# “Hello Canvas” ICE

## Part 0 - Overview

In this exercise you will learn how to obtain a “2D Graphics context” for drawing, and how to draw rectangles and text into the graphics context.

We’ll be discussing this at our next class meeting, so you should have this done before our next class meeting, but it is due at the time listed on the dropbox.

Make sure that you read all of the explanatory notes below and in the start file, so don’t just blast through the exercise and type in the code without thinking about what each line is doing. (If you don’t know what each line is doing, ask!)

## Part I - Get Started

1) Download `template.html` from my courses, rename it to `hello-canvas.html`, and preview it in a browser. You should see a `<canvas>` tag with a gray border, and there is an “init called” log in the console.

2) Add the following code to `init()`:

```
// A - canvas variable points at <canvas> tag
var canvas = document.querySelector(‘canvas’);

// B - the ctx variable points at a “2D drawing context”
var ctx = canvas.getContext(‘2d’);

// C - all fill operations are now in yellow
ctx.fillStyle = ‘yellow’;

// D - fill a rectangle with the current fill color
ctx.fillRect(0,0,640,480);
```

**Explanation:**

A. Get a reference to the `<canvas>` element and store it in a variable named canvas. In this case the canvas variable is scoped to the `init()` function and is not visible outside of `init()`.

B. Get a “2D drawing context” for the canvas element. This drawing context has methods for drawing and other operations, as well as 18 “state variables” that represent things like the current font, the current fill color, and the current line width.

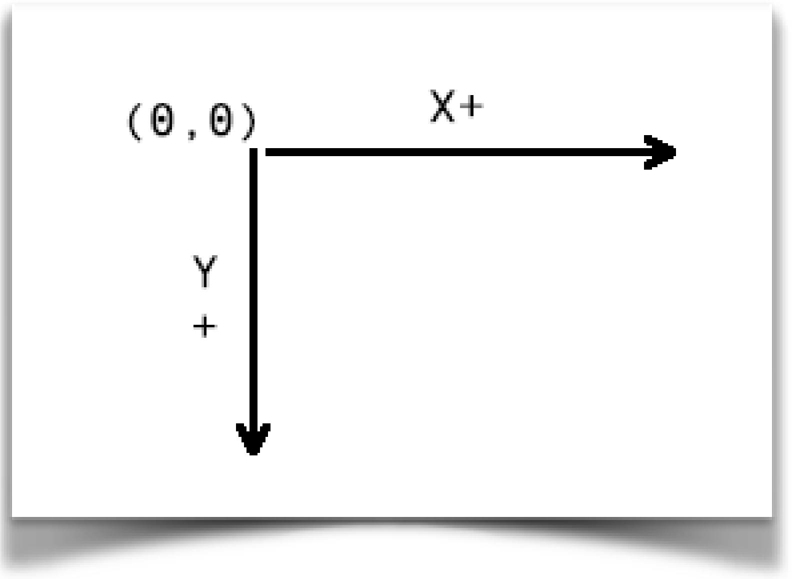
C. The `fillStyle` property of the drawing context sets the color for all operations that involve filling a path. This value can be a solid color, a gradient, or a pattern.

D. The `fillRect(x,y,width,height)` method fills a rectangle with the current fill color. (0,0) is the upper left corner of the canvas, NOT the screen.

Load the page, the entire canvas tag should be yellow. Code not working? Before you raise your hand and ask for help, check the JavaScript console and find the issue yourself.

**Note:** In the canvas coordinate system, the upper-left corner is `(0,0)`.

*x* gets larger moving to the right on the page, and *y* gets larger moving down the page.



## Part II - “Filling text”

1) Add the following code to the end of `init()`

```
// E - set the current font
ctx.font = 'bold 60pt Verdana';

// F - change the current fill color
ctx.fillStyle = ‘#44aa44’;

// G - draw and fill text using current fill color
ctx.fillText(‘Ace Coder’,50,240);
```

**Explanation:**

E. Set the `.font` property of the drawing context used whenever text is draw into the canvas. The `.font` property can hold any valid CSS rule.
Ex. `‘small-caps bold italic 100pt courier,monospace’`
http://www.w3schools.com/cssref/pr_font_font.asp

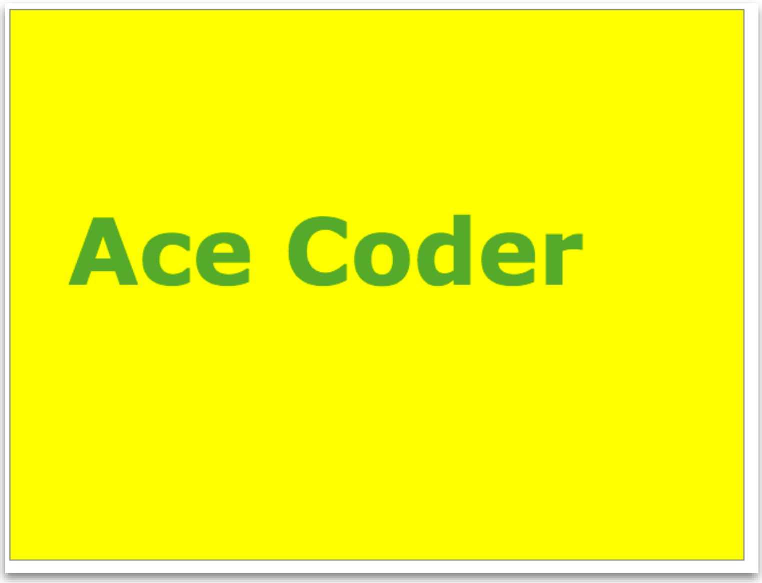
F. Change the `.fillStyle` of the drawing context. All future fill operations will now use this color.

G. Put the text on the screen with `ctx.fillText(“string”,x,y);`
Note that the text and the background rectangle are different colors because we changed the `.fillStyle` to a greenish color right before we filled the text.

You’ve probably noticed that canvas uses the “painter’s model” where our later drawing calls (like the text) are drawn on top of earlier drawing calls (the yellow rectangle).

## Part III - Make some changes

My boring version looks like this:



**Make your version different:**

1) Change the yellow rectangle to a different solid color.

2) Change the text to your name, and change the color of the text to something else. Note: Color can be specified in any legal CSS way, i.e. pre-defined keyword, hex, shorthand hex, RGB value, RGB percentage, RGBA, or even HSL

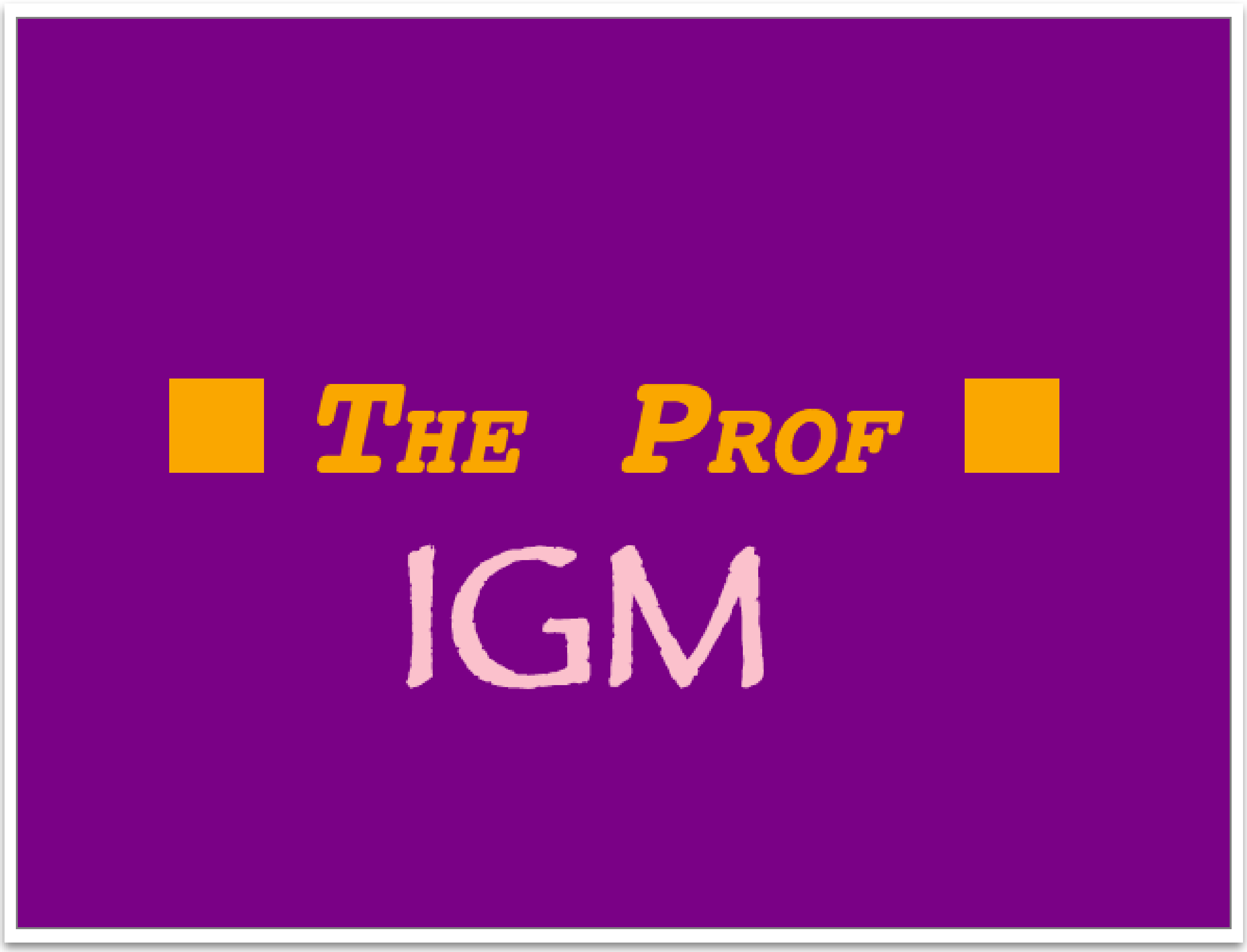
http://www.w3schools.com/cssref/css_colors_legal.asp

3) Change the font of your name to something else.

4) Add another line of text that reads "IGM". Make sure that this text appears in a different color and font than your name did.

5) Add two squares to your canvas (50 x 50 or so) using `ctx.fillRect(x,y,width,height)` – and make sure that the square color is the same as the first text you drew (your name).

Here's an example:

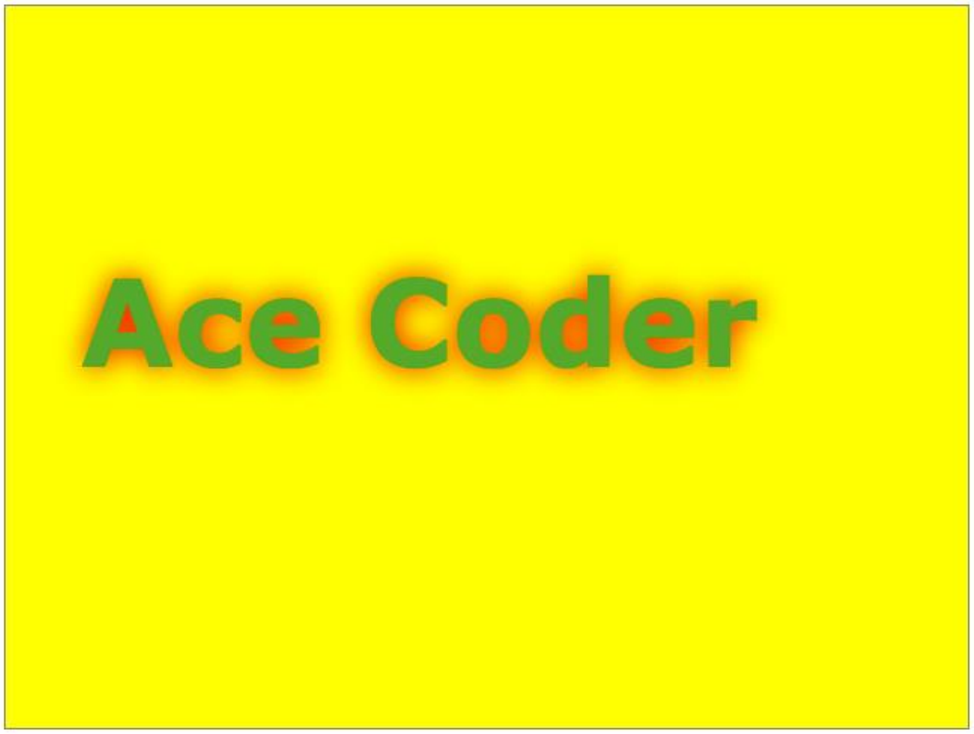


## Part IV - More canvas state variables

1) Add the following code to `init()`, right before you start drawing text.

```
// set shadow properties
ctx.shadowColor = "red";
ctx.shadowOffsetX = 0;
ctx.shadowOffsetY = 0;
ctx.shadowBlur = 30;
```

These are 4 more **drawing state variables** that will effect all of the future drawing operations, until they are changed to other values (or back to the default values).



Note: There are 15 canvas state variables in total, and you can see the list here: http://www.w3.org/TR/2dcontext/#the-canvas-state

## Part V - Drawing Centered Text

One way to draw centered text is to set the text alignment and text baseline before you draw any text. This is accomplished by setting 2 more canvas drawing state variables.

1) Add the following code to `init()`, right before you start drawing text.

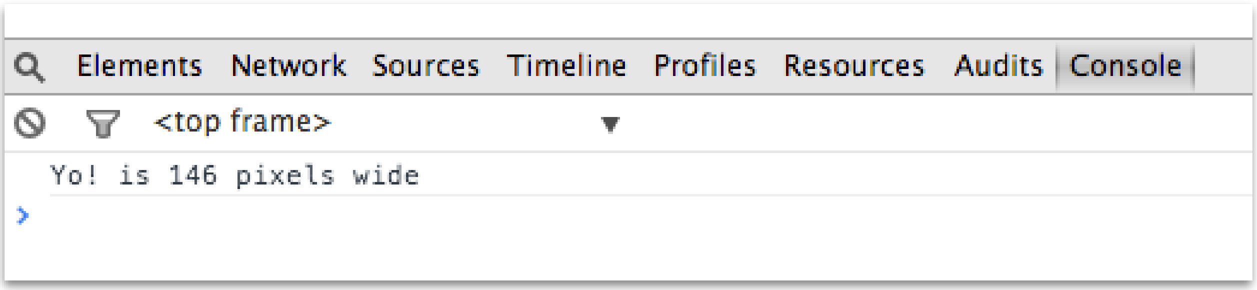
```
// horizontal alignment
ctx.textAlign = “center";

// vertical alignment
ctx.textBaseline = “middle";
```

2) Now if you want to center the text on the canvas, draw it at the center coordinates of the canvas:

```
ctx.fillText('Ace Coder',320,240);
```

Another way to precisely position text is to measure its size with `ctx.measureText()` before you draw it on the screen, and then position it accordingly.

```
// Getting Font Metrics
var metrics = ctx.measureText("Yo!");
console.log("Yo! is " + metrics.width + " pixels wide”);
```

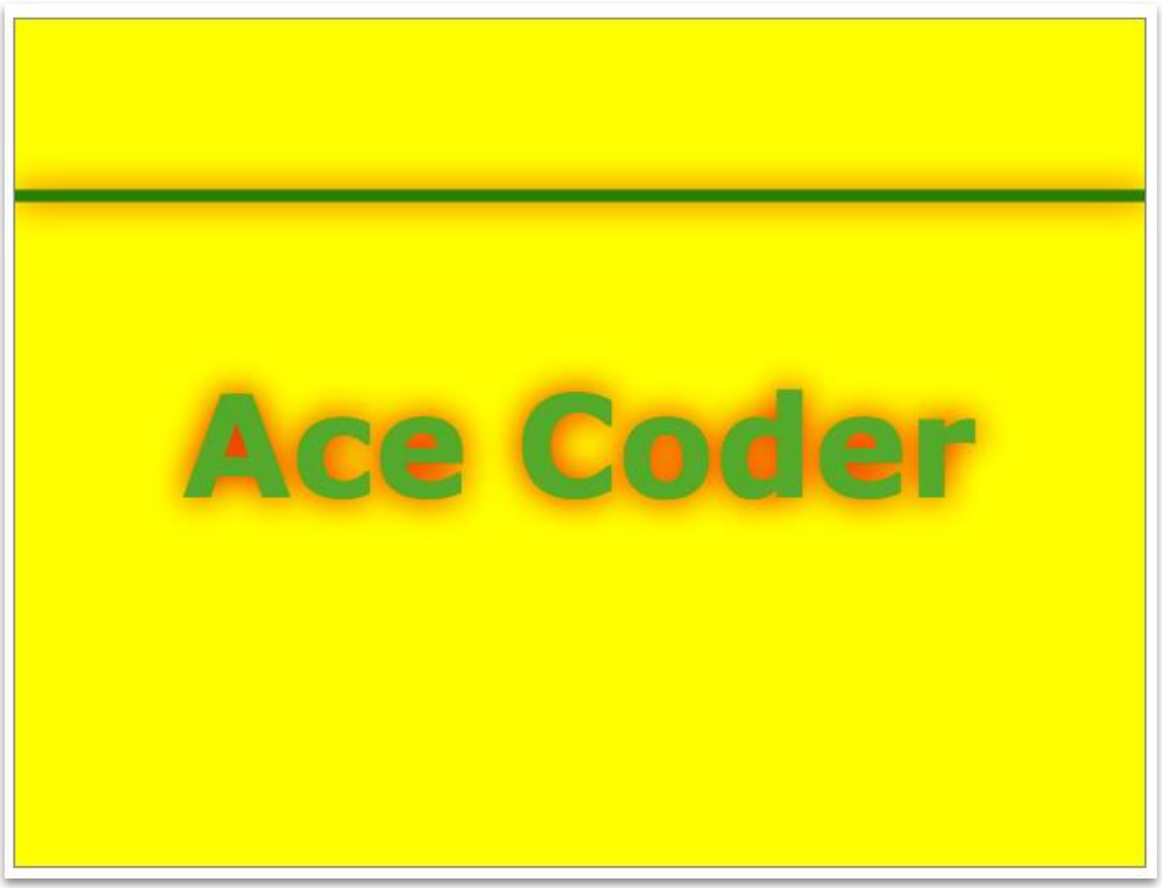


```
Yo! is 146 pixels wide
```

## Part VI - Drawing lines

1) Add the following code to the end of `init()`:

```
// 2 more drawing state variables
ctx.strokeStyle = "green";
ctx.lineWidth = 7;

// start defining a path - in this case just a line
ctx.beginPath();

// move a "pen" to 0,100
ctx.moveTo(0,100);

// draw to 640,100
ctx.lineTo(640,100);

// close the path
ctx.closePath();

// stroke the path so we can see it
ctx.stroke();
```



## Part VII - The drawing state "stack"

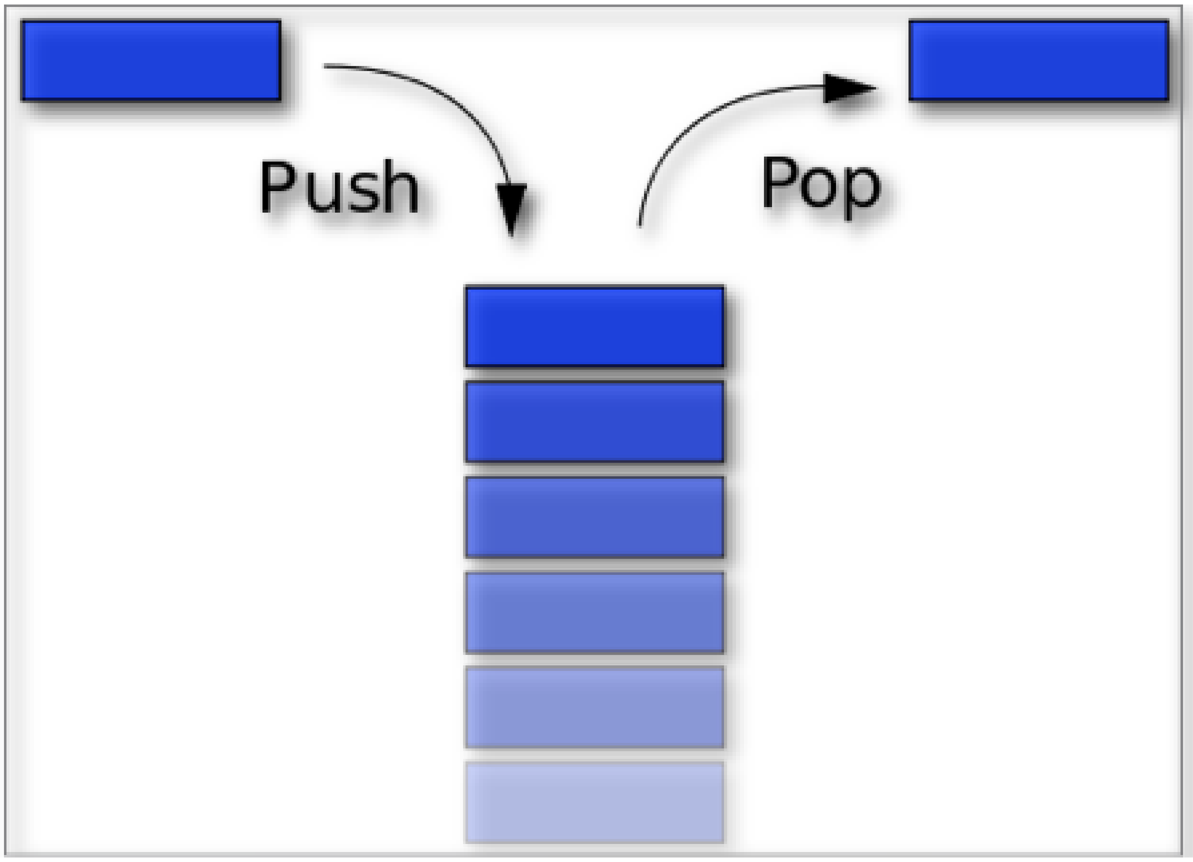
1) Notice that the line we just made has the red blurry shadow like the text does. What if we wanted our line to be free of those effects, but didn't want to change the 4 "shadow related" state variables back manually.

Solution, the drawing state stack!

A) Add this code right BEFORE the "set shadow properties" code from Part IV

```
ctx.save();
```

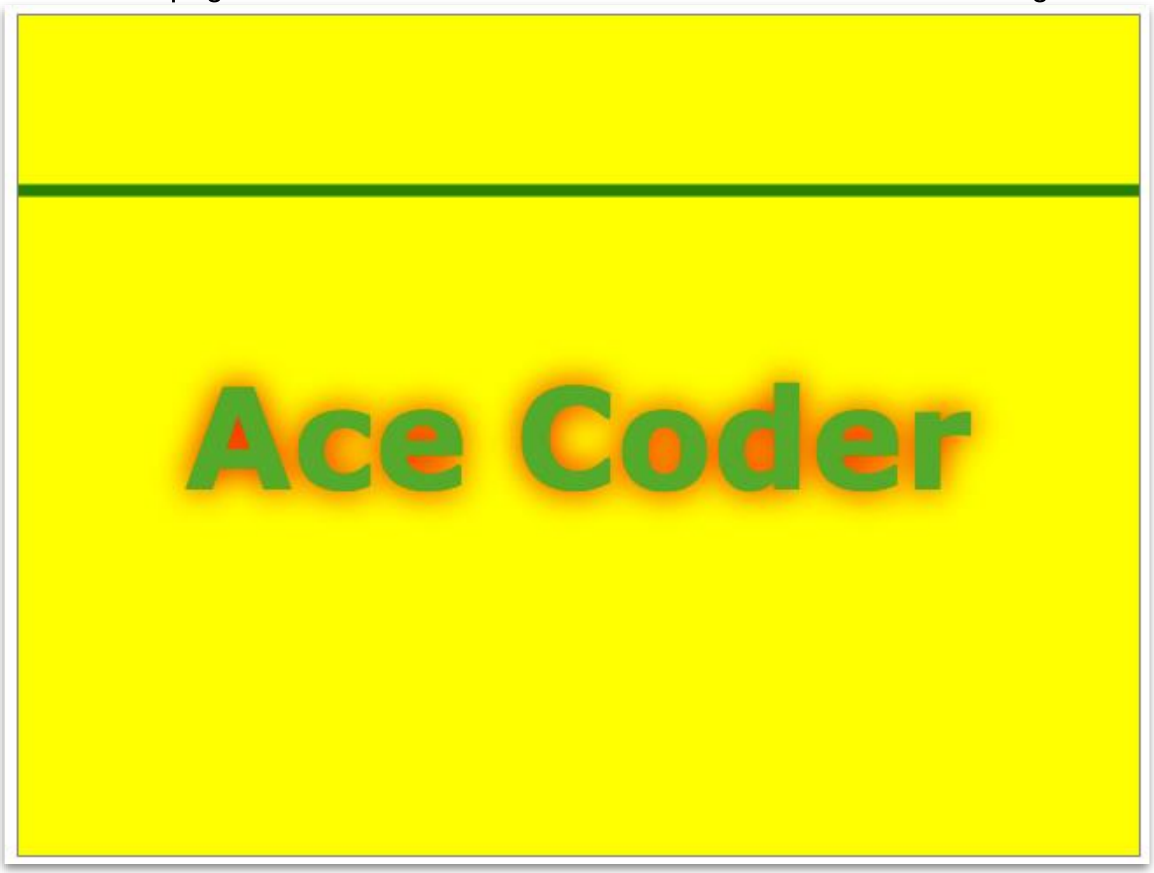
This code will save (push) the values of all 15 drawing state variables onto a "stack"



B) Now we can retrieve those values and get rid of all of the changes we made to the drawing context state variables since the last `ctx.save()`, by adding the following code right BEFORE the code in Part VI where we start drawing our line:

```
ctx.restore(); // this "pop" goes back to the last save()
```

Reload the page - the text will have a red shadow, but the line will no longer have one.



Look over this `ctx.save()` and `ctx.restore()` code carefully. Try moving the `save()` and `restore()` calls to different locations in the code to see how the drawing changes.

This manipulation of the drawing state "stack" will come in handy later, especially when we have to draw rotated or scaled objects.

## Part VIII – Loading Web Fonts

For us to make cool media apps and custom interfaces (app menus, game UI, etc), we're going to want custom fonts. Fonts say a lot about the theme and experience.

Beyond that, web fonts give us the most consistent experiences across mobile and desktop devices. A common practice these days is just to include the font you want in the web page.

Unfortunately, there is a small issue. Custom fonts **DO NOT** load in the browser until they are used. This is for performance reasons.

In HTML tags, this is not a worry and does not cause an issue, but in canvas, we're working with a graphics buffer. The browser does not detect that the font is in use in the buffer.

We need to make sure the browser detects that the font is in use so that canvas can use it. Otherwise, the canvas will show the default font.

*Now there is a proper way to do this in JS by asynchronously downloading the font, but it's a bit advanced at this point in the course. For now we will just use some simple CSS tricks.*

1) Notice the following two web fonts included head of the document. These web fonts are provided by https://fonts.google.com/ (a great place for fonts!).

   These two fonts will not be loaded until they are used.

```
<link href="https://fonts.googleapis.com/css?family=Indie+Flower"
      rel="stylesheet">

<link href="https://fonts.googleapis.com/css?family=Lobster"
      rel="stylesheet">
```

2) Inside of your CSS style tag, add the following.

```
/** Loading fonts without JS loader
    We are making a hidden area on the page where we can put
    text to force the fonts to load, but it will not be
    viewable to the user. Once these fonts are loaded into
    memory, they will be available to the canvas.
**/
#fontLoader {
  height: 0;
  width: 0;
  overflow: hidden;
}
#indie {
  font-family: 'Indie Flower', cursive;
}
#lobster {
  font-family: 'Lobster', cursive;
}
```

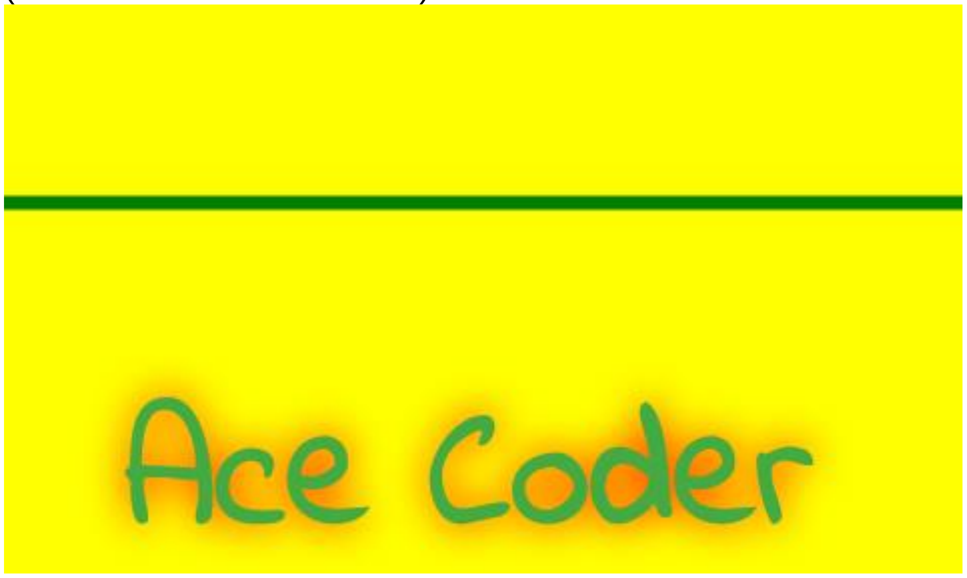
3) Inside your HTML, let's add a section to load our fonts. This will make sure the font is loaded because it's being used in the HTML. Our CSS above prevents the user from seeing it though.

   Since this will load the fonts into memory, our canvas will be able to use them.

```
<canvas width="640" height="480">
     Get a real browser!
</canvas>
<div id="fontLoader">
     <p id="indie">Indie Font</p>
     <p id="lobster">Lobster Font</p>
</div>
```

4) Change your canvas font to Indie Flower or Lobster.

(shown with Indie Flower font)



## Part IX - Finish it up

1) Easy!

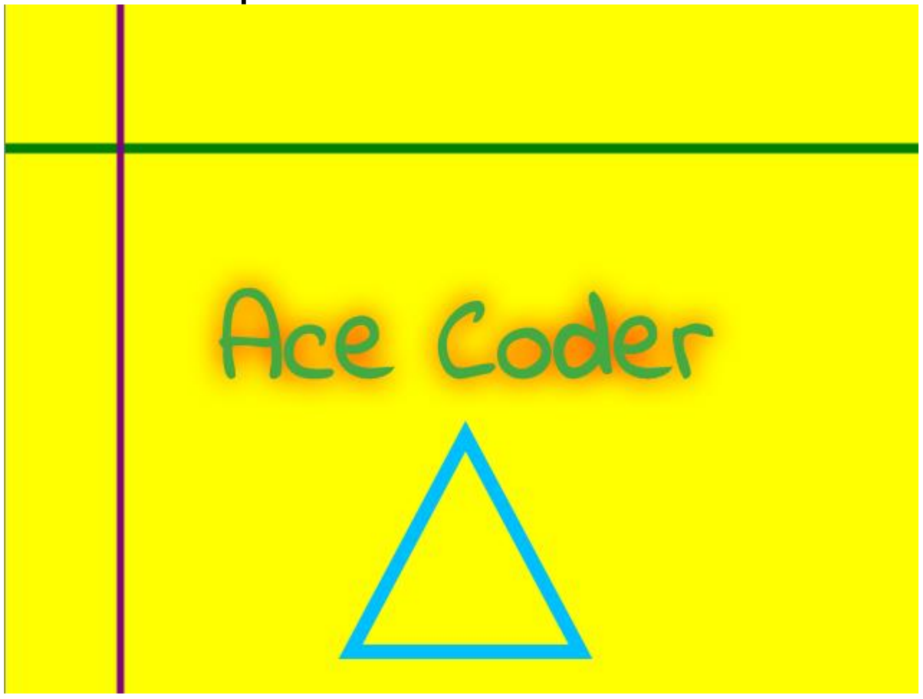
   i) Draw another vertical line on the canvas - give it a different thickness and color than the first line.

   ii) Create a triangle and put it somewhere on the page. Hint: use the the line drawing code as a guide, and add one or two more `ctx.lineTo()` calls. If you can't figure it out, use "the google" for ideas.

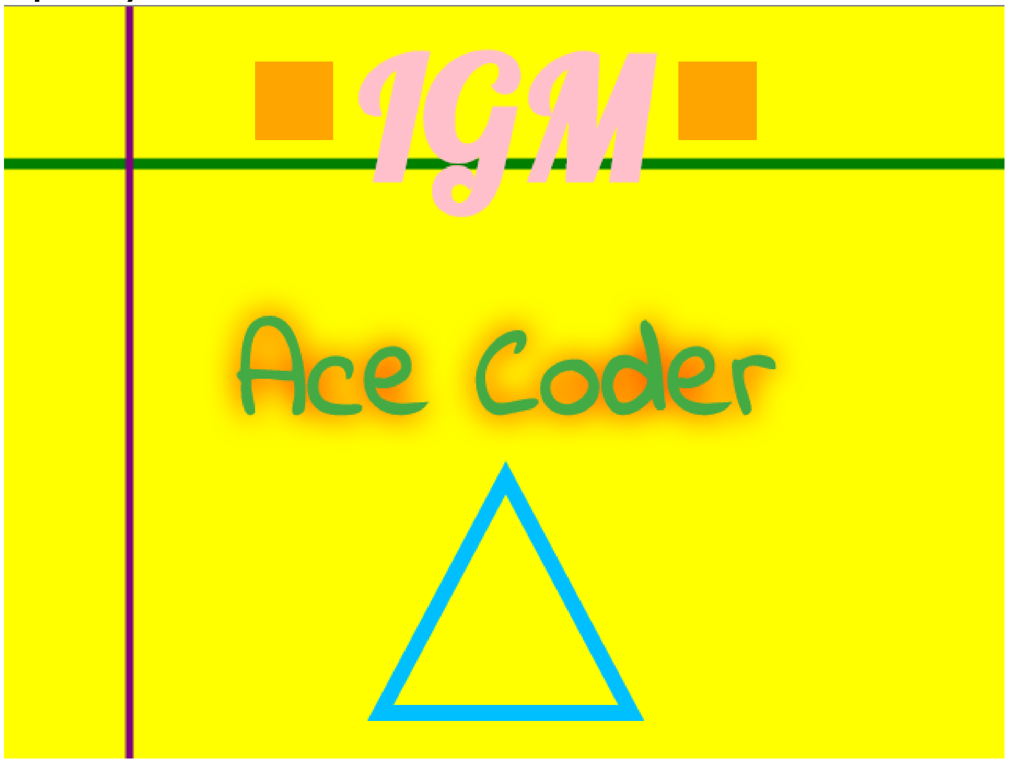
   iii) Add something new! Make this your own creation. You have some creative freedom. You can even use different web fonts.

**2)** Make sure the text and squares from part III are present**. Before the due date, upload your files to banjo, ZIP up your code and post the zip to the dropbox ALONG with a link to your page on banjo in the submission comments.**

**Here's an example:**

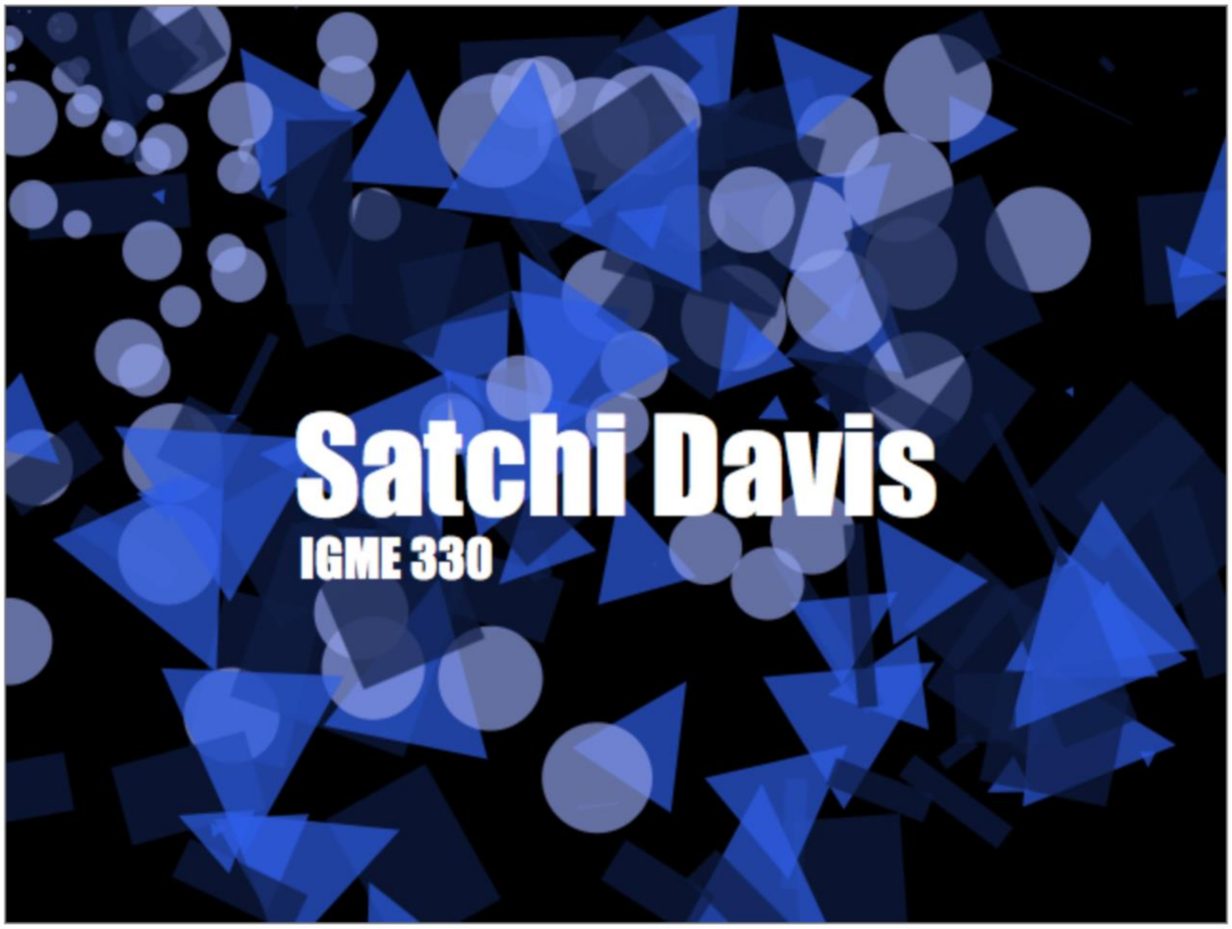


**And don't forget that your final version will also need your additions (text and squares) from Part III:**



**As long as your submission meets the rubric, you have some *creative freedom*. Make something you′re proud of!**

**Here′s a submission that was very nicely done:**



## Submission

### Rubric

| DESCRIPTION | SCORE | VALUE % |
|---|---|---|
| Finished – All sections of the assignment are completed. | | **20** |
| Rectangles Drawn – Rectangles drawn in final version. | | **10** |
| Lines Drawn – Lines drawn in final version. | | **10** |
| Triangles Drawn – Triangles drawn in final version. | | **10** |
| Changes strokeStyle Colors – Changes strokeStyle to various colors. | | **10** |
| Changes fillStyle Colors – Changes fillStyle to various colors. | | **10** |
| Text Drawn - Text drawn in final version. | | **10** |
| Changes Text Font – Changes text font using web font(s). | | **10** |
| Centered Text – Text centered on canvas. | | **10** |
| **Errors Thrown** – Any errors thrown in the console. | | **-15% (this time)** |
| **Additional Penalties** – These are point deductions for poorly written code or improper code, not uploaded to Banjo or other issues. There are no set values for penalties. The more penalties you make the more points you will lose. | | |
| **TOTAL** | | **100%** |

**Before the due date, upload your files to Banjo, ZIP up your code and post the zip to the dropbox ALONG with a link to your page on Banjo in the submission comments.**