# SSUI Web: Assignment 2

Drawing

**Due: October 17th, 2016 @ 11:59pm**

## Overview, Learning Goals, and Grading

The goal of this project is to create a reusable drawing library for creating interactive content on the <canvas> tag. Some libraries already exist for drawing on an HTML canvas (e.g., Raphael, Processing, and fabric.js), but for this project you're going to write your own drawing library, and then use this library to create an interesting doodle of your own! Your library will support hierarchical drawing and perform transformation and clipping on a wide variety of visual elements. This assignment has aims to help you understand:

- How to support construction of an interactor hierarchy (**25 pts**)
    - Manage children and parents in the tree
    - Update the interface when a child is added, removed, or modified
- How to support simple layout abstractions and manipulations (**20 pts**)
    - Creating rows and columns
    - Drawing interactors in their proper locations when they are a child of layout
- How to correctly draw interactors (**25 pts**)
    - Without impinging on the rest of the interface
    - With proper translation and clipping to control frame buffer access
- Separation of concerns between application architecture output and the application developer by creating your own collage (**10 pts**)
- How to write clean usable code through inheritance and good style conventions (See: Google style guide**)** **(20 pts**)

## Files Provided

The starter code for assignment 2 can be found here. It contains the following files:

- doodle-library.js: Skeleton code for the JavaScript library you will be creating.
- utils.js: Utility functions that have been provided for you.
- primitive-test.html: HTML file that is provided for you to test your library.
- primitive-test.js: JavaScript file that is provided for you to test your library.
- container-test.html: HTML file that is provided for you to test your library.
- container-test.js: JavaScript file that is provided for you to test your library.
- primitives.png: Screenshot of test output for first part of tests (i.e. simple primitives)
- containers.png: Screenshot of text output for second part of tests (i.e. complex containers)
- cherry.jpg and kitty.jpg Pictures for the tests.

# Project Overview

This project has two pieces. The first pieces is to create a drawing library for <canvas> that provides a few basic functions to make drawing using canvas easier. The second pieces is to use your library to create a cool doodle (ideas at http://www.google.com/logos/).

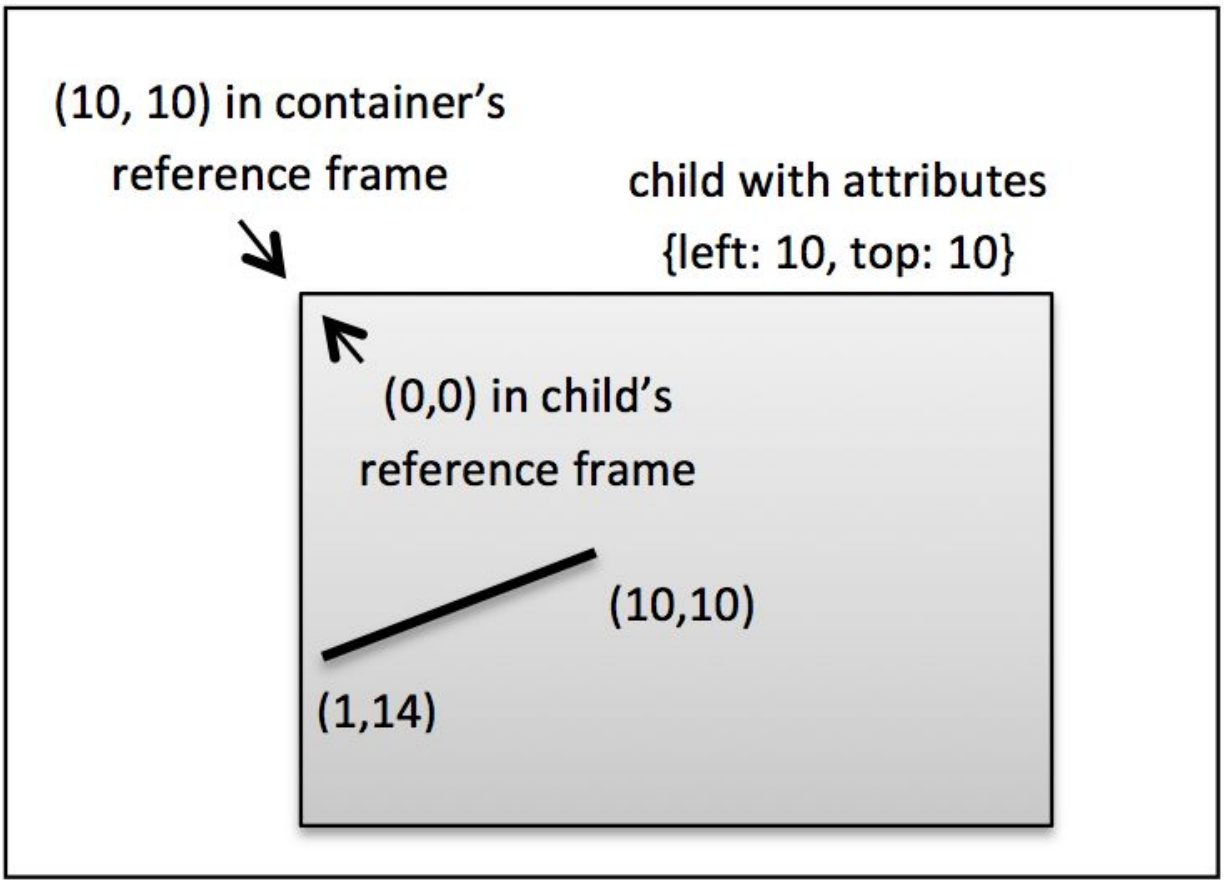
## Part I: Drawing Library

Your drawing library must implement the objects specified below. Your objects should have at least the methods and attributes specified below (though you may add more if you wish). The utils.js file provides a mechanism for executing inheritance, as well as other useful functions for setting default values. Inheritance has already been set up for you in the skeleton code, and you can use it for creating new objects for extra credit.

Your objects will need to follow a specific layout protocol. There is a root entity (a Doodle) which represents the <canvas> element that you're drawing on. Its purpose is to hold and draw all of the elements in your doodle. It cannot be rotated or translated.

Every drawable object in a doodle has left and top coordinates. Left and top specify the reference frame for the object; when the object draws itself, it treats its left and top as the origin, (0, 0). Therefore, x and y coordinates in the attributes for an object, such as a line or a path, are relative to the object's left and top.

For example, the figure below shows a container with a single child container. The child container has a left and top of (10, 10), so it is drawn at (10, 10) in the parent's coordinate system. The line inside the child container, however, treats the left, top point of the container as (0, 0) when drawing its endpoints.




When a container object (such as a Doodle or Container object) draws one of its child elements, it first translates and rotates the child element according to the child's left and top attributes. It then calls the child's draw function. The following are specifications for the objects you must implement. Stubs for these objects are provided in the starter code.

## Doodle

A Doodle is the root container for all drawable elements. It represents the canvas on which all other elements will be drawn. When drawing using this library, the first element created is an instance of this Doodle element. Additional elements are added as children to the root Doodle element.

*Constructor Parameter*

- `context` The drawing context for the object

Methods

- `draw` Draws its children. If a child is not visible, does not draw the child. Scales, translates, and rotates the child according to the child's scale, (left, top), and theta variables.

*Fields*

- `context` The drawing context for the object, generated by a canvas
- `children` An array of the top-level drawable elements in this doodle. Hint: The drawing context is stored in Doodle (the canvas object) and can be passed to methods of a child element to do drawing.

## Primitives

### Text

Inherits from: `Drawable`
A Text object draws text with the given attributes. Note: Text is always drawn up from the bottom of the object. This is because there is no simple way to measure the height of a text string using canvas, so we explicitly define it with height.

*Constructor Parameter*

- `attrs` An object containing values for each of the fields in the object. If the attrs parameter is not specified, or if one of the fields is not specified, use predefined defaults (defined for you).

*Methods*

- `draw` Draw the text using the values defined in attrs.
- `getWidth` Return the width of the text (use the MeasureText helper method provided)
- `getHeight` Return the height of the text (use the MeasureText helper method provided)

*Fields* (in addition to those specified by ancestors)

- `content` The text string to draw.
- `fill` The fill color of the text. Specify in the form of a CSS color.
- `font` The font family to use.
- `size` The size of the font in pt
- `bold` boolean value for if the font is bold or not

### DoodleImage

Inherits from: `Drawable`
A DoodleImage object draws an image. This may be a bit trickier than the other primitives to implement, because it has to load the image file before being drawn. You can get around this by making draw() wait until the image is loaded, or by making the whole Doodle wait to be drawn until all the images are loading. Your solution here might be a little hacky, that's ok.

*Constructor Parameter*

- `attrs` An object containing values for each of the fields in the object. If the attrs parameter is not specified, or if one of the fields is not specified, use predefined defaults (defined for you).

*Methods*

- `draw` Draw the image using the specified source, with the specified width and height.
- `getWidth` Return the width of the image
- `getHeight` Return the height of the image

*Fields* (in addition to those specified by ancestors)

- `width` The width of the image. Default is -1. If image width is -1, use the natural width of the image.
- `height` The height of the image. Default is -1. If image height is -1, use the natural height of the image.
- `src` The location of the image (specify a path).

### Line

Inherits from: `Primitive` (already implemented for you)
Draws a single line.

*Constructor Parameter*

- `attrs` An object containing values for each of the fields in the object. If the attrs parameter is not specified, or if one of the fields is not specified, use predefined defaults (defined for you).

*Methods*

- `draw` Draw the line.
- `getWidth` Return the width of the line's bounding box
- `getHeight` Return the height of the line's bounding box

*Fields* (in addition to those specified by ancestors)

- `startX` Starting x coordinate of line. Must be >=0.
- `startY` Starting y coordinate of line. Must be >=0.
- `endX` Ending x coordinate of line. Must be >=0.
- `endY` Ending y coordinate of line. Must be >=0.

### Rectangle

Inherits from: `Primitive` (already implemented for you)
Draws a rectangle (not filled in).

*Constructor Parameter*

- `attrs` An object containing values for each of the fields in the object. If the attrs parameter is not specified, or if one of the fields is not specified, use predefined defaults (defined for you).

*Methods* (in addition to those specified by ancestors)

- `draw` Draw the rectangle as specified by x, y, width, and height. (just the outline, not filled in.)
- `getWidth` Return the width of the rectangle
- `getHeight` Return the height of the rectangle

*Fields* (in addition to those specified by ancestors)

- `width` width of the rectangle
- `height` height of the rectangle

## Layout Container

Inherits from: `Drawable`

A container is a rectangular object that can have other drawable objects as children. When drawn, the container draws itself as well as all of its children. Specifically, it should do the following: (1) manage its geometry (size and position), (2) manage the hierarchy (storing and manipulating a list of children inside it, and remember its parent), (3) perform layout (determining the position and possibly size of the child objects within it), and (4) draw itself and its children. The container also clips all of its content to its bounds.

*Constructor Parameter*

- `attrs` An object containing values for each of the fields in the object. If the attrs parameter is not specified, or if one of the fields is not specified, use predefined defaults (defined for you).

*Methods*

- `draw` Draws itself and its children. If a child is not visible (visible property set to false), does not draw it. Scales, translates, and rotates children according to each child's scale, (left, top), and theta values.
- `layout` Performs layout on the children objects before it draws them.
- `getWidth` Return the width of the container
- `getHeight` Return the height of the container

*Fields* (in addition to those specified by ancestors):

- `width` The width of the container
- `height` The height of the container
- `borderWidth` How wide the container's border is (in pixels).
- `borderColor` Color of the border.
- `fill` Fill color of the container. Default value is "". If fill is not specified or "", do not fill in the container.
- `children` Drawable objects that are located within the container.

We will create 5 different specialized layout containers.

### Pile

Inherits from: `Container`
Places all of its children at its own top-left corner.
Has the same constructor parameters, methods, and fields as container.

### Row

Inherits from: `Container`
Places its children in a single horizontal row with the children vertically centered. If the children do not fit within the bounds of the row object they are clipped at the right edge.
Has the same constructor parameters, methods, and fields as container.

### Column

Inherits from: `Container`
Places its children in a single vertical column with the children horizontally centered. If the children do not fit within the bounds of the row object they are clipped at the bottom edge.
Has the same constructor parameters, methods, and fields as container.

### Circle

Inherits from: `Container`
Places its children so that their centers (not top-left corners) lie positioned at equal angles around a circular perimeter of a given size. So for example, if there were five child objects they would be placed every 360/5 = 72°. Or if there were four child objects, their centers would be positioned 90° apart, i.e., at the four compass points. Note that this layout ignores any overlap that might occur between children, simply calculating their locations and drawing them there in the order found in the child list. Has the same constructor parameters and methods as container.

*Fields* (in addition to those specified by ancestors):

- `layoutCenterX` The X center of the circle we want to layout
- `layoutCenterY` The Y center of the circle we want to layout
- `layoutRadius` The distance from the center of the circle that the middle of each child should be at

### OvalClip

Inherits from: `Container`
Applies additional clipping to all its children in the form of an oval which fits just inside its defined bounding box. Has the same constructor parameters, methods, and fields as container.

## Testing Part I

We have provided code for you to test out your drawing library. The test code has two different parts. The first part (test-primitives.html) makes a drawing using only the primitives in your library (i.e. no containers). You should probably start implementing only the primitives, and make sure test-primitives.html works before moving on to the next part. The next part (test-containers.html) tests whether your container code works. test-containers.html assumes that test-primitives.html works, so you should do this test second. We have also included images with the expected output. When your code is graded, we will check it against these tests and some additional tests not provided. Feel free to write tests of your own as well!

## Part II: Make a Doodle

Make a doodle similar to a Google doodle to show off your library. Your doodle should have some text in it; however it does not need to say Google. You can use your name as your doodle, for example. I will be grading your doodle based on how interesting it is.

## Show and Tell

We will be picking the nicest doodles and showing them to the class. They may also be put on a public website. Please let me know if you do not want your solution shown in the README file in your assignment.

## Turning Your Program In

The project is due **October 17th, 2016 @ 11:59pm**. You should turn in your assignment via Autolab as a zip archive containing the following files:

- doodle-library.js: Your completed doodle library
- doodle.html: HTML file of your doodle
- doodle.js: JavaScript file of your doodle
- primitive-test.html: Keep all of the test files
- primitive-test.js: Keep all of the test files
- container-test.html: Keep all of the test files
- container-test.js: Keep all of the test files
- screenshot.png: A screenshot of your doodle
- README.txt

Please write a README.txt file which mentions any online sources you used to help with the project, as well as any notes about your programs (i.e. if you couldn't get a particular part of the project to work).