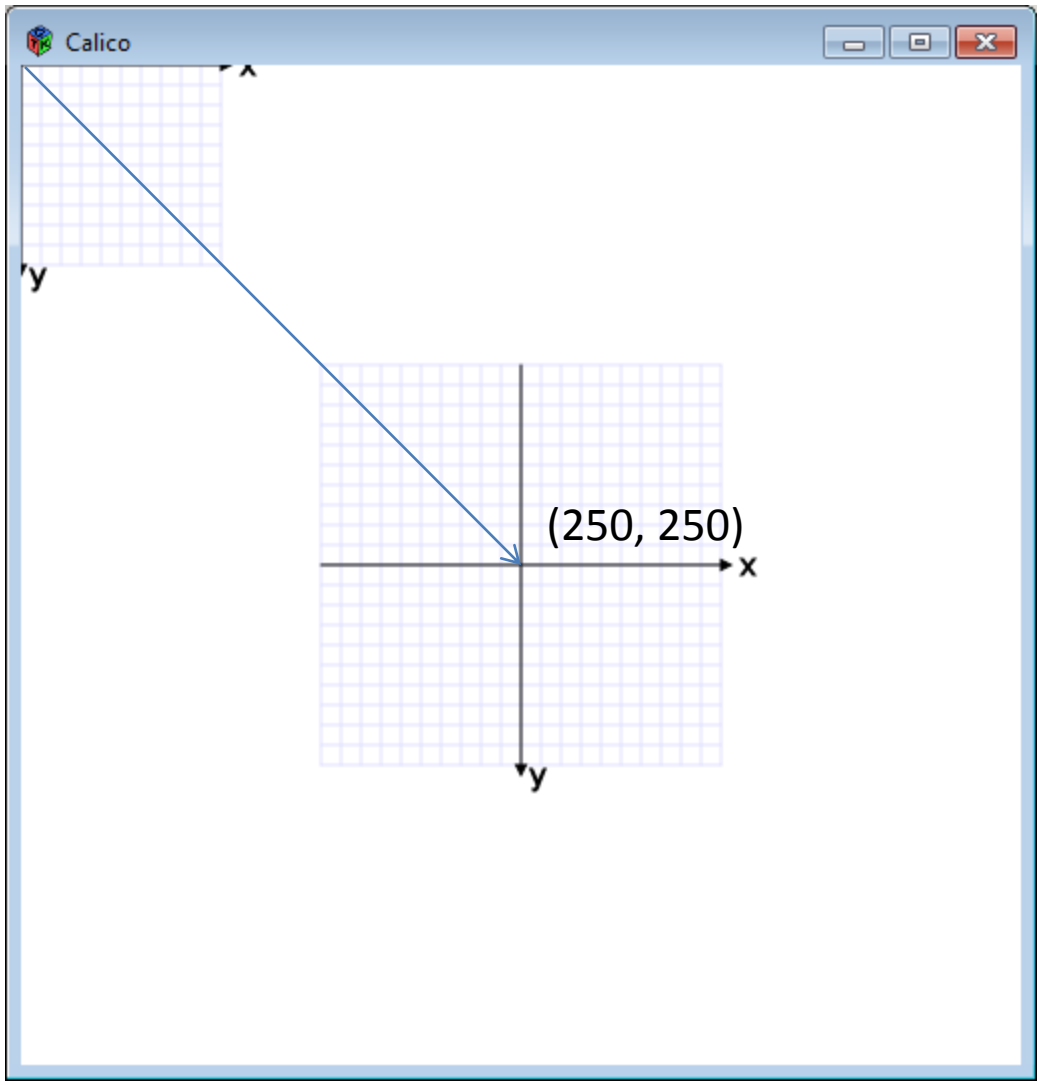
grid.pde

Translate

- The origin of the coordinate system (0,0) is shifted by the given amount in the x and y directions.

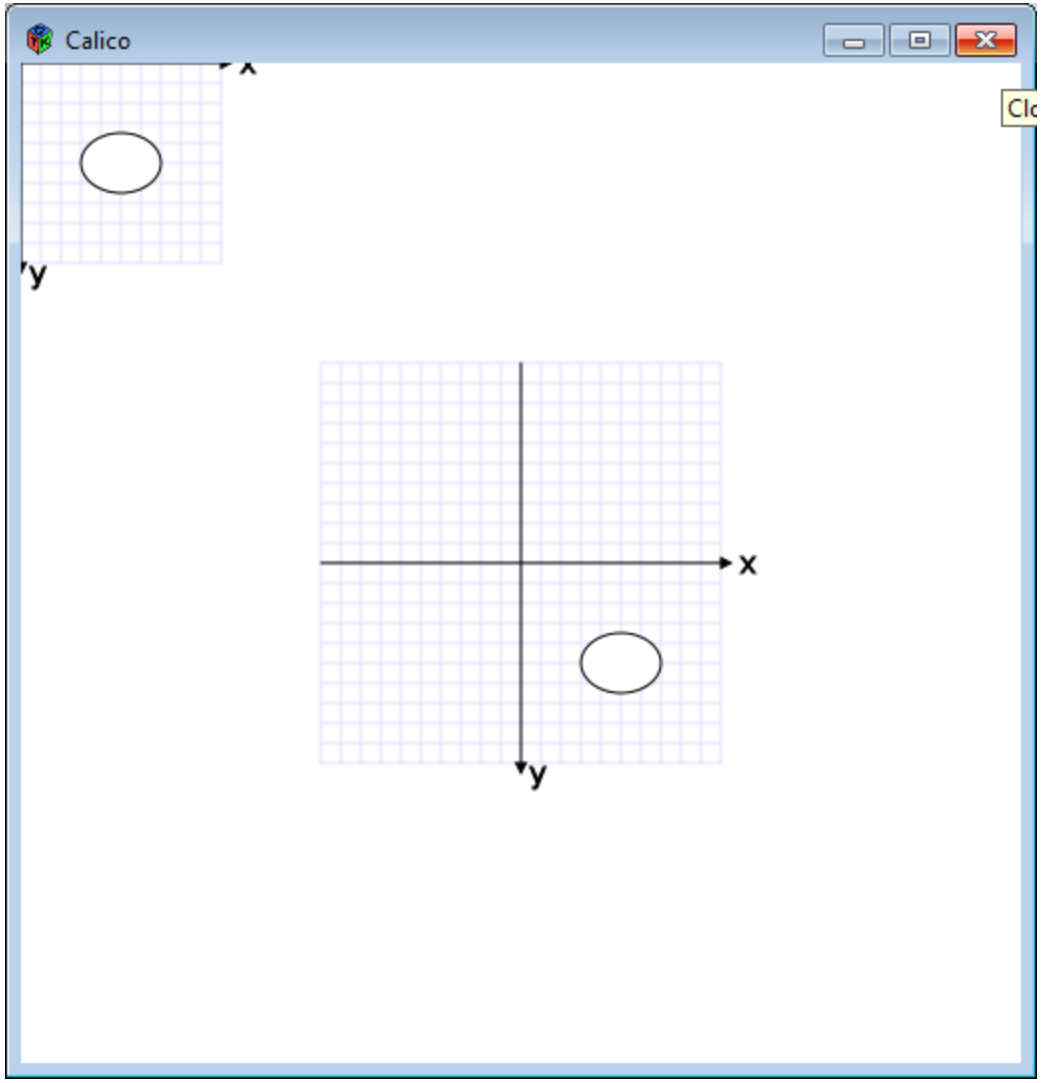
```
translate( x-shift, y-shift)
```

```
from Processing import *
window(500, 500)

background(255)

grid()

translate(250, 250)
grid()
```



grid2.pde

```
from Processing import *
window(500, 500)

background(255)

grid()

fill(255)
ellipse(50, 50, 40, 30)

translate(250, 250)
grid()
fill(255)
ellipse(50, 50, 40, 30)
```

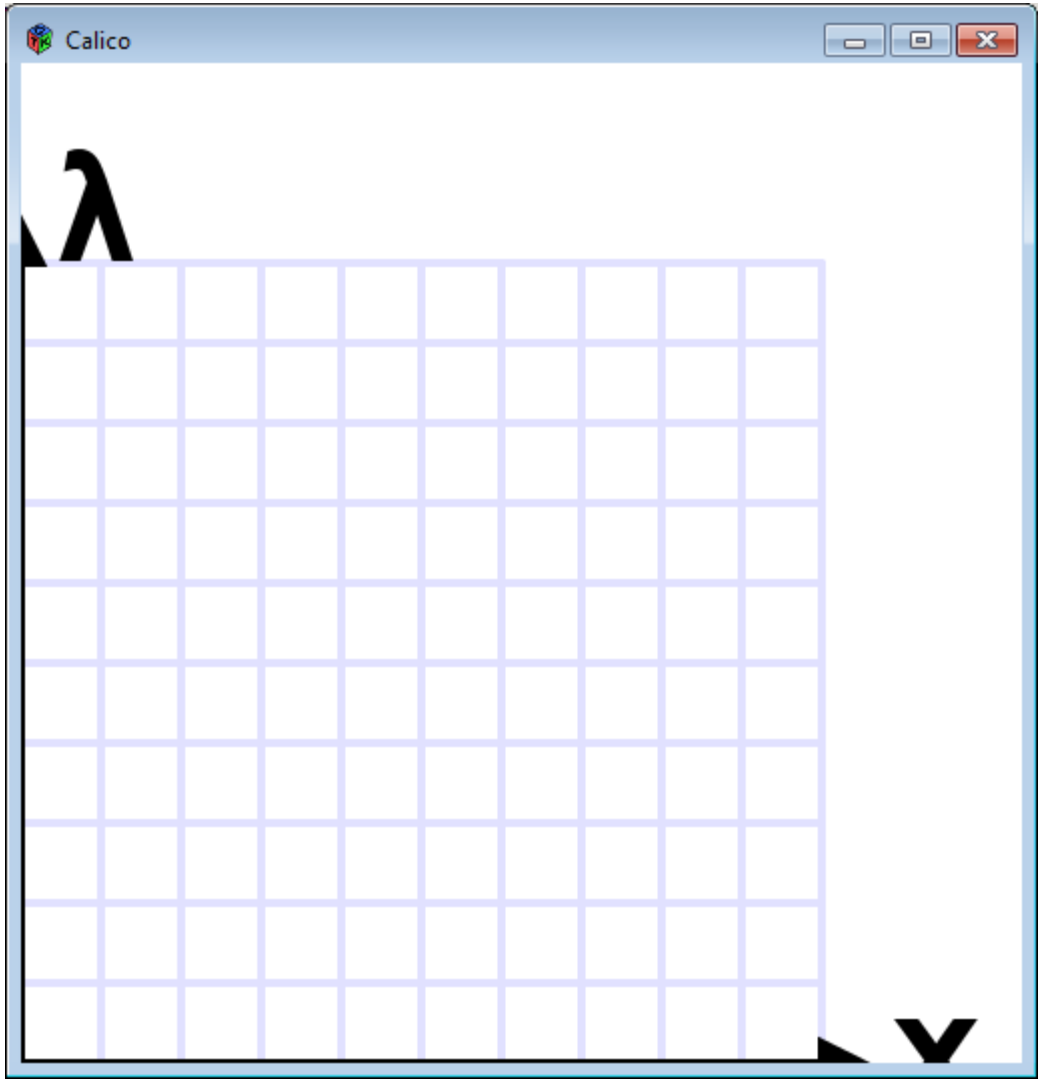


Transformations can be combined

- Combine Scale and Translate to create a coordinate system with the y-axis that increases in the upward direction
- Axes can be flipped using negative scale factors

- Order in which transforms are applied matters!

```
from Processing import *
window(500, 500)

background(255)
#grid()

translate(0, height())
scale(4, -4)
grid()
```



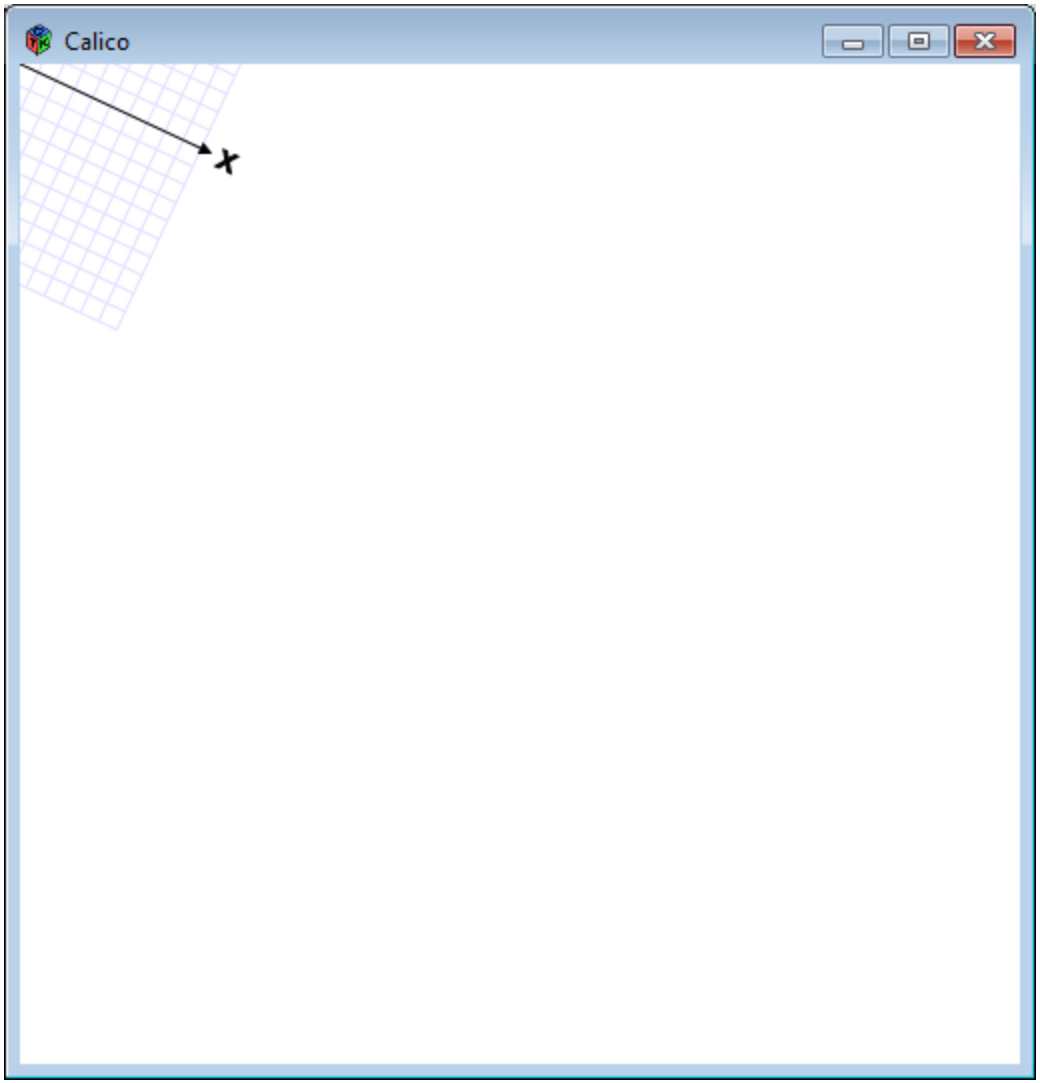
grid.pde

Rotate

- The coordinate system is rotated around the origin by the given angle (in radians).

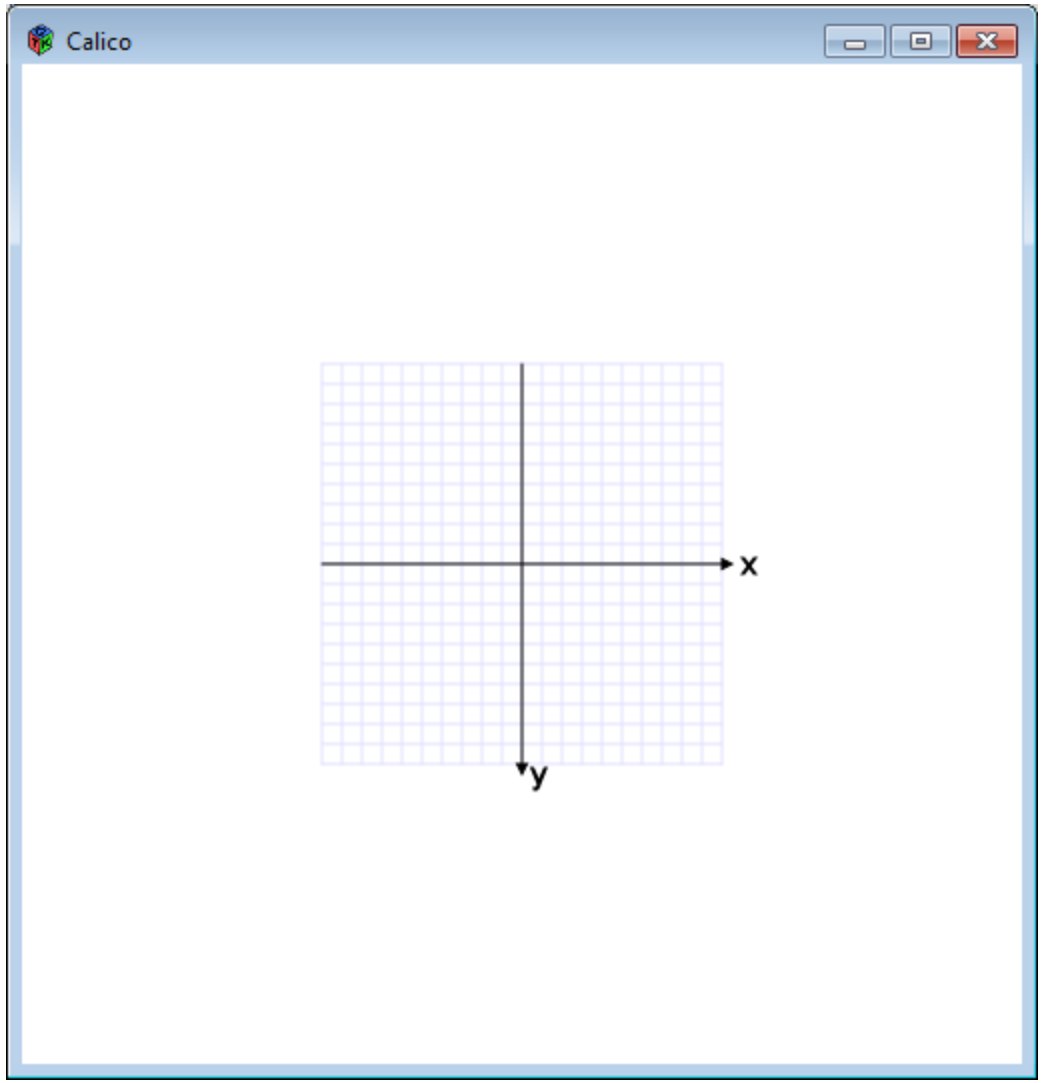
```
rotate( radians )
```

```
from Processing import *
window(500, 500)

background(255)

rotate( radians(25) )
grid()
```



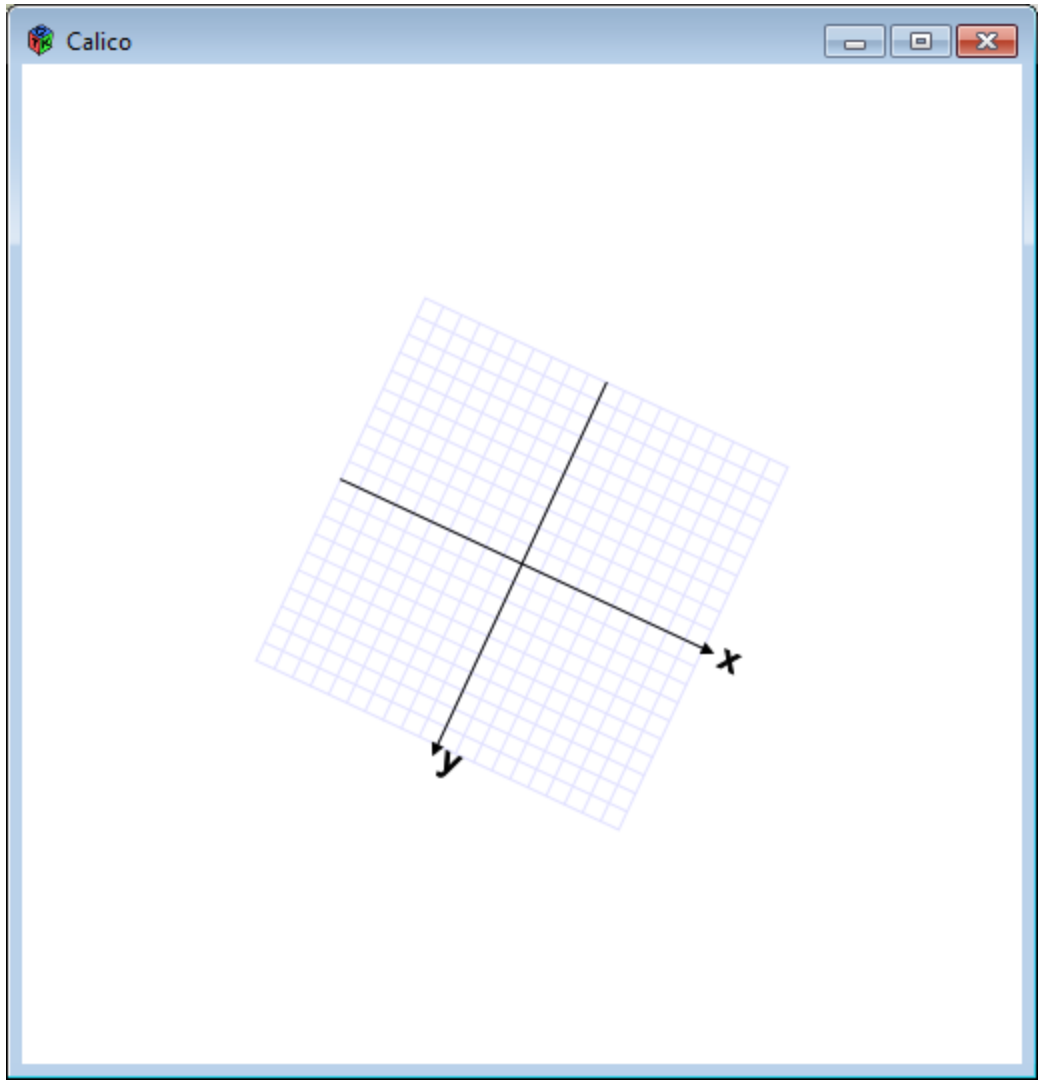
grid.pde

```
from Processing import *
window(500, 500)

background(255)

translate(250.0, 250.0)
#rotate( radians(25) )
#scale( 2 )
grid()
```



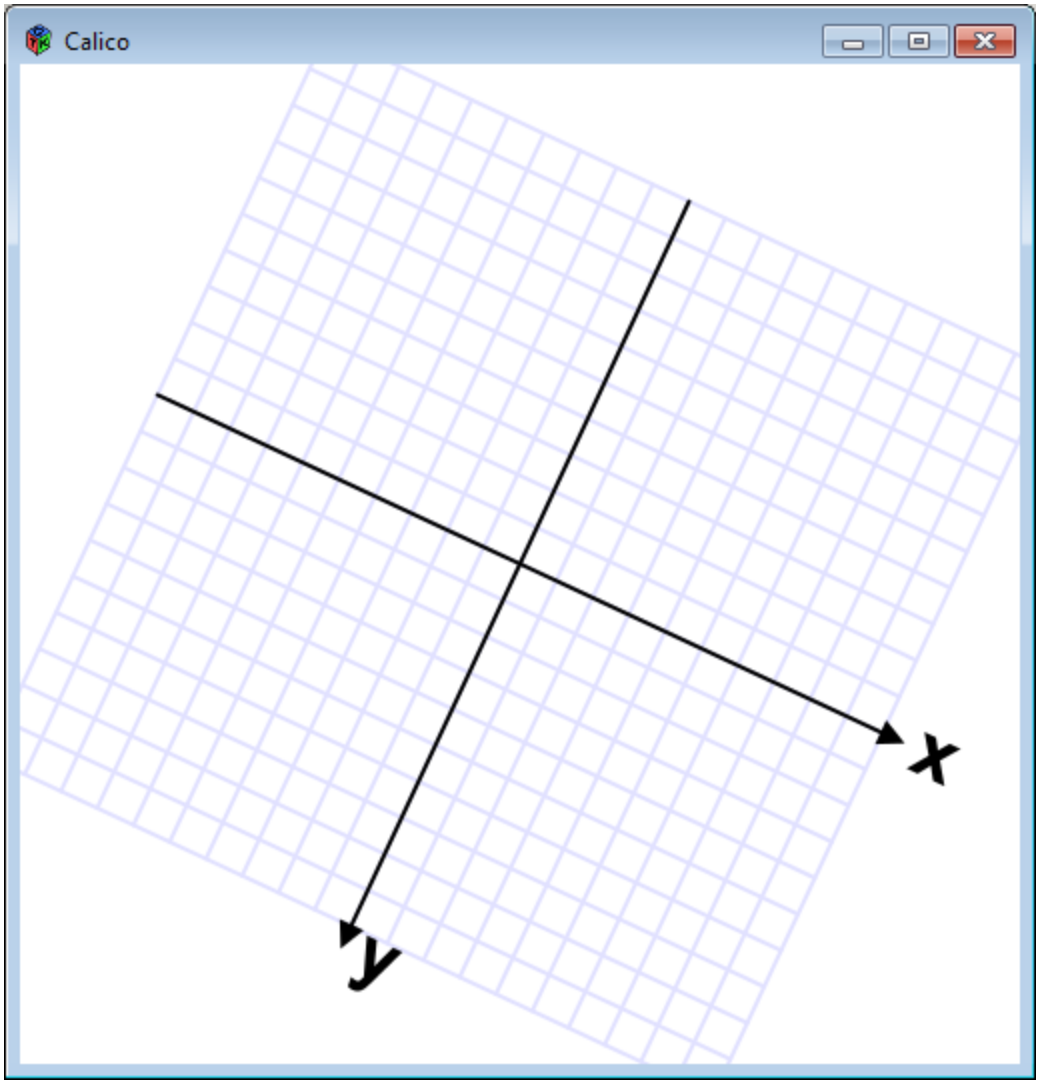
grid.pde

```
from Processing import *
window(500, 500)

background(255)

translate(250.0, 250.0)
rotate( radians(25) )
#scale( 2 )
grid()
```



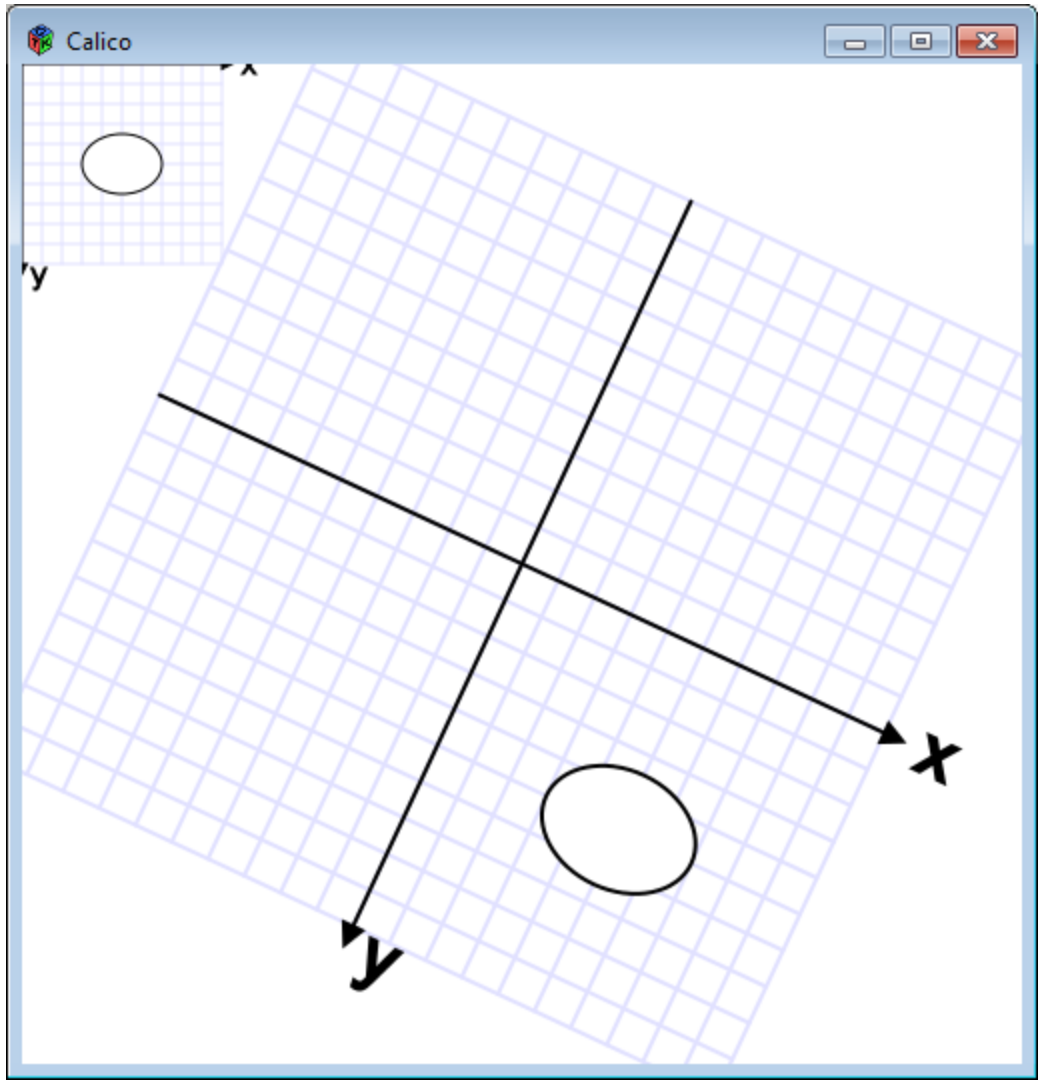
grid.pde

```
from Processing import *
window(500, 500)

background(255)

translate(250.0, 250.0)
rotate( radians(25) )
scale( 2 )
grid()
```



grid.pde

```
from Processing import *
window(500, 500)

background(255)
grid()
fill(255)
ellipse(50, 50, 40, 30)

translate(250.0, 250.0)
rotate( radians(25) )
scale( 2 )
grid()
fill(255)
ellipse(50, 50, 40, 30)
```



grid.pde

Problem

- Transformations accumulate!

- resetMatrix()
  - Roll back all transformations to original

- How roll back some transformations, not all?
  - pushMatrix()
    - Saves the current transformation state
  - Perform transformation and drawing, as needed
  - popMatrix()
    - Restores transformation to state when pushMatrix was called

```
# pushpop1.py
from Processing import *

window(500, 500)

# Translate the origin of the coordinate system
# to the center of the sketch window
translate(250, 250)

# Rotate and draw a line and ellipse
def draw(o, e):
    rotate( radians(5) )
    line(0, 0, 100, 0)          # Drawing args are constant
    ellipse( 100, 0, 10, 10)

# Draw again on mouse pressed
onMousePressed += draw
```

- Can we use resetMatrix() to prevent the accumulation of transformations?
- What if we move translate() into the draw() function?

Some things to remember:

1. Transformations are cumulative.
2. Rotation angles are measured in radians
   - π radians = 180°
   - radians = (PI/180.0) * degrees
3. Order matters
4. Always bracket your transformations with pushMatrix() and popMatrix() unless you explicitly want to accumulate transformations
5. pushMatrix() and popMatrix() can be nested

Transformations

Draw a rectangle centered on the sketch window

```
# transform1.py
from Processing import *

window(500, 500)
background(255)

rectMode(CENTER)

pushMatrix()

translate(250, 250)

noFill()
rect(0, 0, 100, 50)

popMatrix()
```

Transformations

Rotate the previous rectangle by 30 degrees

```
# transform2.py
from Processing import *

window(500, 500)
background(255)
rectMode(CENTER)

pushMatrix()

translate(250, 250)
rotate( radians(30) )

noFill()
rect(0, 0, 100, 50)

popMatrix()
```

Transformations

Draw an animated rotating rectangle

```
# transform3.py
from Processing import *

window(500, 500)
rectMode(CENTER)

angle = 0.0
def draw(o, e):
    global angle
    background(255)

    pushMatrix()

    translate(250, 250)
    rotate( radians(angle) )

    noFill()
    rect(0, 0, 100, 50)

    popMatrix()

    angle += 3

frameRate(30)
onLoop += draw
loop()
```

Transformations

Draw an animated pulsating rectangle

```
# transform4.py
from Processing import *
from math import sin

window(500, 500)
rectMode(CENTER)

size = 1.0
time = 0.0
def draw(o, e):
    global size, time
    background(255)

    pushMatrix()

    translate(250, 250)
    scale( size )

    noFill()
    rect(0, 0, 100, 50)

    popMatrix()

    time += 0.05
    size = sin(time) + 1

frameRate(30)
onLoop += draw
loop()
```

Transformations

Draw an animated rotating and pulsating rectangle

```
# transform5.py
from Processing import *
from math import sin

window(500, 500)
rectMode(CENTER)

size = 1.0
time = 0.0
angle = 0.0
def draw(o, e):
    global size, time, angle
    background(255)

    pushMatrix()

    translate(250, 250)
    rotate( radians(angle) )
    scale( size )

    noFill()
    rect(0, 0, 100, 50)

    popMatrix()

    angle += 3
    time += 0.05
    size = sin(time) + 1

frameRate(30)
onLoop += draw
loop()
```

Transformations

What happens when rectMode is set to CORNER?

```
# transform5.py
from Processing import *
from math import sin

window(500, 500)
rectMode(CORNER)

size = 1.0
time = 0.0
angle = 0.0
def draw(o, e):
    global size, time, angle
    background(255)

    pushMatrix()

    translate(250, 250)
    rotate( radians(angle) )
    scale( size )

    noFill()
    rect(0, 0, 100, 50)

    popMatrix()

    angle += 3
    time += 0.05
    size = sin(time) + 1

frameRate(30)
onLoop += draw
loop()
```

Image Processing Review

- loadPixels()
- updatePixels()
- getPixel()
- setPixel()
- image()

The color data type

- In Calico Processing, ‘color’ is a type of data

  … like number, string, boolean, …

- To create one of these ‘color’ things…

```
myTranspColor = color( red, green, blue, alpha )
myColor       = color( red, green, blue )
myTranspGray  = color( gray, alpha )
myGrayColor   = color( gray )
```

- The signatures of functions to make a ‘color’ match that of background(), fill() and stroke(), exactly!

The color data type

- A ‘color’ can be used as an argument to set background(), fill(), and stroke()

```
myRed = color( 255, 0, 0 )
background( myRed )
fill( myRed )
stroke( myRed )
```

The color data type

- It is possible to extract information about a color.

```
myColor = color( 255, 128, 64, 255 )

re = red( myColor )          # Get the red component
gr = green( myColor )        # Get the green component
bl = blue( myColor )         # Get the blue component
al = alpha( myColor )        # Get the alpha component
sa = saturation( myColor ) # Get the saturation
hu = hue( myColor )          # Get the hue
br = brightness( myColor ) # Get the brightness
```

Accessing the pixels of a sketch

- loadPixels()
  - Loads the color data out of the sketch window into an internal data structure of colors
  - The colors in this internal structure can be modified programmatically

- updatePixels()
  - Copies the color data from the internal data structure back to the sketch window

loadPixels() and updatePixels() must bracket all image processing

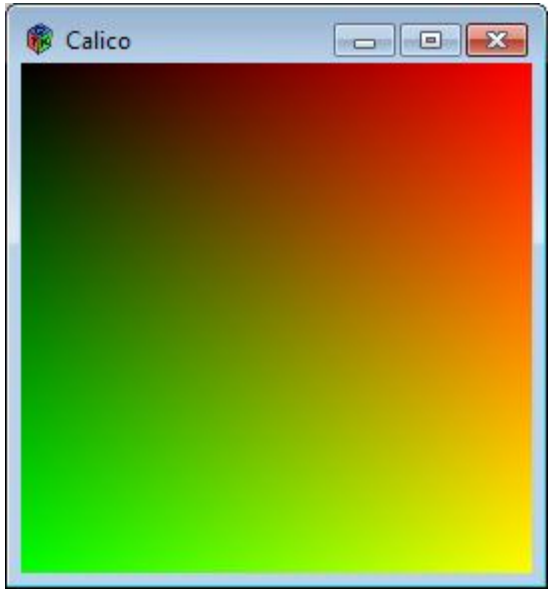
Accessing the pixels of a sketch

- getPixel(i, j)
  - Gets and returns the color at the pixel located at column i, row j.

- setPixel(i, j, color)
  - Sets the color of the pixel at column i, row j, to color, a color object.

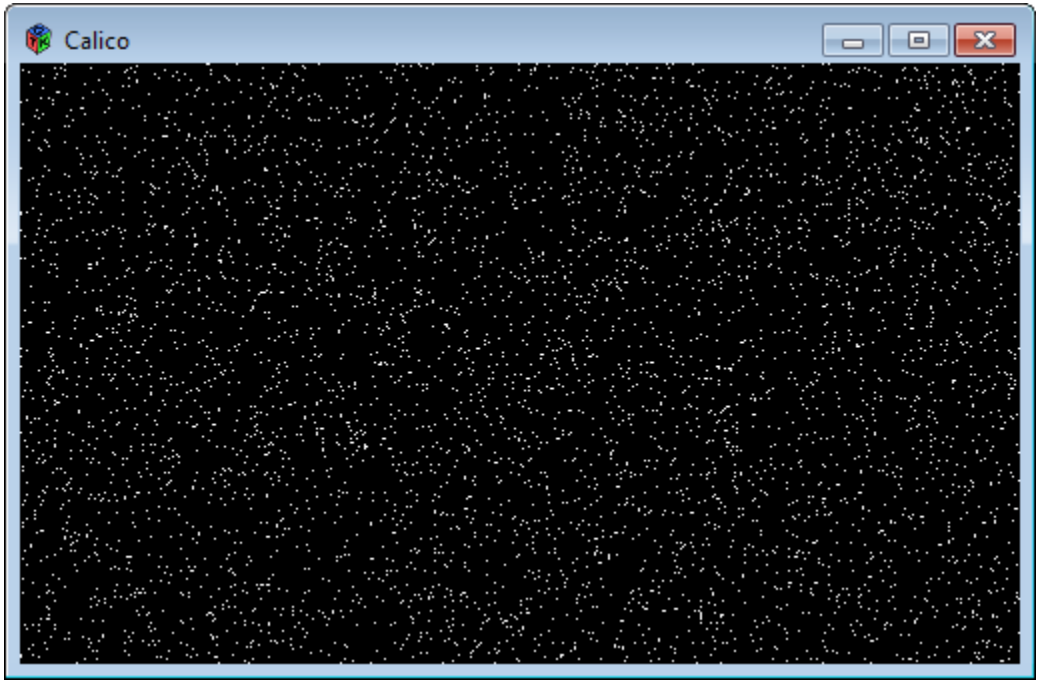
loadPixels() and updatePixels() must bracket all image processing

```
# shade.py
from Processing import *
window(255, 255)

# Load the pixels
loadPixels()

# Set pixel grayscale value to row index
for i in range(255):
    for j in range(255):
        clr = color(i, j, 0)
        setPixel(i, j, clr)

# Update pixels
updatePixels()
```



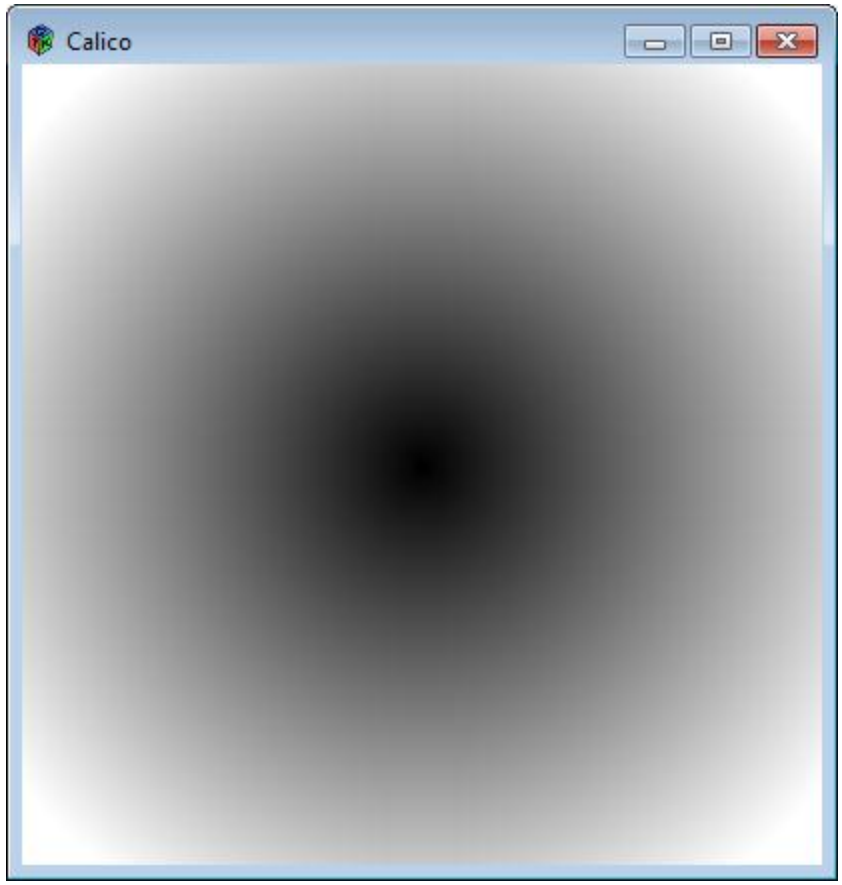
shade.py

```
# noise.py
from Processing import *
window(500, 300)

# Delay all screen updates
# until redraw is called
immediateMode(False)

# Generate the color white
white = color(255)

def noise():
    background(0)
    loadPixels()
    for i in range(5000):
        x = int( random(500) )
        y = int( random(300) )
        setPixel(x, y, white)
    updatePixels()

    # Render current sketch window
    redraw()

# Generate 1000 frames of random noise
for i in range(1000):
    noise()
```



See also colorNoise.py

```
# cone.py
from Processing import *
window(400, 400)

# Get pixel colors from window
loadPixels()

for i in range( 400 ):
    for j in range( 400 ):
        # Compute distance to center.
        b = dist( i, j, 200, 200 )
        b = constrain(b, 0, 255)

        # Set pixel color
        c = color( b )
        setPixel(i, j, c)

# Repaint
updatePixels()
```



```
# ripple.py
from Processing import *
import math

window(400, 400)

# Get pixel colors from window
loadPixels()

for i in range( 400 ):
    for j in range( 400 ):
        # Compute distance to center of sket
        d = dist( i, j, 200, 200 )

        # Compute the sine of the distance a
        b = math.sin(d/5.0)
        b = ((200.0-d)/200.0)*map(b, -1.0, 1.0, 0, 255);

        # Constrain distance to byte range
        b = constrain(b, 0, 255)

        # Set pixel color
        c = color( b )
        setPixel(i, j, c)

# Copy to sketch
updatePixels()
```

Rendering Images in a Sketch

- Image data can be loaded from a file using `loadImage()`, and drawn on a sketch with the `image()` command

```
img = loadImage("myImage.jpg")
image(img, 0, 0)
```

- The `loadImage()` command returns a PImage object (stored in `img` above)

- The PImage object also permits individual pixel color data to be modified.
  - like the sketch window

Pimage Object

Methods

width()	- Returns the width of the image
height()	- Returns the height of the image
loadPixels()	- Loads color data into buffer
updatePixels()	- Copies color data to image
setPixel(i, j, c)	- Sets pixel at i, j to color c
getPixel(i, j)	- Gets pixel color at i, j
get(i, j, w, h)	- Gets part of an image at i, j
	- of width of w and height h

PImage objects are like in-memory sketch windows

```
# warhol.py
from Processing import *

# Load image three times
warhol_r = loadImage("images/andy-warhol2.jpg")
warhol_g = loadImage("images/andy-warhol2.jpg")
warhol_b = loadImage("images/andy-warhol2.jpg")

# Load all pixels to allow color manipulation
warhol_r.loadPixels()
warhol_g.loadPixels()
warhol_b.loadPixels()

# Create window three times image width
w = warhol_r.width()
h = warhol_r.height()
window(3*w, h)

…
```

```
…

# Load pixels in image to allow pixel manipulation
for i in range(w):
    for j in range(h):
        # Use only red color values
        c = warhol_r.getPixel(i, j)
        c = color( red(c), 0, 0 )
        warhol_r.setPixel(i, j, c)

        # Use only green color values
        c = warhol_g.getPixel(i, j)
        c = color( 0, green(c), 0 )
        warhol_g.setPixel(i, j, c)

        # Use only blue color values
        c = warhol_b.getPixel(i, j)
        c = color( 0, 0, blue(c) )
        warhol_b.setPixel(i, j, c)

…
```

…

```
# Update pixels in images
warhol_r.updatePixels()
warhol_g.updatePixels()
warhol_b.updatePixels()

# Draw modified images
image(warhol_r, 0, 0)
image(warhol_g, w, 0)
image(warhol_b, 2*w, 0)
```


Calico

Image Processing

Draw a red X over the entire sketch by setting pixels

```
# pixels1.py
from Processing import *

window(500, 500)
background(255)

r = color(255, 0, 0)

loadPixels()

for i in range(500):
    setPixel(i, i, r)
    setPixel(500-i, i, r)

updatePixels()
```

Image Processing

Draw a blue grid over entire sketch by setting pixels

```
# pixels2.py
from Processing import *

window(500, 500)
background(255)

b = color(0, 0, 255)

loadPixels()

for c in range(0, 500, 10):
    for r in range(500):
        setPixel(c, r, b)
        setPixel(r, c, b)

updatePixels()
```

Image Processing

Draw a circle by setting pixels

```
# pixels3.py
from Processing import *
from math import *

window(500, 500)
background(255)

b = color(0, 0, 255)

loadPixels()

for angle in range(0, 360):
    c = int( 100 * cos( radians( angle ) ) + 250 )
    r = int( 100 * sin( radians( angle ) ) + 250 )
    setPixel(c, r, b)

updatePixels()
```

Image Processing

Draw a horizontal gradient from blue to green by setting pixels

```
# pixels4.py
from Processing import *

window(255, 255)

loadPixels()

for c in range(255):
    for r in range(255):
        clr = color(0, c, 255-c)
        setPixel(c, r, clr)

updatePixels()
```

Image Processing

Read the blue and green color components of the last sketch at the center pixel

```
# pixels5.py
from Processing import *

window(255, 255)

loadPixels()

for c in range(255):
    for r in range(255):
        clr = color(0, c, 255-c)
        setPixel(c, r, clr)

updatePixels()

pxl = getPixel(127, 127)
bl = blue(pxl)
gr = green(pxl)
print( "Blue component: %s" % bl )
print( "Green component: %s" % gr )
```