EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 3 due: 11:59 PM, Monday, October 3
- Assignment 4 due: 11:59 PM, Monday, October 17

Any Questions?

In-Class Problem Solution

- 13-(9-23) In-Class Problem Solution.pptx

Any Questions?

Chapter 17 - Drawing on Canvas

- ~~Displaying Graphics~~
- ~~SVG~~
- ~~The canvas element~~
- ~~Lines and surfaces~~
- ~~Paths~~
- ~~Curves~~
- ~~Drawing a pie chart~~
- ~~Text~~
- Images
- Transformation
- Storing and clearing transformations
- Choosing a graphics interface

Sprites

- When drawImage is given *nine* arguments, it can be used to draw only a fragment of an image.
- The second through fifth arguments indicate the rectangle (x, y, width, and height) in the source image that should be copied.
- The sixth to ninth arguments give the rectangle (on the canvas) into which it should
- be copied.
- This can be used to pack multiple *sprites* (image elements) into a single image file and then draw only the part you need.
- For example, we have this picture containing a game character in multiple poses:

```
<canvas></canvas>
<script>
let cx = document.querySelector("canvas").getContext("2d");
let img = document.createElement("img");
img.src = "img/player.png";
let spriteW = 24, spriteH = 30;
img.addEventListener("load", () => {
  let cycle = 0;
  setInterval(() => {
    cx.clearRect(0, 0, spriteW, spriteH);
    cx.drawImage(img,
    // source rectangle
    cycle * spriteW, 0, spriteW, spriteH,
    // destination rectangle
    0, 0, spriteW, spriteH);
    cycle = (cycle + 1) % 8;
  }, 120);
});
</script>
```

Sprites

- To animate a picture on a canvas, the clearRect method is useful.
- It resembles fillRect, but instead of coloring the rectangle, it makes it transparent, removing the previously drawn pixels.
- We know that each sprite, each subpicture, is 24 pixels wide and 30 pixels high.
- This code loads the image and then sets up an interval (repeated timer) to draw the next frame.

```
<canvas></canvas>
<script>
let cx = document.querySelector("canvas").getContext("2d");
let img = document.createElement("img");
img.src = "img/player.png";
let spriteW = 24, spriteH = 30;
img.addEventListener("load", () => {
  let cycle = 0;
  setInterval(() => {
    cx.clearRect(0, 0, spriteW, spriteH);
    cx.drawImage(img,
    // source rectangle
    cycle * spriteW, 0, spriteW, spriteH,
    // destination rectangle
    0, 0, spriteW, spriteH);
    cycle = (cycle + 1) % 8;
  }, 120); // 120 milliseconds
});
</script>
```

Sprites

- The cycle binding tracks our position in the animation.
- For each frame, it is incremented and then clipped back to the 0 to 7 range by using the remainder operator (%).
- This binding is then used to compute the x-coordinate that the sprite for the current pose has in the picture.

```
<canvas></canvas>
<script>
let cx = document.querySelector("canvas").getContext("2d");
let img = document.createElement("img");
img.src = "img/player.png";
let spriteW = 24, spriteH = 30;
img.addEventListener("load", () => {
  let cycle = 0;
  setInterval(() => {
    cx.clearRect(0, 0, spriteW, spriteH);
    cx.drawImage(img,
    // source rectangle
    cycle * spriteW, 0, spriteW, spriteH,
    // destination rectangle
    0, 0, spriteW, spriteH);
    cycle = (cycle + 1) % 8; //cycle = 0, 1, 2, 3, 4, 5, 6, 7
  }, 120);
});
</script>
```

Any Questions?

Transformation

- Calling the scale method will cause anything drawn after it to be scaled.
- This method takes two parameters, one to set a horizontal scale and one to set a vertical scale.

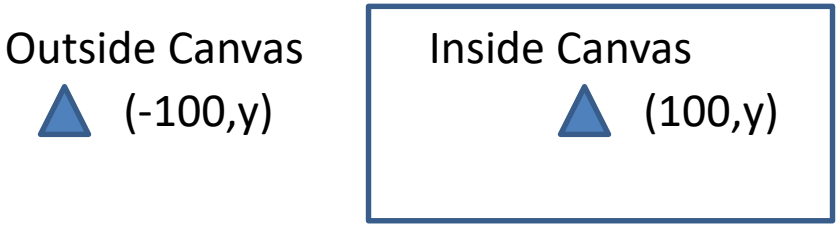
```
<canvas></canvas>
<script>
   let cx = document.querySelector("canvas").getContext("2d");
   cx.scale(3, .5); //scale everything from here on
   cx.beginPath(); //tells Canvas we're going to draw a path
   cx.arc(50, 50, 40, 0, 7); //draw a circle
   cx.lineWidth = 3; //set line width to 3 pixels
   cx.stroke(); //draw it
</script>
```

Because of the call to scale, the circle is drawn three times as wide and half as high.

Any Questions?

Scaling

- Scaling will cause everything about the drawn image, including the line width, to be stretched out or squeezed together as specified.
- Scaling by a negative amount will flip the picture around.
- The flipping happens around point (0,0), which means it will also flip the direction of the coordinate system.
- When a horizontal scaling of -1 is applied, a shape drawn at x position 100 will end up at what used to be position -100.
- So to turn a picture around, we can't simply add cx.scale(-1, 1) before the call to drawImage because that would move our picture outside of the canvas, where it won't be visible.
- You could adjust the coordinates given to drawImage to compensate for this by drawing the image at x position -50 instead of 0.

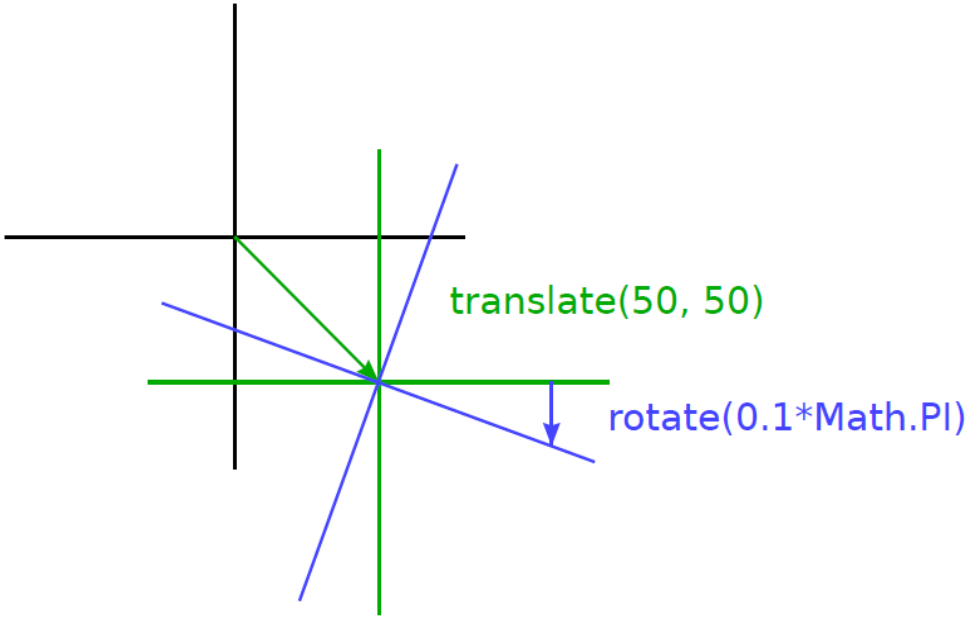


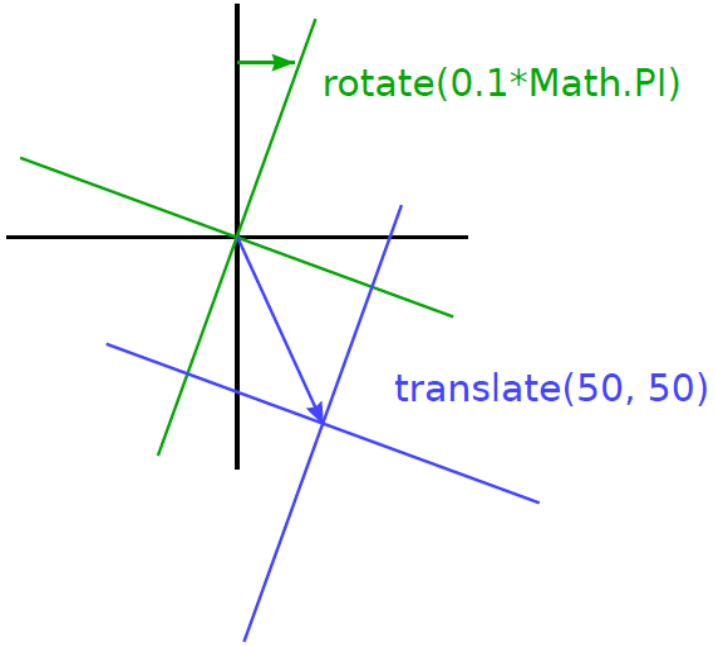
Rotate and Translate

- Another solution, which doesn't require the code that does the drawing to know about the scale change, is to adjust the axis around which the scaling happens.
- You can rotate subsequently drawn shapes with the rotate method …
- and move them with the translate method.
- The interesting and confusing thing is that these transformations *stack*, meaning that each one happens relative to the previous transformations.

Rotate and Translate

- If we first translate the center of the coordinate system to (50,50) and then rotate by 20 degrees (about 0.1π radians), that rotation will happen around point (50,50).





- But if we *first* rotate by 20 degrees and *then* translate by (50,50), the translation will happen in the rotated coordinate system and thus produce a different orientation.
- The order in which transformations are applied matters.

Any Questions?

Flipping an Image

- Remember our sprite that we got to run left to right?
- What if we wanted it to run right to left?

Flipping an Image

- First let's look at how we can flip an image horizontally.
- Then we will look at how to flip the sprite image.

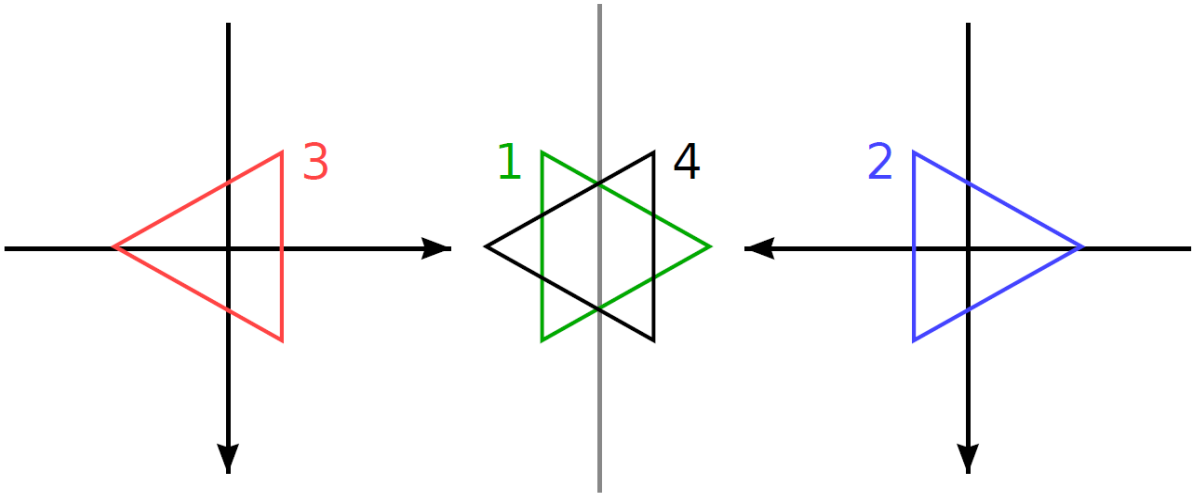
- To flip a picture around the vertical line at a given x position, we can do the following:

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```

Flipping an Image

```
function flipHorizontally(context, around) {
  context.translate(around, 0); //2
  context.scale(-1, 1); //3
  context.translate(-around, 0); //4
}
```

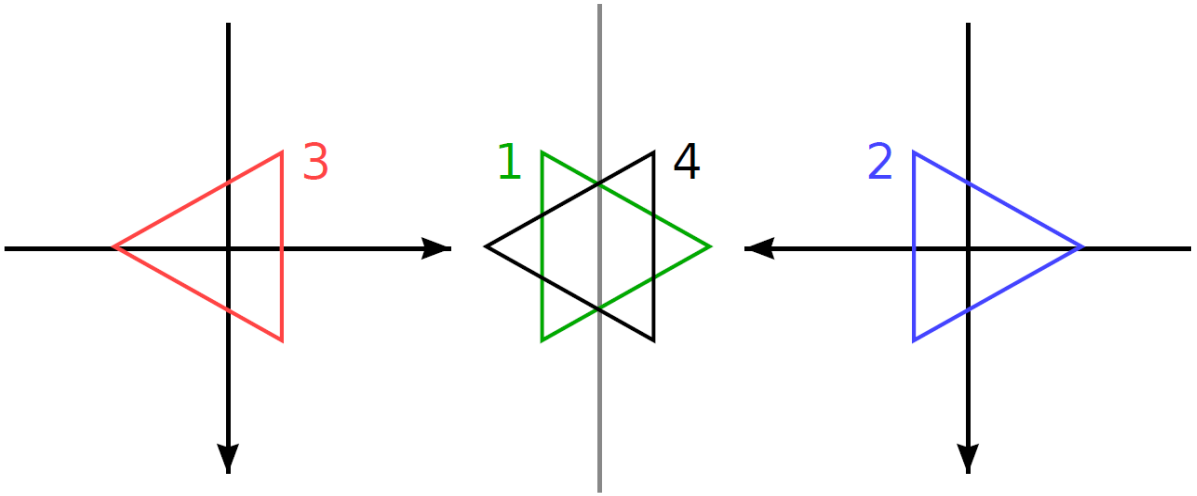
- The picture explains why this works.
- This shows the coordinate systems before and after mirroring across the central line.
- The triangles are numbered to illustrate each step.



Flipping an Image

```
function flipHorizontally(context, around) {
  context.translate(around, 0); //2
  context.scale(-1, 1); //3
  context.translate(-around, 0); //4
}
```

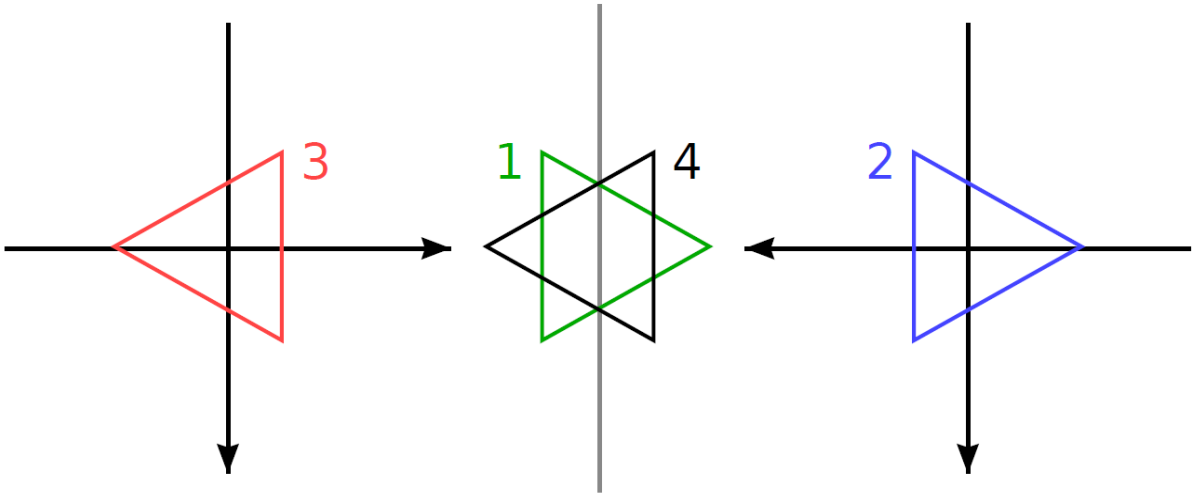
- If we draw a triangle at a positive x position, it would, by default, be in the place where triangle 1 is.
- A call to flipHorizontally first does a translation to the right, which gets us to triangle 2.



Flipping an Image

```
function flipHorizontally(context, around) {
  context.translate(around, 0); //2
  context.scale(-1, 1); //3
  context.translate(-around, 0); //4
}
```

- It then scales, flipping the triangle over to position 3.
- This is not where it should be, if it were mirrored in the given line.
- The second translate call fixes this—it "cancels" the initial translation and makes triangle 4 appear exactly where it should.



Flipping an Image

- We can now draw a mirrored character at position (100,0) by flipping the world around the character's vertical center.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");//create Canvas window
  //next two lines create HTML: <img src="img/player.png">
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;//define width and height of sprite
  //create event handler to wait for Canvas to draw image, i.e., load
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);//flip Canvas window
    //draw flipped sprite in Canvas window
    let cycle = 0;
    setInterval(() => {
     cx.clearRect(0, 0, spriteW, spriteH);
     cx.drawImage(img, cycle * spriteW, 0, spriteW, spriteH, 0, 0, spriteW, spriteH);
     cycle = (cycle + 1) % 8;
    }, 120);});
</script>
```

Any Questions?

Storing and Clearing Transformations

- Transformations stick around.
- Everything else we draw after drawing that mirrored character would also be mirrored.
- That might be inconvenient.
- It is possible to save the current transformation, do some drawing and transforming, and then restore the old transformation.
- This is usually the proper thing to do for a function that needs to temporarily transform the coordinate system.
- First, we save whatever transformation the code that called the function was using.
- Then the function does its thing, adding more transformations on top of the current transformation.
- Finally, we revert to the transformation we started with.

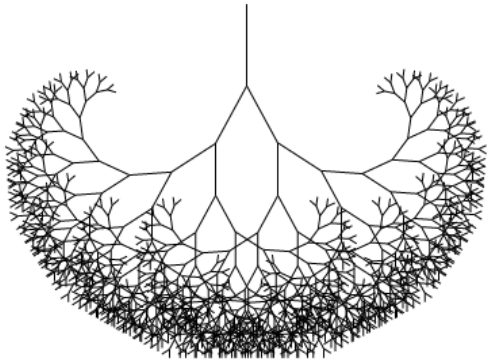
Storing and Clearing Transformations

- The save and restore methods on the 2D Canvas context do this transformation management.
- They conceptually keep a stack of transformation states.
- When you call save, the current state is pushed onto the stack, …
- and when you call restore, the state on top of the stack is taken off and used as the context's current transformation.
- You can also call resetTransform to fully reset the transformation.

Storing and Clearing Transformations

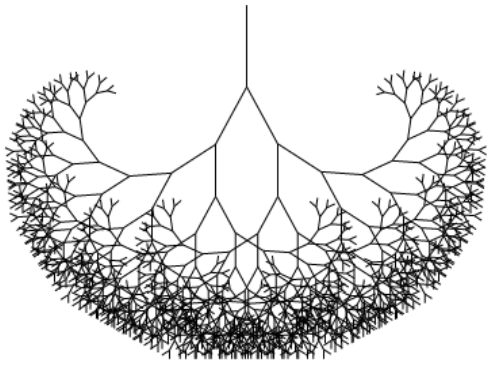
- The branch function in this example illustrates what you can do with a function that changes the transformation and then calls a function (in this case itself), which continues drawing with the given transformation.

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```



Storing and Clearing Transformations

- This function draws a treelike shape by drawing a line, …
- moving the center of the coordinate system to the end of the line, …
- and calling itself twice …
- first rotated to the left …
- And then rotated to the right.
- Every call reduces the length of the branch drawn, …
- and the recursion stops when the length drops below 8.

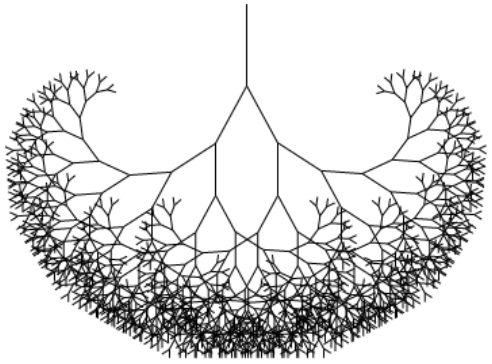


Finish commentin

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length); //draw a line
    if (length < 8) return;
    cx.save();
    cx.translate(0, length); //move coordinate system
    cx.rotate(-angle); //rotate left
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle); //rotate right
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```

Storing and Clearing Transformations

- If the calls to save and restore were not there, …
- the second recursive call to branch would end up with the position and rotation created by the first call.
- It wouldn't be connected to the current branch but rather to the innermost, rightmost branch drawn by the first call.
- The resulting shape might also be interesting, but it is definitely not a tree.



```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```

Any Questions?

Choosing a Graphics Interface

- So when you need to generate graphics in the browser, you can choose between plain HTML, SVG, and Canvas.
- There is no single best approach that works in all situations.
- Each option has strengths and weaknesses.
- Plain HTML has the advantage of being simple.
- It also integrates well with text.
- Both SVG and Canvas allow you to draw text, but they won't help you position that text or wrap it when it takes up more than one line.
- Although you can wrap text in SVG using CSS.
- In an HTML-based picture, it is much easier to include blocks of text.
- SVG can be used to produce crisp graphics that look good at any zoom level.
- Unlike HTML, it is designed for drawing and is thus more suitable for that purpose.

Choosing a Graphics Interface

- Both SVG and HTML build up a data structure (the DOM) that represents your picture.
- This makes it possible to modify elements after they are drawn.
- If you need to repeatedly change a small part of a big picture in response to what the user is doing or as part of an animation, doing it in a Canvas can be needlessly expensive.
- The DOM also allows us to register mouse event handlers on every element in the picture (even on shapes drawn with SVG).
- You can’t do that with Canvas.

Choosing a Graphics Interface

- But Canvas's pixel-oriented approach can be an advantage when drawing a huge number of tiny elements.
- The fact that it does not build up a data structure but only repeatedly draws onto the same pixel surface gives Canvas a lower cost per shape.
- There are also effects, such as rendering a scene one pixel at a time …
- for example, using a ray tracer…
- or postprocessing an image with JavaScript (blurring or distorting it), …
- that can be realistically handled only by a pixel-based approach.

Choosing a Graphics Interface

- In some cases, you may want to combine several of these techniques.
- For example, you might draw a graph with SVG or Canvas but show textual information by positioning an HTML element on top of the picture.
- For nondemanding applications, it really doesn't matter much which interface you choose.

Any Questions?

In-Class Problem

- Write a canvas script to draw the following half of an ellipsoid centered at (50,50) with a radius of 40.
- Include required HTML tags.
- Comment each line of Canvas and HTML code.
- Hint: It is half a circle that is drawn three times as wide and half as high.