Exam 2 Review
Recursion, Transformations, Image Processing

Exam 2 Topics

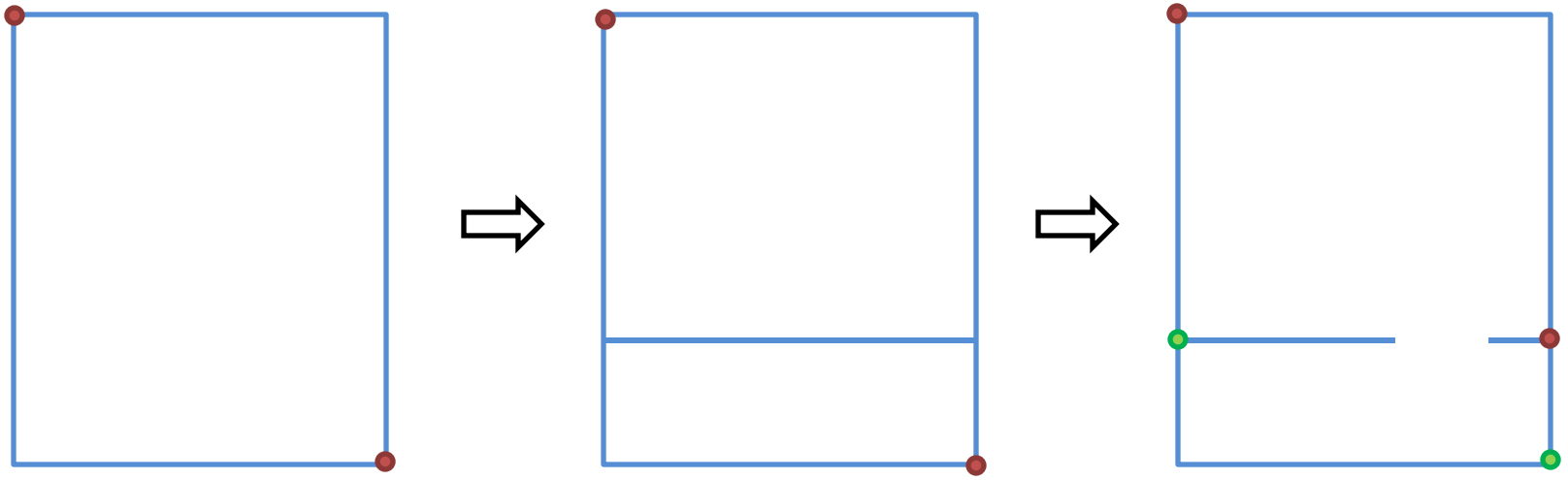
1. Strings and String Manipulation
2. Arrays and Random Numbers
3. Multidimensional Arrays
4. Transformations (Translation, Rotation, Scaling)
5. Sorting and Searching (Algorithms)
6. Image Processing
7. Recursion
8. Debugging

Also, review Exam 1.

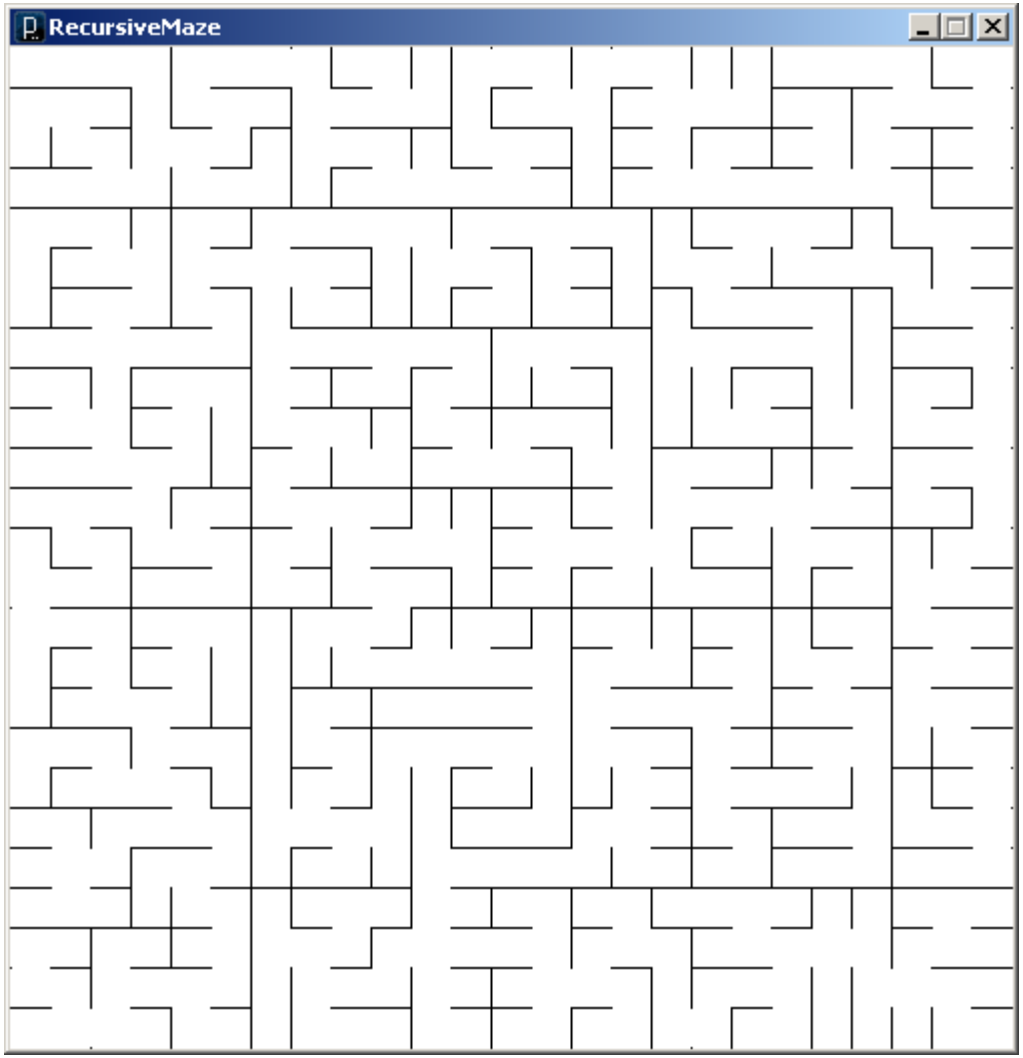
Creating a maze, recursively

1. Start with a rectangular region defined by its upper left and lower right corners
2. Divide the region at a random location through its more narrow dimension
3. Add an opening at a random location
4. Recursively, repeat procedure on two rectangular subregions

A recursive function calls itself.



Inspired by http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm


RecursiveMaze

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1$

$4! = 4 \times 3 \times 2 \times 1$

$5! = 5 \times 4!$

$N! = N \times (N-1)!$

Factorial can be defined in terms of itself

$4! = 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$5! = 5 \times 4!$

$4! = 4 \times 3!$

$3! = 3 \times 2!$

$2! = 2 \times 1$

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10		

Last In First Out (LIFO) Stack of Plates

Call Stack

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10		
factorial(9)	9		

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10		
factorial(9)	9		
factorial(8)	8		

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10		
factorial(9)	9		
factorial(8)	8		
factorial(7)	7		
factorial(6)	6		
factorial(5)	5		
factorial(4)	4		
factorial(3)	3		
factorial(2)	2		
factorial(1)	1		
factorial(0)	0	--	1

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10		
factorial(9)	9		
factorial(8)	8		
factorial(7)	7		
factorial(6)	6		
factorial(5)	5		
factorial(4)	4		
factorial(3)	3		
factorial(2)	2		
factorial(1)	1	1	1
factorial(0)	0	--	1

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10		
factorial(9)	9		
factorial(8)	8		
factorial(7)	7		
factorial(6)	6		
factorial(5)	5		
factorial(4)	4		
factorial(3)	3		
factorial(2)	2	1	2
factorial(1)	1	1	1
factorial(0)	0	--	1

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10		
factorial(9)	9		
factorial(8)	8		
factorial(7)	7		
factorial(6)	6		
factorial(5)	5		
factorial(4)	4		
factorial(3)	3	2	6
factorial(2)	2	1	2
factorial(1)	1	1	1
factorial(0)	0	--	1

```
// Compute factorial of a number
// Recursive implementation

void setup() {}
void draw() {}

void mousePressed() {
  int f = factorial(10);
  println(f);
}

int factorial( int i)
{
  if( i == 0) {
    return 1;
  } else {
    int fim1 = factorial(i-1);
    return i*fim1;
  }
}
```

Call	i	fim1	returns
factorial(10)	10	362880	3628800
factorial(9)	9	40320	362880
factorial(8)	8	5040	40320
factorial(7)	7	720	5040
factorial(6)	6	120	720
factorial(5)	5	24	120
factorial(4)	4	6	24
factorial(3)	3	2	6
factorial(2)	2	1	2
factorial(1)	1	1	1
factorial(0)	0	--	1

```
String[] parts = new String[] {"a", "b", "c", "d"};

void setup() {}
void draw() {}

void mousePressed() {
  String joined = reverseJoin(3);
  println( joined );
}

String reverseJoin( int i ) {
  if (i == 0)
  {
    return parts[0];
  }
  else
  {
    String rjim1 = reverseJoin(i-1)
    return parts[i] + rjim1;
  }
}
```

Call	i	parts[i]	rjim1	returns
reverseJoin(3)	3	"d"	"cba"	"dcba"
reverseJoin(2)	2	"c"	"ba"	"cba"
reverseJoin(1)	1	"b"	"a"	"ba"
reverseJoin(0)	0	"a"	--	"a"

Three ways to transform the coordinate system:

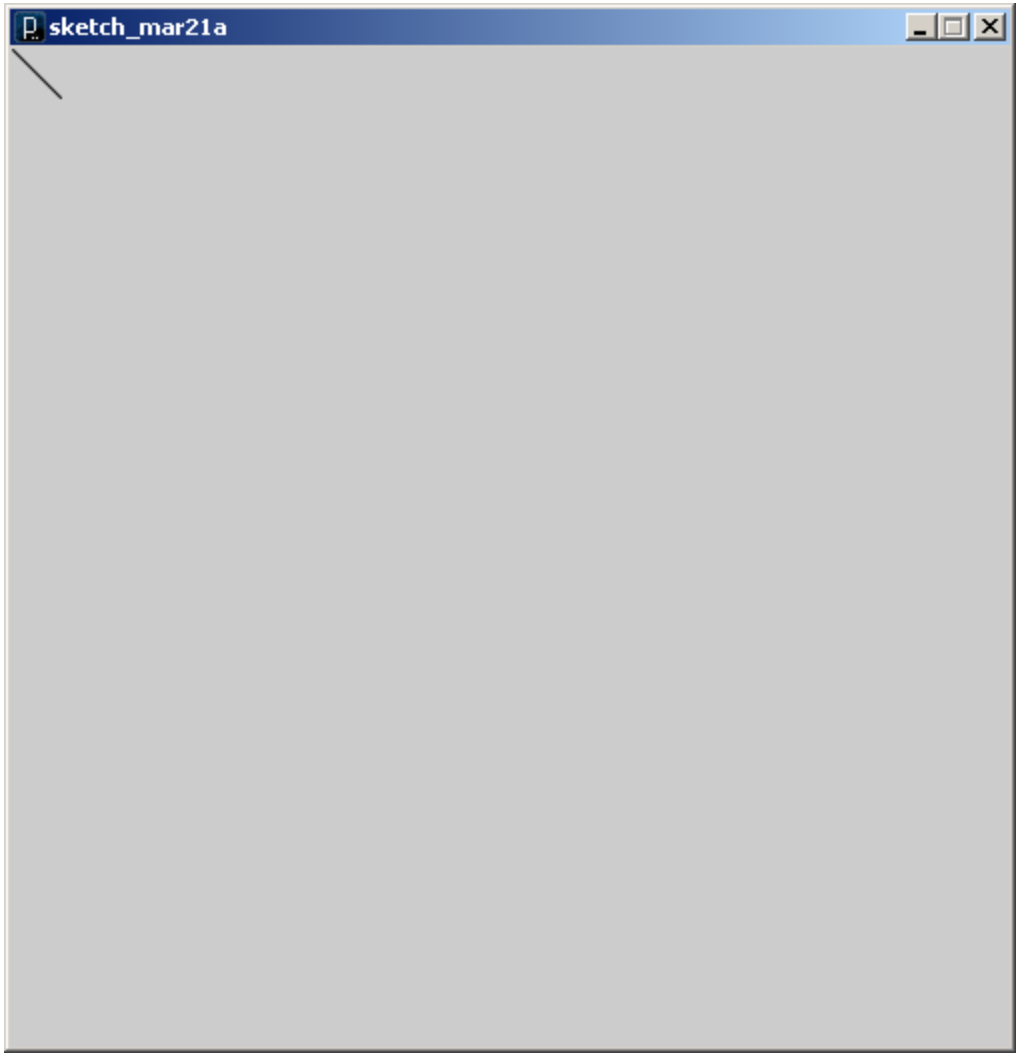
1. **Translate**
   - Move axes left, right, up, down …
2. **Scale**
   - Magnify, zoom in, zoom out …
3. **Rotate**
   - Tilt clockwise, tilt counter-clockwise …

Scale

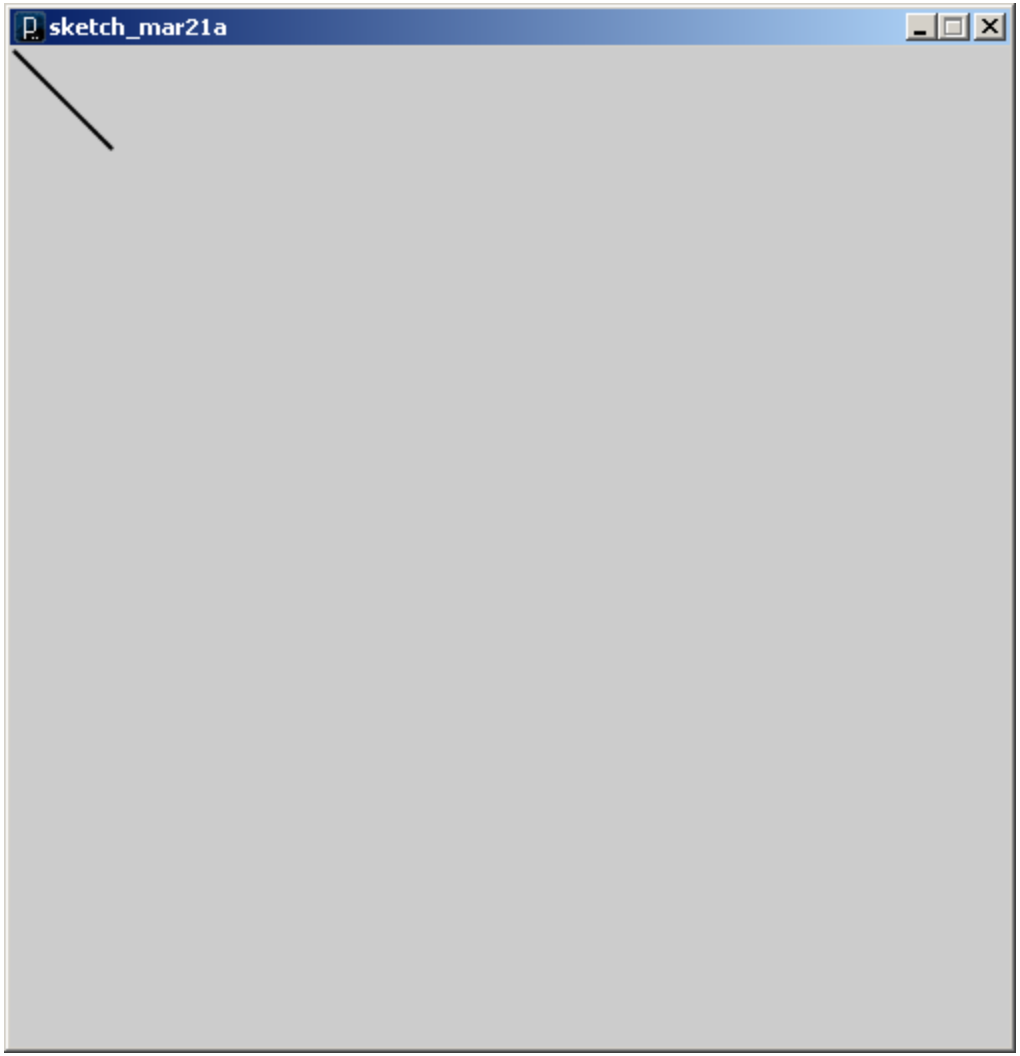
- All coordinates are multiplied by an x-scale-factor and a y-scale-factor.
- The size of everything is magnified about the origin (0,0)
- Stroke thickness is also scaled.

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

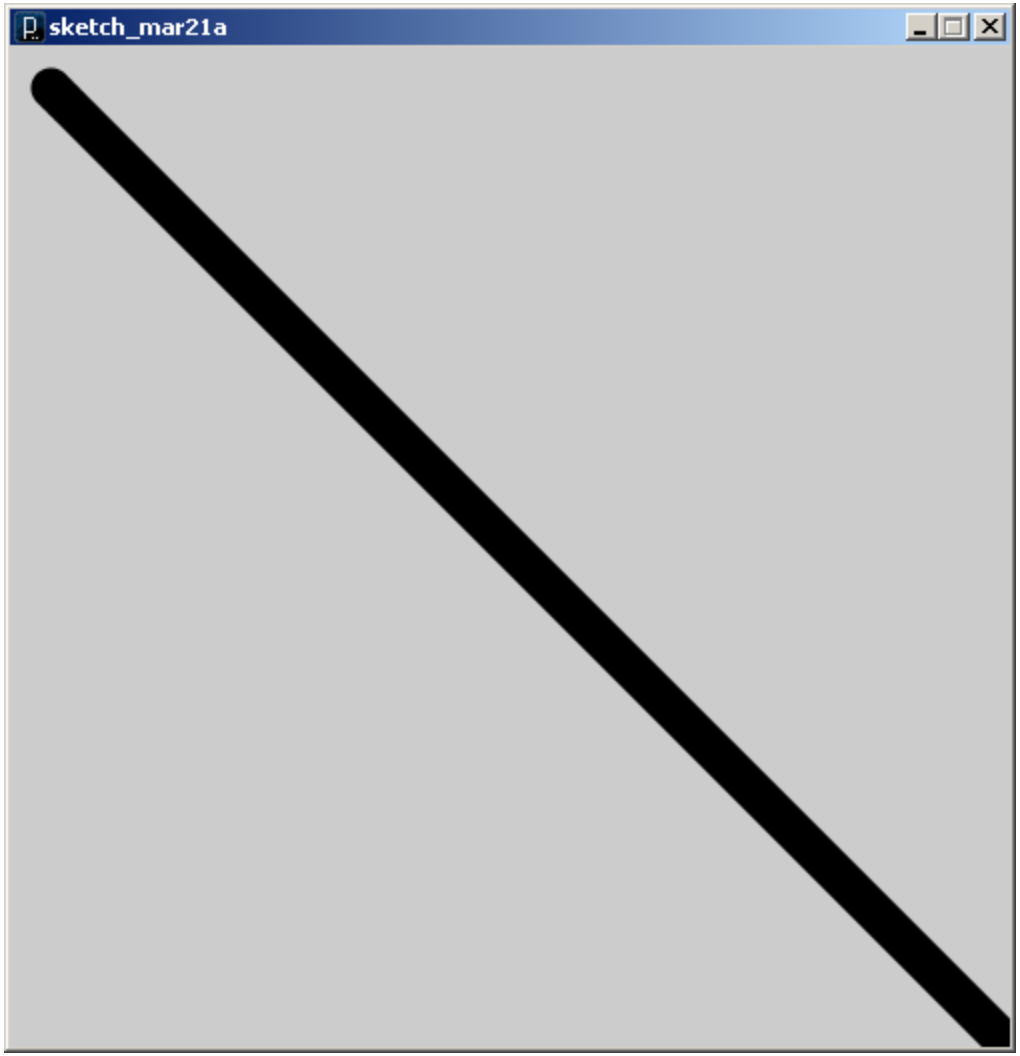
```
void setup() {
  size(500, 500);
  smooth();
  noLoop();

  line(1, 1, 25, 25);
}
```



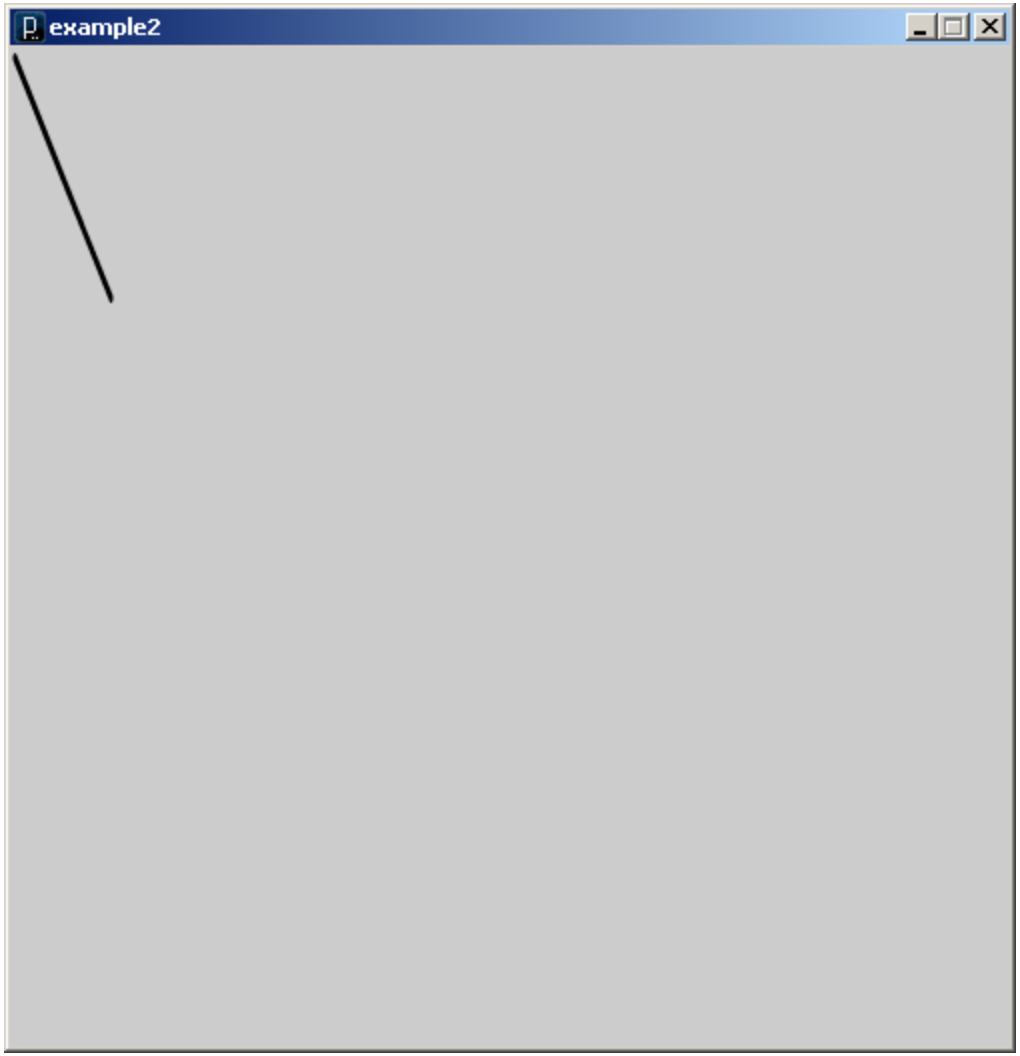
```
void setup() {
  size(500, 500);
  smooth();
  noLoop();

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



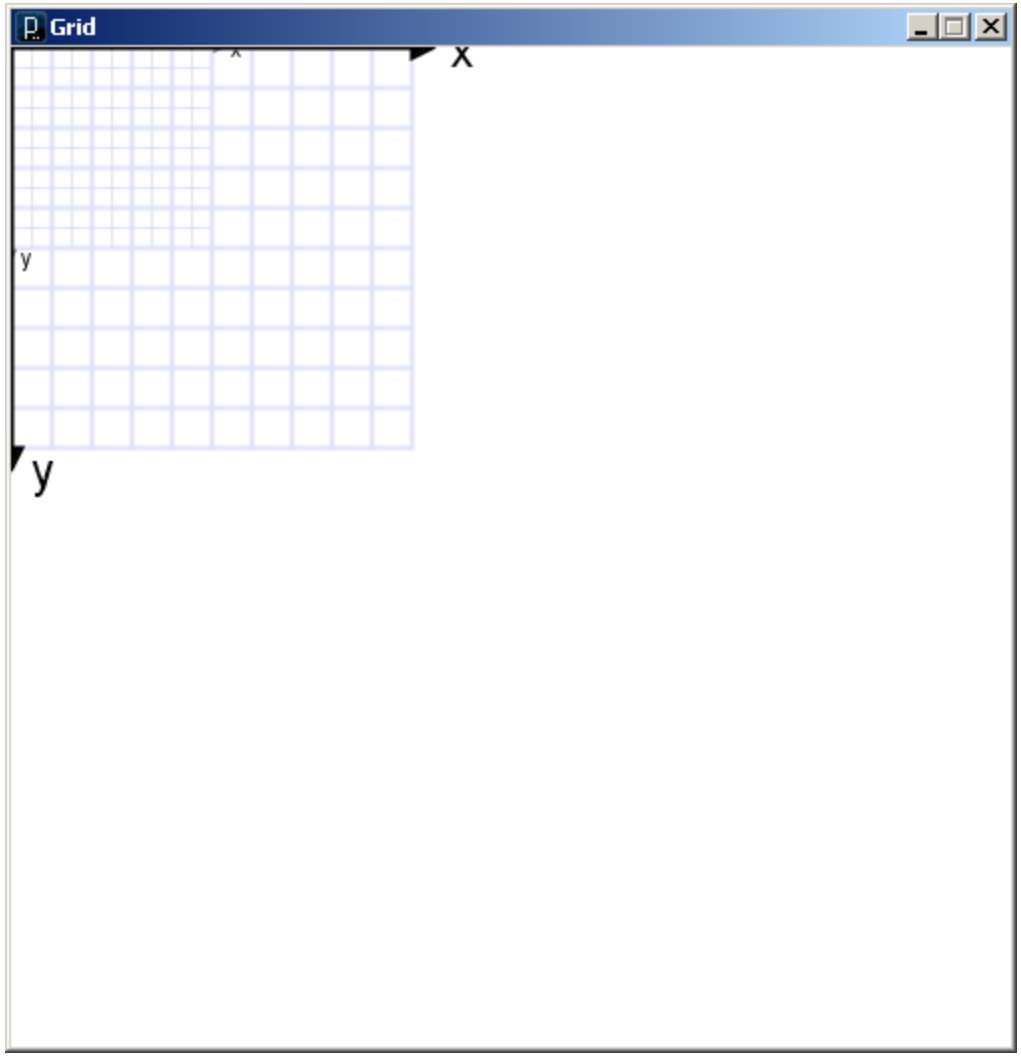
```
void setup() {
  size(500, 500);
  smooth();
  noLoop();

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



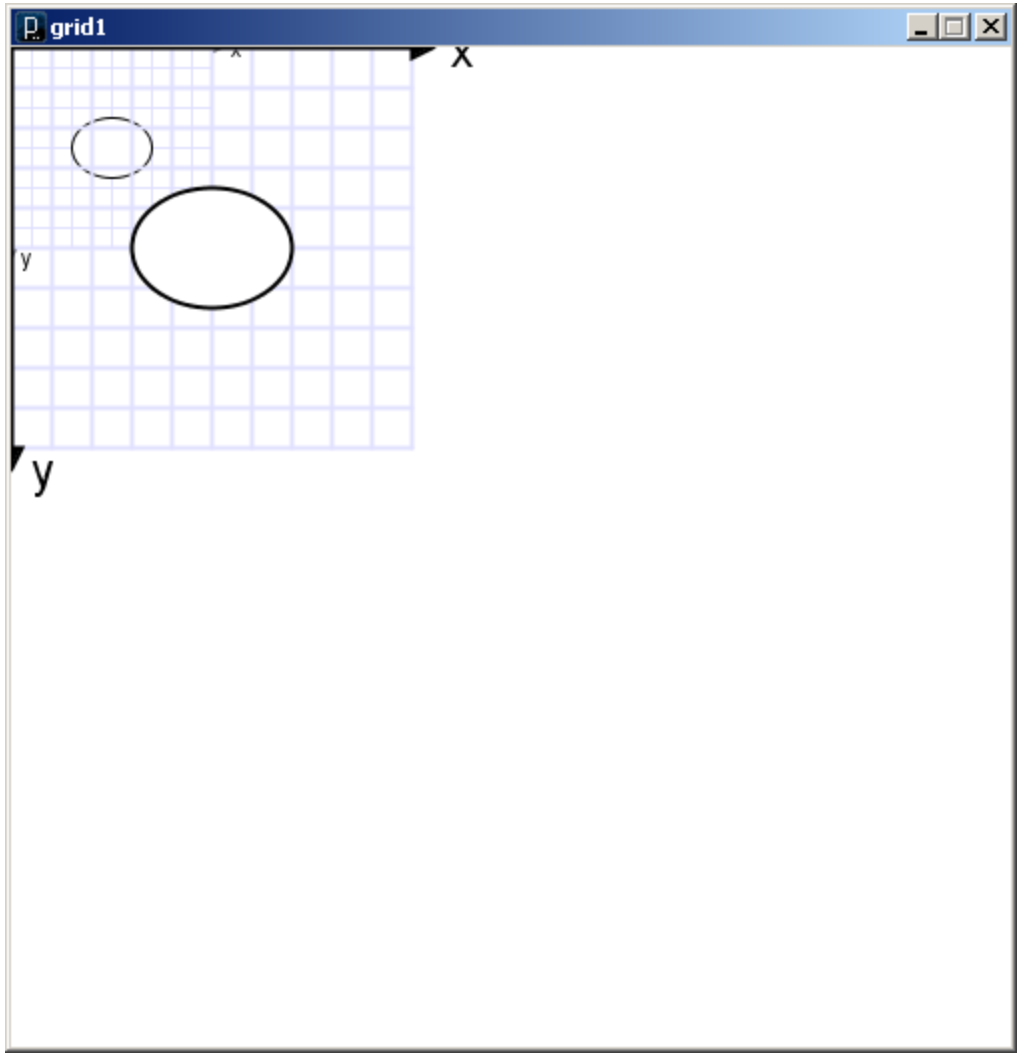
```
void setup() {
  size(500, 500);
  smooth();
  noLoop();

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



```
void setup() {
  size(500, 500);
  background(255);
  smooth();
  noLoop();
}

void draw() {
  grid();
  scale(2,2);
  grid();
}
```



```
void draw() {
  grid();
  fill(255);
  ellipse(50,50,40,30);

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

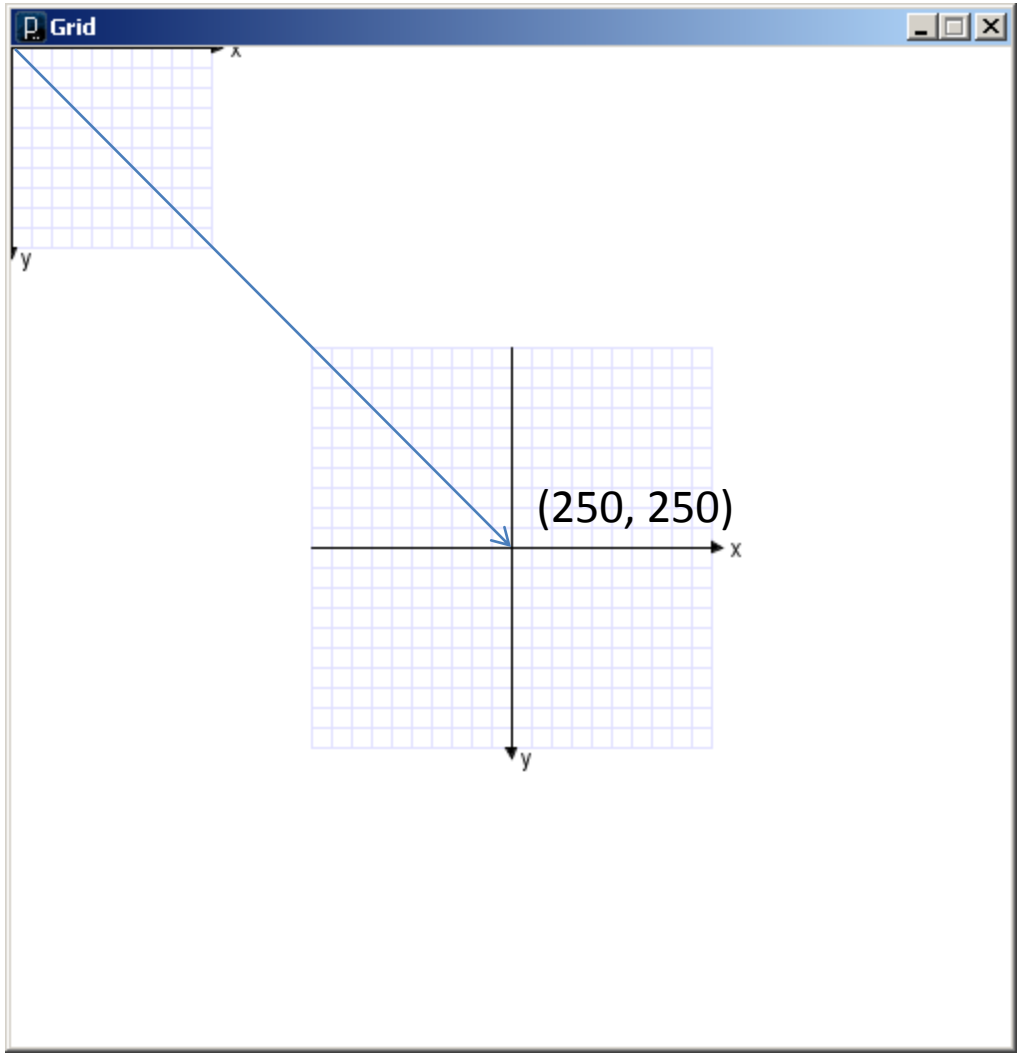


Translate

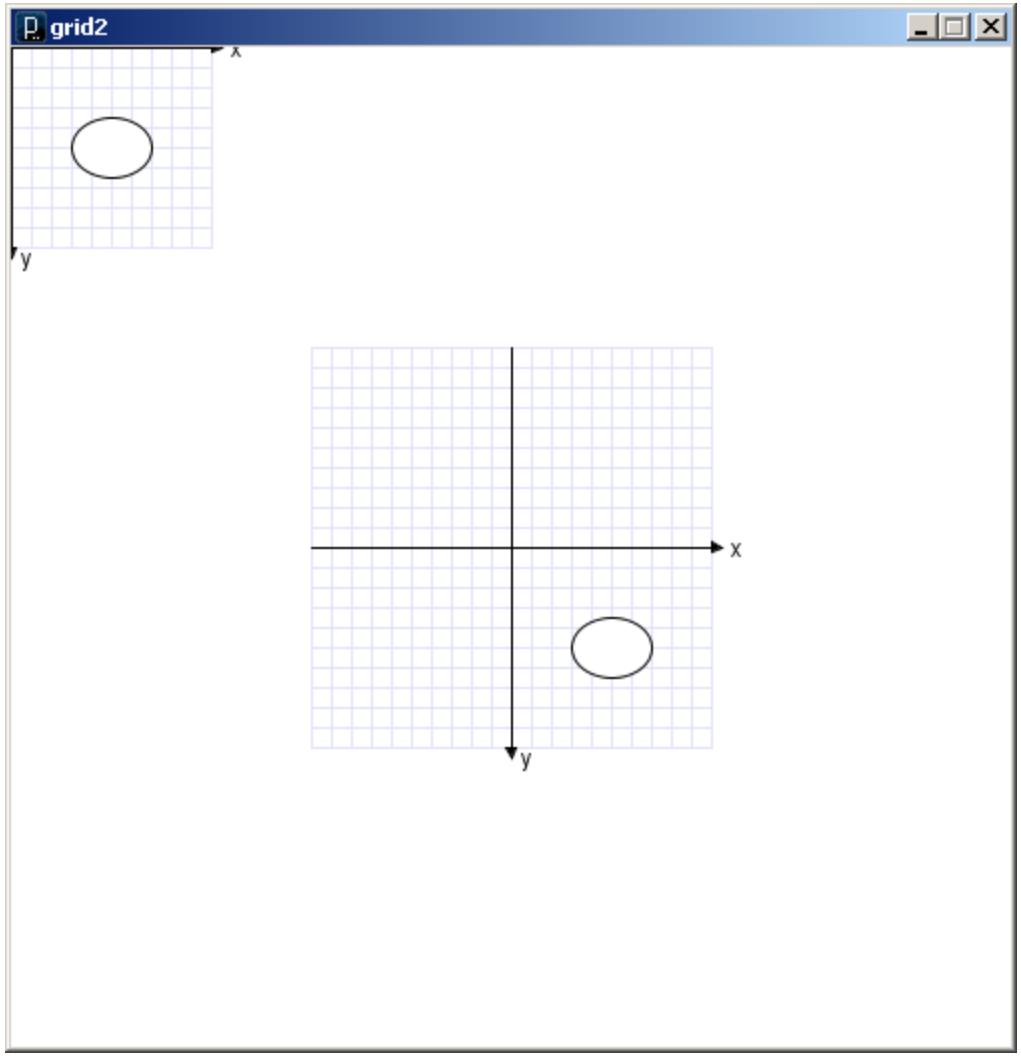
- The origin of the coordinate system (0,0) is shifted by the given amount in the x and y directions.

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

```
void draw() {
  grid();
  translate(250,250);
  grid();
}
```



```
void draw() {
  grid();
  fill(255);
  ellipse(50, 50, 40, 30);

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

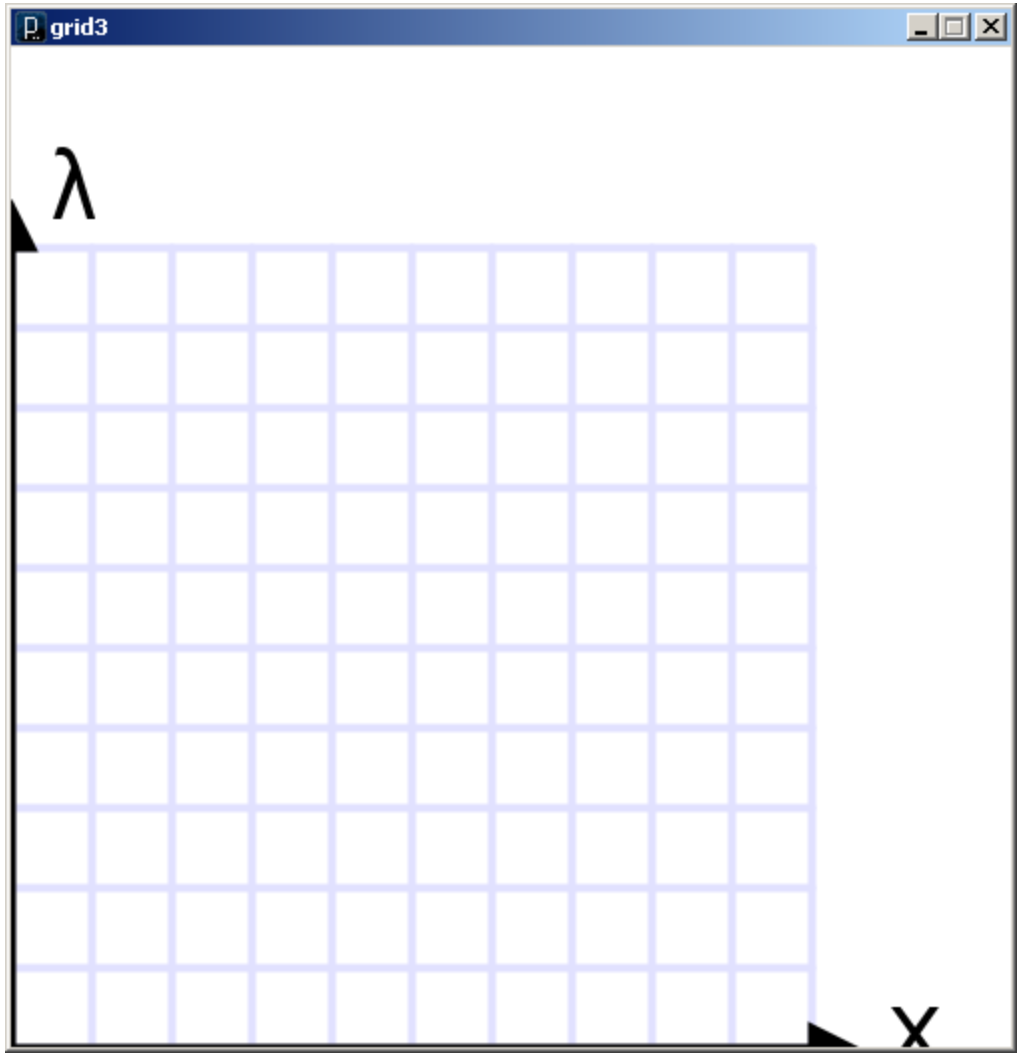


Transformations can be combined

- 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!

```
void draw() {
  translate(0,height);
  scale(4,-4);
  grid();
}
```

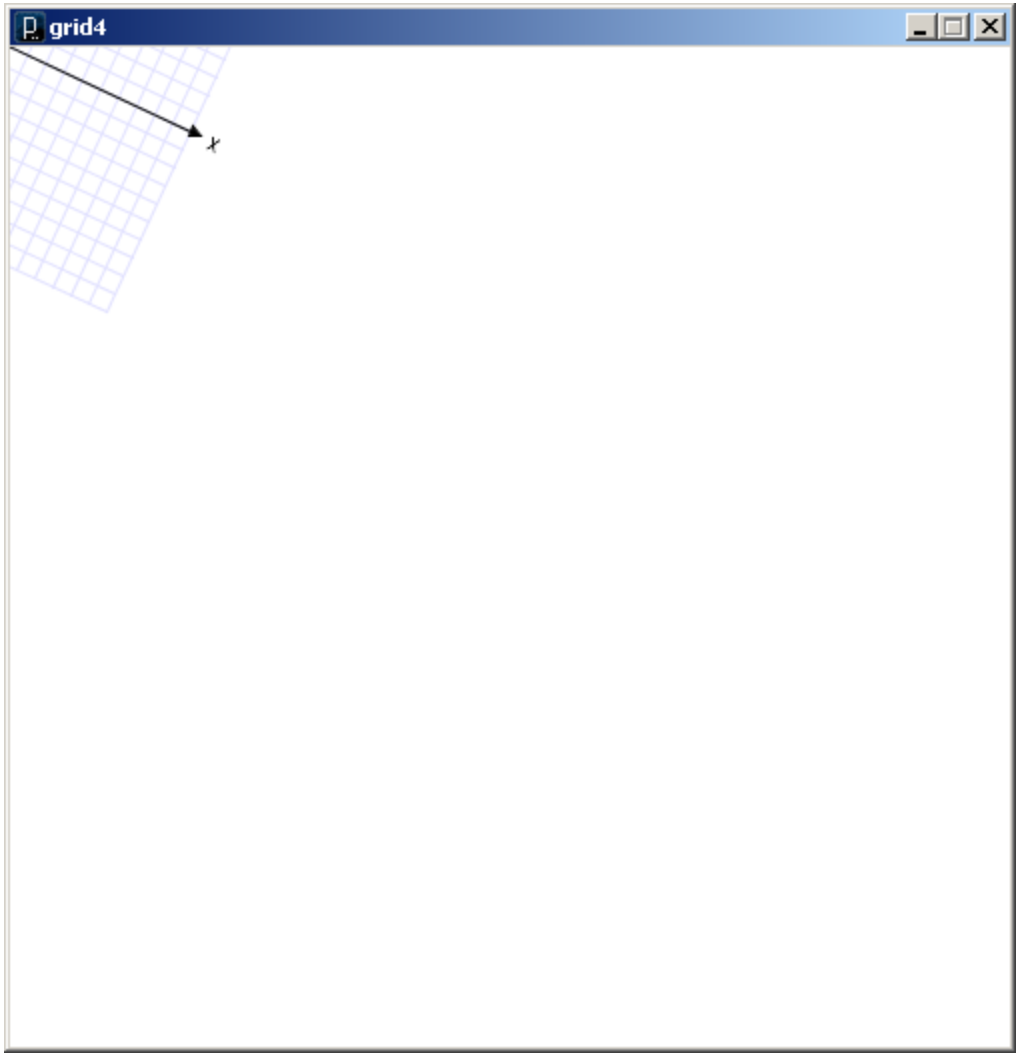


Rotate

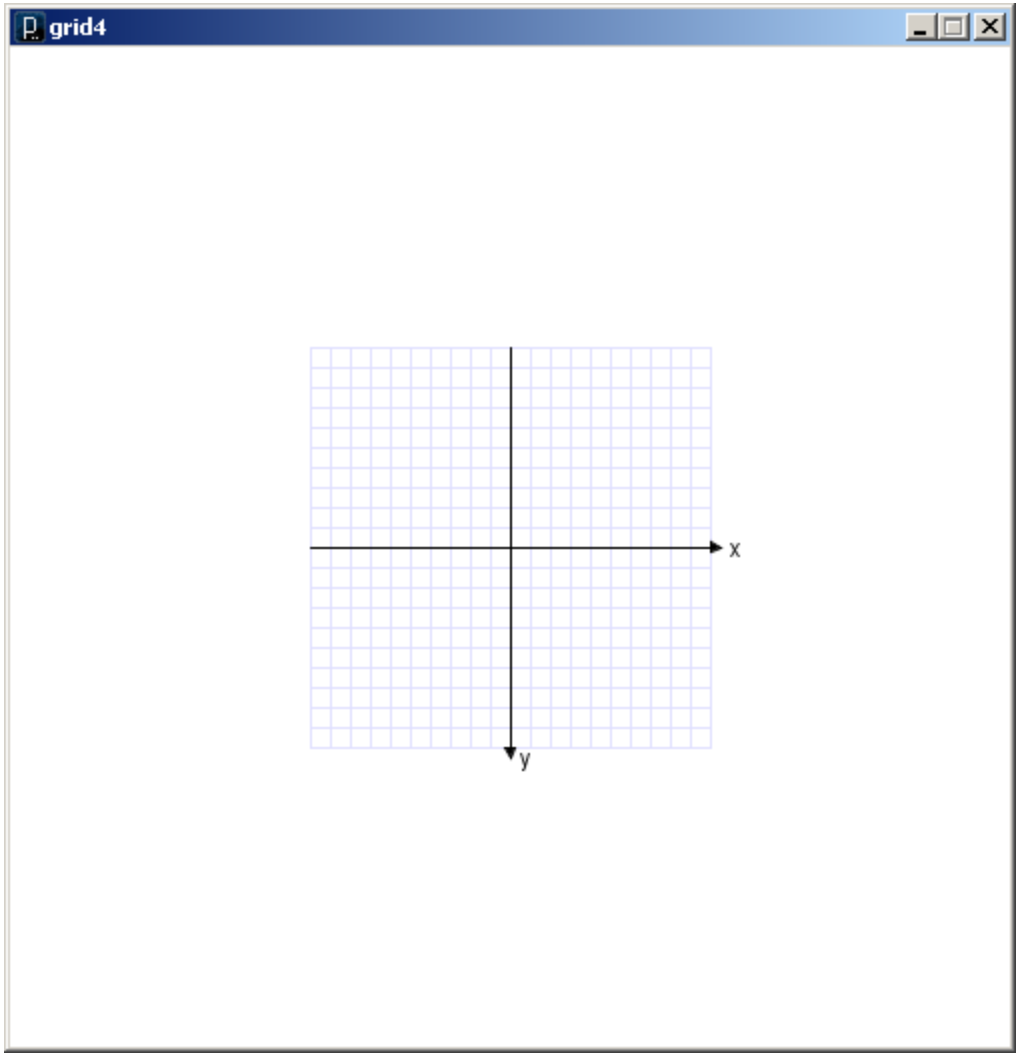
- The coordinate system is rotated around the origin by the given angle (in radians).

```
rotate( radians );
```

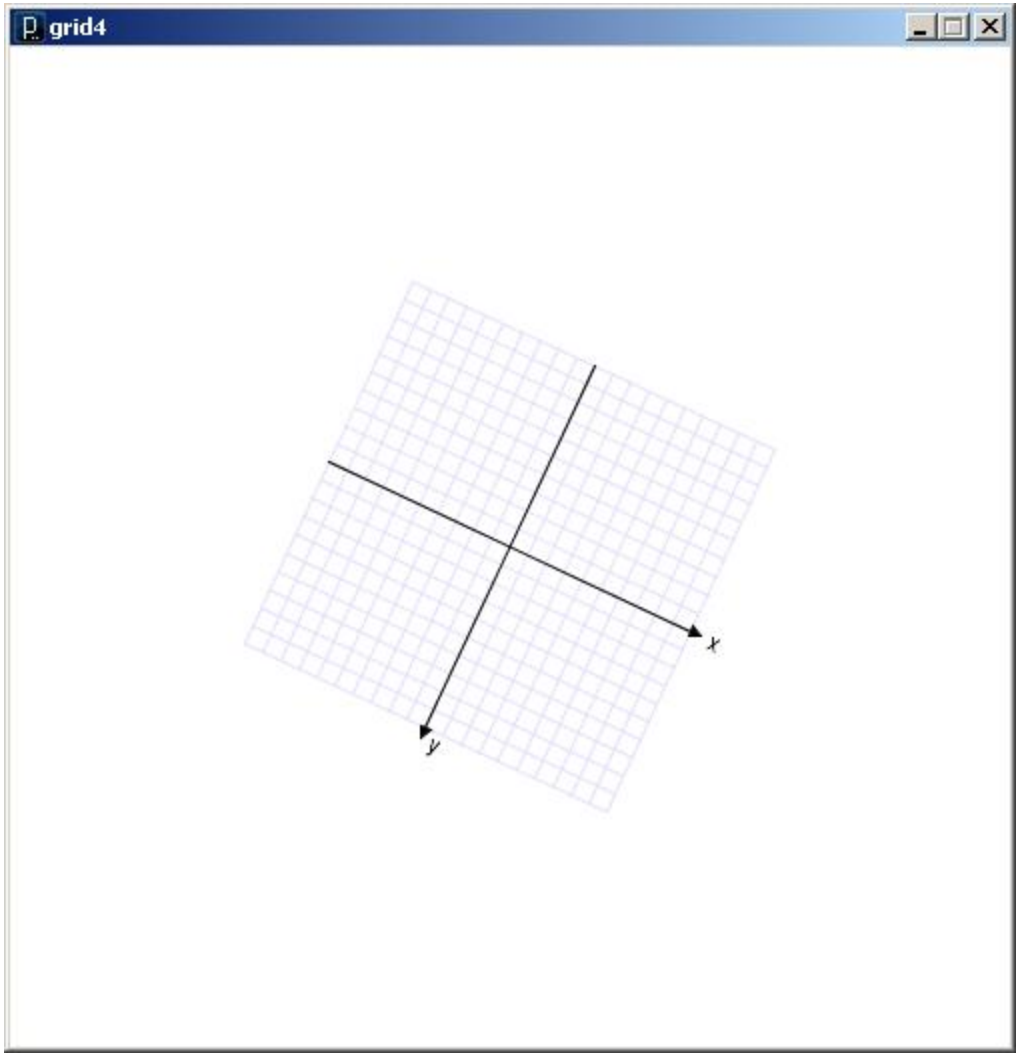
```
void draw() {
  rotate( 25.0 * (PI/180.0) );
  grid();
}
```



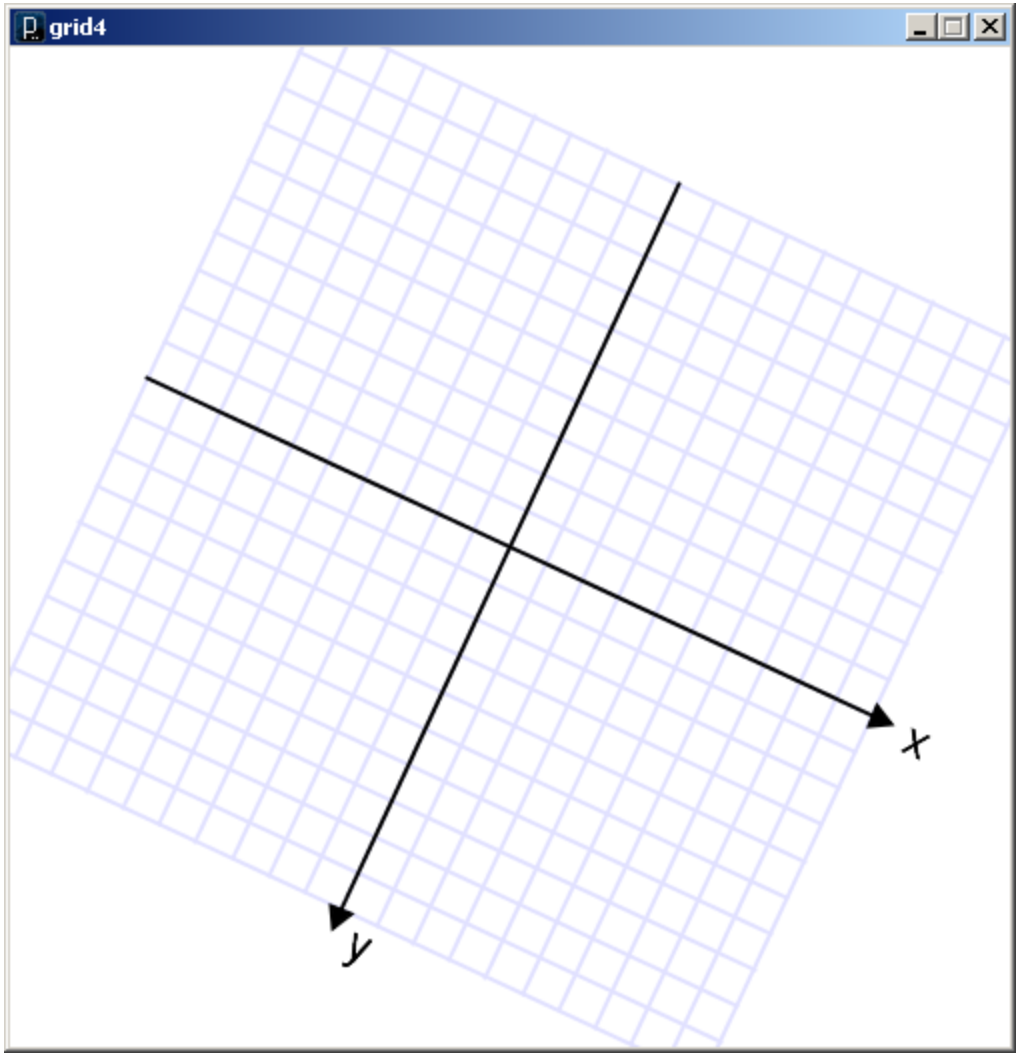
```
void draw() {
  translate(250.0, 250.0);
  //rotate( 25.0 * (PI/180.0) );
  //scale( 2 );
  grid();
}
```



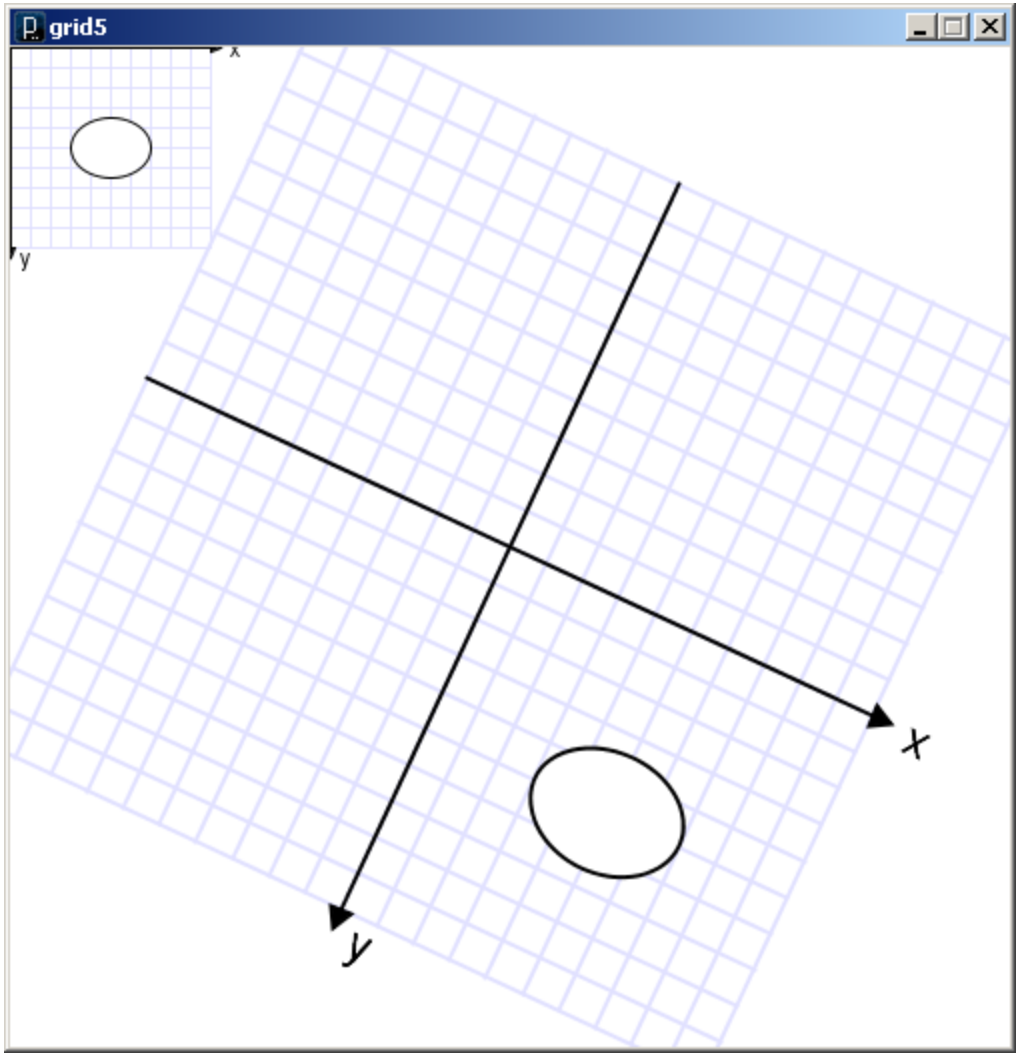
```
void draw() {
  translate(250.0, 250.0);
  rotate( 25.0 * (PI/180.0) );
  //scale( 2 );
  grid();
}
```



```
void draw() {
  translate(250.0, 250.0);
  rotate( 25.0 * (PI/180.0) );
  scale( 2 );
  grid();
}
```



```
void draw() {
  grid();
  fill(255);
  ellipse(50, 50, 40, 30);

  translate(250.0, 250.0);
  rotate( 25.0 * (PI/180.0) );
  scale(2);
  grid();
  fill(255);
  ellipse(50, 50, 40, 30);
}
```



grid5.pde

Last In First Out (LIFO) Stack of Plates

Transformation Matrix Stack

Transformation Matrix Stack

- Transformation matrices can be managed using the **Matrix Stack.** (Recall, a stack is LIFO)
- The current transformation matrix can be temporarily pushed on to the Matrix Stack, and popped off for use later on.
- The Matrix Stack can hold multiple transformation matrices.
- Enables the idea of recursive drawing coordinate systems
    - … when you want to draw a part of something w.r.t. that something's master coordinate system

pushMatrix()

- Pushes a copy of the current transformation matrix onto the Matrix Stack

popMatrix()

- Pops the last pushed transformation matrix off the Matrix Stack and replaces the current matrix

resetMatrix()

- Replaces the current transformation matrix with the identity matrix

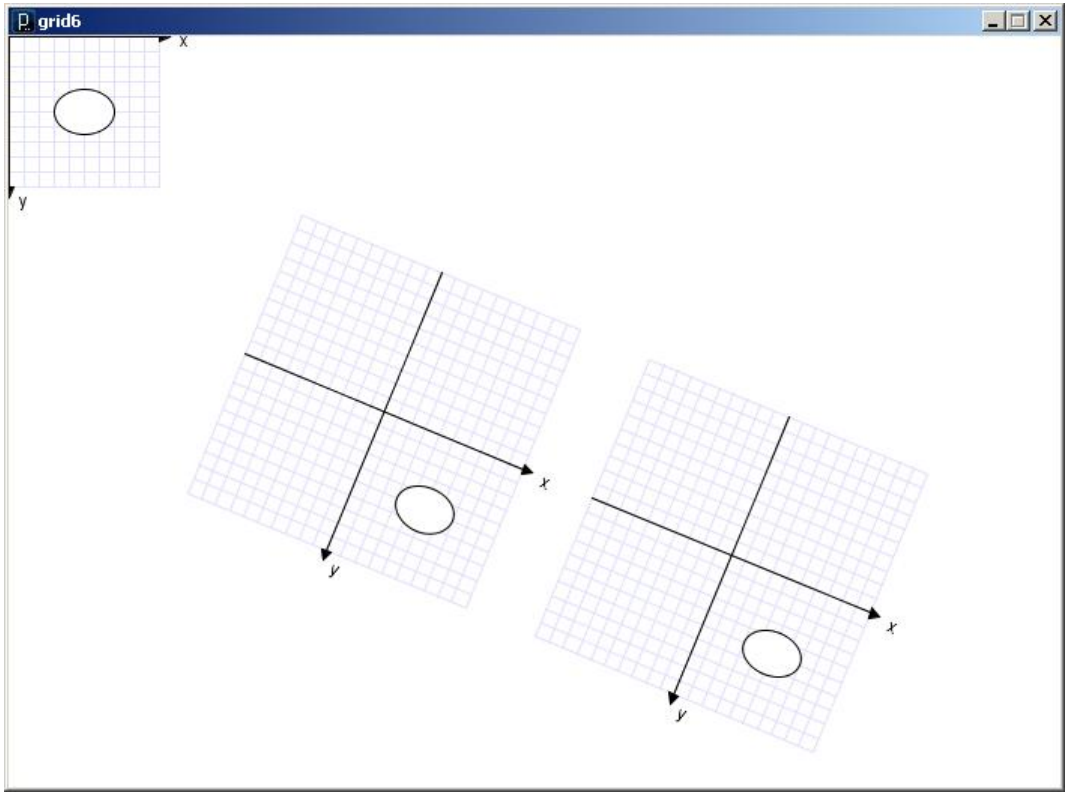
applyMatrix()

- Multiplies the current transformation matrix with a given custom matrix.

printMatrix()

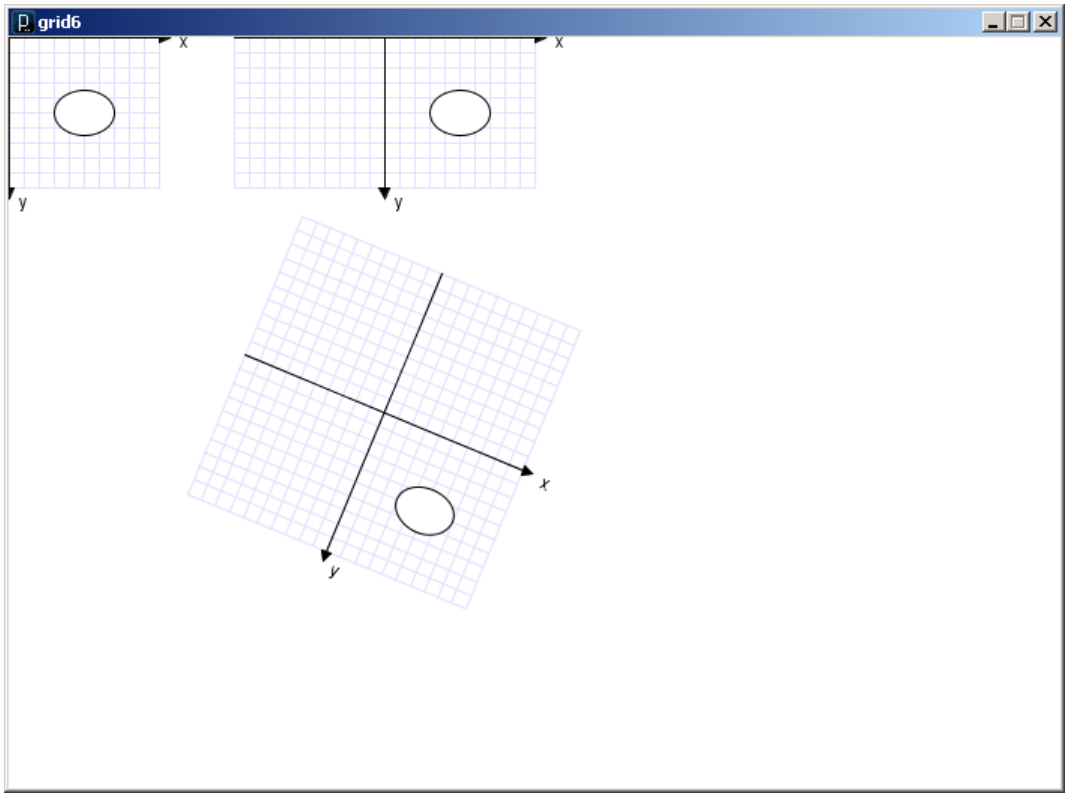
- Prints the current transformation matrix in effect.

```
void draw() {
  grid();
  fill(255);
  ellipse(50, 50, 40, 30);


  translate(250, 250);
  rotate(PI/8.0);

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


  translate(250, 0);

  grid();
  fill(255);
  ellipse(50, 50, 40, 30);
}
```



relXform.pde

```
void draw() {
  grid();
  fill(255);
  ellipse(50, 50, 40, 30);

  pushMatrix();
  translate(250, 250);
  rotate(PI/8.0);

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

  translate(250, 0);

  grid();
  fill(255);
  ellipse(50, 50, 40, 30);
}
```

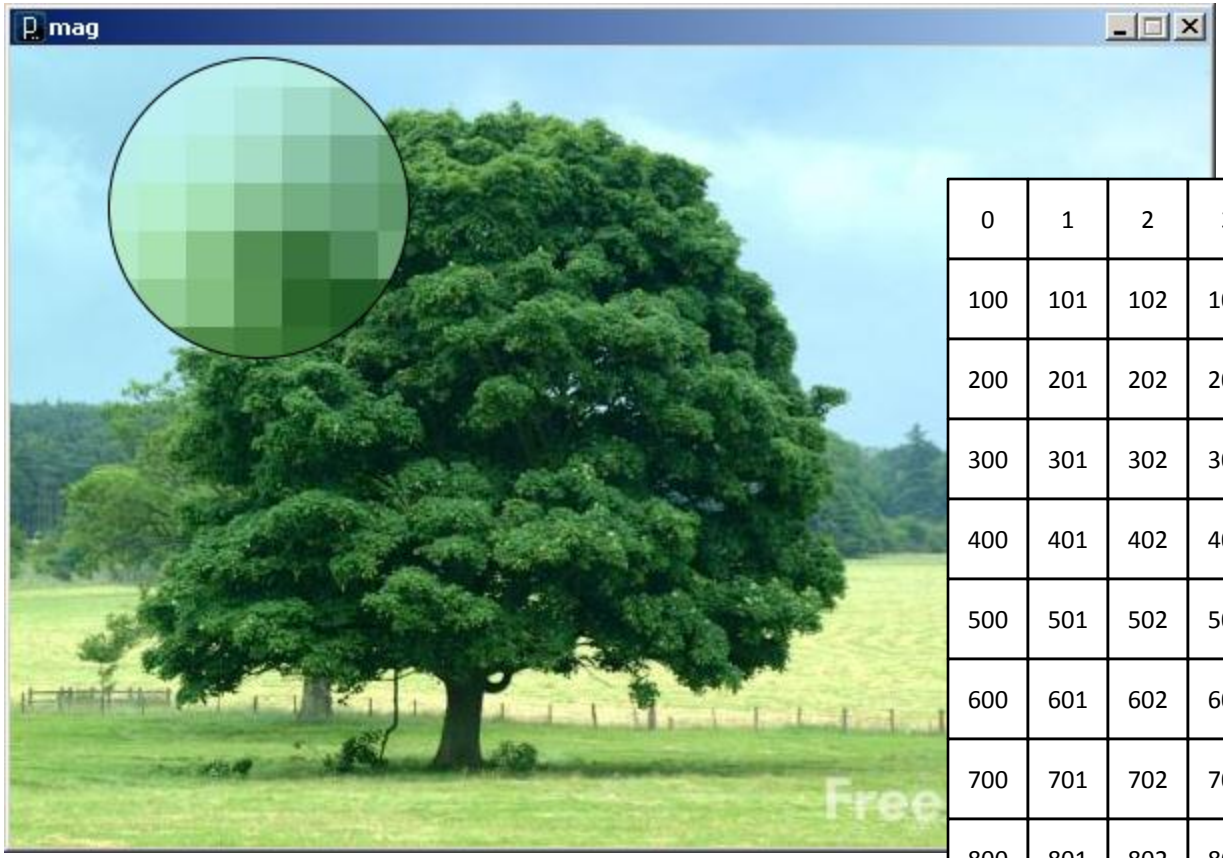


relXform.pde

Some things to remember:

1. Transformations are cumulative.
2. All transformations are cancelled each time draw() exits.
   - They must be reset each time at the beginning of draw() before any drawing.
3. Rotation angles are measured in radians
   - π radians = 180°
   - radians = (PI/180.0) * degrees
4. Order matters

An image is an array of colors



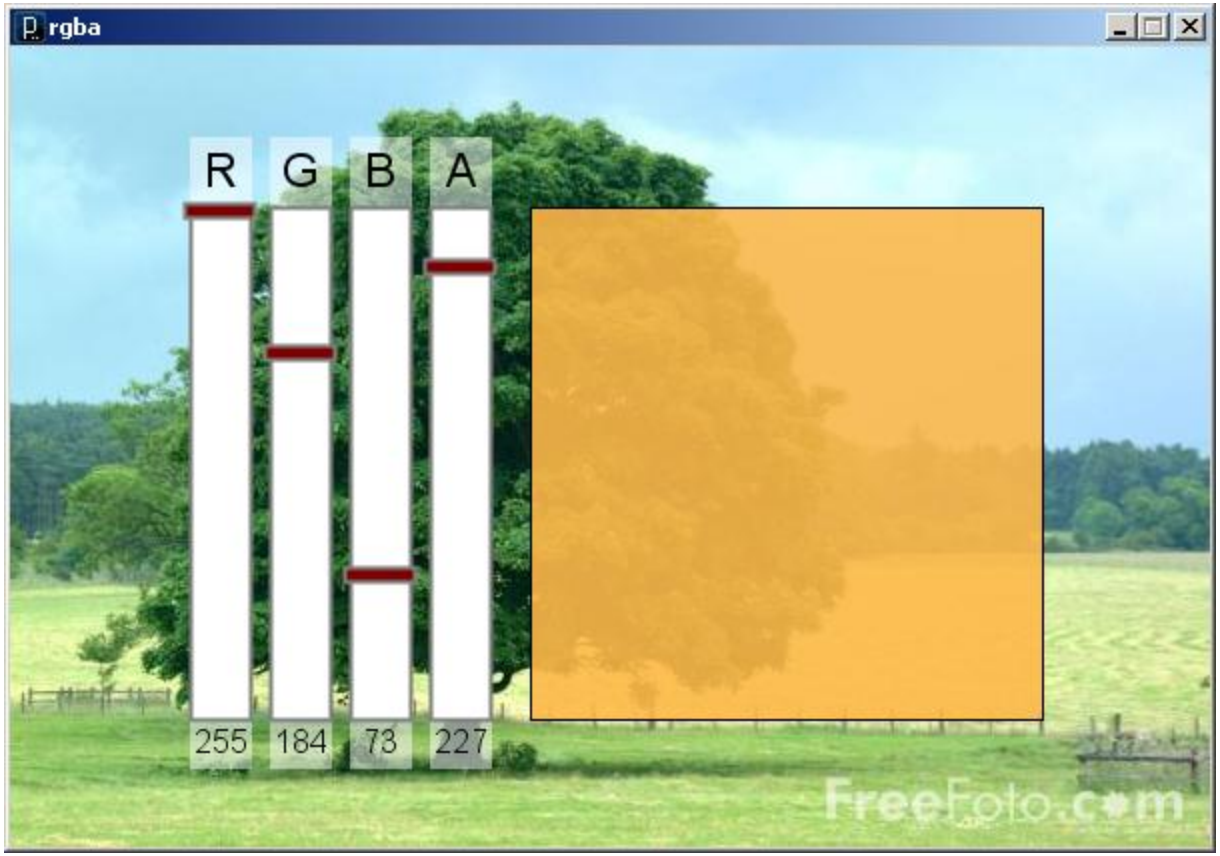
0	1	2	3	…		98	99
100	101	102	103	…		198	199
200	201	202	203	…		298	299
300	301	302	303	…		398	399
400	401	402	403	…		498	499
500	501	502	503	…		598	599
600	601	602	603	…		698	699
700	701	702	703	…		798	799
800	801	802	803	…		898	899
⋮	⋮	⋮	⋮	…		⋮	⋮

Pixel : Picture Element

mag.pde

Color

- A triple of bytes [0, 255]
  - RGB or HSB
- Transparency (alpha)
  - How to blend a new pixel color with an existing pixel color



rgba.pde

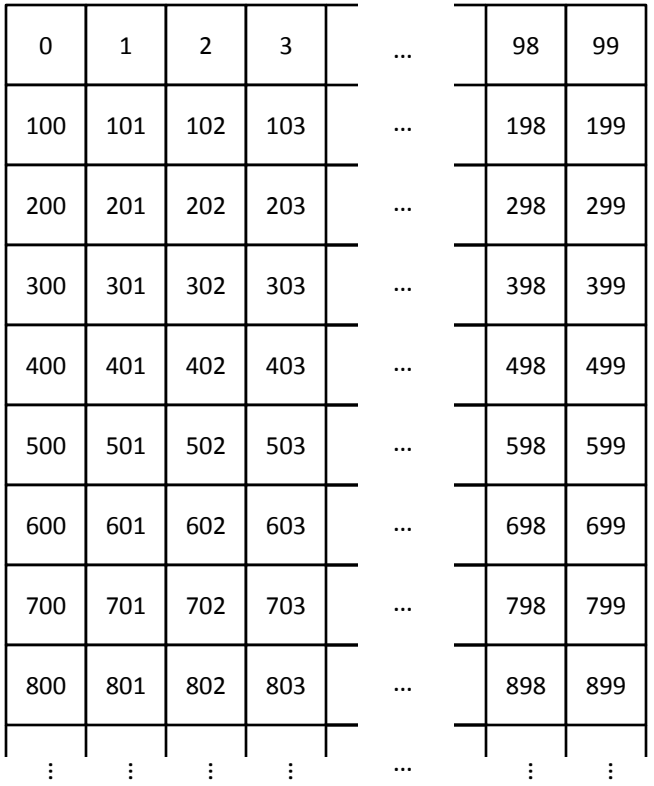
Accessing the pixels of a sketch

- loadPixels()
  - Loads the color data out of the sketch window into a 1D array of colors named pixels[]
  - The pixels[] array can be modified

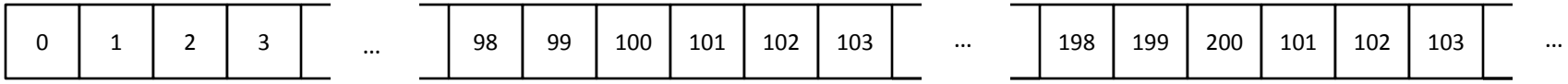
- updatePixels()
  - Copies the color data from the pixels[] array back to the sketch window

A 100-pixel wide image

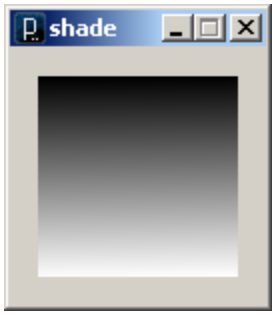
- First pixel at index 0
- Right-most pixel in first row at index 99
- First pixel of second row at index 100



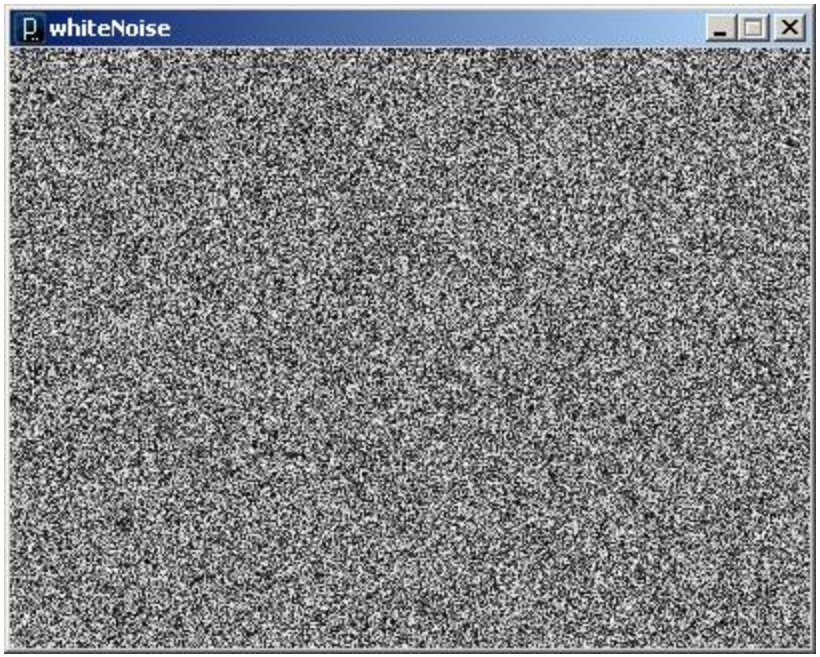
The pixels[] array is one-dimensional



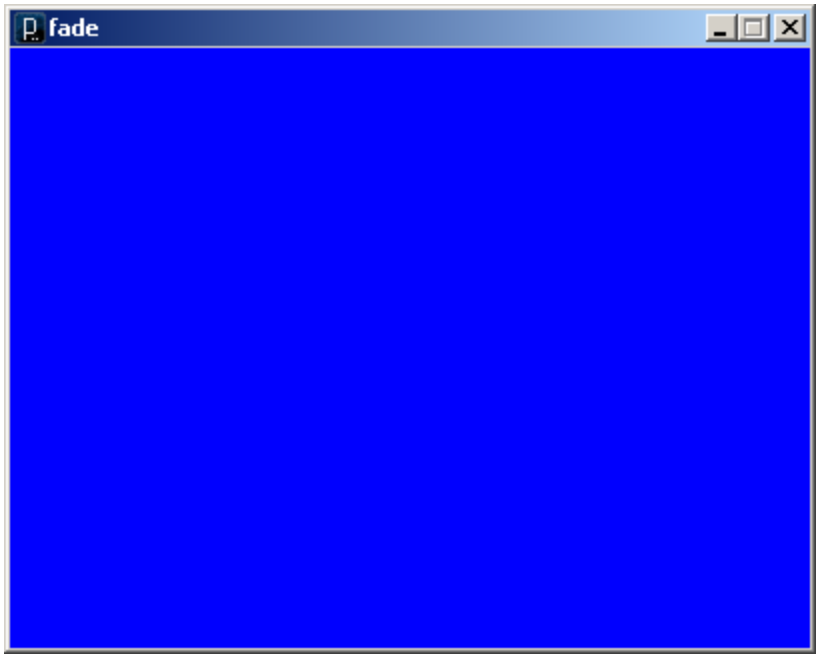
```
// shade
void setup() {
  size(100, 100);

  float b = 0.0;

  // Load colors into the pixels array
  loadPixels();

  // Fill pixel array with a grayscale value
  // based on pixel array index
  for (int i=0; i<pixels.length; i++) {
    b = map(i, 0, 10000, 0, 255);
    pixels[i] = color(b);
  }

  // Update the sketch with pixel data
  updatePixels();
}
```



```
// whiteNoise

void setup() {
  size(400, 300);
}

void draw() {
  float b;

  // Load colors into pixels array
  loadPixels();

  // Fill pixel array with a random grayscale value
  for (int i=0; i<pixels.length; i++) {
    b = random(0, 255);
    pixels[i] = color(b);
  }

  // Update the sketch with pixel data
  updatePixels();
}
```



See also colorNoise.pde




```
// fade red to blue
void setup() {
  size(400, 300);
  background(255, 0, 0);
}

void draw() {

  // Load colors into the pixels array
  loadPixels();

  for (int i=0; i<pixels.length; i++)
  {
    color p = pixels[i];       // Get color from pixels array

    float r = red( p );        // Extract color components
    float g = green( p );
    float b = blue( p );

    r = r * 0.99;              // Fade red to blue
    b = 255-r;

    pixels[i] = color(r,g,b);  // Rebuild and replace color
  }

  // Update the sketch with pixel data
  updatePixels();
}
```

fade.pde

Accessing Pixels as a 2D Array

- Pixels can be accessed as a 2D array using the following formula:

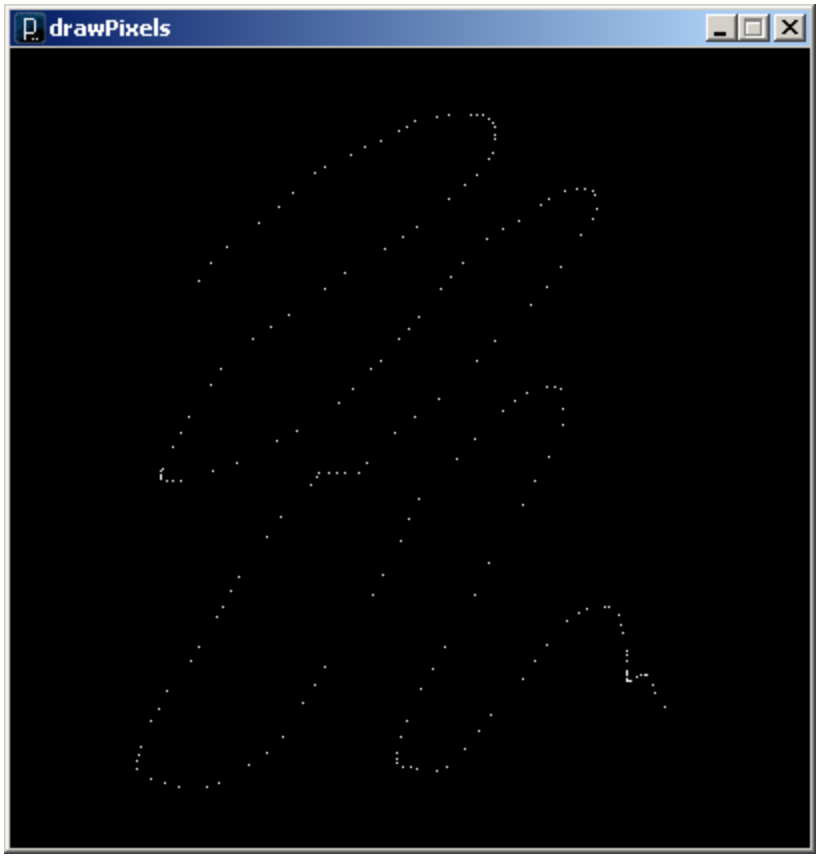
```
Index = #Columns*(Row-1) + (Col-1)
```

Check it …

- Using 0-based indexes…

```
int b = r*width + c;
pixels[i] = color(b);
```

```
// drawPixels
void setup() {
  size(400, 400);
  background(0);
}

void draw() {
  if (mousePressed == true)
  {
    // Load colors into the
    // pixels array
    loadPixels();

    // Compute pixel location
    // from mouse coordinates
    int i = mouseY*width + mouseX;

    // Set pixel color
    pixels[i] = color(255);

    // Update the sketch with pixel data
    updatePixels();
  }
}
```



drawPixels.pde

What does the following program print? Justify your answer if you're not certain.

```
void setup() {
  int n = op(5,3);
  println( n );
}

int op (int val, int divisor) {

  if (val < divisor) {
    return val;
  } else {
    int v = op(val - divisor, divisor);
    return v;
  }
}
```

Which built-in mathematical operator does the op() function simulate?

Add the necessary transformations to `draw()` to render the rectangle at the center of the sketch, twice its size, and rotated by 45 degrees (PI/4 radians).

```
void setup() {
  size(400, 400);
  rectMode(CENTER);
}

void draw() {

  // Add transformations here



  rect( 0, 0, 50, 50);
}
```

Add the necessary code within the nested for-loops to color all pixels white, except for pixels on the diagonal (when r == c).

```
void setup() {
  size(100,100);

  loadPixels();

  for (int r=0; r<width; r++) {
    for (int c=0; c<height; c++) {

      // Add code here



    }  // Closing brace for the c-loop
  } // Closing brace for the r-loop

  updatePixels();
}
```

Optional Assignment 8 Hints

Evaluating Logical Expressions

Negation (!A)

A	!A
false	true
true	false

Conjunction "AND" (A && B)

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

Disjunction "OR" (A || B)

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

Derive new tables by combining operators…

1. *If I've already had two desserts, then don't serve me any more. Otherwise, I'll take another, thank you.*

   A = had_dessert_1, B = had_dessert_2

!(A && B) or !A || !B

A	B	!(A && B)
true	true	false
true	false	true
false	true	true
false	false	true

2. *I'll have dessert, as long as it is not flan (A) or beef jerky (B).*

Evaluating Bitwise Operators

NOT (~a)

a	~a
0	1
1	0

AND (a & b)

a	b	a & b
1	1	1
1	0	0
0	1	0
0	0	0

OR (a | b)

a	b	a | b
1	1	1
1	0	1
0	1	1
0	0	0

But you can't save a single bit by itself … where would you put it?
At a minimum, you must operate on whole bytes.

```
int b1 = 45;
int b2 = 179;
int b3 = b1 & b2;
println( b3 );
// b3 = 33
```

```
  00101101
& 10110011
  --------
  00100001
```

bits.pde

Force an individual bit to 0

- Bitwise-AND with a number containing all bits set to 1 except where you want to set to 0

```
  00101101          00101100
& 11111110        & 11111110
  --------          --------
  00101100          00101100
```

Resulting bit is 0, regardless of starting bit.

Force an individual bit to 1

- Bitwise-OR with a number containing all bits set to 0 except where you want to set the bit to 1

```
  00101101          00101100
| 00000001        | 00000001
  --------          --------
  00101101          00101101
```

Resulting bit is 1, regardless of starting bit.

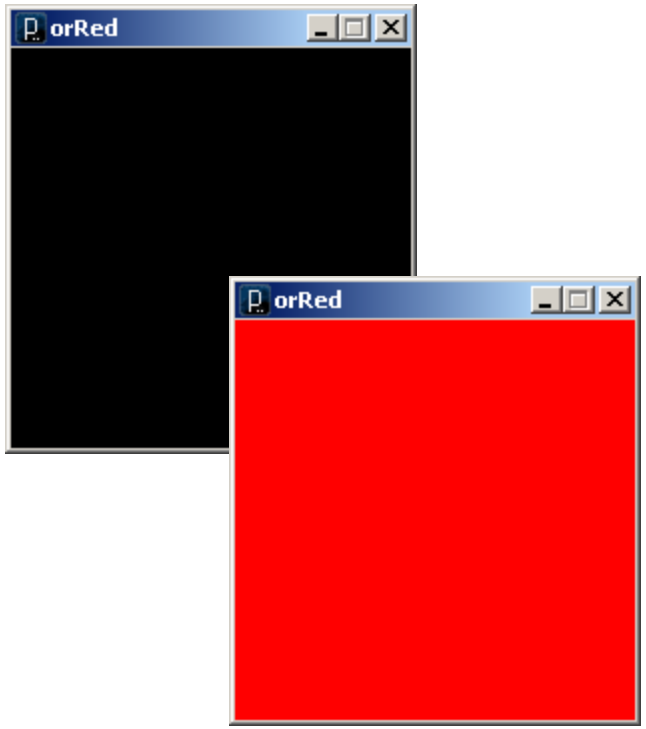
Colors are 4-byte primitives in Processing

```
void setup() {
  color c = color(109, 62, 98, 207);

  float r = red(c);
  float g = green(c);
  float b = blue(c);
  float a = alpha(c);

  println("red: "   + r);
  println("green: " + g);
  println("blue: "  + b);
  println("alpha: " + a);
}
```

colorbits.pde

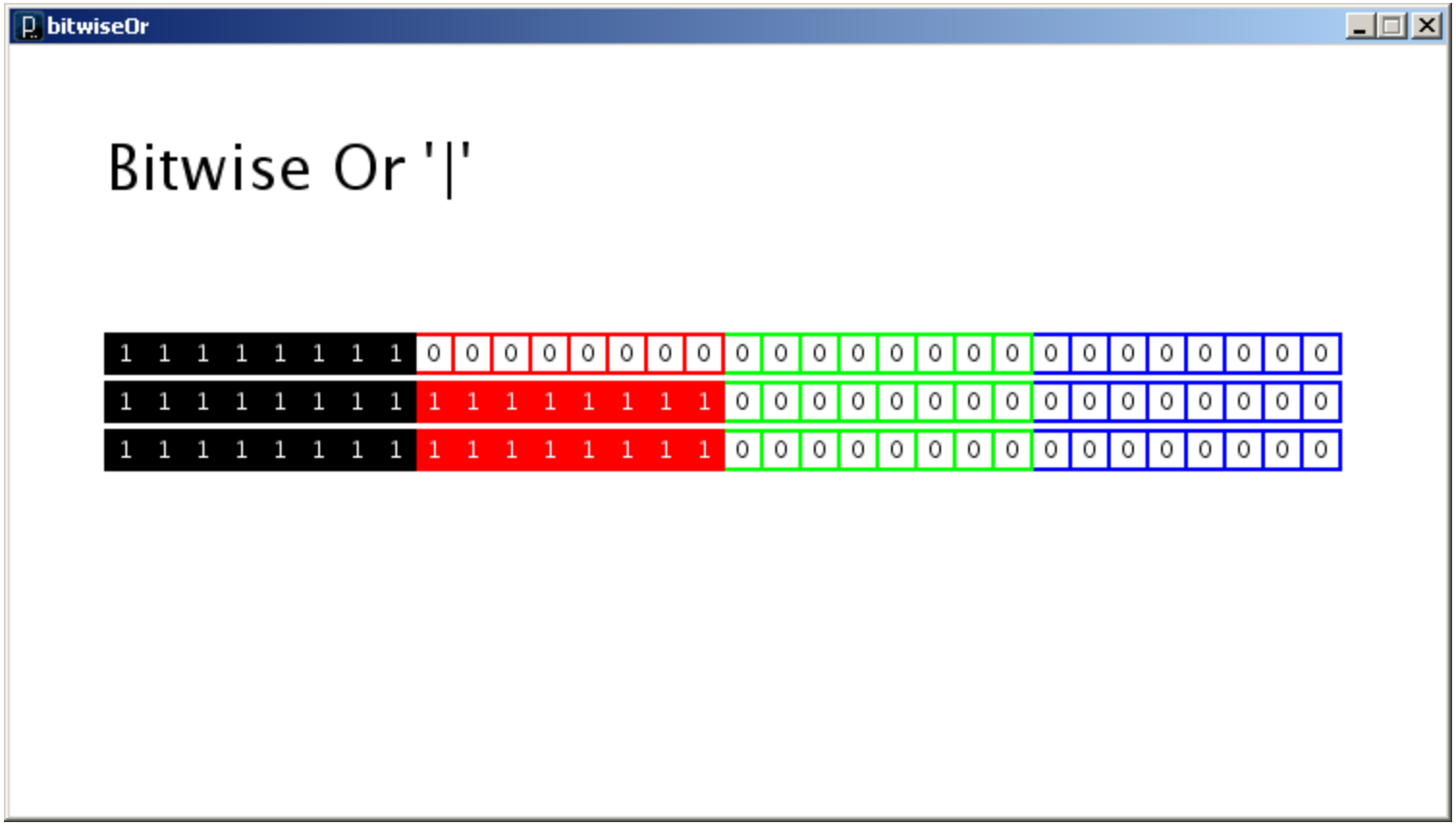
Colors can be manipulated with bitwise operations

```
//orRed.pde
void setup() {
  size(200, 200);
  background(0);
}

void draw() {}

void mousePressed() {
  loadPixels();

  // Bits to or with
  color orColor = color(255, 0, 0, 255);

  // Set red bits
  for (int i=0; i<pixels.length; i++) {
    color c = pixels[i];
    c = c | orColor;
    pixels[i] = c;
  }

  updatePixels();
}
```



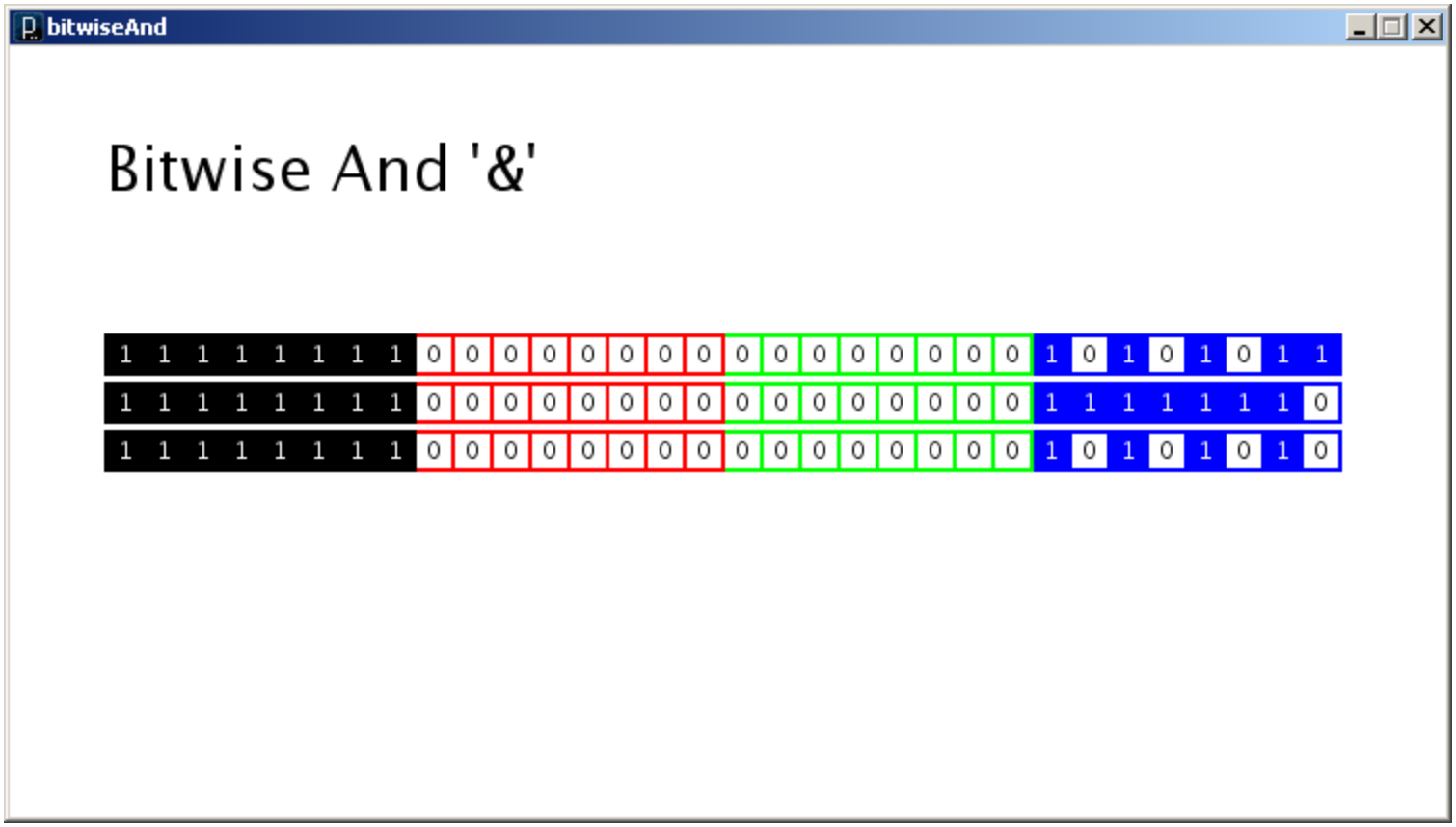
orRed.pde

Bitwise-ORing Colors



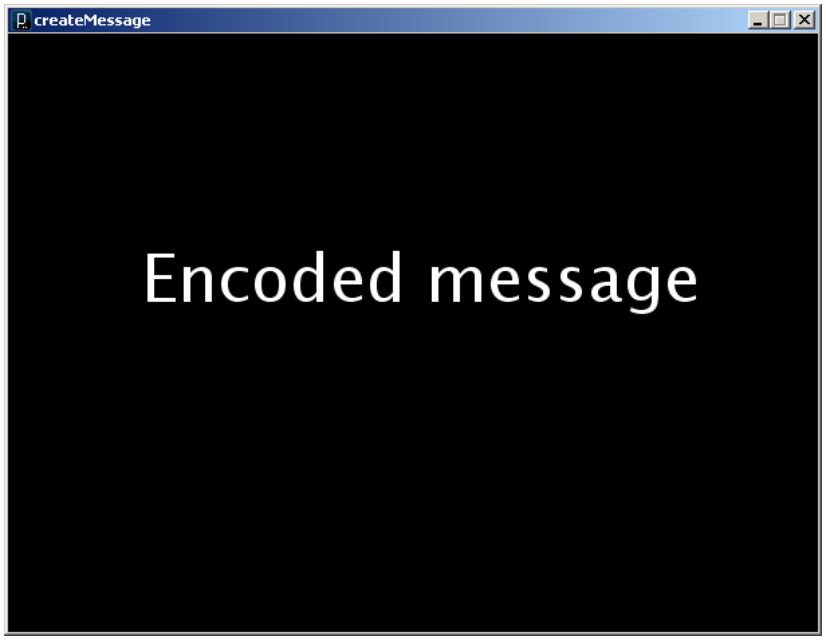
bitwiseOr.pde

Bitwise-ANDing Colors



bitwiseAnd.pde

```
// createMessage.pde

// This program can be used to generate an image containing
// a hidden message to be encoded within another picture.

void setup() {
  size(604, 446);      // The size of the generated image must
                       // exactly match the picture into which
                       // the image will be encoded.
  background(0);       // Black background
  fill(255);           // White lettering
  textSize(48);
  text("Encoded message", 100, 200);
  save("msg.png");

}
```



```
// encode.pde
PImage img;  // To hold unencoded picture
PImage msg;  // To hold image to be encoded into the picture

color delBit = color(255,255,254,255);  // Used to bitwise-OR with color to remove last bit
color addBit = color(0,0,1,0);          // Used to bitwise-AND with color to add last bit

void setup() {
  img = loadImage("bmc.png");   // Load the unencoded picture
  msg = loadImage("msg.png");   // Load the image toencode into the picture
  size(img.width, img.height);  // Size the sketch to be the same size as the picture
  image(img, 0, 0);             // Display the picture
}

void draw() {}

// When the mouse is clicked, encode b&w image into the other
void mousePressed() {
  img.loadPixels();  // Load pixels for the picture
  msg.loadPixels();  // Load pixels for the b&w image to be encoded

  // Loop over all pixels in the picture
  for (int i=0; i<img.pixels.length; i++) {

    // Get pixel color in both the picture and the b&w message to be hidden
    color ip = img.pixels[i];
    color mp = msg.pixels[i];

    // Strip right-most bit from picture pixel
    ip = ip & delBit;

    // Add the bit from the hidden message to picture pixel color's right-most bit
    ip = ip | (mp & addBit);

    // Replace the encoded pixel back into the picture
    img.pixels[i] = ip;
  }

  // Update pixels and save the encoded picture to a new file
  img.updatePixels();
  image(img, 0, 0);
  img.save("encoded.png");
}
```

Unencoded and Encoded Images

```
// decode.pde
PImage img;

color readBit = color(0,0,1,0);   // Used to bitwise-AND with pixel color to read last bit

void setup() {
  img = loadImage("encoded.png"); // Load encoded image
  size(img.width, img.height);    // Display encoded image on the sketch
  image(img, 0, 0);
}

void draw () { }

// When the mouse is pressed, decode the secret message and display.
void mousePressed() {

  loadPixels();           // Load the sketch pixels into the pixel's array
  img.loadPixels();       // Load pixels of encoded image into the image's pixels[] array.

  // Loop over all pixels in encoded image.
  for (int i=0; ...       // 1. Complete the for-loop so that it accesses all image pixels

    // Copy ith pixel color from the image to pixVal.
    color pixVal = ...    // 2. How does one access the ith pixel from an image?

    // Extract the encoding bit using a bitwise-and (&) of pixVal and readBit.
    int encodingBit = ... // 3. How does one do a bitwise-AND to read a bit from the pixel

    // If value of the encodingBit is greater than 0, set pixVal to white (color(255))
    // Otherwise, set pixVal to black (color(0)).
    if (...               // 4. How does one build this if-else statement to properly set pixVal?
      pixVal = ...
    } else {
      pixVal = ...
    }

    // Reset the ith pixel in the sketch to pixVal.
    pixels[i] = ...       // 5. How does one set the ith pixel in a sketch?
  }

  // Update the pixels in the sketch to reveal the secret message.
  updatePixels();
}
```


decode
Encoded message