

Lecture 3: JavaScript Graphics

Written by Ben Yan, from materials by Jerry Cain

CS106AX JSGraphics Library

This lecture's code—centered around visual and graphical programs—relies on a delightful, in-house library **JSGraphics.js**. This library can be summoned by adding a single line to our HTML files, i.e.

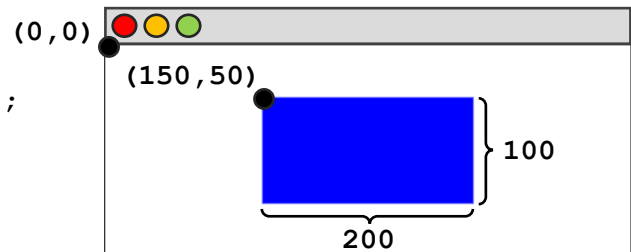
```
<html>
  <head>
    <script type='text/javascript' src='JSGraphics.js'></script>
  </head>
</html>
```

Blue Rectangle

The following program first defines a graphics window or canvas measuring **500** pixels in width and **200** pixels in height. We then create a rectangle of width **200** pixels and height **100** pixels, painted with a solid blue fill, and the rectangle's **top-left corner** placed at coordinate **(x, y) = (150, 50)**.

Note that unlike traditional math, **JavaScript defines the origin (0, 0) as the upper-left corner of the window**, and **values for the y coordinate increase as you go downward** 😊. Weird, but consistent!

```
function BlueRectangle() {
  let gw = GWindow(500, 200);
  let rect = GRect(150, 50, 200, 100);
  rect.setColor("Blue");
  rect.setFilled(true);
  gw.add(rect);
}
```



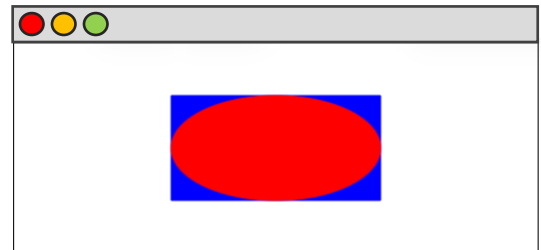
Note the eponymous methods used to create and add the **GWindow** and **GRect** objects:

- `let gw = GWindow(width, height)`
- `let rect = GRect(x, y, width, height)` or `GRect(width, height)`
- `gw.add(rect)` or `gw.add(rect, x, y)`, especially if we didn't specify coordinates above.

Rectangle and Oval

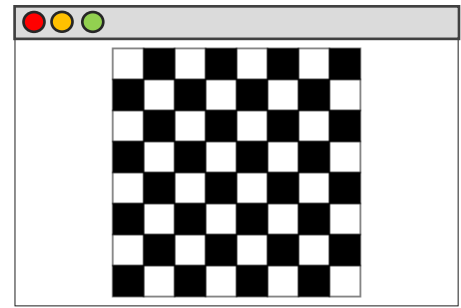
This program extends the one above, by adding a solid red, overlapping **GOval** object to the window, with identical dimensions and top-left coordinates to the rectangle. Note that newer objects can eclipse objects that were added earlier, a layering arrangement called a **stacking order**.

```
function GRectPlusGOval() {
  /* insert here the same code above! */
  let oval = GOval(150, 50, 200, 100);
  oval.setFilled(true);
  oval.setColor("Red");
  gw.add(oval);
} // syntax is similar, GOval(x, y, width, height) or GOval(width, height)
```



Checkerboard

A more visually elaborate and computationally interesting program creates **a grid of filled (black) and non-filled (white) squares to resemble a checkerboard**. The board is centered both horizontally and vertically on the graphics window, just like this.



Assume the following constants have been defined ☺.

```
const GWINDOW_WIDTH = 500;          /* Width of the graphics window */
const GWINDOW_HEIGHT = 300;        /* Height of the graphics window */
const N_COLUMNS = 8;               /* Number of columns */
const N_ROWS = N_COLUMNS;          /* Number of rows */
const SQUARE_SIZE = 35;           /* Size of a square in pixels */
```

First, we create a graphics window with the specified, not necessarily square dimensions.

```
let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
```

We compute the **top-left coordinate (x0, y0) of the grid**. The full width of the grid is **N_COLUMNS * SQUARE_SIZE**, which leaves a margin of **gw.getWidth() - N_COLUMNS * SQUARE_SIZE**, split equally on the left and right sides. The left margin gives **x0**, and we can calculate **y0** similarly.

```
let x0 = (gw.getWidth() - N_COLUMNS * SQUARE_SIZE) / 2;
let y0 = (gw.getHeight() - N_ROWS * SQUARE_SIZE) / 2;
```

Next, to generate the 2D grid of squares, we'll need to **iterate over each row and column in a doubly nested loop**. For ease, let's focus on a single square at a time—say the one at row **i** and column **j**, for $0 < i < N_ROWS$ and $0 < j < N_COLUMNS$ (note the 0-indexing). Its top-left coordinate is given by

```
let x = x0 + j * SQUARE_SIZE;
let y = y0 + i * SQUARE_SIZE;
```

Note these are offsets from the grid's top-left corner (**x0, y0**). Next, we can create the square **GRect**.

```
let sq = GRect(x, y, SQUARE_SIZE, SQUARE_SIZE);
```

Now, we need to decide if the square should be black (filled) or white (non-filled). **It turns out for all black squares, the row-column sum $i + j$ is even, and for white squares, $i + j$ is odd.** Why is this?

Well, **all squares in one rising diagonal have the same row-column sum, and colors alternate between diagonals**. All squares in the 0th diagonal (just the top-left square) have $i + j = 0$, all squares in the 1st diagonal have $i + j = 1$, all squares in the 2nd diagonal have $i + j = 2$, and so on. As such,

```
let filled = (i + j) % 2 !== 0; // true if black, false if white
sq.setFilled(filled);
gw.add(sq); // don't forget to add the object to the GWindow!
```

And, at long last, we have the main program in its full, checkered glory ☺.

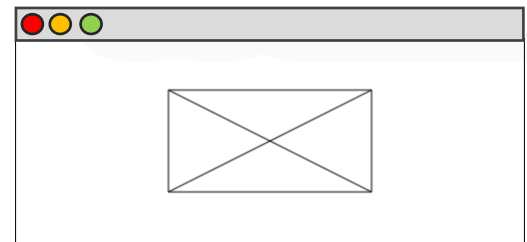
```
function Checkerboard() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let x0 = (gw.getWidth() - N_COLUMNS * SQUARE_SIZE) / 2;
  let y0 = (gw.getHeight() - N_ROWS * SQUARE_SIZE) / 2;

  for (let i = 0; i < N_ROWS; i++) {
    for (let j = 0; j < N_COLUMNS; j++) {
      let x = x0 + j * SQUARE_SIZE;
      let y = y0 + i * SQUARE_SIZE;
      let sq = GRect(x, y, SQUARE_SIZE, SQUARE_SIZE);
      let filled = (i + j) % 2 !== 0;
      sq.setFilled(filled);
      gw.add(sq);
    }
  }
}
```

Crossed Box

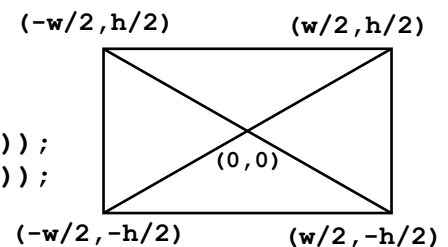
This final program draws an **envelope-looking box with a rectangle and its two diagonal lines**. Our constants are:

```
const GWINDOW_WIDTH = 500; /* in pixels */
const GWINDOW_HEIGHT = 200;
const BOX_WIDTH = 200;
const BOX_HEIGHT = 100;
```



This exercise debuts the **GCompound** class, which makes it possible to combine and layer several **GObjects** into one composite structure. It's akin to a **GWindow** in that you can add objects to it, and akin to a **GObject** in that you can add it to the window. It possesses its own internal coordinate system, and you add objects to the **GCompound** relative to its internal origin. As an example, the function below creates a **GCompound** as architected above, with the diagonals intersecting at the box's center / origin.

```
function createCrossedBox(width, height) {
  let box = GCompound();
  box.add(GRect(-width/2, -height/2, width, height));
  box.add(GLine(-width/2, -height/2, width/2, height/2));
  box.add(GLine(-width/2, height/2, width/2, -height/2));
  return box;
}
```



Note: This makes use of the **GLine(x0, y0, x1, y1)** method to draw a line between two points.

When we add the **GCompound** to the graphics window, we simply set the real coordinates of its local origin—and voilá, all the individual components will appear relative to that point! For example, the function below adds the crossed box so that its internal origin coincides with the center of the canvas.

```
function DrawCrossedBox() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let box = createCrossedBox(BOX_WIDTH, BOX_HEIGHT);
  gw.add(box, gw.getWidth() / 2, gw.getHeight() / 2);
}
```