GRAPHICS TECHNIQUES

Today

- Last time we looked at some ways we can do graphics in Java.
- Today we will look a bit under the hood, understanding what needs to be done to make these routines work.
- We'll also look a bit at some of the other Graphics options in Java.
- In particular we'll look at some of the options for manipulating images.

2D Graphics

- The 2D API in Java is made up of:

  `java.awt`

  `java.awt.color`

  `java.awt.font`

  `java.awt.geom`

  `java.awt.image`

  `java.awt.print`

- We'll look at some of the features of these, especially those that relate back to the basic drawing primitives from last week

- An instance of `java.awt.Graphics2D` is called a *graphics context*.
- A graphics context provides methods for drawing:
  - Shapes
  - Text
  - Images
- It also holds contextual information about the drawing area.

- Four ways to get a `graphics2D` object.
  1. From AWT or Swing as the result of issuing a paint request on a component
     A graphics context is created and passed to `paint()` or `update()`
  2. Directly from an offscreen image buffer
     Here we ask for the context directly — see double buffering.
  3. By copying an existing `Graphics2D` object
     Can do this using `create()`. Helpful in doing complex drawings where different copies draw on the same area.
  4. Directly from an onscreen component
     Almost always a mistake.

- Each time you call `paint()`, the windowing system gives the component a new `Graphics2D` object to play with.
- Thus all previous context is lost (and needs to be reset if you want it).
- AWT components may also have an `update()` method, which allows context to be carried forward.

  Not used by Swing.

- For backward compatibility, `paint()`'s argument is always an object of type `Graphics`.
- You will likely want to cast this to a `Graphics2D` to take advantage of the features of the 2D API.

The rendering pipeline

- `Graphics2D` has four rendering operations:
  1. Draw an outline with `draw()`
  2. Fill a shape with `fill()`
  3. Write text with `drawString()`
  4. Draw an image with `drawImage()`
- The way these things are achieved depends on the context.

- The current *paint* determines the color or pattern used to fill a shape.
- The *stroke* determines how outlines are drawn

  (Like the nib on a pen).
- The current *font* determines the font.
- We can perform *transformations*.
- There is a *compositing rule* that determines how a new operation is combined with an old one.
- All rendering is restricted to the interior of the *clipping* shape.
- The *rendering hints* control the tradeoff between speed and what the result looks like.

- Let's look at examples of all of these things.

- We can transform an image, for example by moving it in the frame, and by rotating it:

```
g2.translate(cx,  cy);
g2.rotate(theta * Math.PI/180);
```

- We'll look at how these operations are achieved later on.

- This compositing command:

```
AlphaComposite ac =
  AlphaComposite.getInstance(
    AlphaComposite.SRC_OVER, (float).75);
g2.setComposite(ac);
```

  sets the transaprency to 50%, allowing objects to be send through those drawn "on top" of each other.

- Any drawn object is clipped by the clipping shape in place when that object is drawn.
- This sets the clipping shape to be an ellipse:

```
Shape e =
   new Ellipse2D.Double(-cx, -cy, cx*2, cy*2);
g2.clip(e);
```

- Rendering hints effect all the drawing operations. This:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
```

  turns on anti-aliasing, to smooth out rough edges.

- These examples can all be found in the `Iguana` project.
- Play with this to see how the various bits of the rendering operation work.

- Having looked at the methods Java provides for doing these some basic graphics operations, we'll take a look at what goes on behind these operations.
- We'll focus on the way that images are manipulated.

Transformations

- *Geometrical transformations* allow you to manipulate drawings.
- Mathematical manipulation of the points that make up a drawing.
- Three basic transformations:
  - translating
  - scaling
  - rotating
- *Matrix* manipulation is used, so need some basic maths.

Why matrices?

- We describe graphical objects with sets of coordinates:
  - x and y for 2D graphics.
  - x, y and z for 3D graphics.
  - also add θ_i when we are interested in orientation.
    (for example if we are having an object move)
- A set of coordinates can easily be written as a matrix:

$$\begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix}$$

- This representation allows us to handle all objects, however many coordinates describe them, in the same way.

  (It is also a handy shortcut for writing down the math.)

A quick guide to matrix math

- *Scalar*: a single number, either integer or real-valued.
- *Vector*: essentially a group of scalars that are manipulated together; e.g., two scalars to represent a point in 2-dimensional space or three scalars to represent a point in 3-dimensional space.
- Example vectors:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \text{ or } \begin{bmatrix} x \\ y \end{bmatrix} \text{ or } \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{ or } \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- Two vectors of the same size can be added together by adding their components; the result is another vector of the same size:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1+4 \\ 2+5 \\ 3+6 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 9 \end{bmatrix}$$

- Two vectors of the same size can be multiplied by each other to compute what is called the *dot product* (or *inner product*); the result is a scalar
- The dot product ($\cdot$) is computed by multiplying the vectors' individual components and then adding them together:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 4 + 10 + 18 = 32$$

- The general formula is:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = (x_1 \times x_2) + (y_1 \times y_2) + (z_1 \times z_2)$$

- A *matrix* is just a vector with more than one column, e.g.:

$$\begin{bmatrix} 1 & 5 & 3 \\ 8 & 1 & 4 \\ 2 & 9 & 1 \end{bmatrix}$$

- We can multiply a matrix (or a vector) by a scalar:

$$2 \times \begin{bmatrix} 1 & 5 \\ 8 & 1 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 2 \times 1 & 2 \times 5 \\ 2 \times 8 & 2 \times 1 \\ 2 \times 2 & 2 \times 9 \end{bmatrix} = \begin{bmatrix} 2 & 10 \\ 16 & 2 \\ 4 & 18 \end{bmatrix}$$

- Multiplying two matrices together is like computing a bunch of dot products between the rows of one matrix and the columns of the other
- So in order to be able to multiply two matrices together, the number of rows in one has to be the same as the number of columns in the other
- The general formula for matrix multiplication is:

$$C_{(row,col)} = \sum_{s=1}^{m} A_{(row,s)} * B_{(s,col)}$$

where A is an $n \times m$ matrix and B is an $m \times p$ matrix and C is an $n \times p$ matrix

- So, for example, we can compute:

$$C = \begin{bmatrix} x_1 & y_1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

- Or, in the notation of the general formula:

$$C = \begin{bmatrix} A_{(1,1)} & A_{(2,1)} \end{bmatrix} \begin{bmatrix} B_{(1,1)} \\ B_{(1,2)} \end{bmatrix}$$

where A is an 1×2 matrix and B is an 2×1 matrix and C is an 1×1 matrix (otherwise called a scalar); i.e., $m = 1$, $n = 2$ and $p = 2$

$$C = (A_{(1,1)} \times B_{(1,1)}) + (A_{(1,2)} \times B_{(2,1)})$$

- Alternatively, we can compute:

$$C = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \begin{bmatrix} x_2 & y_2 \end{bmatrix}$$

- Which, in the notation of the general formula, is:

$$C = \begin{bmatrix} A_{(1,1)} \\ A_{(1,2)} \end{bmatrix} \begin{bmatrix} B_{(1,1)} & B_{(2,1)} \end{bmatrix}$$

where A is an 2×1 matrix and B is an 1×2 matrix and C is an 2×2 matrix; i.e., $m = 1$, $n = 2$ and $p = 2$

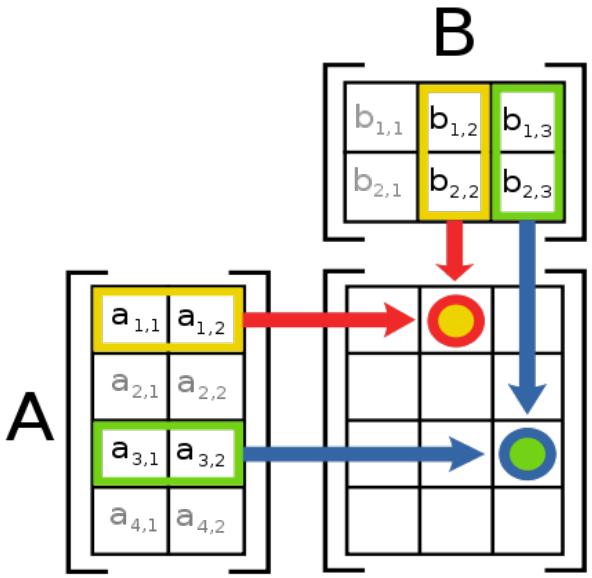
- We can think of this second case as:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \begin{bmatrix} x_2 & y_2 \end{bmatrix} = \begin{bmatrix} x_1 \times x_2 & x_1 \times y_2 \\ y_1 \times x_2 & y_1 \times y_2 \end{bmatrix}$$

- We can also write this as:

$$\begin{aligned}
C_{(1,1)} &= \sum_{s=1}^{1} A_{(1,s)} \times B_{(s,1)} = A_{(1,1)} \times B_{(1,1)} = x_1 \times x_2 \\
C_{(1,2)} &= \sum_{s=1}^{1} A_{(1,s)} \times B_{(s,2)} = A_{(1,1)} \times B_{(1,2)} = x_1 \times y_2 \\
C_{(2,1)} &= \sum_{s=1}^{1} A_{(2,s)} \times B_{(s,1)} = A_{(2,1)} \times B_{(1,1)} = y_1 \times x_2 \\
C_{(2,2)} &= \sum_{s=1}^{1} A_{(2,s)} \times B_{(s,2)} = A_{(2,1)} \times B_{(1,2)} = y_1 \times y_2
\end{aligned}$$

- A nice visualization of the general case of multiplying two matrices is:



- My high school math teacher called this "diving rows into columns".

  Each row in the lefthand matrix is combined with each column in the right hand matrix.

- If you check the previous examples, you'll find they fit into this mould.

- We'll also use the idea of a *transpose*, basically swapping rows for colums:

$$\begin{bmatrix} x & y & 1 \end{bmatrix}^T = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

and

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}^T = \begin{bmatrix} x & y & 1 \end{bmatrix}$$

- We'll use this to make some of the operations easier to write down.

Transformations

- we will discuss three types of transformations:
  - *translating*: shifting an object to a new location, without changing its size or shape
  - *scaling*: changing the size of an object
  - *rotation*: turning an object about a particular point, without changing the object's size or shape
- We will discuss these in 2-dimensions only.
- The same approach works for 3D, the matrices are just bigger.

- Each transformation can be descibed as a matrix computation.
- These are described in the next few slides
- Note that the examples shown use X-windows graphics coordinate conventions, where the origin $(0, 0)$ is in the upper left corner of the screen, x increases moving to the right and y increases moving down.

Translation

- *Translating* is shifting an object to a new location, without changing its size or shape.
- We'll talk about translating in 2 dimensions, meaning in the x and/or y directions.
- We need to know how much to translate (shift, move) the object in each direction.
- We call these values Δx and Δy, respectively.
- To translate point (x, y) by Δx units along the x axis and Δy units along the y axis we just use addition:

$$x' = x + \Delta x$$
$$y' = y + \Delta y$$

- In matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

- Translation is often done in conjunction with other transformations, which require matrix multiplication rather than addition.
- As a result, we usually describe translation as:

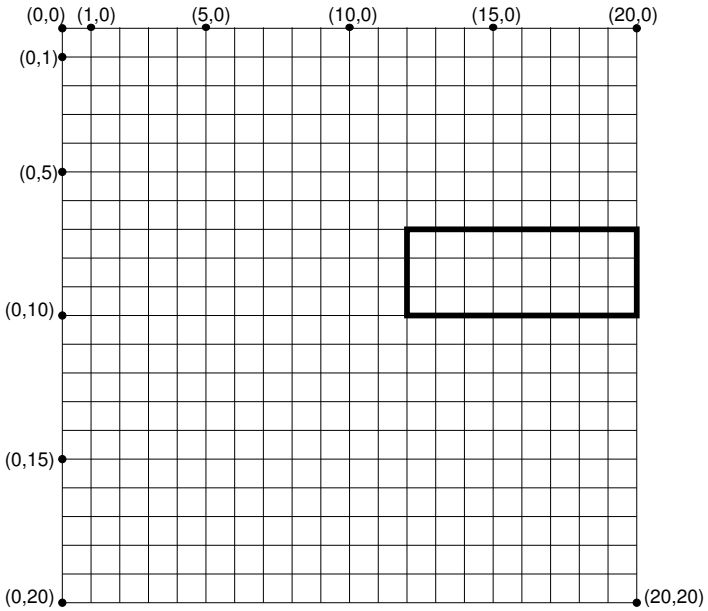
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- The translation is *exactly the same* as doing the addition:

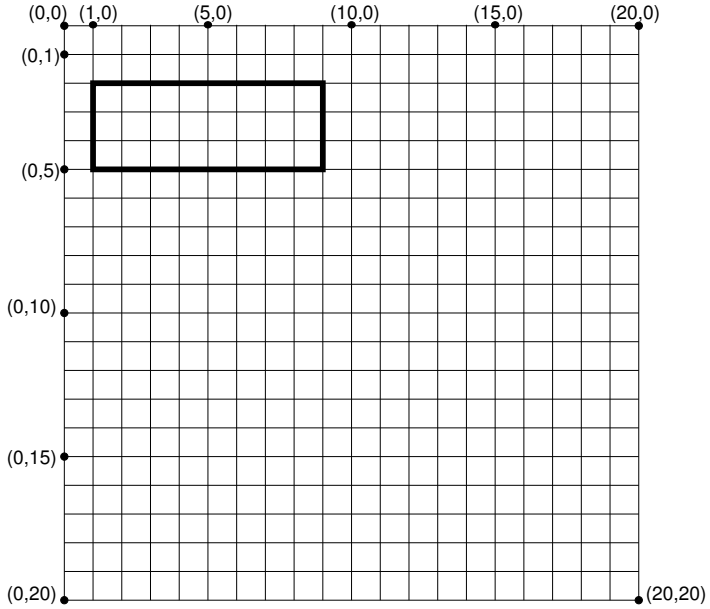
$$
\begin{aligned}
\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} x \times 1 + y \times 0 + 1 \times \Delta x \\ x \times 0 + y \times 1 + 1 \times \Delta y \\ x \times 0 + y \times 0 + 1 \times 1 \end{bmatrix} \\
&= \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ 1 \end{bmatrix}
\end{aligned}
$$

An example of translation

- Given the rectangle in 2D space bounded by $(12, 7)$ and $(20, 10)$, translate the rectangle so that the upper left corner is at $(1, 2)$.
- Here is the original rectangle:



- Translation is a one-step process, ending up here:



- In order for the new upper left corner to end up at $(1, 2)$, we first calculate the Δx and Δy values necessary for the translation:

$$\begin{aligned} \Delta x &= 1 - x_1 \\ &= 1 - 12 \\ &= -11 \end{aligned}$$

$$\begin{aligned} \Delta y &= 2 - y_1 \\ &= 2 - 7 \\ &= -5 \end{aligned}$$

- After this translation we know the upper left corner will be at $(1, 2)$, but we have to apply the translation to the bottom right corner to know where that ends up.

- This gives:

$$
\begin{aligned}
\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & -11 \\ 0 & 1 & -5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 20 * 1 + 10 * 0 + 1 * -11 \\ 20 * 0 + 10 * 1 + 1 * -5 \\ 20 * 0 + 10 * 0 + 1 * 1 \end{bmatrix} \\
&= \begin{bmatrix} 9 \\ 5 \\ 1 \end{bmatrix}
\end{aligned}
$$

- This means that the translated rectangle is bounded by $(1, 2)$ and $(9, 5)$

Scaling

- *Scaling* is changing the size of an object.
- This can be done *uniformly*, where the size of all dimensions change by the same amount.
- Can also be done *non-uniformly*, where the size of each dimension changes by a different amount.
- Again, we'll talk about scaling in 2 dimensions, meaning in the x and/or y dimensions

- If s_x and s_y represent how much we want to scale the object in the x and y dimensions, respectively, then the formulas are:

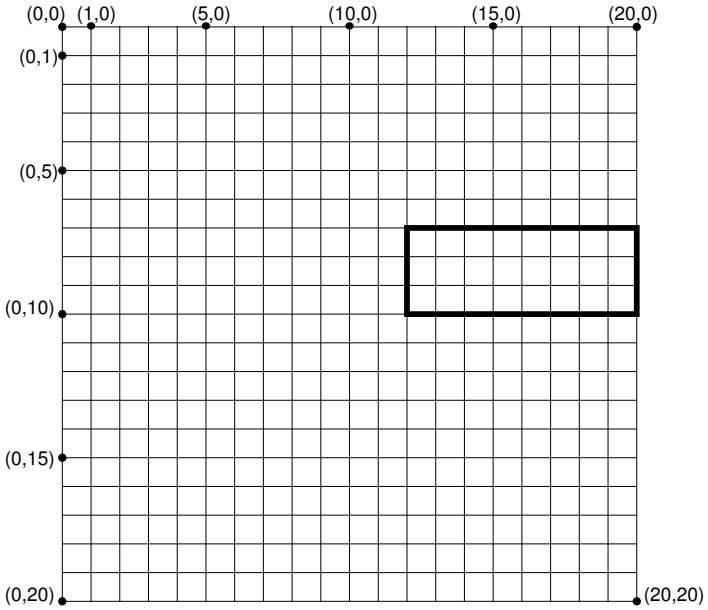
$$x' = x \times s_x$$
$$y' = y \times s_y$$

- In matrix form:

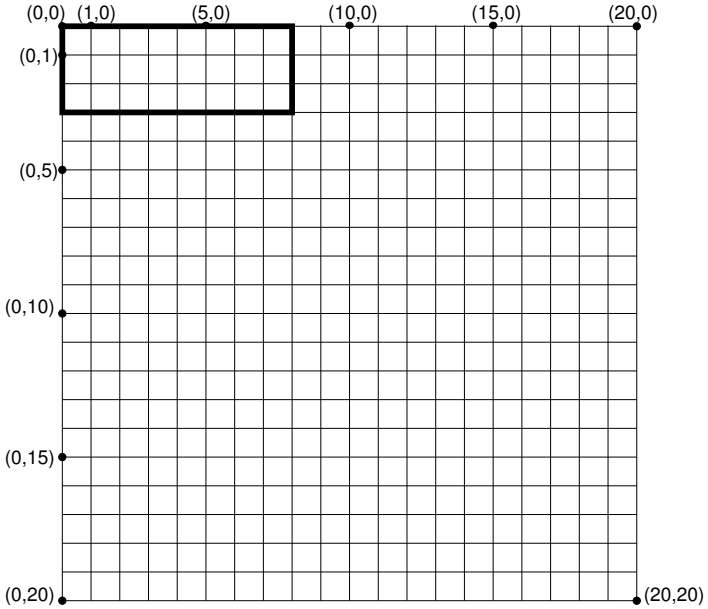
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

An example of scaling

- Given the rectangle in 2D space bounded by $(12, 7)$ and $(20, 10)$, use scaling to increase the size of the rectangle threefold ($\times 3$), keeping the location of its upper left corner the same.
- Here is the original rectangle:



- This is a 3-step process.
- First, we translate the original rectangle back to the origin $(0, 0)$:



- This is done by first calculating the Δx and Δy values necessary for the translation of the upper left corner:

$$\Delta x = 0 - x_1 = 0 - 12 = -12$$
$$\Delta y = 0 - y_1 = 0 - 7 = -7$$

- Then we apply this translation to the bottom right corner:

$$
\begin{aligned}
\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & -12 \\ 0 & 1 & -7 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 20 \times 1 + 10 \times 0 + 1 \times -12 \\ 20 \times 0 + 10 \times 1 + 1 \times -7 \\ 20 \times 0 + 10 \times 0 + 1 \times 1 \end{bmatrix} \\
&= \begin{bmatrix} 8 \\ 3 \\ 1 \end{bmatrix}
\end{aligned}
$$

- With both corners fixed, we know what the new position of the rectangle is.

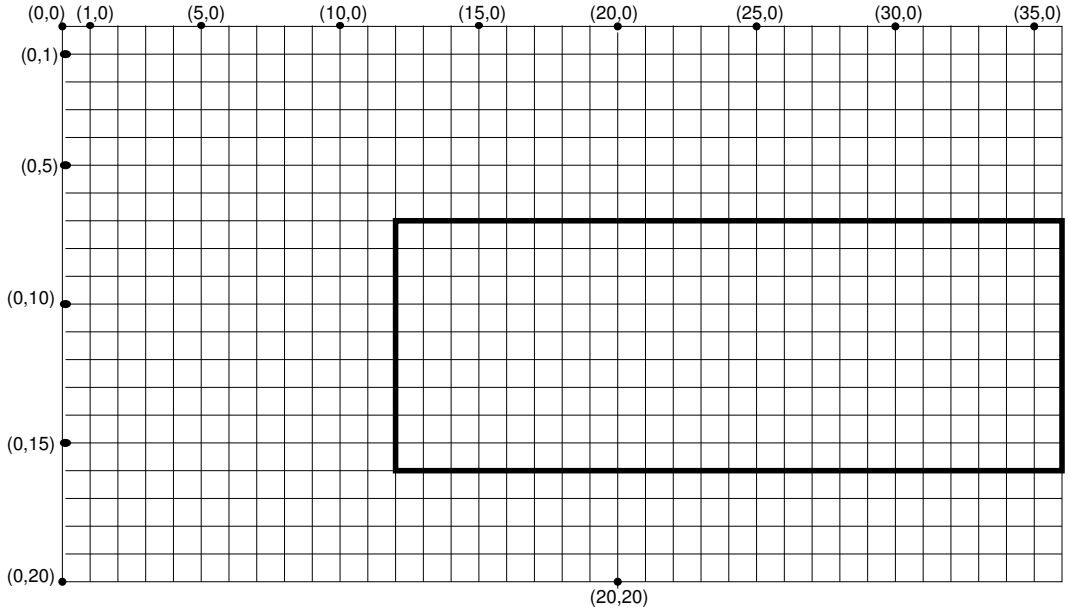
- Second, we perform the scaling on the rectangle at the origin:

- We use the given scaling factor, $s_x = 3$ and $s_y = 3$.
- This is what happens to the lower right corner:

$$\begin{aligned}
\begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\
&= \begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 8 \\ 3 \end{bmatrix} \\
&= \begin{bmatrix} 8 \times 3 + 3 \times 0 \\ 8 \times 0 + 3 \times 3 \end{bmatrix} \\
&= \begin{bmatrix} 24 \\ 9 \end{bmatrix}
\end{aligned}$$

- Since the top left corner is set to $(0, 0)$, we don't have to do anything — if we do the multiplication, we will still end up with $(0, 0)$.

- The third and final step is to translate the scaled rectangle from the origin back to its original location, as shown in figure D, where the upper left corner of the rectangle is in the original place:



- To do this, we just apply the opposite translation we did earlier, using $\Delta x = 12$ and $\Delta y = 7$ on both sets of coordinates.

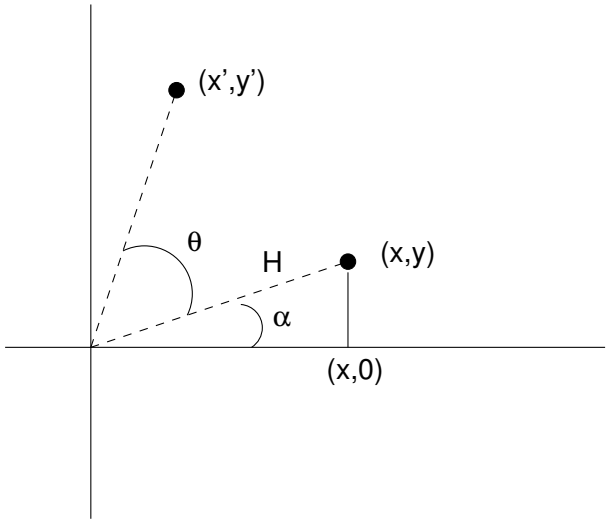
Rotation

- We handle rotation by matrix multiplication as well.
- *Rotation* is turning an object about a particular point, without changing the object's size or shape
- Typically, rotation is computed about the origin
- So in order to rotate an object about one of its points, you first have to translate the object so that that point is at the origin, perform the rotation and then translate the object back to where it came from.
- Here, we'll talk about how to rotate about the origin

- The amount of rotation is expressed in terms of an angle, θ
- The formula for rotating a point by θ about the origin is:

$$x' = x.cos(\theta) - y.sin(\theta)$$
$$y' = x.sin(\theta) + y.cos(\theta)$$

- We get the formula by starting from a point (x, y) that we want to rotate θ degrees:



- If we drop a line from that point perpendicular to the x-axis, intersecting at $(x, 0)$, we get a right-angle triangle described by the three points: (x, y), $(x, 0)$ and $(0, 0)$

- The angle between the two lines that meet at the origin, is α
- The rules of trigonometry tells us a few things about this triangle:

$$cos(\alpha) = x/H$$
$$sin(\alpha) = y/H$$

  Where H is the hypotenuse, the longest side in a right triangle; in this case, the distance from the origin to the point (x, y)
- in other words:

$$x = H.cos(\alpha)$$
$$y = H.sin(\alpha)$$

  or

$$H = cos(\alpha)/x$$
$$H = sin(\alpha)/y$$

- we can do the same thing with the new, rotated point (x', y'), by dropping a perpendicular to the x-axis; the distance from the origin to the new point stays the same (i.e., $= H$) and the angle between the two lines that meet at the origin, is now $\alpha + \theta$, where θ is the angle of rotation.
- This gives us:

$$cos(\alpha + \theta) = x'/H$$
$$sin(\alpha + \theta) = y'/H$$

or

$$x' = H.cos(\alpha + \theta)$$
$$y' = H.sin(\alpha + \theta)$$

- Here we need to use some trigonometric identities:

$$cos(\alpha + \theta) = cos(\alpha).cos(\theta) - sin(\alpha).sin(\theta)$$
$$sin(\alpha + \theta) = sin(\alpha).cos(\theta) + sin(\theta).cos(\alpha)$$

- Then we can subsitute back in:

$$x' = H.(cos(\alpha).cos(\theta) - sin(\alpha).sin(\theta))$$
$$y' = H.(sin(\alpha).cos(\theta) + sin(\theta).cos(\alpha))$$

  which equals

$$x' = H.cos(\alpha).cos(\theta) - H.sin(\alpha).sin(\theta)$$
$$y' = H.sin(\alpha).cos(\theta) + H.sin(\theta).cos(\alpha)$$

- Now we can refer back to our original equations about x and y and do another substitution:

$$x' = x.cos(\theta) - y.sin(\theta)$$
$$y' = x.sin(\theta) + y.cos(\theta)$$

- We can represent these formulas using matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

An aside

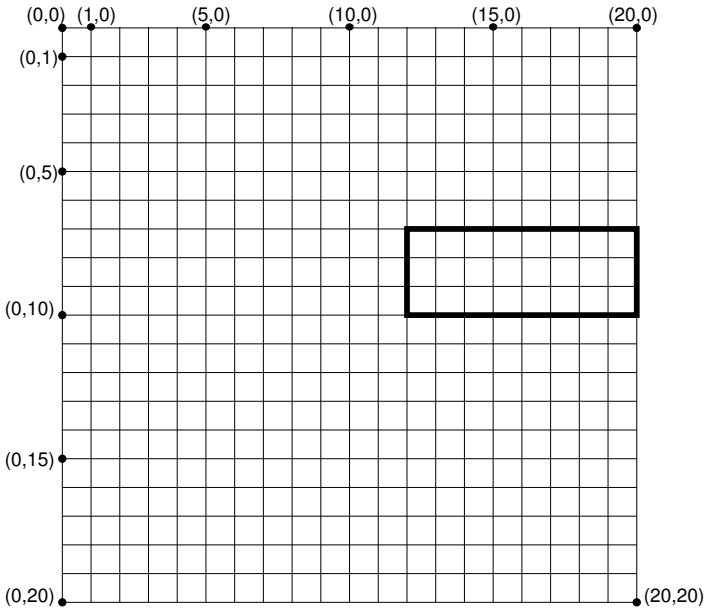
- To be completely general, once we start rotating, we don't just care about the position of a figure, we also care about its orientation.
  - We need to track that α that was in the diagram above.
- So we expand our notion of co-ordinate to include orientation, and the transformation becomes:

$$\begin{bmatrix} x' \\ y' \\ \alpha \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \alpha \end{bmatrix}$$

- But this is beyond our scope in this lecture.

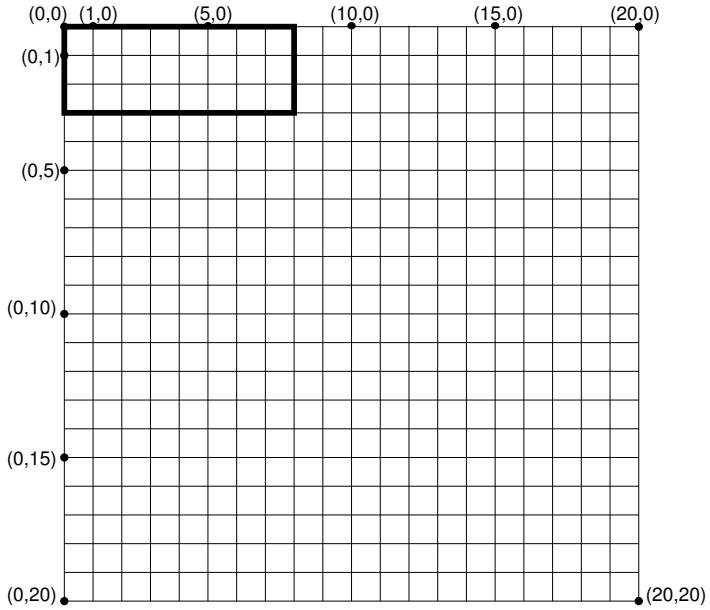
A rotation example

- Given the rectangle in 2D space bounded by $(12, 7)$ and $(20, 10)$, rotate the rectangle by $90\,\mathrm{deg}$, about its upper left corner
- Here's the original rectangle:

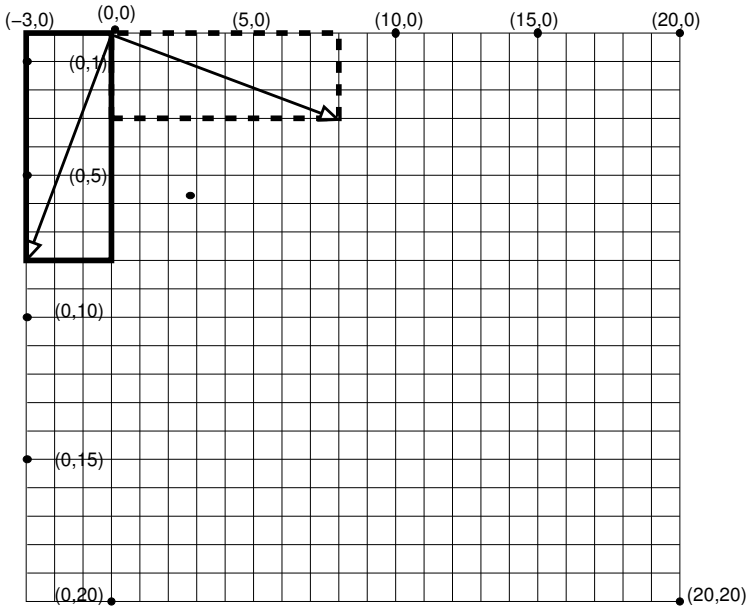


- Rotation is a 3-step process, rather like scaling.

- First, in order to perform rotation, we translate the original rectangle back to the origin.
- This is done the same way as the first step in the scaling example, giving us:



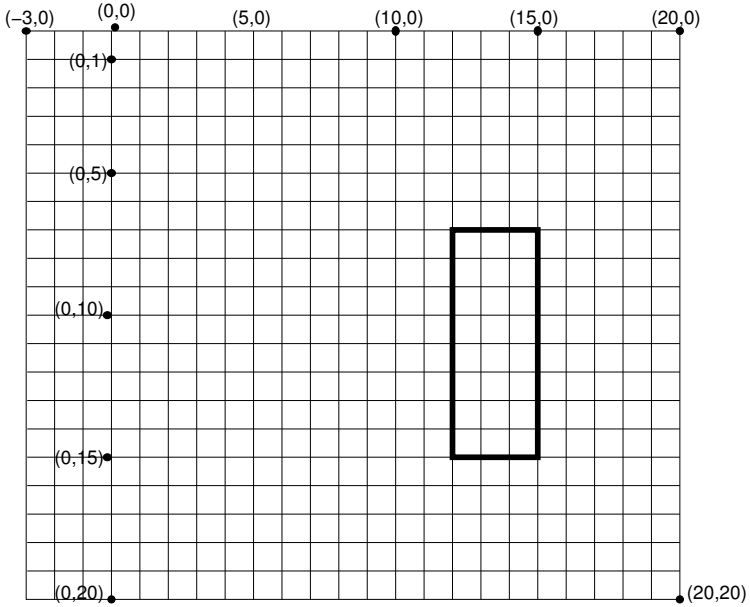
- Second, we perform the rotation on the rectangle at the origin, taking us to this position:



- We use the given rotation factor, $\theta = 90$ and the facts that $cos(90) = 0$ and $sin(90) = 1$
- Thus the transformation is:

$$\begin{aligned}
\begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\
&= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 8 \\ 3 \end{bmatrix} \\
&= \begin{bmatrix} 0 * 8 + -1 * 3 \\ 1 * 8 + 0 * 3 \end{bmatrix} \\
&= \begin{bmatrix} -3 \\ 8 \end{bmatrix}
\end{aligned}$$

- The third and final step is to translate the scaled rectangle from the origin back to its original location, as shown in figure D, where the upper left corner of the rectangle is in the original place: $(12, 7)$.
- Here, we just apply the opposite translation we did earlier:



- This means that the rotated rectangle is bounded by $(12, 7)$ and $(9, 15)$

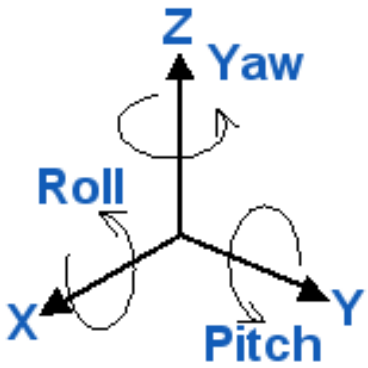
Other shapes

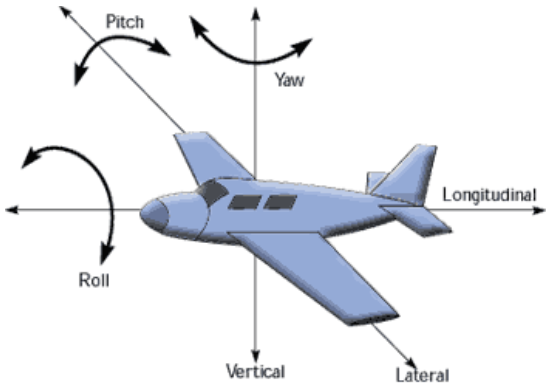
- Rectangles make for nice examples because they are easily described with two points.
  - Of course Java uses slightly different parameters, but they are equivalent.
- To do our own transformations, we just do the same operations on these (different) parameters.
- As you saw last time, other Java shapes are defined using different sets of parameters.
- Similar caclulations on these parameters will achieve the transformations

More dimensions

- Basically we just have larger matrices.
- However, rotation gets quite a few extra parameters since we now have three axes of rotation:
  - Yaw
  - Pitch
  - Roll

  as they are sometimes called.

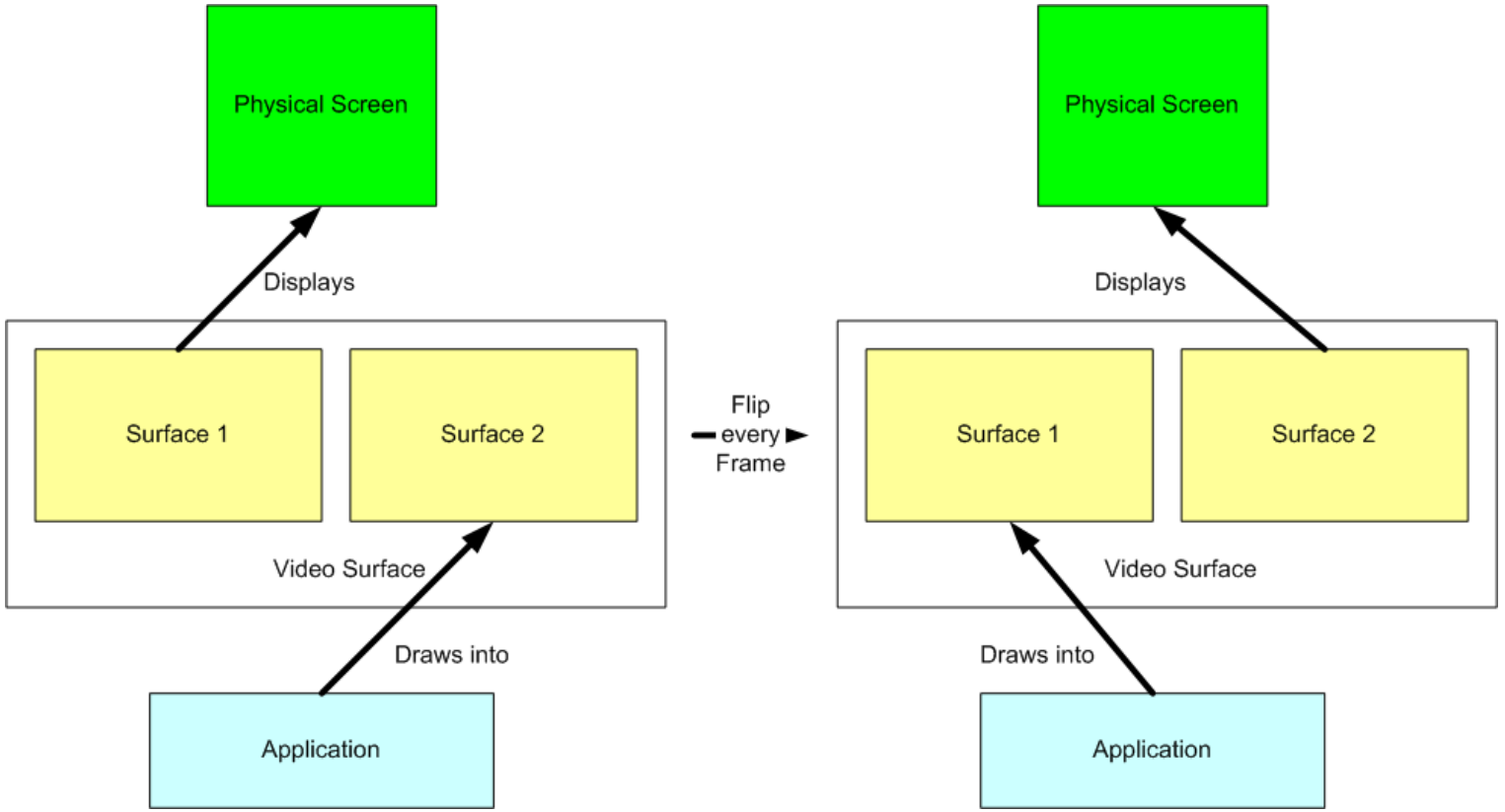




- Also known as:
  - Pan
  - Tilt
  - Roll

Double buffering

- The problem with drawing complex images is that the processor might not be able to keep up with what is going on.
- Every `repaint()` needs the whole screen to be redrawn.
- See `DrawImage` for an example.
- *Double buffering* is a way to overcome the flicker we see.



- We have `paint()` draw an image in an off-screen buffer while displaying an image that we drew earlier.
- Then flip them.

- When you use Swing components in Swing containers you get double buffering by default.
- Not the case with AWT.

- How can we get double buffering when it is not built in?
- Hold a copy of the current image.
- Let another method modify this image, and call `repaint()` when it is done.
- See example of `DoodlePad`

Using clipping to limit drawing

- The problem we saw with `DragImage()` is that `repaint()` is drawing the whole component.
- We can limit the amount of work it needs to do.
- See `DragImageWClipping`
- Not as effective as double buffering.

Animation

- With double buffering, animation is simple.
- Display one image.
- Create a second image off-screen.
- Make the object that does the image manipulation a `Runnable`.
- Run in in a thread.
- Have its `run()` method sleep between `repaint()`s

- Constructor includes:

```
Thread t = new Thread(this);
t.start();
```

- Method `myUpdateMethod()` prepares the data for `paint()` to use.

- Run is:

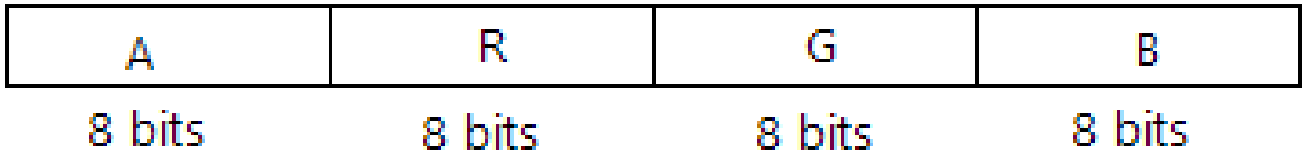
```
public void run(){
  try{
    while(true){
      myUpdatemethod();
      repaint();
      Thread.sleep(1000/24);
    }
  }
  catch(InterruptedException e){}
}
```

- This updates the screen 24 times a second, fast enough to look continuous.

Working with images

- `java.awt.image.BufferedImage` lets you manipulate the "inside" of an image directly.
- Basically an array of pixels, with information on color.
- Supports standard ways to handle image data, allowing you to deal with RGB or HSV for example.
- There are a set of image *filters* which will generate new `BufferedImage`s from old ones.

- Individual pixels are just large numbers.
- Here is how an ARGB pixel is represented:



- The "Alpha" part contains transparency information, R, G and B are red, green and blue.

- The operations are member of the class `BufferedImageOp`, for example:
  - `AffineTrasnformOp`: Scale, rotate or shear image:

    $$x' = a.x + b$$

    is an affine transformation.
  - `ColorConvertOp`: convert between color models.
  - `ConvolveOp`: convolve the image.
  - `LookupOp`: process using a lookup table.
  - `RescaleOp`: change pixel values.

```
destination = op.filter(source, null)
```

or

```
op.filter(source, destination)
```

- In addition to these filters, it is obviously possible to write methods that directly manipulate the pixels.

Summary

- This lecture rounded out our treatments of graphics.
  - Much more to say, both about what Java contains, and what the field of computer graphics covers.
- We looked at some basic computer graphics manipulations that underpin all drawing operations.
- We also looked at some of the support that Java provides for handling graphics.