Lectures 6 and 7:

Output 1:
Basic 2D Computer Graphics

05-431/631 Software Structures for User Interfaces (SSUI)

Fall, 2022

Logistics (9/15/2022)

- Readings for this lecture are on Canvas: "Files / Special Course Readings" folder
  - Foley-VanDam-graphics.pdf

Logistics (9/20/2022)

- Alex changed office hours – now Mondays 1-2pm virtually
  - See Zoom section of Canvas for link

- Homework 2 due 1 week from today

What are "Graphics"

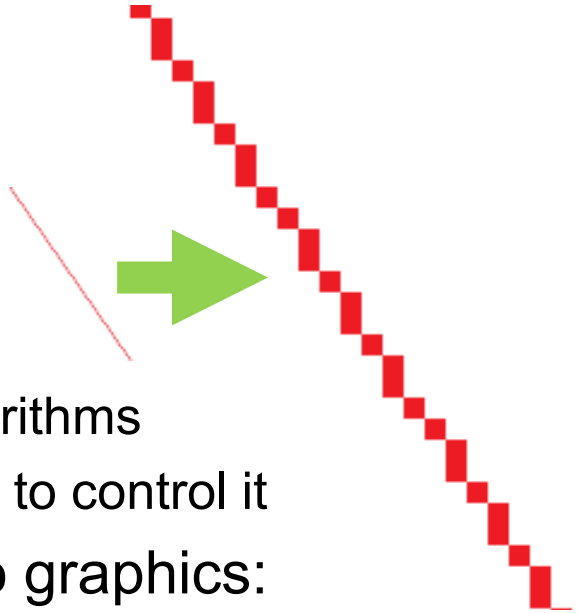
- All visual output shown to users
  - *Includes* textual output
  - Only 2D for now
- So far, mostly html or html generated from JS
- Mostly styled text and images
- Also, areas of colors – mostly rectangles or rounded rectangles
- Borders on regions

- Now, adding in "real" graphical objects:
  - Other shapes – lines, circles, polygons, etc.
  - More properties on other graphics

Why talk about Graphics?

- To draw application-specific graphical objects
- Lines, rectangles, text
- Mac, Windows, Linux, Android, iOS, web, … all have approximately the same way of describing graphics
- There are some complexities that are worth looking at
- There are 2 models, we (and homework 3) will cover both!

Rendering Graphics



- Graphics are rendered onto the screen
- Decide exactly which pixels to draw in which color
  - We won't cover the low-level rendering algorithms
  - Do need to know what is going on, and how to control it
- JavaScript provides 2 built-in ways to do graphics:
  - SVG – Scalable Vector Graphics = "Drawing"
  - Canvas = "Painting"

Drawing vs. Painting programs

- Drawing = SVG
- Painting = Canvas
- Hybrid (both)

Drawing vs. Painting programs

- Drawing
  - PowerPoint
  - MacDraw
  - Adobe Illustrator
  - Adobe InDesign
- Painting
  - Microsoft Paint
  - MacPaint
  - Snagit Editor
- Hybrid (both)
  - Photoshop

Drawing vs. Painting programs

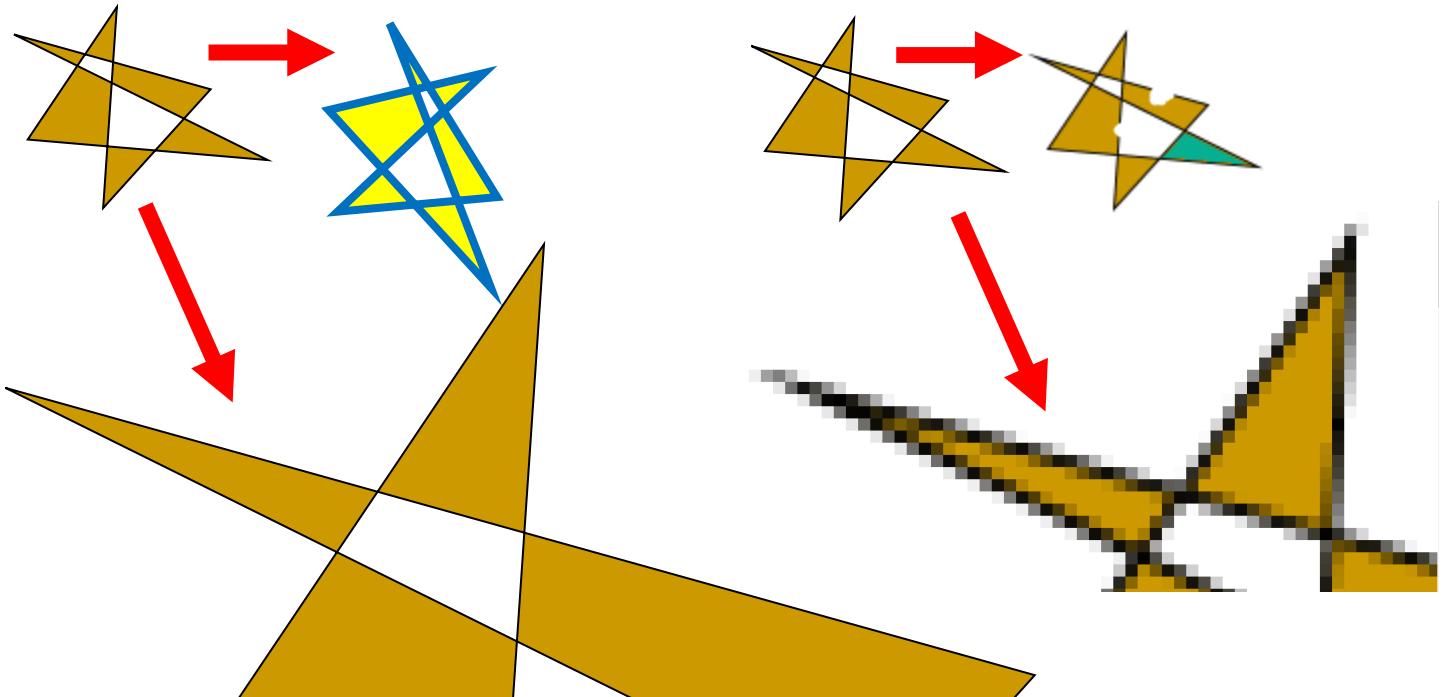
- Drawing
  - Graphical objects maintain their integrity *after* being drawn
- Painting
  - Objects just become pixels *after* being drawn

Drawing vs. Painting programs

- Drawing
  - Graphical objects maintain their integrity *after* being drawn
  - Shapes are reinterpreted as mathematical entities
  - Can move, change properties of all objects at any time
  - Rotation, change overlapping
  - Can zoom in continuously
- Painting
  - Objects just become pixels *after* being drawn
  - Can draw arbitrary shapes
  - Can touch up and individually edit the pixels anywhere
  - Supports “flood fill” (paint can)
  - Lose “resolution” and see pixels when zoom in

Drawing vs. Painting programs

- Drawing
  - Graphical objects maintain their integrity *after* being drawn
- Painting
  - Objects just become pixels *after* being drawn

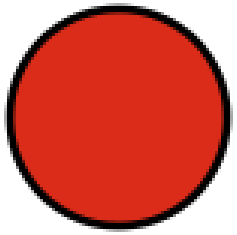


Drawing vs. Painting programs

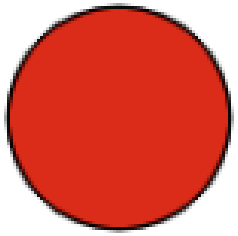
- Homework 3 – you will make a hybrid system
  - (Full specification still in progress)
- Draw on one “layer”
- Paint on another “layer”
  - “Layer” = collection of graphical objects that are treated separately from graphics on other layers
- Super-simplified version of Photoshop or other hybrid editing programs

Drawing each Object

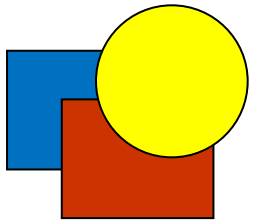
- Drawing an object can be done in either model
- “objectness” disappears after drawing is complete for painting programs
- Completely different models!
  - Note: different border size, both are “3”
-
```
<svg height="100" width="100">
  <circle cx="50" cy="50" r="40"
      stroke="black" strokewidth="3"
      fill="red" />
</svg>
```
  - https://www.w3schools.com/graphics/svg_circle.asp

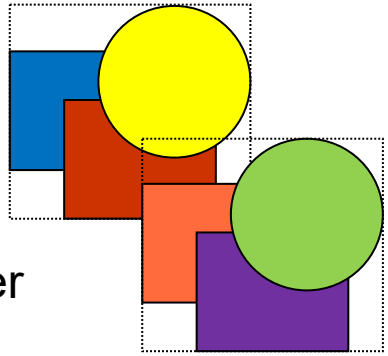


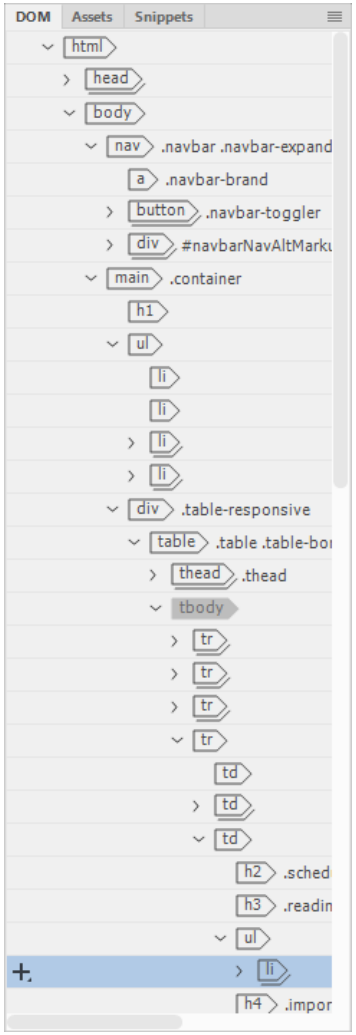
-
```
let c = document.getElementById("myCanvas");
let ctx = c.getContext("2d");
ctx.beginPath();
ctx.lineWidth = 3;
ctx.arc(50, 50, 40, 0, 2 * Math.PI);
ctx.stroke();
ctx.fillStyle = "red";
ctx.fill();
```
  - https://www.w3schools.com/tags/canvas_arc.asp



2D and covering



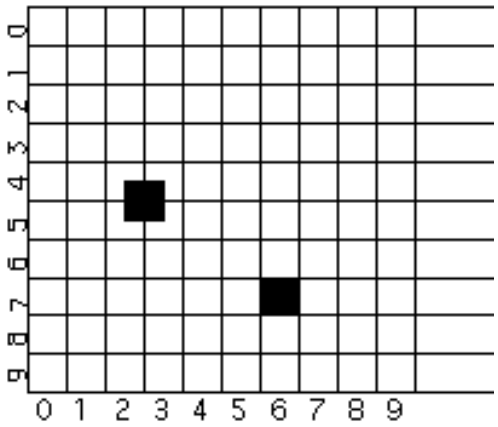




- Objects are drawn in order, back to front
  - Same as a drawing program like PowerPoint
- Containers – recursive
  - E.g., for DOM
    - Groups in many systems
    - Divs and other elements can *contain* others
  - All items in a container are above/below other containers
  - Group itself is usually see-through
- Commands to change the order for java, SVG
  - Java: `setComponentZOrder()` on any container
  - SVG: remove from DOM and re-add at the desired z-order
  - Not canvas!

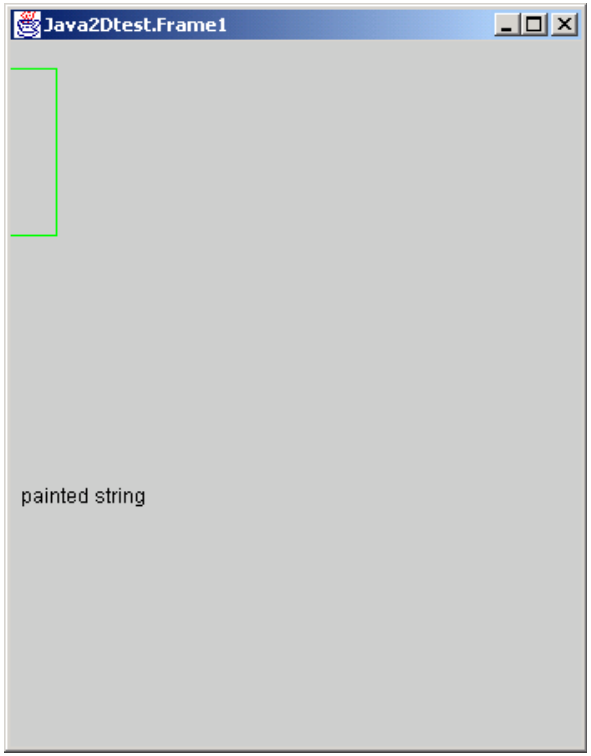
Coordinates for Drawing

- Origin
  - Typically 0,0 in top left
  - Comes from text handling and raster scan
  - Java 2D allows customization
- Different from conventional axes
- Coordinates of pixels:
  - Center of pixel?
  - Corner of pixel?
- Matters for lines



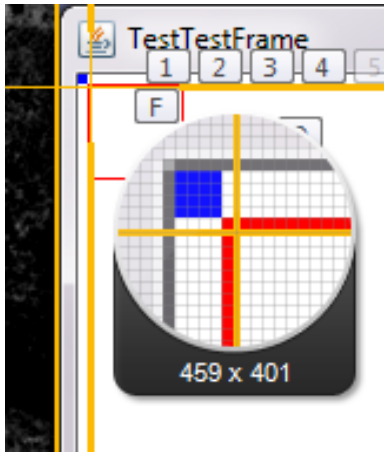
Issue: Window Coordinates

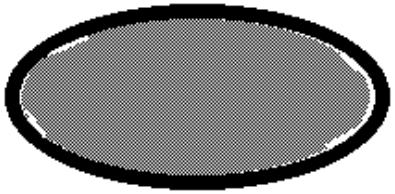
- Where is 0,0 with respect to the window's *inside* or *outside* border?
- CreateWindow (10, 10, 100, 100)
  - Inside or outside?
  - Different for point vs. W/H?
  - What is the size of window border?
- JS – just inside



Drawing Primitives

- Drawing Objects:
  - Graphics, graphics2D java APIs: http://docs.oracle.com/javase/8/docs/api/
  - Canvas / svg for JavaScript – all draw different pixels!
  - P1 and P2 or P1 and W/H?
    - void graphics.drawRect (int x, int y, int width, int height)
      Draws the outline of the specified rectangle. (also fillRect)
  - Inclusive or exclusive?
  - Which pixels are turned on for DrawRectangle (2,2, 8,8)?
  - Suppose you draw another rectangle next to it?
  - Suppose draw filled and outline rectangle with the *same* coordinates?
    - JavaScript SVG can control the order: https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/paint-order
  - What about for ellipse?



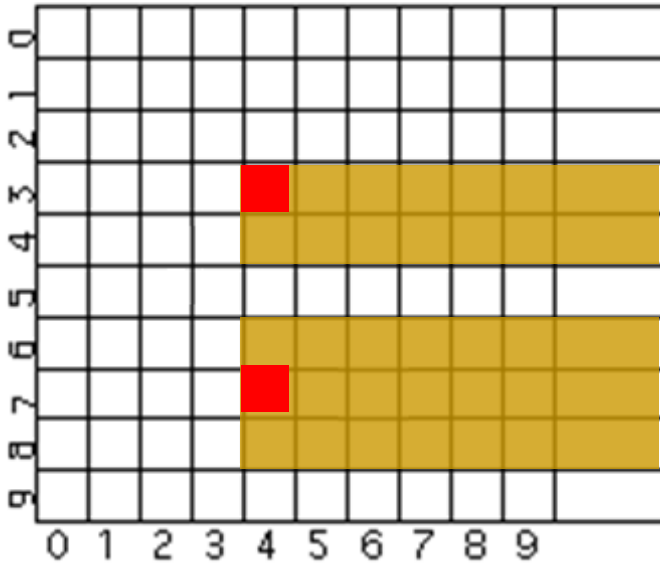


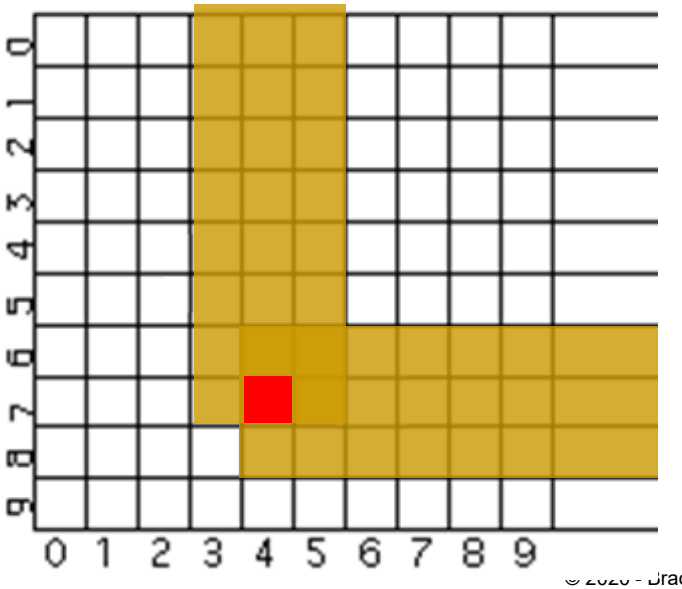
```
graphics.setColor(Color.red);
graphics.drawRect(4, 4, 40, 40);
graphics.setColor(Color.blue);
graphics.fillRect(0,0,4,4);
```

Primitives, 2

- DrawLine has similar concerns
  - Thick lines often go to *both sides* of the coordinates
  - Option in JavaScript for fully inside
- drawPolyline takes a sequence of points
  - Endpoints of each segment drawn?
  - Last end-point drawn?
  - Closed vs. open; may draw first point twice

Where to draw lines?



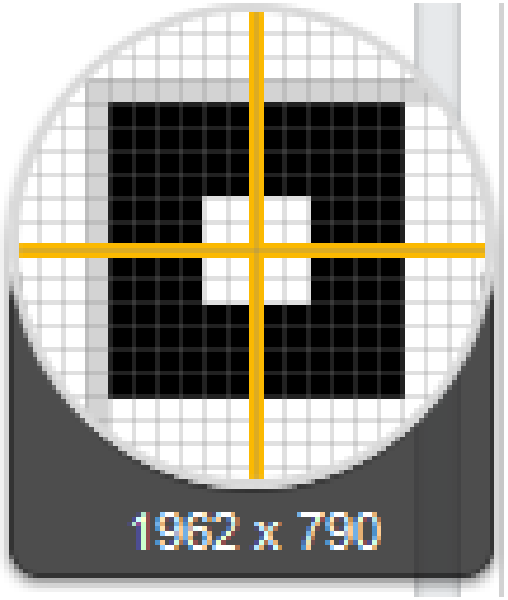


Inside or outside?

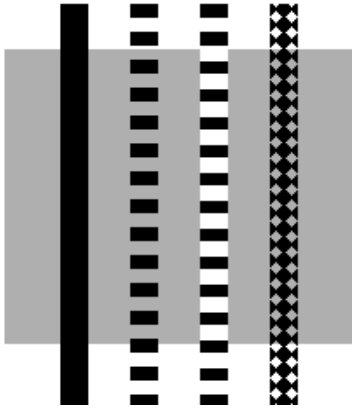
- How many pixels across are painted for line width = 4, rectangle width = 8?

```
let c = document.getElementById("myCanvas");
let ctx = c.getContext("2d");
ctx.lineWidth = 4;
ctx.strokeRect(2, 2, 8, 8);
```

12



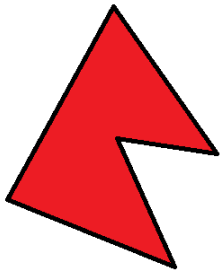
Line Properties

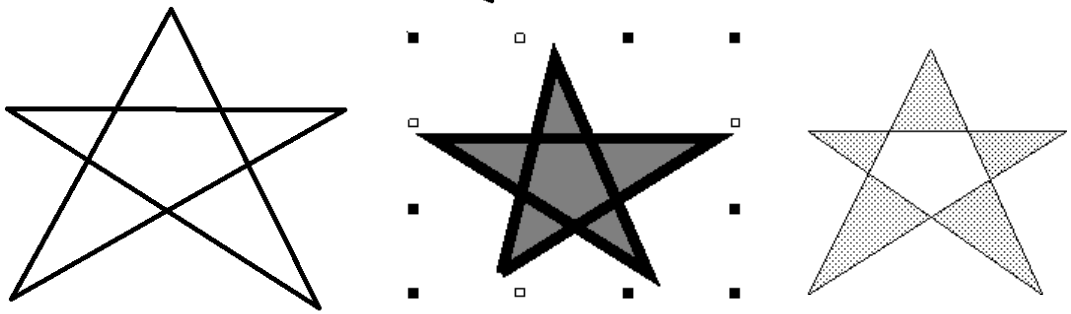


- LineStyles
  - Width
  - Solid, dashed 1110001110000111000, "double-dashed", patterned
- Cap-style: butt, round, projecting (by 1/2 linewidth):

Polylines

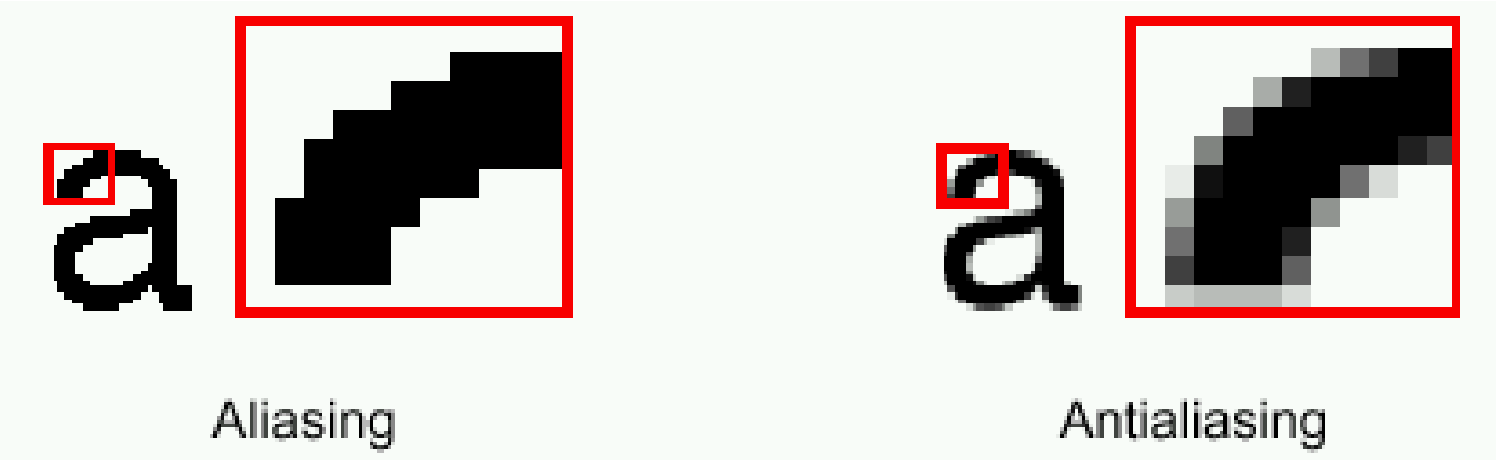
- End-caps: miter, round, bevel:
  - Miter = point, up to 11 degrees
    - JS miterLimit
  - Round = circle of the line width
  - Bevel = fill in notch with straight lin
- Filled, what parts?
  - "Winding rule"
    - JS: fill-rule:nonzero
  - "Odd parity rule"
    - JS: fill-rule:evenodd
    - Used by Java
  - JavaScript has both!
    - `fill-rule:winding`
    - `fill-rule:evenodd`





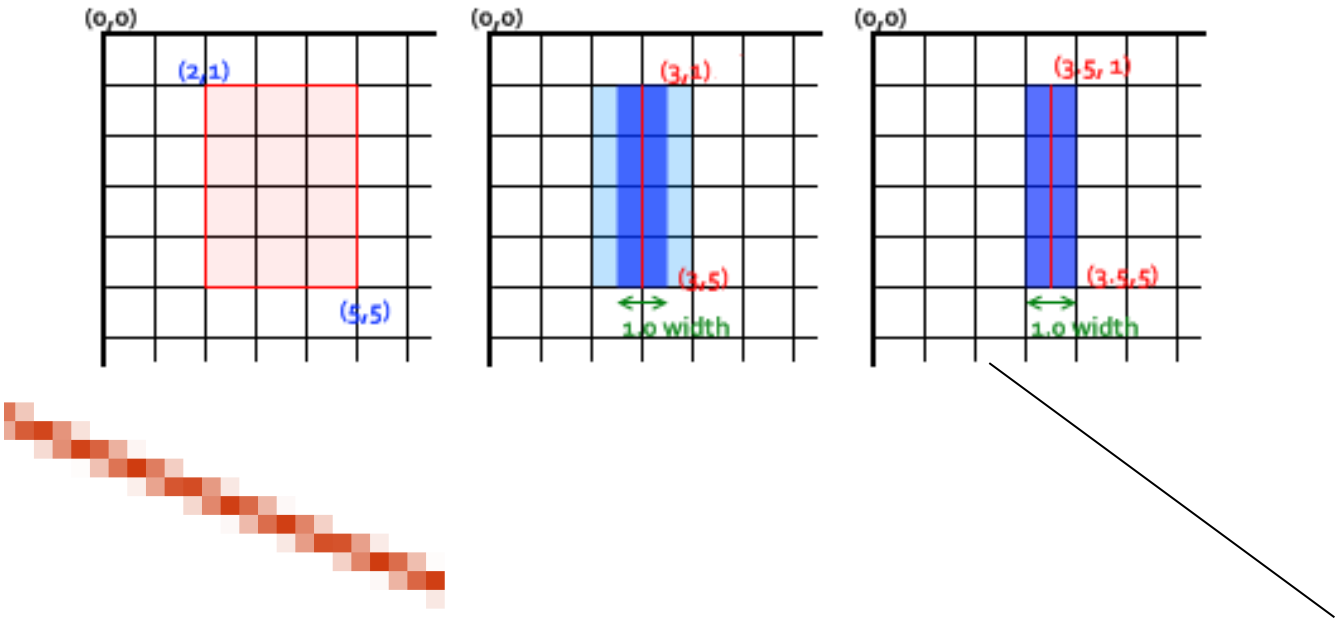
Anti-Aliasing

- Making edges appear smooth by using blended colors
- Useful for text and all lines
- Supported by Java RenderingHints parameter to Graphics2D
- JavaScript – always on, controlled by the browser



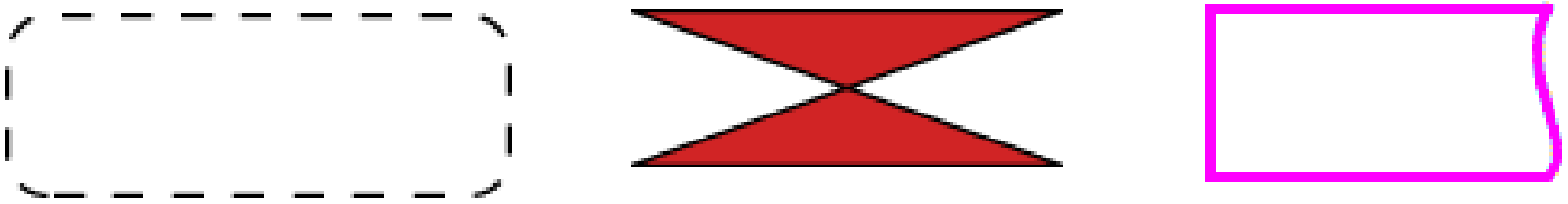
Anti-aliasing discussion

- https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Applying_styles_and_colors



Java2D & JavaScript Canvas Path Model

- Others (AWT, SVG) draw by drawing shapes (drawRect, <rect>, etc.)
- Path model: Define a path first, then stroke or fill it
  - Used in Java, Macintosh, Postscript, JS Canvas
- Can create a beginPath() and add moveTo, lineTo's, curveTo (etc.) to it, and then call stroke() or fill(), etc.



JavaScript Canvas

- Build up a path and then “stroke” or “fill” it
  - Implicit “default” path, or explicit path
  - Global (hidden) data structure holds the path and all parameters

```
function draw() {
  const canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    const ctx = canvas.getContext('2d');

    ctx.beginPath();
    ctx.moveTo(75, 50);
    ctx.lineTo(100, 75);
    ctx.lineTo(100, 25);
    ctx.closePath(); //optional for fill
    ctx.fill();
  }
}
```

ref

More JavaScript Canvas examples

```
<body>
  <canvas id="myCanvas" width="100%" height="100%"></canvas>
…
  <script>
      let c = document.getElementById("myCanvas");
      let ctx = c.getContext("2d");
      ctx.strokeStyle = "red";
      ctx.moveTo(10, 10);
      ctx.lineTo(20, 200);
      ctx.stroke();

      ctx.fillStyle = "green";
      ctx.fillRect(20, 20, 150, 100);
  </script>
</body>
```

JavaScript Canvas

```
<body>
  <canvas id="myCanvas" width="100%" height="100%"></canvas>
…
  <script>
       ctx.strokeStyle = "red";
       ctx.moveTo(10, 10);
       ctx.lineTo(20, 200);
       ctx.stroke();

       ctx.fillStyle = "green";
       ctx.fillRect(20, 20, 150, 100);
       ctx.moveTo(0,0);
       ctx.lineWidth = 5;
       ctx.strokeStyle = "blue";
       ctx.lineTo(0,100);
       ctx.stroke();

</script>
  </body>
```

JavaScript Canvas

```
<body>
  <canvas id="myCanvas" width="100%" height="100%"></canvas>
…
  <script>
        ctx.strokeStyle = "red";
        ctx.moveTo(10, 10);
        ctx.lineTo(20, 200);
        ctx.stroke();

        ctx.fillStyle = "green";
        ctx.fillRect(20, 20, 150, 100);
        ctx.moveTo(0,0);
        ctx.lineWidth = 5;
        ctx.strokeStyle = "blue";
        ctx.lineTo(0,100);
        ctx.stroke();

</script>
  </body>
```

- *Redraws previous stroke!*

JavaScript Canvas

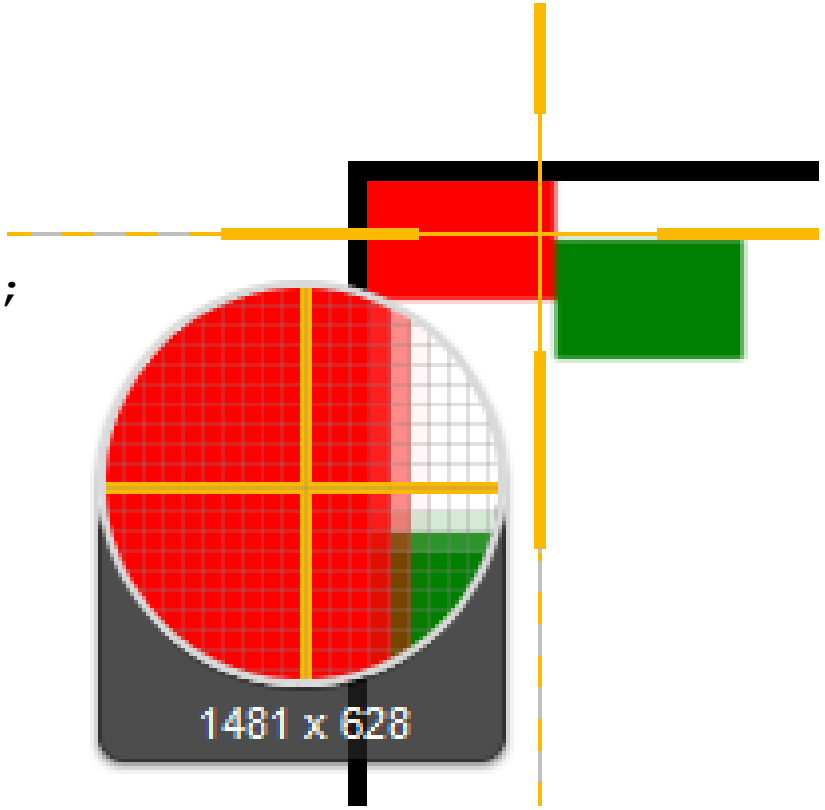
```
<body>
  <canvas id="myCanvas" width="100%" height="100%"></canvas>
…
  <script>
        ctx.strokeStyle = "red";
        ctx.moveTo(10, 10);
        ctx.lineTo(20, 200);
        ctx.stroke();

        ctx.beginPath();
        ctx.moveTo(10, 250);
        ctx.fillStyle = "green";
        ctx.fillRect(20, 20, 150, 100);
        ctx.moveTo(0,0);
        ctx.lineWidth = 5;
        ctx.strokeStyle = "blue";
        ctx.lineTo(0,100);
        ctx.stroke();

</script>
  </body>
```

- Need beginPath() between strokes!

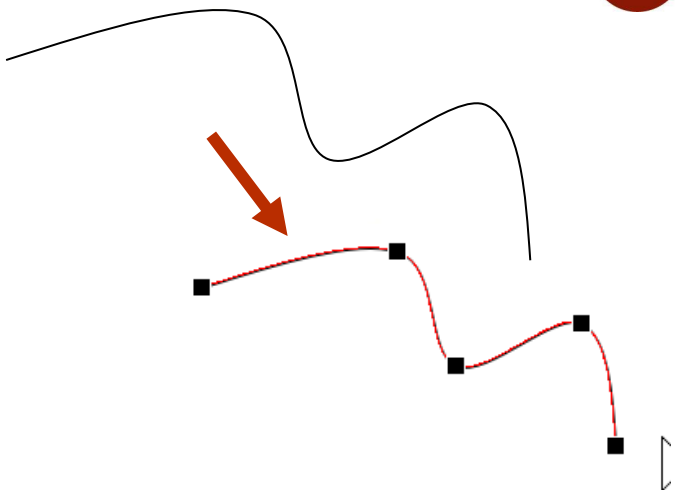
JavaScript Canvas

```
…
ctx.fillStyle = "red";
ctx.fillRect(0, 0, 20, 20);
ctx.fillStyle = "green";
ctx.fillRect(20, 10, 20, 20);
…
```

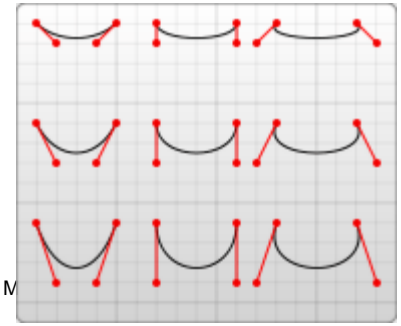
- Anti-aliasing makes it hard to control which pixels are on

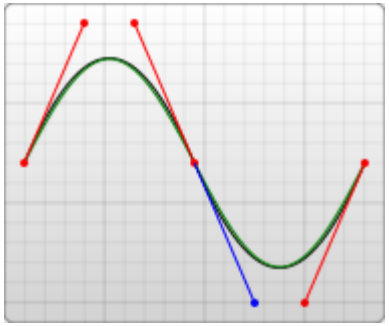


Splines



- Curves defined by “cubic” equation
  - $x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$
    $y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$
  - Well-defined techniques from graphics (not covered here – see Foley&vanDam)
- Defined based on “control” points
  - Different kinds do or don’t go through the control points
  - Available in both SVG and Canvas in JavaScript
  - “Bézier” curves
    - endpoints are on the curve, and control points are not
- Different from PowerPoint

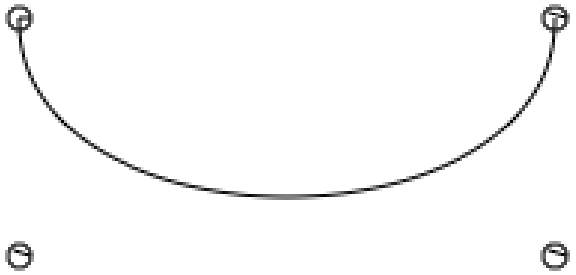




Bezier Curve Example

```
var c = document.getElementById("myCanvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.moveTo(20, 20);
ctx.bezierCurveTo(20, 100, 200, 100, 200, 20);
ctx.moveTo(20, 20);
ctx.arc(20,20,4,0,2*Math.PI);
ctx.moveTo(18, 98);
ctx.arc(20,100,4,0,2*Math.PI);
ctx.moveTo(198, 98);
ctx.arc(200,100,4,0,2*Math.PI);
ctx.moveTo(198, 18);
ctx.arc(200,20,4,0,2*Math.PI);
```

ref



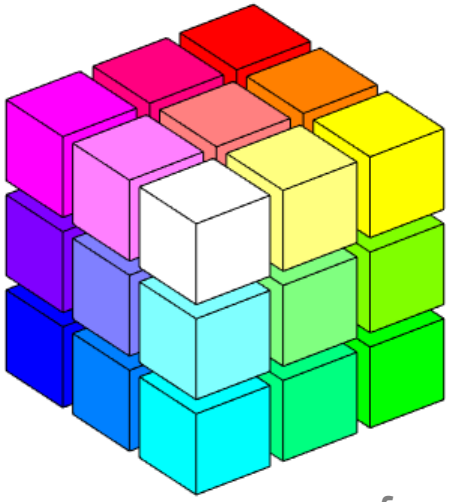
ctx.stroke();

- Note that Bezier curves do *not* go through the intermediate control points

Color Models

- *See online color picker / converter: https://colorizer.org/ or w3schools*
- RGB -- Additive color primaries
- CMY -- Cyan, Magenta, Yellow
  - complements of red, green, blue; subtractive primaries
  - colors are specified that are removed from white light, instead of added to black (no light) as in RGB
- YIQ -- used in color TVs in US (NTSC)
  - Y is luminance, what is shown on black and white TVs
  - I and Q encode chromaticity

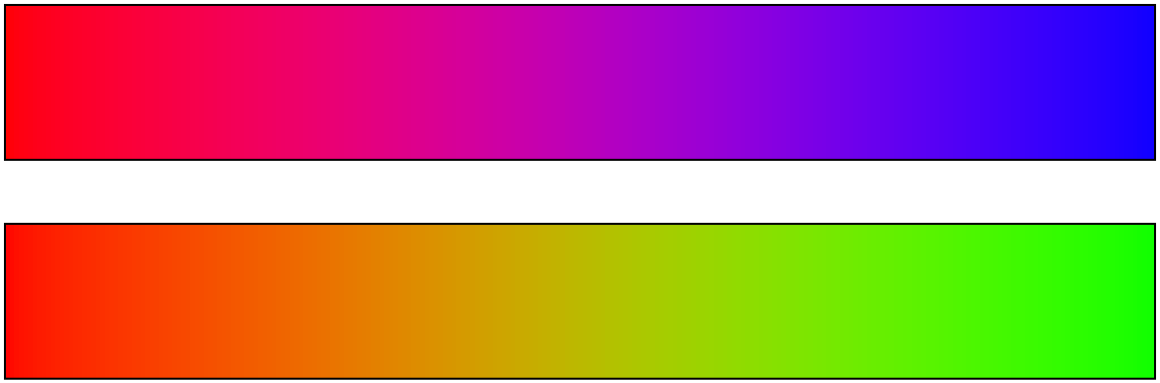
$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ 0.60 & -0.28 & -0.32 \\ 0.21 & -0.52 & 0.31 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$



ref

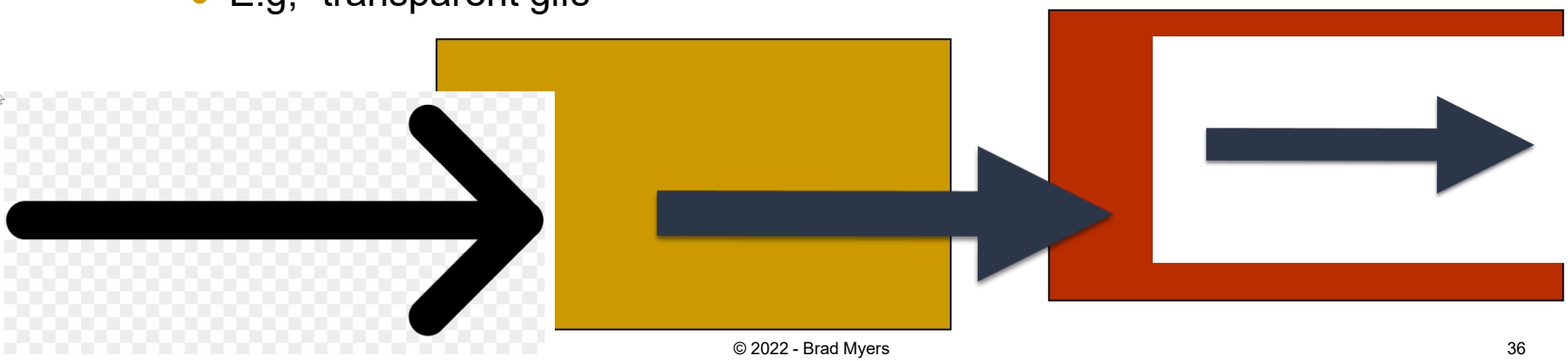
Color, cont.

- HSV -- Hue, Saturation and Value (brightness) or HSL (Luminance)
  - user oriented, intuitive appear of artist's hint, shade, tone
  - simple algorithm in text to convert, but not a linear mapping
- Interpolating between colors can be done using different models, with different resulting intermediate colors



Transparency of Color

- Original model used only opaque paint
  - Modeled hardcopy devices this was developed for (at Xerox PARC)
- Current systems now support “paint” that combines with “paint” already under it
  - e.g., translucent paint (“alpha” values)
- Intermediate
  - Icons and images can select one “transparent” color
    - E.g, “transparent gifs”

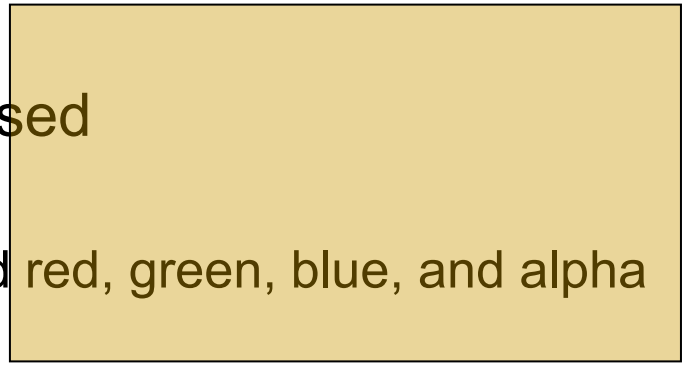


Paint with transparency

- Postscript model originated the “alpha blending” approach
  - Dominant model for hardcopy
- Java2D and JS drawing models also takes this approach

Alpha Blending

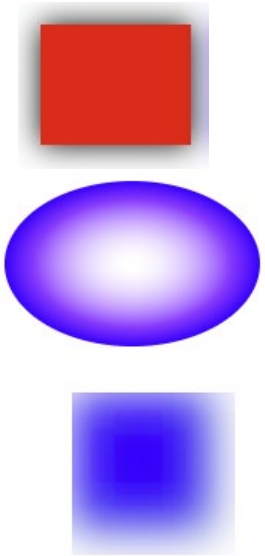
- Alpha is percent of this color to be used
  - rgba(red, green, blue, alpha)
  - Creates an rgba color with the specified red, green, blue, and alpha values
    - rgb in 0..255
    - a in the range 0.0 - 1.0



- Reverse of PowerPoint!
  - PowerPoint is % transparent so 100% = see through; 0 = opaque
  - Percent transparent = 1-percent alpha

Other painting parameters

- Shadows
- Many kinds of gradients
- Filters and blurring
- 3D (WebGL) …..



Fonts and drawing strings

- Font provides description of the shape of a collection of chars
  - Shapes are called “glyphs”
- Plus information e.g., about how to advance after drawing a glyph
- And aggregate info for the whole collection

Fonts

- Typically specified by:
  - A family or typeface
    - e.g., courier, helvetica, times roman
  - A size (normally in “points”)
  - A style
    - e.g., plain, italic, bold, bold & italic
    - other possibles (from mac): underline, outline, shadow

Font examples

- Fonts: Times, Helvetica, Courier, Symbol ✳❶❍✺❑●, *Zapf Chancery*
  - — Fixed width ("pitch") ("monospaced type"): W. . I@i
  - — Variable ("proportial") width: W..I@i
- Style: **Bold**, *Italic*, Underline, Outline, etc.
- Size: in "points" = 1/72 of inch.

  24 pts, 18 pts, 14 points, 10 points, 7 points

  *Not*screen (pixel) size: 7x9

Sizes can be deceiving (24 pt New York, bold)

Sizes can be deceiving (24 pt Monotype Corsiva)

Points

- An odd and archaic unit of measurement
  - 72.27 points per inch
    - Origin: 72 per French inch (!)
  - Postscript rounded to 72/inch
    - Most have followed
  - Early Macintosh: point==pixel (1/75th)

FontMetrics

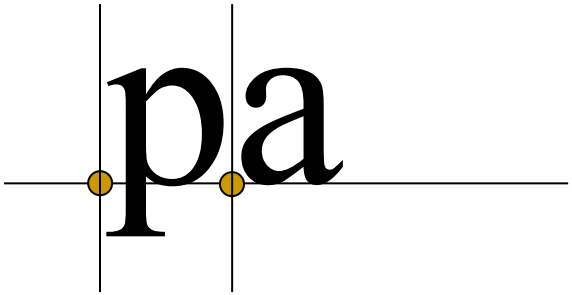
- Objects that allow you to measure characters, strings, and properties of whole fonts

Reference point and baseline

- Each glyph has a reference point
  - Draw a character at x,y, reference point will end up at x,y (not top-left)
  - Reference point defines a **“baseline”**

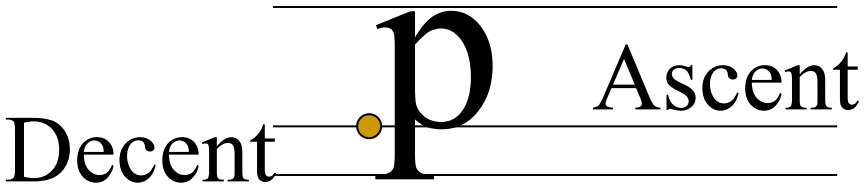
Advance width

- Each glyph has an “advance width”
  - Where reference point of next glyph goes along baseline



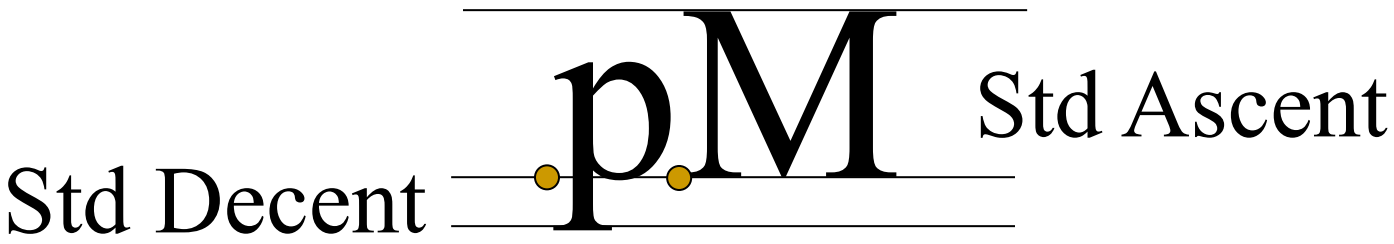
Ascent and decent

- Glyphs are drawn both above and below baseline
  - Distance below: “decent” of glyph
  - Distance above: “ascent” of glyph



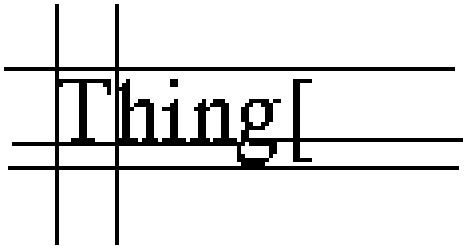
Standard ascent and decent

- Font as a whole has a standard ascent and standard decent



Leading

- Leading = space between lines of text
  - Pronounce “led”-ing after the lead strips that used to provide it
  - space between bottom of standard decent and top of standard ascent
    - i.e., interline spacing



Height

- Height of character or font
  - ascent + decent + leading
  - not standard across systems: on some systems doesn't include leading (but does in Java)
    - New question: is the leading above or below the text in Java?

Other Parameters

- Kerning: overlapping of characters: VA, ff, V.
- Stroke: Element of a character that would have originally been created with a single pen stroke
- Em: Equal to the font's point size. So an "Em-dash" in a 18 point font is 18points wide: (option-shift-underline on Mac)
- En: Half font's point size. "En-dash" is 9 points wide in 18 point font: (Mac: option-underline)
  - - DASHES – DASHES—DASHES

Types of Fonts

- **Bitmap fonts**: look bad when scaled up. Best appearance at native resolution.

  Times vs. Storybook

  - Sometimes used for dingbats, wingdings
- **Postscript fonts**: by Adobe, described by curves and lines so look good at any resolution, often hard to read when small
- **TrueType fonts**: similar to Postscript: font is a program

  abcd

  - Supported by Java: java.awt.font.TRUETYPE_FONT
- **OpenType**, etc. – web fonts are scalable

Encoding of Characters

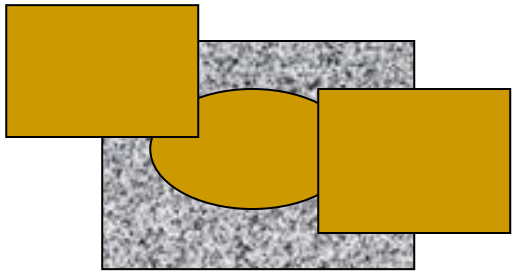
- Conventional ASCII
  - One byte per character
  - Various special characters in lower and upper part of fonts
    - Depends on OS, font, etc.
- Unicode: http://www.unicode.org
  - 16 bits per character
  - All the world's languages
  - Java and web use Unicode

Images

- Pictures created externally
  - “Bitmaps”, “Pixmaps”
- Various encodings
  - bmp, pict, gif, tiff, jpeg, png, …
- Issues:
  - Origin for when used as a cursor
  - Encodings for transparency
    - Windows cursors and gif files
    - Java uses alpha compositing
    - gif & png do support it, jpg does not

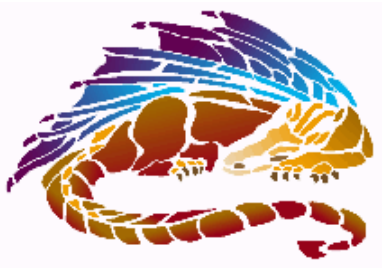
Clipping and "Stencils"

- X windows, Mac, etc. can clip drawing to a set of rectangles
  - Must be non-overlapping
  - Important for refresh
  - Can make drawing more efficient
- SVG – <clipPath> attribute
  - Define clipping path for children objects
- JS Canvas: ctx.clip();
  - Clip to arbitrary shape



"Stencils"

- Model is like the stencils used in crafts
  - Only see paint through the "holes"
- Used for transparency, icons, other effects
- Uses alpha compositing and shape clip mechanisms already discussed



The Starry Night

The Starry Night

Coordinate Transformations

- Supports
  - Translate - move
  - Rotate
  - Scale – change size (including flip = -scale)
  - Shear
- Can modify any shape, including text
- To fully understand, need matrix algebra:
  - Affine transformations are based on two-dimensional matrices of the following form:

$$\begin{bmatrix} a & c & t_x \\ b & d & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{where } x' = ax + cy + t_x \text{ and } y' = bx + dy + t_y$$

How Parameters are Passed

- How pass parameters for drawing operations?
- Issue: Lots of parameters control the drawing of objects.
  - X drawline has at least 19
  - How many for Canvas or SVG?

DrawLine Parameters

1. Window in which to draw
2. X1
3. Y1
4. X2
5. Y2
6. relative-p
7. line-width
8. draw function
9. background-color
10. foreground-color
11. cap style
12. line style (solid, dashed, double-dashed)
13. dash pattern
14. dash offset
15. stipple bitmap
16. stipple origin X
17. stipple origin Y
18. clip mask
19. plane mask (for drawing on specific planes)

How Pass Parameters?

- Three basic possibilities
  - Pass all parameters with each operation
    - DrawLine(70,30,70,200, Red, ...)
    - - too many parameters
    - Not really used by any modern system

Passing Parameters, 2

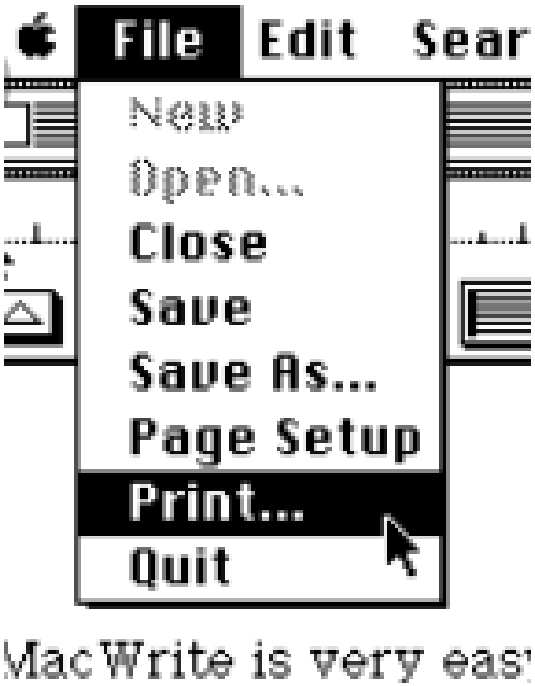
- All parameters stored in the system
  - Used by Macintosh, Display Postscript, etc.
  - Sometimes called the “pen”
  - Example (pseudo code):
    - SetColor(Red)
      MoveTo(70, 30)
      DrawTo(70, 200)
  - + Fewer parameters to calls
  - + Don't have to set properties don't care about
  - - Interrupts, multi-processing, may result in unexpected settings

Passing Parameters, 3

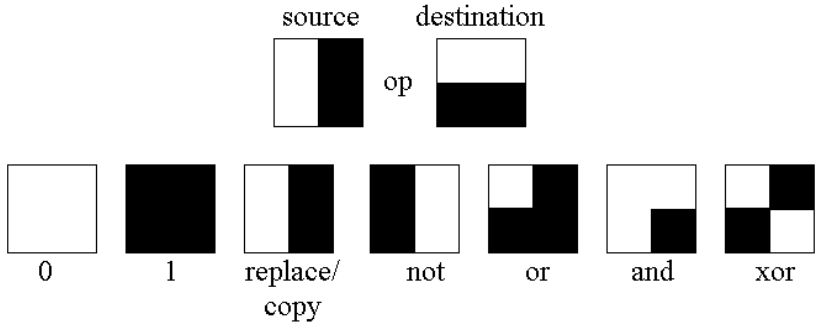
- Store parameters in an object
  - JavaScript canvas “context”
    - `canvas.getContext("2d");`
  - X = “graphics context”
  - Windows = “device context”
    - corresponds to surface, but can push and pop
  - Java
    - “Graphics”, Graphics2D, Graphics3D objects
      - Lots of settings
  - Android
    - Has BOTH graphics object and Paint object
    - Parameters are in the Paint object

Historical reference

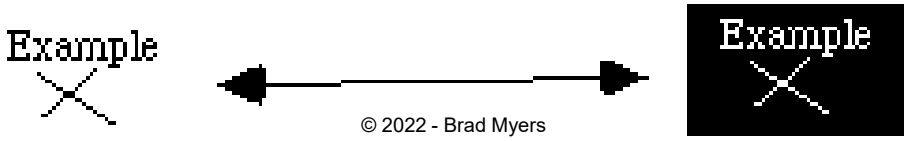
- Early machines were all monochrome
  - Each pixel was black or white
- Slow graphics
- Tricks for highlighting and "grey"
  - "halftone" – every other pixel on



Draw Function



- Replace (COPY)
- XOR
- And, OR, NOT, etc.
- Makes it important to do the points only once
- Anything XOR BLACK = inverted anything, XOR again and get original:
- AND useful for making holes
- Doesn't work well in color
- Java: Paint or XOR (setXORMode or setPaintMode)



RasterOp (BitBlt, CopyArea)

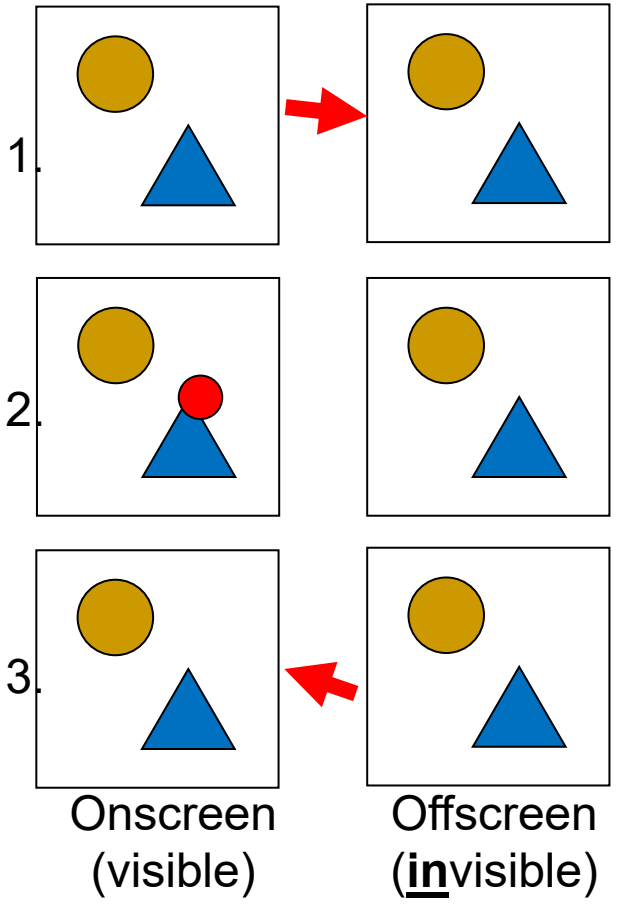
- Copy an area of the screen
  graphics.copyArea (int x, int y, int width, int height, int dx, int dy)
  Copies an area of the component by a distance specified by dx and dy.
- Used to have ability to combine with destination using Boolean combinations
- Useful for moving, scrolling, erasing & filling rectangles, etc.
- SmallTalk investigated using it for rotate, scaling, etc.
- Not nearly as useful in color

Double Buffering

- (Needed for Homework 3)
- Save an extra picture offscreen
  - Smoother animations
  - Save hidden parts of windows
- = “Backing store”
- Use two buffers for special effects, faster refresh
- “Save-under” for pop-ups
- Use this for the temporary canvas for homework3
  - Need a way to draw interim feedback

Double Buffering (for a Canvas)

- 1. Make an off-screen copy of screen
- 2. Draw interim object on-screen
- 3. Erase by copying off-screen one to on-screen
- Repeat steps 2-3 as interim object moves
- (or make permanent by copying to off-screen)



Double Buffering for JavaScript Canvas

- In workarea, have 3 full-size elements on top of each other
  - Create a temporary canvas on top of the “regular” canvas

```
<div id="workarea">
  <canvas id="workarea-canvas" width="800" height="800"></canvas>
  <canvas id="tempCanvas" width="800" height="800"></canvas>
  <svg id="workarea-svg" width="800" height="800"></svg>
</div>
```

  - Remember, later one is on top – SVG is on top of Canvas, by specification
- Make sure to control which one is visible

```
workareaCanvas.style.display = "block";
tempCanvas.style.display = "none";
```

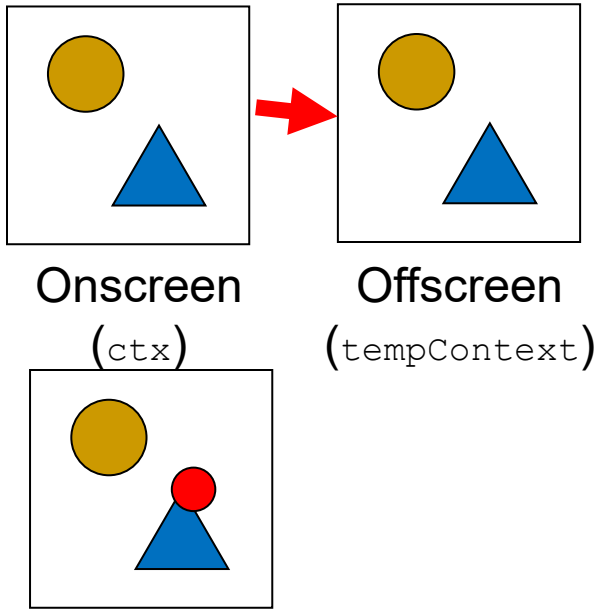
- Get temp’s context for drawing:

```
let tempContext = tempCanvas.getContext("2d");
```

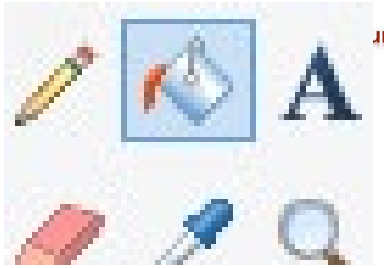
- Copy contents from real (`ctx`) to temp canvas use: drawImage:
  https://www.w3schools.com/tags/canvas_drawimage.asp

```
tempContext.drawImage(workareaCanvas, 0, 0);
```

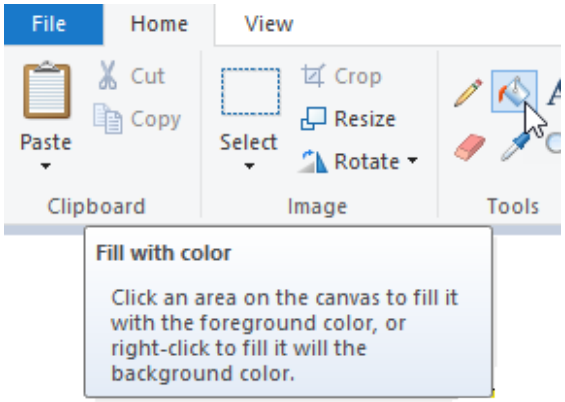
- Now can draw on-screen, and easily restore old view

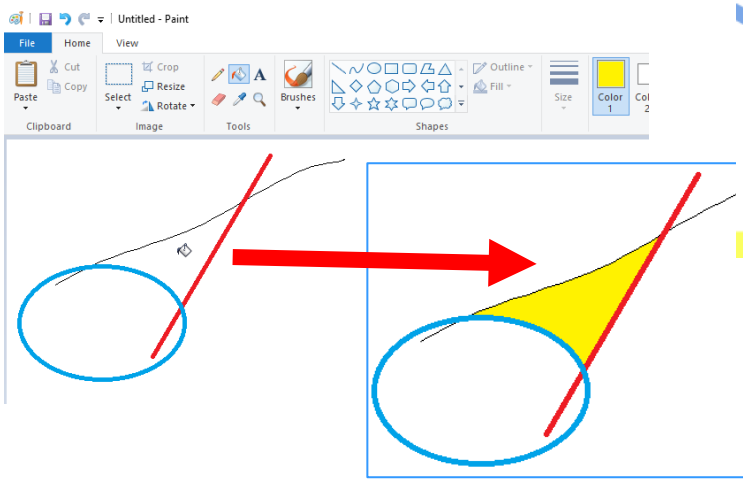


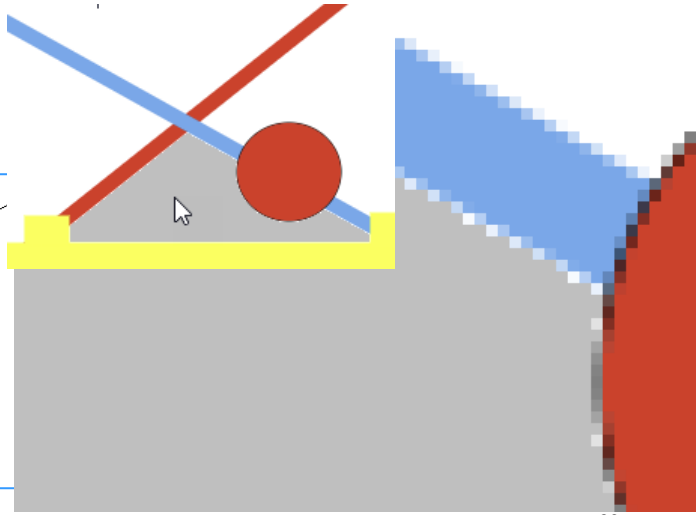
Flood Fill



- You will do this for homework 3
  - We give you an implementation
- Only available in painting programs
- Issue: floodfill is SLOW, so don't worry if it is taking a while
  - Added a wait cursor
  - Try not to click in the background! = 32 seconds!
- Issue: anti-aliasing
  - Don't worry about anti-aliasing for hw3







HW 3 hints: Modes

- Similar to HW2 – need to enable and disable many handlers, and whole Canvas or SVG
  - Lots of global variables to keep track of the modes
  - Or use an enum, and think of it as a state machine
    - No enums in JS, but can use strings or const numbers
    - State machines will be covered in lecture 9
- Grey out controls using styles *and* remove their event handlers, e.g.,
  - Both handler: `borderColorSelection.classList.add("disabled");`
  - Other handlers: `borderColorSelection.classList.remove("disabled");`
  - Or can put a `div` in front of them that is ½ transparent and also takes the events
- Make svg or canvas layer appear and disappear using `display`

```
svg.style.display = "block";  // regular display
canvas.style.display = "none"; // invisible
```

Radio buttons

- Make the 3 radio buttons be mutually exclusive: use same "`name`" property
  - Handlers for each button controls modes for other controls and layers