



**CODE
PROJECT**
For those who code

articles

Q&A

forums

lounge

Search for articles, questions, tips



Build a Prototype Web-Based Diagramming App with SVG and Javascript



Marc Clifton, 2 Apr 2018



5.00 (1 vote)

Rate this:



Learning how to programmatically manipulate SVG in Javascript



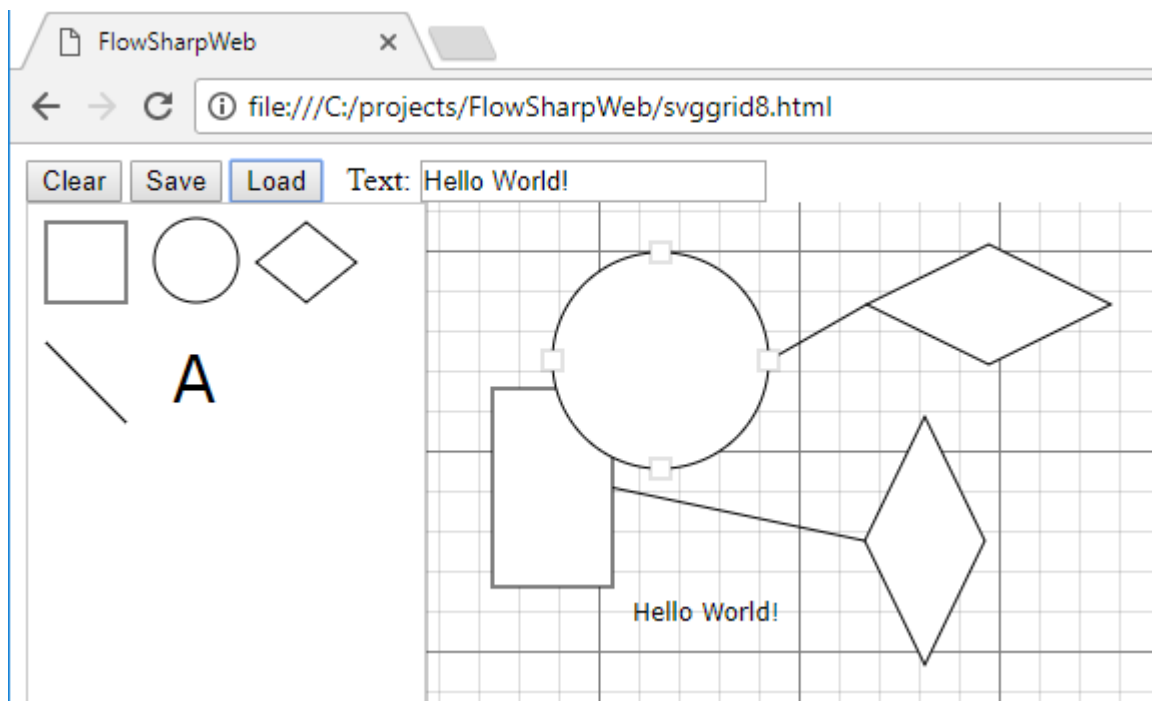
Download FlowSharpWeb.zip - 35.1 KB

Download and unzip the file then open the file "FlowSharpWeb.html" to launch the app in the default browser.

Or...

[jsfiddle of this code](#)

[On GitHub](#)



Contents

- Introduction
 - No Third Party Libraries
 - Describing Why and How
 - draw.io
- Prototype Build
- Creating a Scrollable Grid
 - Simulating a Virtual Surface
 - Scrolling the Grid - Mouse Events
 - Wiring up the Mouse Events
 - Best Practice
 - The Event Handlers
 - Handling Moving the Mouse off the Grid
- Resizing the Grid - Our First Dynamic SVG
- Adding Some Static Shapes
- Moving Shapes Around
 - The Mouse Controller
 - The Shape Object Model
 - The SvgObject Class
 - The SvgElement Class
 - The Circle Class
 - The Surface Class
 - Wrapping Up Moving Shapes
- A Toolbox and Dynamic Shape Creation
 - Supporting Classes
 - Initialization
 - The Toolbox Shapes
 - The SvgToolboxElement Class
 - The ToolboxController Class
 - Constructor
 - Toolbox Controller onMouseDown
 - Determining a Click Event
 - Toolbox Controller onMouseUp
 - Toolbox Controller onMouseMove
 - Mouse Controller onMouseUp
- Saving and Restoring the Diagram Locally
 - Saving the Diagram
 - Loading the Diagram
- Lines and Anchor Points
 - The Complexities of Selecting A Line
 - Point Class and Shape Rectangle
- Anchors
 - Line Length and Orientation
- Anchor Drag Operations for Other Shapes
 - Circles
 - Diamonds

- Rectangles
- Text
 - Changing Text
- Refactoring the Prototype to use an MVC Pattern
 - Models, Views, and Controllers
 - The Text Model
 - The Text Controller
 - The Text View
 - The Base Model
 - The Base View
 - Creating a Shape Programmatically
 - Serialization
 - Deserializing
- Connecting Lines
 - Connection Points
 - Detecting Shapes That We're Near
 - Showing Connection Points
 - Removing Connection Points
 - Connecting to a Shape
 - Updating Connections when the Shape is Moved
 - Updating Connections when the Shape is Resized
 - Disconnecting Connections
- Removing A Shape
- Conclusion

Introduction

I've been wanting to learn about SVG for a while now, and there are certainly any number of helpful websites on creating SVG drawings and animations. But I didn't want to learn how to create static (or even animated) SVG drawings, I wanted to learn how to use SVG dynamically:

- Create, modify, and remove SVG elements dynamically.
- Hook events for moving elements around, changing their attributes, etc.
- Save and restore a drawing.
- Discover quirks and how to work around them.

That's what this article is about -- it will only teach you SVG and Javascript in so far as to achieve the goals outlined above. However, what it will teach you is how to create dynamic SVG drawings, and what better way to do this than to actually create a simple drawing program. Then again, I learned a lot about both SVG and modern Javascript writing this article.

No Third Party Libraries

No third party libraries are used in this code. There were a few useful Javascript functions (in particular, the FileSaver) that I included in the source here that came from elsewhere (see the article for where), but there are no dependencies on any SVG manipulation frameworks. This code doesn't even use jQuery. In my opinion, this makes it a lot easier to learn SVG from the ground up -- you're not dealing with SVG + YAF (Yet Another Framework.)

Describing Why and How

The point of this code and the code comments is to describe why I'm doing things in a certain way, and how I'm doing them. As this was a learning experience for myself, any time I had to reach out to the web to figure something out, I reference the source of the information -- this turns out to mostly be StackOverflow references!

draw.io

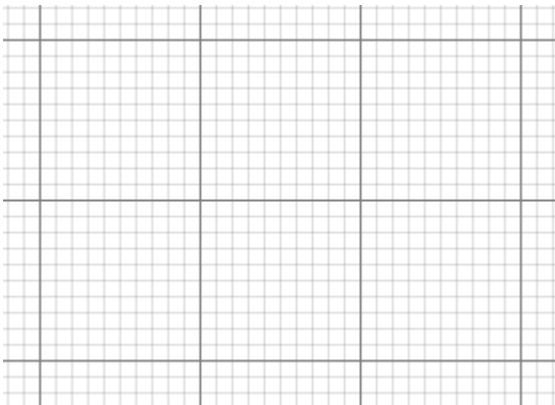
One of the best online SVG drawing programs is draw.io and I'm not going to attempt to recreate it. However, like many things, it is often useful to "roll your own" to understand how the technologies is used. The online program draw.io is a nice front end for [mxgraph](https://jgraph.github.io/mxgraph/docs/manual.html), which has excellent documentation at <https://jgraph.github.io/mxgraph/docs/manual.html>. Also see their API specifications which supports PHP, .NET, Java, and Javascript. If you're looking for a polished drawing program, similar to Visio, look at draw.io. If you want to *learn* about how this stuff is done, that's what this article is for.

That said, let's begin!

Prototype Build

The first two-thirds of this article is a prototype build. It vets the basic functionality and manipulation of the SVG DOM with UI interaction and diagram persistence. In particular, I implemented a very shallow view-controller architecture which gets completely replaced later on in the article. In the section that begins "Refactoring the Prototype" I move to a full model-view-controller, which cleans up a lot of the kludgy workarounds that you'll see here. Making the transition wasn't that painful -- 90% of the code was re-used in a proper MVC model, with the most significant changes occurring in the mouse controller and toolbox controller. And all cases of **instanceof** have been removed, something I considered quite a kludge in itself.

Creating a Scrollable Grid



The first thing I wanted to learn how to do was create a grid that scrolls. It was easy enough to find an example on the Internet which I used as a starting point:

[Hide](#) [Copy Code](#)

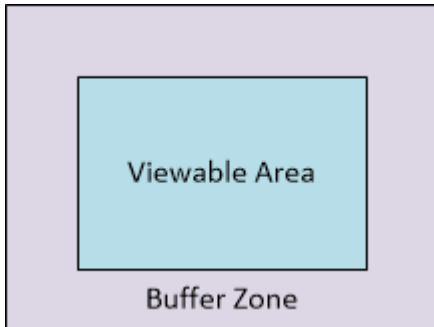
```
<svg id="svg" width="801" height="481" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <pattern id="smallGrid" width="8" height="8" patternUnits="userSpaceOnUse">
      <path d="M 8 0 H 0 V 8" fill="none" stroke="gray" stroke-width="0.5" />
    </pattern>
    <pattern id="grid" width="80" height="80" patternUnits="userSpaceOnUse">
      <rect width="80" height="80" fill="url(#smallGrid)" />
      <!-- draw from upper right to upper left, then down to lower left -->
      <!-- This creates the appearance of an 80x80 grid when stacked -->
      <path d="M 80 0 H 0 V 80" fill="none" stroke="gray" stroke-width="2" />
    </pattern>
  </defs>

  <!-- a trick from my old Commodore 64 days is to extend the scrolling region beyond the viewport
  and use mod 80 to reset the position to simulate a virtual space. -->
  <rect transform="translate(0, 0)" id="surface" x="-80" y="-80" width="961" height="641" fill="url(#grid)"
/>
</svg>
```

As I mentioned, I'm not going to go into the details of SVG but I will point out the core features:

- There are two grids -- an outer grid every 80 pixels and an inner grid every 8 pixels.
- The "grid" is actually created by drawing only two lines: the top line (from right to left) and the left edge, from upper left to bottom left. That's what the "M 80 0 H 0 V 80" does -- it creates a path starting at (80, 0), draws a horizontal line to (0, 0) and then a vertical line to (0, 80).
- The initial transform is a placeholder -- "translate(0, 0)" doesn't actually do anything.

Simulating a Virtual Surface



Notice that the rectangle is drawn with an off-screen buffer zone of (-80, -80) and (width + 80*2, height + 80*2). This is an old trick from which I used to program scrolling games on the Commodore 64 -- you would render the viewing area to include an off-screen buffer zone so that scrolling could be done simply by performing a translate (or on the C64, changing the screen memory pointer.) When scrolling a repeating pattern, one "translates" the viewable area +/- 80 mod 80 (the width and height of the grid) and it appears to the user as if there is an infinite virtual surface.

Scrolling the Grid - Mouse Events

The user scrolls the grid with a "drag" operation:

- Mouse down to start.
- Move mouse, which scrolls the grid.
- Mouse up when done.

We'll keep track of the following variables:

[Hide](#) [Copy Code](#)

```
var mouseDown = false;
var mouseDownX = 0;
var mouseDownY = 0;
var gridX = 0;
var gridY = 0;
```

Wiring up the Mouse Events

This is very simple (but we'll see later that it gets more complicated because for actual shapes that may be removed from the drawing, we will want to unhook the event handlers):

[Hide](#) [Copy Code](#)

```
function initializeSurface() {
  var svg = document.getElementById("svg");
  var surface = svg.getElementById("surface");
  surface.addEventListener("mousedown", onMouseDown, false);
  surface.addEventListener("mouseup", onMouseUp, false);
  surface.addEventListener("mousemove", onMouseMove, false);
  surface.addEventListener("mouseleave", onMouseLeave, false);
}

initializeSurface();
```

Best Practice

Technically, we could just get the surface element directly from the document:

Hide Copy Code

```
var svgSurface = document.getElementById("surface");
```

but I suppose using the **svg** element helps to prevent the possibility that the HTML has an element of the same name, particularly since we don't know how the programmer might create additional HTML.

The Event Handlers

Here we handle the **mousedown**, **mouseup**, and **mousemove** events:

Hide Shrink ▲ Copy Code

```
const LEFT_MOUSE_BUTTON = 0;

function onMouseDown(evt) {
  if (evt.button == LEFT_MOUSE_BUTTON) {
    evt.preventDefault();
    mouseDown = true;
    mouseDownX = evt.clientX;
    mouseDownY = evt.clientY;
  }
}

function onMouseUp(evt) {
  if (evt.button == LEFT_MOUSE_BUTTON) {
    evt.preventDefault();
    mouseDown = false;
  }
}

function onMouseMove(evt) {
  if (mouseDown) {
    evt.preventDefault();
    var mouseX = evt.clientX;
    var mouseY = evt.clientY;
    var mouseDX = mouseX - mouseDownX;
    var mouseDY = mouseY - mouseDownY;
    gridX += mouseDX;
    gridY += mouseDY;
    mouseDownX = mouseX;
    mouseDownY = mouseY;
    var svg = document.getElementById("svg");
    var surface = svg.getElementById("surface");
    var dx = gridX % 80;
    var dy = gridY % 80;
    surface.setAttribute("transform", "translate(" + dx + "," + dy + ")");
  }
}
```

A few things to note:

- Apparently, some browsers such as Firefox has default drag/drop handling so we call **evt.preventDefault()** to, well, prevent the default handling of the event.
- The variables **gridX** and **gridY** track the absolute offset grid.
- The grid is translated by this absolute offset modulus 80 so we don't exceed the boundaries of the buffer zone.
- Apparently there used to be some confusing between the W3C standard (left button == 0) and Microsoft's concept, (left button == 1) but that seems to be very old information -- testing on Chrome and Edge, the left button value (as well as right and middle values) are consistent across these two browsers.

Handling Moving the Mouse off the Grid

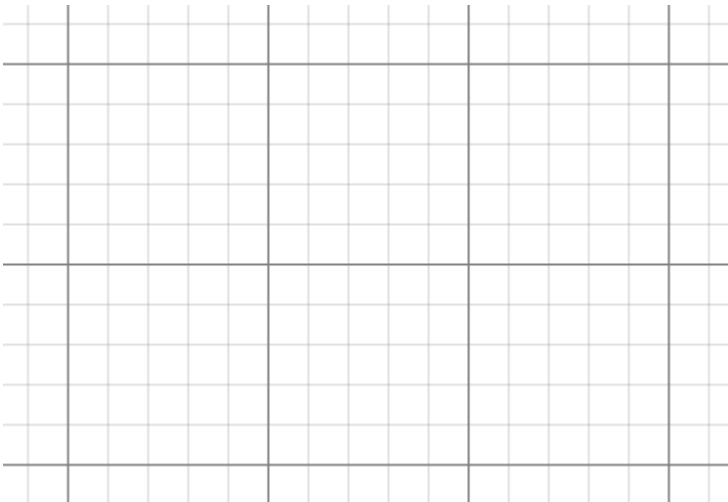
There is no concept of "mouse capture," so when the user drags the surface and the mouse moves outside of the SVG element, events, such as **mouseup**, are no longer received. If the user stops dragging *outside of the SVG element* by releasing the mouse button, the code is still in the dragging state because the **mouseup** event didn't fire. So instead, when the mouse cursor leaves the element, we simulate a **mouseup** event by handling the **mouseleave** event.

[Hide](#) [Copy Code](#)

```
surface.addEventListener("mouseleave", onMouseLeave, false);

// If the mouse moves out of the surface area, the mouse up event will not trigger,
// so we clear the mouseDown flag so that scrolling does not resume "by itself"
// when the user moves the mouse back onto the surface, which would otherwise
// require the user to click to clear the mouseDown flag.
function onMouseLeave(evt) {
  evt.preventDefault();
  mouseDown = false;
}
```

Resizing the Grid - Our First Dynamic SVG



Of course, all the code above is hard-coded for a grid of dimensions 80x80 with inner grid spacing of 8x8. We would like this to actually be user configurable. To do this, it's useful to rename some ID's and add additional ID's to the pattern definitions:

[Hide](#) [Copy Code](#)

```
<defs>
  <pattern id="smallGrid" width="8" height="8" patternUnits="userSpaceOnUse">
    <path id="smallGridPath" d="M 8 0 H 0 V 8" fill="none" stroke="gray" stroke-width="0.5" />
  </pattern>
  <pattern id="largeGrid" width="80" height="80" patternUnits="userSpaceOnUse">
    <rect id="largeGridRect" width="80" height="80" fill="url(#smallGrid)" />
    <!-- draw from upper right to upper left, then down to lower left -->
    <!-- This creates the appearance of an 80x80 grid when stacked -->
    <path id="largeGridPath" d="M 80 0 H 0 V 80" fill="none" stroke="gray" stroke-width="2" />
  </pattern>
</defs>
```

For reasons that will become clear in the next section, I've also added a group around the rectangle that represents the grid:

[Hide](#) [Copy Code](#)

```
<g id="surface" transform="translate(0, 0)" x="-80" y="-80" width="961" height="641" >
  <rect id="grid" x="-80" y="-80" width="961" height="641" fill="url(#largeGrid)" />
</g>
```

We need to track the width and height setting of the larger rectangles for modulus operator:

[Hide](#) [Copy Code](#)

```
// The default:
var gridCellW = 80;
var gridCellH = 80;
```

and is used in the mousemove handler:

[Hide](#) [Copy Code](#)

```
var dx = gridX % gridCellW;
var dy = gridY % gridCellH;
```

Given this function which changes the grid spacing to the screenshot at the start of this section, large grid is 100x100, small grid is 20x20:

[Hide](#) [Copy Code](#)

```
resizeGrid(100, 100, 20, 20);
```

Here's the implementation:

[Hide](#) [Shrink](#) ▲ [Copy Code](#)

```
// Programmatically change the grid spacing for the larger grid cells and smaller grid cells.
function resizeGrid(lw, lh, sw, sh) {
  gridCellW = lw;
  gridCellH = lh;
  var elLargeGridRect = document.getElementById("largeGridRect");
  var elLargeGridPath = document.getElementById("largeGridPath");
  var elLargeGrid = document.getElementById("largeGrid");

  var elSmallGridPath = document.getElementById("smallGridPath");
  var elSmallGrid = document.getElementById("smallGrid");

  var elSvg = document.getElementById("svg");
  var elSurface = document.getElementById("surface");
  var elGrid = document.getElementById("grid");

  elLargeGridRect.setAttribute("width", lw);
  elLargeGridRect.setAttribute("height", lh);

  elLargeGridPath.setAttribute("d", "M " + lw + " 0 H 0 V " + lh);
  elLargeGrid.setAttribute("width", lw);
  elLargeGrid.setAttribute("height", lh);

  elSmallGridPath.setAttribute("d", "M " + sw + " 0 H 0 V " + sh);
  elSmallGrid.setAttribute("width", sw);
  elSmallGrid.setAttribute("height", sh);

  elGrid.setAttribute("x", -lw);
  elGrid.setAttribute("y", -lh);

  var svgW = +elSvg.getAttribute("width");
  var svgH = +elSvg.getAttribute("height");

  elSurface.setAttribute("width", svgW + lw * 2);
  elSurface.setAttribute("height", svgH + lh * 2);

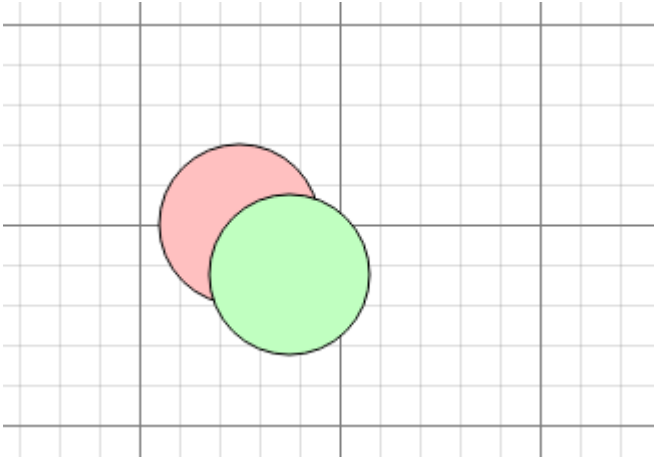
  elSurface.setAttribute("x", -lw);
  elSurface.setAttribute("y", -lh);

  elSurface.setAttribute("width", svgW + lw * 2);
  elSurface.setAttribute("height", svgH + lh * 2);
}
```

That's a lot of manipulation of the DOM elements. What we're doing is:

- Resetting the width and height of the outer grid rectangle and the pattern dimensions.
- Resetting the width and height of the inner grid pattern dimensions.
- Changing the path of the out and inner grids to reflect the new dimensions.
- Resizing the buffer zone and surface dimensions.

Adding Some Static Shapes



Remember the group that I added around the grid rectangle? We'll now add another group that is used for shapes, and we'll put a couple static shapes in that group:

[Hide](#) [Copy Code](#)

```
<g id="objects" transform="translate(0, 0)">
  <circle cx="150" cy="100" r="40" stroke="black" stroke-width="1" fill="#FFC0C0" />
  <circle cx="175" cy="125" r="40" stroke="black" stroke-width="1" fill="#C0FFC0" />
</g>
```

Now, with a simple addition to the **mousemove** event, we can translate all the elements in the "objects" group as well so that they move as the surface is scrolled:

[Hide](#) [Copy Code](#)

```
function onMouseMove(evt) {
  if (mousedown) {
    evt.preventDefault();
    var mouseX = evt.clientX;
    var mouseY = evt.clientY;
    var mouseDX = mouseX - mouseDownX;
    var mouseDY = mouseY - mouseDownY;
    gridX += mouseDX;
    gridY += mouseDY;
    mouseDownX = mouseX;
    mouseDownY = mouseY;
    var surface = document.getElementById("surface");

    var dx = gridX % gridCellW;
    var dy = gridY % gridCellH;
    surface.setAttribute("transform", "translate(" + dx + "," + dy + ")");

    var objects = document.getElementById("objects");
    objects.setAttribute("transform", "translate(" + gridX + "," + gridY + ")");
  }
}
```

The reason we use two separate groups is:

- The surface is always translated modulus the large grid size
- The objects *on the surface* must be translated by the absolute scroll offset.

If we don't keep the two regions separate, we get the strange effect that the shapes return to their original positions as a result of the modulus operation. Obviously we don't want that.

Moving Shapes Around

At this point, we have to start getting more sophisticated about how mouse events are captured -- each shape (including the surface) must handle its own mouse events. However, what the event does is not always the same -- for example, scrolling the surface grid is different than moving a shape in the "objects" group. Later on, even more complicated mouse move activities will require tracking the state of the operation -- are we moving the shape, resizing it, rotating it, etc.?

It's a big leap, but it really is of benefit to create an actual **MouseController** class and to create a shape controller class for the specialized behaviors of the different shapes. If we do this now, it becomes a lot easier to continue expanding the capabilities of what so far has just been a playground to test things out.

The Mouse Controller

The **MouseController** class does a few things for us:

- It tracks the shape being dragged. This is important because the user can move the mouse in a larger increment than the size of the shape. When this happens, the mouse "escapes" the shape and it no longer receives **mousemove** events. So once a shape (including the surface grid) is "captured" by **mousedown** event, the **mousemove** events are passed along to the controller responsible for that shape.
- It maps shape ID's to shape controllers. This allows the mouse controller to route mouse events to the controller associated with the shape.
- It implements some basic behavioral features such as where the user clicked and the basic logic of mouse down -> drag -> mouse up operations. Later on additional states can be added besides dragging -- states such as resizing.

The implementation is rather basic right now, building on what we did before:

Hide Shrink ▲ Copy Code

```
const LEFT_MOUSE_BUTTON = 0;

class MouseController {
  constructor() {
    this.mouseDown = false;
    this.controllers = {};
    this.activeController = null;
  }

  // Create a map between then SVG element (by it's ID, so ID's must be unique) and its controller.
  attach(svgElement, controller) {
    var id = svgElement.getAttribute("id");
    this.controllers[id] = controller;
  }

  detach(svgElement) {
    var id = svgElement.getAttribute("id");
    delete this.controllers[id];
  }

  // Get the controller associated with the event and remember where the user clicked.
  onMouseDown(evt) {
    if (evt.button == LEFT_MOUSE_BUTTON) {
      evt.preventDefault();
      var id = evt.currentTarget.getAttribute("id");
      this.activeController = this.controllers[id];
      this.mouseDown = true;
      this.mouseDownX = evt.clientX;
      this.mouseDownY = evt.clientY;
    }
  }
}
```

```

// If the user is dragging, call the controller's onDrag function.
onMouseMove(evt) {
    evt.preventDefault();

    if (this.mouseDown && this.activeController != null) {
        this.activeController.onDrag(evt);
    }
}

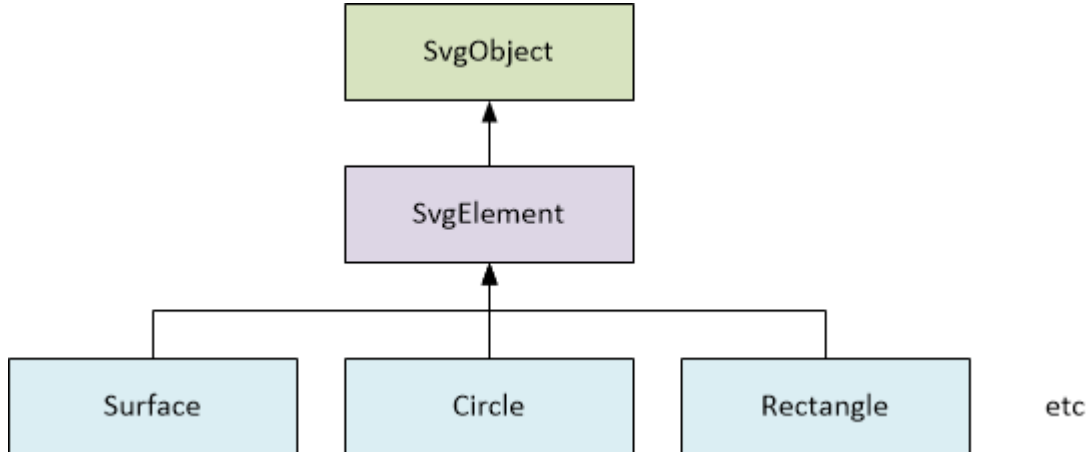
// Any dragging is now done.
onMouseUp(evt) {
    if (evt.button == LEFT_MOUSE_BUTTON) {
        evt.preventDefault();
        this.clearSelectedObject();
    }
}

// Any dragging is now done.
onMouseLeave(evt) {
    evt.preventDefault();
    if (this.mouseDown && this.activeController != null) {
        this.activeController.onMouseLeave();
    }
}

clearSelectedObject() {
    this.mouseDown = false;
    this.activeController = null;
}
}

```

The Shape Object Model



The diagram above illustrates the shape object model I've put together.

The SvgObject Class

This is the root class which keeps track of:

- The mouse controller (a shared object between all shapes).
- The shape's translation (it's offset from origin). I've seen various techniques for this by using attributes directly in the elements tag rather than parsing the `transform="translate(x, y)"` string in order to update the translation, but I'd rather keep this as variables in the shape's class instance.
- An event registration method so that when the shape is removed, all its associated event handlers can be unhooked.
- Default implementations for the basic drag operation math and other events.
- Binding the event handler to "this" as the default class instance or to a specified class instance (usually the mouse controller.)

```

class SvgObject {
  constructor(mouseController, svgElement) {
    this.mouseController = mouseController;
    this.events = [];

    // These two parameters are actually the shape TRANSLATION, not the absolute coordinates!!!
    this.X = 0;
    this.Y = 0;

    // These two parameters are the relative change during the CURRENT translation.
    // These is reset to 0 at the beginning of each move.
    // We use these numbers for translating the anchors because anchors are always
    // placed with an initial translation of (0, 0)
    this.dragX = 0;
    this.dragY = 0;

    this.mouseController.attach(svgElement, this);
  }

  // Register the event so that when we destroy the object, we can unwire the event listeners.
  registerEvent(element, eventName, callbackRef) {
    this.events.push({ element: element, eventName: eventName, callbackRef: callbackRef });
  }

  destroy() {
    this.unhookEvents();
  }

  registerEventListener(element, eventName, callback, self) {
    var ref;

    if (self == null) {
      self = this;
    }

    element.addEventListener(eventName, ref = callback.bind(self));
    this.registerEvent(element, eventName, ref);
  }

  unhookEvents() {
    for (var i = 0; i < this.events.length; i++) {
      var event = this.events[i];
      event.element.removeEventListener(event.eventName, event.callbackRef);
    }

    this.events = [];
  }

  startMove() {
    this.dragX = 0;
    this.dragY = 0;
  }

  updatePosition(evt) {
    var mouseX = evt.clientX;
    var mouseY = evt.clientY;
    var mouseDX = mouseX - this.mouseController.mouseDownX;
    var mouseDY = mouseY - this.mouseController.mouseDownY;
    this.X += mouseDX;
    this.Y += mouseDY;
    this.mouseController.mouseDownX = mouseX;
    this.mouseController.mouseDownY = mouseY;
  }

  onMouseLeave(evt) { }
}

```

The SvgElement Class

This class extends the `SvgObject` class, providing default mouse event registration and shape drag implementation:

[Hide](#) [Copy Code](#)

```
class SvgElement extends SvgObject {
  constructor(mouseController, svgElement) {
    super(mouseController, svgElement);
    this.element = svgElement;
    this.registerEventListener(this.element, "mousedown", mouseController.onMouseDown, mouseController);
    this.registerEventListener(this.element, "mouseup", mouseController.onMouseUp, mouseController);
    this.registerEventListener(this.element, "mousemove", mouseController.onMouseMove, mouseController);
  }

  onDrag(evt) {
    this.updatePosition(evt);
    this.element.setAttribute("transform", "translate(" + this.X + "," + this.Y + ")");
  }
}
```

Most of the time the "this" that is used to bind the event callback to the handling class instance will be the mouse controller, but the functionality has been provided to use the class instance registering the event (this is the default behavior) or some other class instance to which we want to bind the handler.

The Circle Class

The `Circle` class demonstrates the most basic of elements in which all the default behaviors can be utilized. It merely extends the `SvgElement` class.

[Hide](#) [Copy Code](#)

```
class Circle extends SvgElement {
  constructor(mouseController, svgElement) {
    super(mouseController, svgElement);
  }
}
```

The Surface Class

This class is much more complicated as it has to handle all the things we talked about before regarding scrolling the grid and objects on the grid. Note how it extends the `mouseleave` event. We want this to pass through the mouse controller's test to ensure that a drag operation is occurring when the mouse "leaves" the shape. Depending on the selected shape (the active controller) the behavior is different:

- in case of leaving the surface, the surface class is implemented such that the drag operation is cleared.
- In the case of leaving a shape, nothing happens as we want the shape to catch up to mouse position.

[Hide](#) [Shrink](#) ▲ [Copy Code](#)

```
class Surface extends SvgElement {
  constructor(mouseController, svgSurface, svgObjects) {
    super(mouseController, svgSurface);
    this.svgObjects = svgObjects;
    this.gridCellW = 80;
    this.gridCellH = 80;

    this.registerEventListener(this.svgSurface, "mouseleave", mouseController.onMouseLeave,
mouseController);
  }

  onDrag(evt) {
    this.updatePosition();
    var dx = this.X % this.gridCellW;
```

```

    var dy = this.Y % this.gridCellH;
    this.scrollSurface(dx, dy, this.X, this.Y);
}

onMouseLeave() {
    this.mouseController.clearSelectedObject();
}

scrollSurface(dx, dy, x, y) {
    // svgElement is the surface.
    this.svgElement.setAttribute("transform", "translate(" + dx + "," + dy + ")");
    this.svgObjects.setAttribute("transform", "translate(" + x + "," + y + ")");
}

function resizeGrid(lw, lh, sw, sh) {
    this.gridCellW = lw;
    this.gridCellH = lh;
    var elLargeGridRect = document.getElementById("largeGridRect");
    var elLargeGridPath = document.getElementById("largeGridPath");
    var elLargeGrid = document.getElementById("largeGrid");

    var elSmallGridPath = document.getElementById("smallGridPath");
    var elSmallGrid = document.getElementById("smallGrid");

    var elSvg = document.getElementById("svg");
    var elSurface = document.getElementById("surface");
    var elGrid = document.getElementById("grid");

    elLargeGridRect.setAttribute("width", lw);
    elLargeGridRect.setAttribute("height", lh);

    elLargeGridPath.setAttribute("d", "M " + lw + " 0 H 0 V " + lh);
    elLargeGrid.setAttribute("width", lw);
    elLargeGrid.setAttribute("height", lh);

    elSmallGridPath.setAttribute("d", "M " + sw + " 0 H 0 V " + sh);
    elSmallGrid.setAttribute("width", sw);
    elSmallGrid.setAttribute("height", sh);

    elGrid.setAttribute("x", -lw);
    elGrid.setAttribute("y", -lh);

    var svgW = elSvg.getAttribute("width");
    var svgH = elSvg.getAttribute("height");

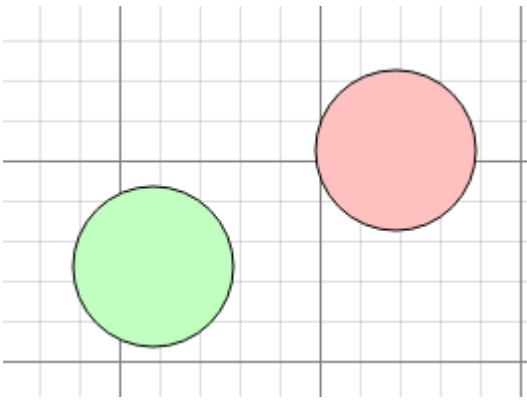
    elSurface.setAttribute("width", svgW + lw * 2);
    elSurface.setAttribute("height", svgH + lh * 2);

    elSurface.setAttribute("x", -lw);
    elSurface.setAttribute("y", -lh);

    elSurface.setAttribute("width", svgW + lw * 2);
    elSurface.setAttribute("height", svgH + lh * 2);
}
}

```

Wrapping Up Moving Shapes



To get this all to work, we need to add ID's to the two static circles in the "objects" group:

[Hide](#) [Copy Code](#)

```
<g id="objects" transform="translate(0, 0)">
  <circle id="circle1" cx="150" cy="100" r="40" stroke="black" stroke-width="1" fill="#FFC0C0" />
  <circle id="circle2" cx="175" cy="125" r="40" stroke="black" stroke-width="1" fill="#C0FFC0" />
</g>
```

We then create the class instances and in the constructor pass in the mouse controller instance and shape element:

[Hide](#) [Copy Code](#)

```
(function initialize() {
  var mouseController = new MouseController();
  var svgSurface = document.getElementById("surface");
  var svgObjects = document.getElementById("objects");
  var svgCircle1 = document.getElementById("circle1");
  var svgCircle2 = document.getElementById("circle2");
  var surface = new Surface(mouseController, svgSurface, svgObjects);
  surface.resizeGrid(100, 100, 20, 20);
  new Circle(mouseController, svgCircle1);
  new Circle(mouseController, svgCircle2);
})();
```

That's it! But where are we actually dragging the shapes? This may have escaped the casual reader--it is happening in the SvgElement class!

[Hide](#) [Copy Code](#)

```
onDrag(evt) {
  this.updatePosition(evt);
  this.element.setAttribute("transform", "translate(" + this.X + "," + this.Y + ")");
}
```

Any shape that derives from **SvgElement** inherits the ability to be dragged around the surface. For example, we'll add a rectangle:

[Hide](#) [Copy Code](#)

```
<rect id="nose" x="200" y="150" width="40" height="60" stroke="black" stroke-width="1" fill="#C0C0FF" />
```

Define the class **Rectangle**, which doesn't override anything yet, just like **Circle**:

[Hide](#) [Copy Code](#)

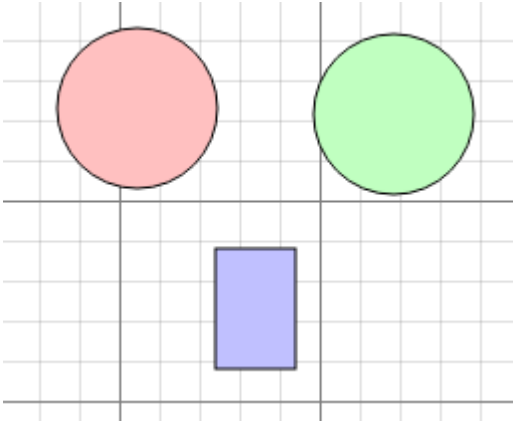
```
class Rectangle extends SvgElement {
  constructor(mouseController, svgElement) {
    super(mouseController, svgElement);
  }
}
```

and instantiate the shape with the associated SVG element:

[Hide](#) [Copy Code](#)

```
new Rectangle(mouseController, document.getElementById("nose"));
```

and we get (after moving the shapes around):

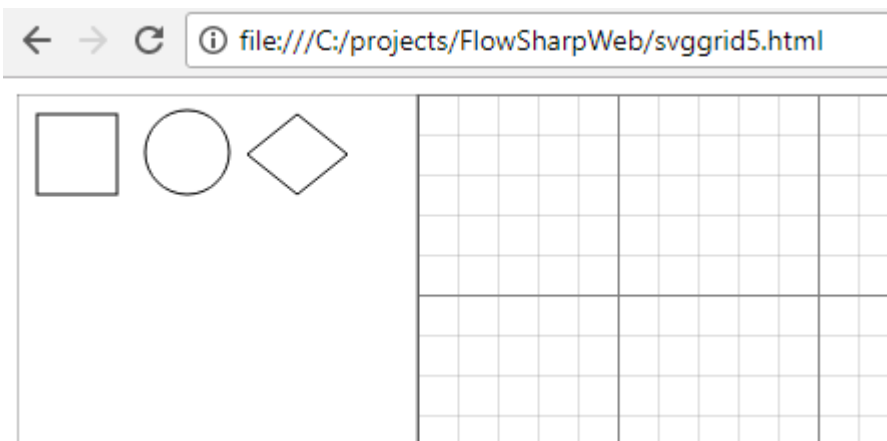


A Toolbox and Dynamic Shape Creation

Let's make what we're doing more useful by adding a toolbox so we can drag and drop new shapes on the surface. The toolbox will be the third group, making it the topmost group so that everything else (grid and objects) are always rendered *behind* the toolbox:

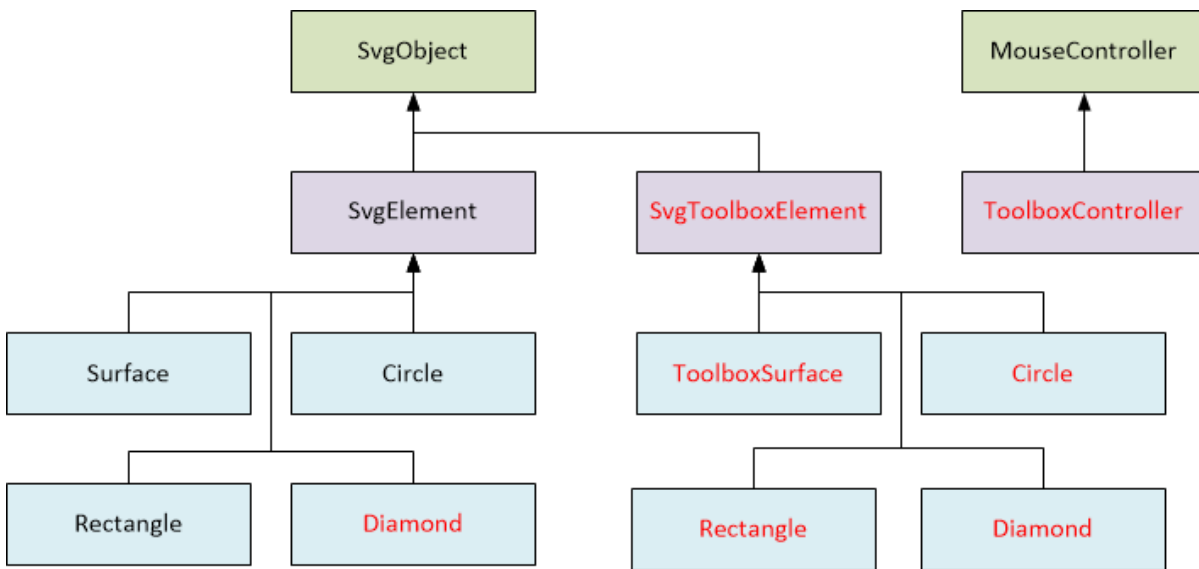
[Hide](#) [Copy Code](#)

```
<g id="toolboxGroup" x="0" y="0" width="200" height="480">
  <rect id="toolbox" x="0" y="0" width="200" height="480" fill="#FFFFFF" stroke="black" stroke-width="0.5"
/>
  <rect id="toolboxRectangle" x="10" y="10" width="40" height="40" stroke="black" stroke-width="1"
fill="#FFFFFF" />
  <circle id="toolboxCircle" cx="85" cy="29" r="21" stroke="black" stroke-width="1" fill="#FFFFFF" />
  <path id="toolboxDiamond" d="M 140 10 L 115 30 L 140 50 L 165 30 Z" stroke="black" stroke-width="1"
fill="#FFFFFF" />
</g>
```



Supporting Classes

We're going to need a some additional classes (indicated with the red text):



Initialization

Here's the entire initialization code (I've removed the static shapes we had previously):

Hide Shrink ▲ Copy Code

```

const SVG_ELEMENT_ID = "svg";
const SVG_SURFACE_ID = "surface";
const SVG_TOOLBOX_SURFACE_ID = "toolboxSurface";
const SVG_OBJECTS_ID = "objects";

(function initialize() {
  var mouseController = new MouseController();
  var svgSurface = getElement(SVG_SURFACE_ID);
  var svgToolboxSurface = getElementBy(SVG_TOOLBOX_SURFACE_ID);
  var svgObjects = getElement(SVG_OBJECTS_ID);

  var surface = new Surface(mouseController, svgSurface, svgObjects);
  surface.resizeGrid(100, 100, 20, 20);

  var toolboxController = new ToolboxController(mouseController);

  // So we can handle mouse drag operations when the mouse moves onto the toolbox surface...
  var toolboxSurface = new ToolboxSurface(toolboxController, svgToolboxSurface);

  // The surface mouse controller needs to know the toolbox controller to finish
  // a toolbox drag & drop operation.
  mouseController.setToolboxController(toolboxController);
  // To compensate for translations when doing a toolbox drag&drop
  mouseController.setSurfaceShape(surface);
  toolboxController.setSurfaceShape(surface);

  new ToolboxRectangle(toolboxController, getElement("toolboxRectangle"));
  new ToolboxCircle(toolboxController, getElement("toolboxCircle"));
  new ToolboxDiamond(toolboxController, getElement("toolboxDiamond"));
})();

```

Notice some changes (will be explained next):

- There are setter methods to tell the mouse controller (which handles the surface) about the toolbox controller and surface "shape." This will be explained shortly.
- The toolbox controller is derived from the mouse controller, as it is a specialized mouse controller for handling drag & drop as well as "click and drop" operations.
- The toolbox controller needs to know about the surface "shape."
- Lastly, we initialize the objects backing the toolbox shapes.

I also added a simple helper method that, granted, I'm not using everywhere, but is less typing:

[Hide](#) [Copy Code](#)

```
function getElement(id) {
  var svg = document.getElementById(SVG_ELEMENT_ID);
  var el = svg.getElementById(id);

  return el;
}
```

Also, the better practice here is that we're looking for element ID's in the "svg" element, not the document.

The Toolbox Shapes

The toolbox shapes all implement the following functions:

- createElement - this creates an element with a starting location to place it on the surface next to the toolbox. This is used for "click and drop" operations.
- createElementAt - this creates an element at the specified location. This is used for "drag and drop" operations.
- createShape - instantiates the associated non-toolbox shape.

So, for example (choosing the diamond because it's a bit more complex):

[Hide](#) [Shrink](#) ▲ [Copy Code](#)

```
class ToolboxDiamond extends SvgToolboxElement {
  constructor(toolboxController, svgElement) {
    super(toolboxController, svgElement);
  }

  // For click and drop
  createElement() {
    var el = super.createElement('path',
      { d: "M 240 100 L 210 130 L 240 160 L 270 130 Z", stroke: "black", "stroke-width": 1, fill: "#FFFFFF"
    });

    return el;
  }

  // For drag and drop
  createElementAt(x, y) {
    var points = [
      { cmd: "M", x: x-15, y: y-30 },
      { cmd: "L", x: x - 45, y: y },
      { cmd: "L", x: x-15, y: y + 30 },
      { cmd: "L", x: x + 15, y: y }];

    var path = points.reduce((acc, val) => acc = acc + val.cmd + " " + val.x + " " + val.y, "");
    path = path + " Z";
    var el = super.createElement('path', { d: path, stroke: "black", "stroke-width": 1, fill: "#FFFFFF" });

    return el;
  }

  createShape(mouseController, el) {
    var shape = new Diamond(mouseController, el);

    return shape;
  }
}
```

All the toolbox shapes follow the above template.

The SvgToolboxElement Class

The base class for all toolbox elements wires up the mouse events for toolbox shape elements to the *toolboxController*. It also provides a common method for creating an element and setting its attributes, including creating a unique ID for the element:

Hide Shrink ▲ Copy Code

```
class SvgToolboxElement extends SvgObject {
  constructor(toolboxController, svgElement) {
    super(toolboxController, svgElement);
    this.toolboxController = toolboxController;
    this.registerEventListener(svgElement, "mousedown", toolboxController.onMouseDown, toolboxController);
    this.registerEventListener(svgElement, "mouseup", toolboxController.onMouseUp, toolboxController);
    this.registerEventListener(svgElement, "mousemove", toolboxController.onMouseMove, toolboxController);
    this.svgns = "<a href='http://www.w3.org/2000/svg'>http://www.w3.org/2000/svg</a>";
  }

  // Create the specified element with the attributes provided in a key-value dictionary.
  createElement(elementName, attributes) {
    var el = document.createElementNS(this.svgns, elementName);

    // Create a unique ID for the element so we can acquire the correct shape controller
    // when the user drags the shape.
    el.setAttributeNS(null, "id", this.uuidv4());

    // Create a class common to all shapes so that, on file load, we can get them all and re-attach them
    // to the mouse controller.
    el.setAttributeNS(null, "class", SHAPE_CLASS_NAME);

    // Add the attributes to the element.
    Object.entries(attributes).map(([key, val]) => el.setAttributeNS(null, key, val));

    return el;
  }

  // From SO: <a href='https://stackoverflow.com/questions/105034/create-guid-uuid-in-javascript'>https://stackoverflow.com/questions/105034/create-guid-uuid-in-javascript</a>
  uuidv4() {
    return ([1e7] + -1e3 + -4e3 + -8e3 + -1e11).replace(/[018]/g, c =>
      (c ^ crypto.getRandomValues(new Uint8Array(1))[0] & 15 >> c / 4).toString(16))
  }
}
```

The ToolboxController Class

Most of how the "click and drop" and "drag and drop" behavior is handled here. Remember that this class derives from *MouseController*, however it also needs to be initialized with the *surface* mouse controller -- it gets interesting (or perhaps confusing) to have two mouse controllers in this class!

Constructor

Hide Copy Code

```
class ToolboxController extends MouseController {
  // We pass in the mouse controller that the surface is using so we can
  // pass control over to the surface mouse controller when dragging a shape.
  constructor(mouseController) {
    super();
    this.mouseController = mouseController;
    this.draggingShape = false;
  }
}
```

As the comments state, we need the *surface* mouse controller so that for drag & drop operations, we can pass off the shape dragging to the *surface* mouse controller. When dragging, a non-toolbox shape is created. This shape wires up the mouse events using the *surface* mouse controller, which is why we need to pass control over to that controller. The alternative would be to tell the shape how to route the mouse events, and once the shape is dropped onto the surface, the events would have to be detached from the toolbox controller and attached to the surface mouse controller. So it merely shoves the problem around. Still, there might be a better way to do this.

Toolbox Controller onMouseDown

[Hide](#) [Copy Code](#)

```
onMouseDown(evt) {  
    super.onMouseDown(evt);  
}
```

We let the base class handle this behavior. The event is wired up to the toolbox controller.

Determining a Click Event

[Hide](#) [Copy Code](#)

```
isClick(evt) {  
    var endDownX = evt.clientX;  
    var endDownY = evt.clientY;  
  
    var isClick = Math.abs(this.startDownX - endDownX) < TOOLBOX_DRAG_MIN_MOVE &&  
        Math.abs(this.startDownY - endDownY) < TOOLBOX_DRAG_MIN_MOVE;  
  
    return isClick;  
}
```

While we could use the "onclick" event, I want finer grained control and I don't want to deal with whether the click event fires after a mouse up "click and drag" vs. a mouse up "drag and drop." OK, I still have to worry about that, but it makes more sense (to me at least) to just handle this in the mouse up event.

Toolbox Controller onMouseUp

[Hide](#) [Copy Code](#)

```
// If this is a "click", create the shape in a fixed location on the surface.  
// If this is the end of a drag operation, place the shape on the surface at  
// the current mouse position.  
onMouseUp(evt) {  
    if (this.isClick(evt) && !(this.activeController instanceof ToolboxSurface)) {  
        // Treat this as a click.  
        var el = this.activeController.createElement();  
  
        // The new shape is attached to the grid surface's mouse controller.  
        var shape = this.activeController.createShape(this.mouseController, el);  
        this.setShapeName(el, shape);  
  
        // Account for surface translation (scrolling)  
        shape.translate(-this.surfaceShape.X, -this.surfaceShape.Y);  
  
        // Use the mouse controller associated with the surface.  
        this.dropShapeOnSurface(SVG_OBJECTS_ID, el, shape);  
        this.mouseDown = false;  
    }  
}
```

Note that we prevent anything from happening if the user clicks on the toolbox surface itself.

This is the heart of the "click & drag" behavior. A click is determined by a mouse up event occurring within a motion "window." After that:

- The "real" shape is created.
- Translated to account for surface translation.
- Dropped onto the surface.
- Cleanup.

Dropping the shape onto the surface involves appending the shape to the "objects" group and telling the surface mouse controller about the shape:

Hide Copy Code

```
dropShapeOnSurface(groupName, svgElement, shapeController) {
  getElement(groupName).appendChild(svgElement);
  this.mouseController.attach(svgElement, shapeController);
}
```

Toolbox Controller onMouseMove

Hide Shrink ▲ Copy Code

```
// If the user is dragging, we create a new shape that can be dragged onto
// the surface. When the drag operation ends, the shape is transferred to the surface.
onMouseMove(evt) {
  if (this.mouseDown) {
    evt.preventDefault();
    if (this.draggingShape) {
      // Our toolbox surface picked up the event instead of the shape. Handle
      // as if the shape got the event.
      super.onMouseMove(evt);
    } else {
      // Make sure a shape has been selected rather than dragging the toolbox surface.
      if (!(this.activeController instanceof ToolboxSurface)) {
        if (!this.isClick(evt)) {
          var endDownX = evt.clientX;
          var endDownY = evt.clientY;
          var el = this.activeController.createElementAt(endDownX, endDownY);
          // Here, because we're dragging, the shape needs to be attached to both the toolbox controller
          and the surface's mouse controller
          // so that if the user moves the shape too quickly, either the toolbox controller or the surface
          controller will pick it up.
          var shape = this.activeController.createShape(this.mouseController, el);
          this.setShapeName(el, shape);
          // set the shape name so we can map shape names to shape constructors when loading a diagram.
          el.setAttributeNS(null, "shapeName", shape.constructor.name);
          shape.mouseController.mouseDownX = endDownX;
          shape.mouseController.mouseDownY = endDownY + 30; // Offset so shape is drawn under mouse.
          this.createShapeForDragging(el, shape);
          this.draggingShape = true;
        }
      }
    }
  }
}
```

This is the most complicated piece. The above code handles:

- If the user moves the shape quickly, the toolbox surface might get the event, so we handle the default behavior which is to update the shape's translation.
 - One caveat -- if the mouse is moving within the click window, the surface mouse controller doesn't have the active shape yet, so nothing happens.
- We also don't want the user dragging the toolbox surface itself. At least not yet. Maybe this will scroll the shapes in the toolbox later.
- A drag operation begins only when the user has moved the mouse enough to not be considered a click event.
- The shape is handed off to the *surface* mouse controller at this point.

When the shape is created for dragging, it is actually appended to the *toolbox* SVG group, so it stay in the foreground while the user moves the shape over to the grid. Later we have to move the shape to the *objects* SVG group.

Hide Copy Code

```

// Place the shape into the toolbox group so it's topmost, and attach the shape to mouse our toolbox mouse
controller
// and the surface mouse controller so off-shape mouse events are handled correctly.
createShapeForDragging(e1, shape) {
    // The shape is now under the control of the surface mouse controller even though we added it to our
    toolbox group.
    // This is because the shape wires up the surface mouse controller events.
    // The only thing the toolbox controller will see is the onMouseMove when the user moves the mouse too
    fast and the
    // mouse events end up being handled by the toolbox controller (or, if over the surface, the surface
    controller.)
    this.dropShapeOnSurface(SVG_TOOLBOX_ID, e1, shape);

    // We need to know what shape is being moved, in case we (the toolbox controller) start to receive mouse
    move events.
    this.attach(e1, shape);
    this.activeController = shape;

    // The surface mouse controller also needs to know what shape is active and that we are in the "mouse
    down" state.
    this.mouseController.activeController = shape;
    this.mouseController.mouseDown = true;
}

```

At this point, the surface mouse controller has control!

Mouse Controller onMouseUp

[Hide](#) [Copy Code](#)

```

// Any dragging is now done.
onMouseUp(evt) {
    if (evt.button == LEFT_MOUSE_BUTTON && this.activeController != null) {
        evt.preventDefault();
        // Allows the toolbox controller to finish the drag & drop operation.
        this.toolboxController.mouseUp();
        this.clearSelectedObject();
    }
}

```

As stated above, the *surface* mouse controller has control of the shape when the drag operation begins. When it receives a mouse up event, it gives the toolbox controller the opportunity to finish any toolbox drag & drop operation:

[Hide](#) [Copy Code](#)

```

// Handles end of drag & drop operation, otherwise, does nothing -- toolbox item was clicked.
mouseUp() {
    if (this.draggingShape) {
        // Account for surface translation (scrolling)
        this.activeController.translate(-this.surfaceShape.X, -this.surfaceShape.Y);

        var e1 = this.activeController.svgElement;

        // Move element out of the toolbox group and into the objects group.
        getElement(SVG_TOOLBOX_ID).removeChild(e1);
        getElement(SVG_OBJECTS_ID).appendChild(e1);
        this.dragComplete(e1);
    }
}

```

Here the element is moved from the topmost foreground position (in the toolbox SVG group) to the objects SVG group. We also have to account for any surface translation so the shape appears exactly where it is when the user concludes the drag operation. Lastly, we clean up the toolbox controller's state:

[Hide](#) [Copy Code](#)

```

dragComplete(e1) {
  this.draggingShape = false;
  this.detach(e1);
  this.mouseDown = false;
  this.activeController = null;
}

```

Hide Copy Code

```

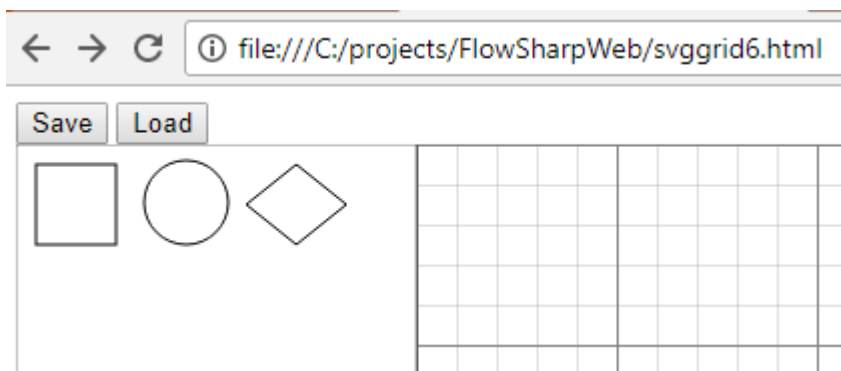


```

Phew! All done! (Except maybe getting the shapes to click & drop at more uniform location.)

Saving and Restoring the Diagram Locally

We have enough happening now that before doing anything else, I think it's a good idea to look at how shapes are saved and loaded locally. The implementation I present here is very rudimentary -- it automatically initiates a download which will go into the Downloads folder, and in Chrome, any existing file causes the filename to be appended with (n) where n is an incrementing number. At some point I will enhance this functionality using the HTML5 FileSystem API (<https://www.html5rocks.com/en/tutorials/file/filesystem/> and <https://dev.w3.org/2009/dap/file-system/pub/FileSystem/>), part of the WebAPI. But for now, it writes "data.svg" and on load, let's you select the directory and file. Rather than focusing on the UI for saving/loading diagrams, the point here is to focus on the mechanics of actually saving and loading the diagram itself. Between figuring out how to write the Javascript, working with the SVG DOM, and fixing bugs, this took two days!



To begin with, I added Save and Load buttons at the top of the page:

Hide Copy Code

```

<div>
  <!-- <a href="https://stackoverflow.com/questions/1944267/how-to-change-the-button-text-of-input-type-file%20--">https://stackoverflow.com/questions/1944267/how-to-change-the-button-text-of-input-type-file --> -->
  <!-- creates a hidden file input on routes the button to clicking on that tag -->
  <button onclick="saveSvg()">Save</button>
  <button onclick="document.getElementById('fileInput').click();">Load</button>
  <input type="file" id="fileInput" style="display:none;" />
</div>

```

The trick here, to avoid the default behavior of a file input element, was to hide the input element, as to SO link showed me.

Saving the Diagram

Saving the SVG *locally* took some research and resulted in this code:

Hide Copy Code

```

document.getElementById(FILE_INPUT).addEventListener('change', readSingleFile, false);

```

Hide Copy Code

```
// https://stackoverflow.com/questions/23582101/generating-viewing-and-saving-svg-client-side-in-browser
function saveSvg() {
  var svg = getElement(SVG_OBJECTS_ID);
  // <a href="https://developer.mozilla.org/en-US/docs/Web/API/XMLSerializer">https://developer.mozilla.org/en-US/docs/Web/API/XMLSerializer</a>
  var serializer = new XMLSerializer();
  var xml = serializer.serializeToString(svg);
  // Prepend the xml with other things we want to save, like the surface translation and grid spacing.
  xml = "<diagram>" + surface.serialize() + "</diagram>" + xml;
  var blob = new Blob([xml], { 'type': "image/svg+xml" });

  // We're using <a
  href="https://github.com/eligrey/FileSaver.js/">https://github.com/eligrey/FileSaver.js</a>
  // but with the "export" (a require node.js thing) removed.
  // There are several forks of this, not sure if there's any improvements in the forks.
  saveAs(blob, FILENAME);
}
```

As the comments point out, I'm using FileSaver.js, written by "eligrey." Thank goodness for open source -- this works in Chrome and Edge (the two browsers I tested) and has support for other browser's nuances as well. Of particular note here:

- We're giving the surface (and maybe other objects later on) the opportunity to save their state. The surface needs to save:
 - It's translation.
 - The grid spacing.
- This data, as an XML string, is prepended to the SVG data.

In the surface class, this is implemented as:

Hide Copy Code

```
// Create an XML fragment for things we want to save here.
serialize() {
  var el = document.createElement("surface");
  // DOM adds elements as lowercase, so let's just start with lowercase keys.
  var attributes = {x : this.X, y : this.Y, gridcellw : this.gridCellW, gridcellh : this.gridCellH,
    cellw : this.cellW, cellh : this.cellH}
  Object.entries(attributes).map(([key, val]) => el.setAttribute(key, val));
  var serializer = new XMLSerializer();
  var xml = serializer.serializeToString(el);

  return xml;
}
```

That was the easy part -- the file is *downloaded* into the browser's default download location.

Loading the Diagram

The first step is to actually read the file data locally:

Hide Copy Code

```
// https://w3c.github.io/FileAPI/
// https://stackoverflow.com/questions/3582671/how-to-open-a-local-disk-file-with-javascript
// Loading the file after it has been loaded doesn't trigger this event again because it's
// hooked up to "change", and the filename hasn't changed!
function readSingleFile(e) {
  var file = e.target.files[0];
  var reader = new FileReader();
  reader.onload = loadComplete;
  reader.readAsText(file);
  // Clears the last filename(s) so loading the same file will work again.
  document.getElementById(FILE_INPUT).value = "";
}
```


This function uses the WebAPI's **FileReader** class. The most interesting thing here is clearing the filename from the input element. As the comment points out, if we don't do this, we can't re-load the diagram if it has the same filename. Very annoying for testing.

When the load completes (I didn't implement any error checking / validation that the file is actually a diagram file):

Hide Copy Code

```
function loadComplete(e) {
  var contents = e.target.result;
  var endOfDiagramData = contents.indexOf(END_OF_DIAGRAM_TAG);
  var strDiagram = contents.substr(0, endOfDiagramData).substr(START_OF_DIAGRAM_TAG.length);
  var xmlDiagram = stringToXml(strDiagram);
  // Deserialize the diagram's surface XML element to restore grid spacing and grid translation.
  surface.deserialize(xmlDiagram);
  var svgData = contents.substr(endOfDiagramData + END_OF_DIAGRAM_TAG.length)
  replaceObjects(contents);
}
```

Several things happen:

- The data (as a string) is separated out into the diagram "state" information -- currently just the surface state -- and the SVG data.
- The surface state is restored.
- The "objects" element is replaced.

The surface state is deserialized and restored:

Hide Copy Code

```
// Deserialize the xml fragment that contains the surface translation and grid dimensions on a file load.
deserialize(xml) {
  var obj = xmlToJson(xml);
  var attributes = obj.surface.attributes;
  // Note the attributes, because they were serialized by the DOM, are all lowercase.
  // OK to assume all ints?
  this.X = parseInt(attributes.x);
  this.Y = parseInt(attributes.y);
  this.gridCellW = parseInt(attributes.gridcellw);
  this.gridCellH = parseInt(attributes.gridcellh);
  this.cellW = parseInt(attributes.cellw);
  this.cellH = parseInt(attributes.cellh);
  var dx = this.X % this.gridCellW;
  var dy = this.Y % this.gridCellH;
  this.resizeGrid(this.gridCellW, this.gridCellH, this.cellW, this.cellH);
  this.svgElement.setAttribute("transform", "translate(" + dx + "," + dy + ")");
}
```

The deserializer xmlToJson was found at the link in the comments. I made a minor tweak to the code described in that link:

Hide Shrink ▲ Copy Code

```
function stringToXml(xmlStr) {
  // <a href="https://stackoverflow.com/a/3054210/2276361">https://stackoverflow.com/a/3054210/2276361</a>
  return (new window.DOMParser()).parseFromString(xmlStr, "text/xml");
}

// https://davidwalsh.name/convert-xml-json
function xmlToJson(xml) {
  var obj = {};

  if (xml.nodeType == 1) { // element
    // do attributes
    if (xml.attributes.length > 0) {
      obj["attributes"] = {};

      for (var j = 0; j < xml.attributes.length; j++) {
        var attribute = xml.attributes.item(j);
        obj["attributes"][attribute.nodeName] = attribute.nodeValue;
      }
    }
  }
}
```

```

} else if (xml.nodeType == 3) { // text
  obj = xml.nodeValue;
}

// do children
if (xml.hasChildNodes()) {
  for(var i = 0; i < xml.childNodes.length; i++) {
    var item = xml.childNodes.item(i);
    var nodeName = item.nodeName;

    if (typeof(obj[nodeName]) == "undefined") {
      obj[nodeName] = xmlToJson(item);
    } else {
      if (typeof(obj[nodeName].push) == "undefined") {
        var old = obj[nodeName];
        obj[nodeName] = [];
        obj[nodeName].push(old);
      }

      obj[nodeName].push(xmlToJson(item));
    }
  }
}

return obj;
};

```

Next, the "objects" element is replaced. To do this, I wrapped the "objects" element in an group so that the child element of the wrapping group can be manipulated.

[Hide](#) [Copy Code](#)

```

<!-- Also, we create an outer group so that on file load, we can remove
      the "objectGroup" and replace it with what got loaded. -->
<g id="objectGroup">
  <g id="objects" transform="translate(0, 0)"></g>
</g>

```

The Javascript code:

[Hide](#) [Copy Code](#)

```

// Replace "objects" with the contents of what got loaded.
function replaceObjects(contents) {
  mouseController.destroyAllButSurface();
  var objectGroup = getElement(OBJECT_GROUP_ID);
  var objects = getElement(SVG_OBJECTS_ID);
  objectGroup.removeChild(objects);
  // <a href="https://stackoverflow.com/questions/38985998/insert-svg-string-element-into-an-existing-svg-
  tag">https://stackoverflow.com/questions/38985998/insert-svg-string-element-into-an-existing-svg-tag</a>
  objectGroup.innerHTML = contents;
  createShapeControllers();
  // re-acquire the objects element after adding the contents.
  var objects = getElement(SVG_OBJECTS_ID);
  surface.svgObjects = objects;
}

```

Several things happen in order to replace the "objects" element:

- All objects except the "surface" currently on the surface are "destroyed." This means that:
 - Their events are unwired.
 - They are detached from the mouse controller.
- The child "objects" element is removed.
- The outer group's inner HTML is replaced with the SVG data that was loaded from the file.
- Next, the backing shape controller classes need to be instantiated. This means:
 - Wiring up their events.

- Attaching them to the mouse controller.
 - Fixing up their position so the object knows how they've been translated.
- Lastly:
 - The new "objects" element is acquired.
 - The surface controller is told about the new "objects" element.

Why are we destroying all SVG elements except the "surface" element? The surface element is effectively our placeholder element for the grid and handles the scrolling of the surface. We don't need to replace that element, so we ignore it:

Hide Copy Code

```
destroyAllButSurface() {
  Object.entries(this.controllers).map(([key, val]) => {
    if (!(val instanceof Surface)) {
      val.destroy();
    }
  });
}
```

Creating the shape controllers is done with a lookup to map the **shapename** attribute to the function that instantiates the correct shape controller:

Hide Copy Code

```
var elementNameShapeMap = {
  Rectangle: (mouseController, svgElement) => new Rectangle(mouseController, svgElement),
  Circle: (mouseController, svgElement) => new Circle(mouseController, svgElement),
  Diamond: (mouseController, svgElement) => new Diamond(mouseController, svgElement)
};
```

As an aside, where did the **shapename** attribute come from? This is created when the shape is click & dropped or drag & dropped by the toolbox. In the **ToolboxController** class:

Hide Copy Code

```
setShapeName(el, shape) {
  // set the shape name so we can map shape names to shape constructors when loading a diagram.
  // <a href="https://stackoverflow.com/questions/1249531/how-to-get-a-javascript-objects-class">https://stackoverflow.com/questions/1249531/how-to-get-a-javascript-objects-class</a>
  el.setAttributeNS(null, SHAPE_NAME_ATTR, shape.constructor.name);
}
```

Also, in the **SvgToolboxElement** class, we add a class attribute that makes it easy to get all the SVG elements in the "objects" group:

Hide Copy Code

```
// Create a class common to all shapes so that, on file load, we can get them all and re-attach them
// to the mouse controller.
el.setAttributeNS(null, "class", SHAPE_CLASS_NAME);
```

Javascript for creating the shape controllers:

Hide Copy Code

```
// The difficult part -- creating the shape controller based on the element's shapeName attribute to the
// shape controller class counterpart.
function createShapeControllers() {
  var els = getElements(SHAPE_CLASS_NAME);

  for (let el of els) { // note usage "of" - ES6. note usage "let" : scope limited to block.
    let shapeName = el.getAttribute(SHAPE_NAME_ATTR);
    let creator = elementNameShapeMap[shapeName];
    let shape = creator(mouseController, el);
    // Annoyingly, we DO have to parse the translation to set the X and Y properties of the shape!
    let transform = el.getAttribute("transform");
    let transforms = parseTransform(transform);
```

```

let translate = transforms["translate"];
// We assume integers?
shape.X = parseInt(translate[0]);
shape.Y = parseInt(translate[1]);
}
}

```

Annoyingly, we have to actually parse the transform because I don't add attributes to the SVG element for the shape controller. This is done using some code I found on SO:

[Hide](#) [Copy Code](#)

```

// https://stackoverflow.com/questions/17824145/parse-svg-transform-attribute-with-javascript
function parseTransform(transform) {
  var transforms = {};
  for (var i in a = transform.match(/(\w+\\((\\-?\\d+\\.?\\d*e?\\-?\\d*,?)\\))+/g)) {
    var c = a[i].match(/[\\w\\.\\-]+/g);
    transforms[c.shift()] = c;
  }

  return transforms;
}

```

We now have a simple mechanism for saving and loading a diagram. When new shapes are added, only the `elementNameShapeMap` needs to be updated.

Lines and Anchor Points

The last thing I want to create for this article is the ability to draw simple lines that can connect shapes. Easier said than done, as this means we will need some additional diagram state information so that we know what lines are attached to what shapes so that when the shape moves, the line is updated as well. I'm not even going to deal with [arrows](#) yet!

The Complexities of Selecting A Line

After creating the `Line` and `ToolboxLine` classes following the same template as the other shape controller classes, adding a line element to the toolbox group and wiring up the shape controller:

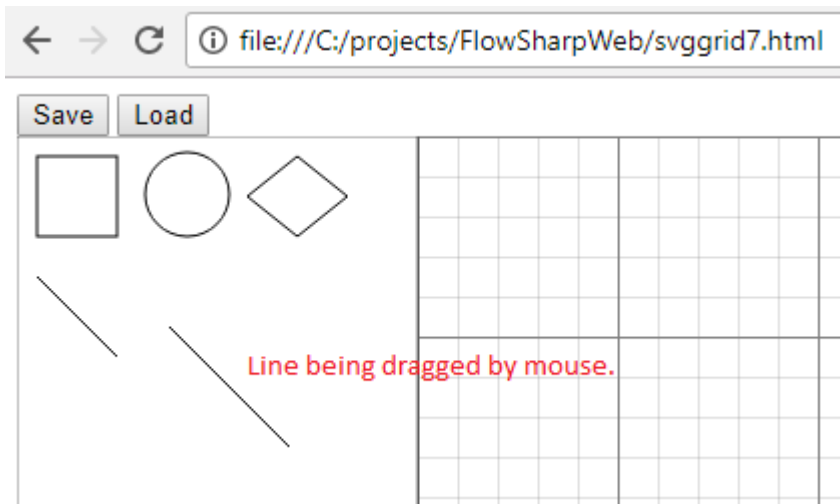
[Hide](#) [Copy Code](#)

```

<line id="toolboxLine" x1="10" y1="70" x2="50" y2="110" stroke="black" stroke-width="1" fill="#FFFFFF" />
new ToolboxLine(toolboxController, getElement(TOOLBOX_LINE_ID));

```

we encounter the first problem -- it's nearly impossible to actually select the line because the line is so thin -- you have to click exactly on the pixels of the line in order to select it. The best solution to this issue seems to be to create a group with two lines: the actual line and a transparent line with a larger width ([reference](#).) This is what we want to do once the shape is on the drawing, but for the toolbox, we don't want the user to have to be that precise, so instead we'll create a transparent rectangle so that visually, anywhere in the "box" formed by the toolbox line shape will work. Back to the toolbox group:



This works quite well (the commented out transparent line is for future reference):

[Hide](#) [Copy Code](#)

```
<g id="toolboxLine">
  <line id="line" x1="10" y1="70" x2="50" y2="110" stroke="black" stroke-width="1" fill="#FFFFFF" />
  <rect id="hiddenLine" x="10" y="70" width="40" height="40" stroke="black" stroke-opacity="0" fill-
opacity="0"/>
  <!--<line id="line2" x1="10" y1="70" x2="50" y2="110" fill="#FFFFFF" stroke-opacity="0" fill-
opacity="0"/>-->
</g>
```

While we're at it, we can anticipate the next problem -- clicking and dragging the line's endpoints so that it's easier to change the line's length and orientation. Let's look at how this group is rendered:

[Hide](#) [Copy Code](#)

```
<g id="toolboxLine">
  <rect id="hiddenLine" x="10" y="70" width="40" height="40" stroke="black" stroke-opacity="0" fill-
opacity="0"/>
  <line id="line2" x1="10" y1="70" x2="50" y2="110" fill="#FFFFFF" stroke="black" stroke-width="20"/>
  <line id="line" x1="10" y1="70" x2="50" y2="110" stroke="red" stroke-width="1" fill="#FFFFFF" />
</g>
```



Notice that the stroke width doesn't cause the larger line to extend beyond the extents of the red line with stroke width 1. From a UI perspective, this means that the user would have to select a line endpoint by being "inside" the line -- selecting the line endpoint near the "outside" won't result in the mouse events being handled by the line. Again, we can fix this by creating transparent rectangles around the line endpoints which will represent the clickable area for selecting a line endpoint. When rendered without transparency, we get this -- the red areas are the clickable areas to select the shape in the toolbox and to select the endpoints once the line has been drawn on the surface:



[Hide](#) [Copy Code](#)

```
<g id="toolboxLine">
  <rect id="lineHiddenSelectionArea" x="10" y="70" width="40" height="40" stroke="red" stroke-width="1"
fill="#FFFFFF"/>
  <rect id="endpoint1" transform="translate(10, 70)" x="-5" y="-5" width="10" height="10" stroke="red"
stroke-width="1" fill="#FFFFFF"/>
  <rect id="endpoint2" transform="translate(50, 110)" x="-5" y="-5" width="10" height="10" stroke="red"
stroke-width="1" fill="#FFFFFF"/>
</g>
```

```
stroke-width="1" fill="#FFFFFF" />
<line id="line" x1="10" y1="70" x2="50" y2="110" fill="#FFFFFF" stroke="black" stroke-width="1" />
</g>
```

In reality, when creating a line on the surface, we'll deal with anchors slightly differently rather than adding them to the "toolboxLine" group.

- For the toolbox, we don't need to the endpoint rectangles or the larger transparent stroke line.
- For the line when it's on the diagram surface, we need to endpoint rectangles and the larger transparent stroke line for easy selection by the mouse.
 - In reality, when creating a line on the surface, we'll deal with anchors slightly differently rather than adding them to the "toolboxLine" group, as described in the next section.

This means that we have some specialized work to do when creating the element when it is dropped onto the surface (showing only the `createElement` function):

[Hide](#) [Copy Code](#)

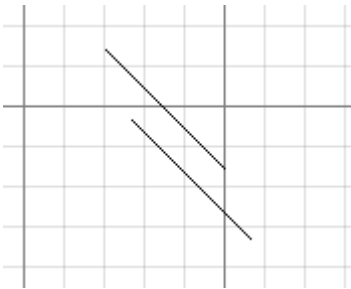
```
createElement() {
  var el = super.createElement('g', {});
  el.appendChild(super.createElement('line', { x1: 240, y1: 100, x2: 300, y2: 160, "stroke-width": 20,
  stroke: "black", "stroke-opacity": "0", "fill-opacity": "0" }));
  el.appendChild(super.createElement('line', { x1: 240, y1: 100, x2: 300, y2: 160, fill: "#FFFFFF",
  stroke: "black", "stroke-width": 1 }));

  return el;
}
```

Of note here is something important:

- The stroke "color" must be initialized in the transparent line, otherwise the outer group does not include it in its region and the wider transparent line is not selectable!

I also had to write a `createChildElement` function which differs from `createElement` only in that it does not create a `class` attribute, as we don't want these child elements to be mapped to the shape controllers -- only the outer group.



We can now click & drop and drag & drop a line onto the surface and then drag the line around.

Point Class and Shape Rectangle

At this point, I'm going to digress briefly -- it's time to create a Point class and functions that help us get the diagonal corners of a shape. Each shape has nuances. Lines have (x1,y1) and (x2,y2) attributes, rectangles have (x, y) and (width, height) attributes, circles have (cx, cy) and (r) attributes and paths, well, having a bounding rectangle. I want to unify this mess. We'll define a simple Point class:

[Hide](#) [Copy Code](#)

```
class Point {
  constructor(x, y) {
    this.X = x;
    this.Y = y;
  }

  translate(x, y) {
    this.X += x;
  }
}
```

```
    this.Y += y;

    return this;
}
}
```

Yes, there's already an [SVGPoint](#) object as well as a [DOMPoint](#) object, but I'm not using either, as I want behaviors, like `translate`, that these objects don't provide.

Now we can implement getting the upper left and lower right corners for each shape, *translated* to the absolute coordinate of the shape.

Rectangle:

[Hide](#) [Copy Code](#)

```
getULCorner() {
    var p = new Point(+this.svgElement.getAttribute("x"), +this.svgElement.getAttribute("y"));
    p = this.getAbsoluteLocation(p);

    return p;
}
```

[Hide](#) [Copy Code](#)

```
getLRCorner() {
    var p = new Point(+this.svgElement.getAttribute("x") + +this.svgElement.getAttribute("width"),
                     +this.svgElement.getAttribute("y") + +this.svgElement.getAttribute("height"));
    p = this.getAbsoluteLocation(p);

    return p;
}
```

Circle:

[Hide](#) [Copy Code](#)

```
getULCorner() {
    var p = new Point(+this.svgElement.getAttribute("cx") - +this.svgElement.getAttribute("r"),
                     +this.svgElement.getAttribute("cy") - +this.svgElement.getAttribute("r"));
    p = this.getAbsoluteLocation(p);

    return p;
}

getLRCorner() {
    var p = new Point(+this.svgElement.getAttribute("cx") + +this.svgElement.getAttribute("r"),
                     +this.svgElement.getAttribute("cy") + +this.svgElement.getAttribute("r"));
    p = this.getAbsoluteLocation(p);

    return p;
}
```

Line:

[Hide](#) [Copy Code](#)

```
getULCorner() {
    var line = this.svgElement.children[0];
    var p = new Point(+line.getAttribute("x1"), +line.getAttribute("y1"));
    p = this.getAbsoluteLocation(p);

    return p;
}

getLRCorner() {
    var line = this.svgElement.children[0];
    var p = new Point(+line.getAttribute("x2"), +line.getAttribute("y2"));
    p = this.getAbsoluteLocation(p);
}
```

```
    return p;
}
```

Diamond:

Paths are interesting because `getBoundingClientRect` returns the shape's location *already translated*. As the function name indicates, it returns the client (in screen coordinates) location, so we have to translate it to the root SVG element's location.

[Hide](#) [Copy Code](#)

```
getULCorner() {
    var rect = this.svgElement.getBoundingClientRect();
    var p = new Point(rect.left, rect.top);
    this.translateToSvgCoordinate(p);

    return p;
}

getLRCorner() {
    var rect = this.svgElement.getBoundingClientRect();
    var p = new Point(rect.right, rect.bottom);
    this.translateToSvgCoordinate(p);

    return p;
}
```

And for the two helper functions:

[Hide](#) [Copy Code](#)

```
getAbsoluteLocation(p) {
    p.translate(this.X, this.Y);
    p.translate(this.mouseController.surface.X, this.mouseController.surface.Y);

    return p;
}

// https://stackoverflow.com/questions/22183727/how-do-you-convert-screen-coordinates-to-document-space-in-a-scaled-svg
translateToSvgCoordinate(p) {
    var svg = document.getElementById(SVG_ELEMENT_ID);
    var pt = svg.createSVGPoint();
    var offset = pt.matrixTransform(svg.getScreenCTM().inverse());
    p.translate(offset.x, offset.y);
}
```

Anchors

Next, we want to be able to change the line length and its orientation. This will be a useful exercise as the behavior is similar to resizing a shape. In WinForm app `FlowSharp`, I had each shape determine the anchor points for sizing. We'll do the same thing here. We finally have something to implement in the shape's controller classes! The idea here is that when the mouse hovers over a shape, the anchors magically appear so the user has an indication of where to click & drag to modify the shape. In the mouse controller, we'll add an `onMouseOver` event handler and add it to the events that get wired up in the shape's controller `SvgElement` base class:

[Hide](#) [Copy Code](#)

```
this.registerEventListener(svgElement, "mouseover", mouseController.onMouseOver, mouseController);
```

The event handler:

[Hide](#) [Copy Code](#)

```
onMouseOver(evt) {
    var id = evt.currentTarget.getAttribute("id");
    var hoverShape = this.controllers[id];
```



```

// On drag & drop, anchors are not shown because of this first test.
// We do this test so that if the user moves the mouse quickly, we don't
// re-initialize the anchors when the shape catches up (resulting in
// a mousemove event again.
if (this.activeController == null) {
  if (hoverShape instanceof SvgElement &&
      !(hoverShape instanceof ToolboxController) &&
      !(hoverShape instanceof Surface)) {
    this.displayAnchors(hoverShape);
  } else {
    this.removeAnchors();
    this.anchors = [];
  }
}
}

displayAnchors(hoverShape) {
  var anchors = hoverShape.getAnchors();
  this.showAnchors(anchors);
  this.anchors = anchors;
}

```

Anchors will be displayed between the "objects" and "toolbox" groups, so that anchors are on top of every other shape but below the toolbox:

[Hide](#) [Copy Code](#)

```

<g id="objectGroup">
  <g id="objects" transform="translate(0, 0)"></g>
</g>
<g id="anchors"></g>
<g id="toolbox" x="0" y="0" width="200" height="480">
...

```

Creating anchors:

[Hide](#) [Copy Code](#)

```

showAnchors(anchors) {
  // not showing?
  if (this.anchors.length == 0) {
    var anchorGroup = getElement(ANCHORS_ID);
    // Reset any translation because the next mouse hover will set the anchors directly over the shape.
    anchorGroup.setAttribute("transform", "translate(0, 0)");

    anchors.map(anchor => {
      var el = this.createElement("rect",
        { x: anchor.X - 5, y: anchor.Y - 5, width: 10, height: 10,
          fill: "#FFFFFF", stroke: "black", "stroke-width": 0.5});
      anchorGroup.appendChild(el);
    });
  }
}

// TODO: Very similar to SvgToolboxElement.createElement. Refactor for common helper class?
createElement(name, attributes) {
  var svgns = "<a href='http://www.w3.org/2000/svg'>http://www.w3.org/2000/svg</a>";
  var el = document.createElementNS(svgns, name);
  el.setAttribute("id", Helpers.uuidv4());
  Object.entries(attributes).map(([key, val]) => el.setAttributeNS(null, key, val));

  return el;
}

```

Removing anchors:

[Hide](#) [Copy Code](#)

```

removeAnchors() {
  // already showing?
  if (this.anchors.length > 0) {
    var anchorGroup = getElement(ANCHORS_ID);

    // <a href="https://stackoverflow.com/questions/3955229/remove-all-child-elements-of-a-dom-node-in-javascript">https://stackoverflow.com/questions/3955229/remove-all-child-elements-of-a-dom-node-in-javascript</a>
    // Will change later.
    anchorGroup.innerHTML = "";
    // Alternatively:
    //while (anchorGroup.firstChild) {
    //  anchorGroup.removeChild(anchorGroup.firstChild);
    //}
  }
}

```

Notice the resetting of the anchor group translation above in **showAnchors** when the anchors are first drawn. One touch-up is that we now also need to translate the anchors group when the object is being dragged in the **SvgElement** class:

Hide Copy Code

```

onDrag(evt) {
  this.updatePosition(evt);
  this.svgElement.setAttribute("transform", "translate(" + this.X + "," + this.Y + ")");
  getElement(ANCHORS_ID).setAttribute("transform", "translate(" + this.X + "," + this.Y + ")");
  getElement(ANCHORS_ID).setAttribute("transform", "translate(" + this.dragX + "," + this.dragY + ")");
}

```

The **dragX** and **dragY** coordinates are reset on the mouse down event in the **MouseController**:

Hide Copy Code

```

onMouseDown(evt) {
  if (evt.button == LEFT_MOUSE_BUTTON) {
    evt.preventDefault();
    var id = evt.currentTarget.getAttribute("id");
    this.activeController = this.controllers[id];
    this.mouseDown = true;
    this.mouseDownX = evt.clientX;
    this.mouseDownY = evt.clientY;
    this.startDownX = evt.clientX;
    this.startDownY = evt.clientY;
    this.activeController.startMove(); // <----- added this
  }
}

```

We do this because the anchor group always begins with a translation of (0,0) so we need to know the translation relative to the current drag operation. There's two more nuances:

- Because the user can move the mouse quickly (off the shape), the **mouseover** event will re-fire (the mouse leaves the shape and when the shape catches up, the **mouseover** event is fired again.) For this reason we check to see if there's an active shape controller (a shape is actually being dragged), which is set when the user clicks down on a shape. The side-effect to this is actually a nice one -- the anchor points are not shown when the shape is drag & dropped from the toolbox because the surface mouse controller has an active shape controller.
- However, this has the unintentional side-effect of not showing the anchors when the shape is dropped after a drag & drop operation and the mouse is still over the just-dropped shape.

To fix the second problem, the toolbox controller must initiate the display of the anchors once the shape is dropped after a drag & drop operation:

Hide Copy Code

```

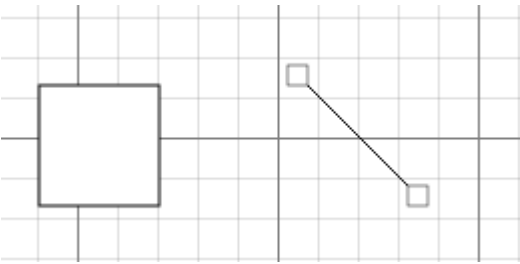
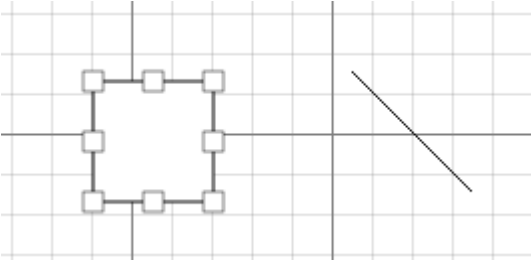
dragComplete(e1) {
  this.draggingShape = false;
  this.detach(e1);
  this.mouseDown = false;
}

```

```

this.mouseController.displayAnchors(this.activeController); // <--- I added this line
this.activeController = null;
}

```



Shapes now have anchors! We also have the visual side-effect of now showing the user what shape is about to be selected if the user wants to drag the shape.

Line Length and Orientation

Now that we have anchors displaying, we have to get the anchors working. We'll focus on the line shape. Notice in the above picture that the anchor is drawn on top of the line. This helps us select the anchor for dragging rather than the entire line. If the line was on top of the anchor it would be possible for the user to click exactly on the line, missing the anchor.

To begin with, we need to be able to specify the function that is called when the anchor is moved. This is a change in the anchors array that is passed back to the mouse controller when displaying anchors:

[Hide](#) [Copy Code](#)

```

getAnchors() {
  var corners = this.getCorners();
  var anchors = [{ anchor: corners[0], onDrag: this.moveULCorner.bind(this) },
                 { anchor: corners[1], onDrag: this.moveLRCorner.bind(this) }];

  return anchors;
}

```

- Notice the **bind**, so that the "this" in the event handler function is the **Line** object. Sigh.

The Line class implements the handlers (notice the extra anchor parameter which I discuss shortly):

[Hide](#) [Copy Code](#)

```

// Move the (x1, y1) coordinate.
moveULCorner(anchor, evt) {
  // Use movementX and movementY - this is much better than dealing with the base class X or dragX stuff.
  // Do both the transparent line and the visible line.
  this.moveLine("x1", "y1", this.svgElement.children[0], evt.movementX, evt.movementY);
  this.moveLine("x1", "y1", this.svgElement.children[1], evt.movementX, evt.movementY);
  this.moveAnchor(anchor, evt.movementX, evt.movementY);
}

// Move the (x2, y2) coordinate.
moveLRCorner(anchor, evt) {
  this.moveLine("x2", "y2", this.svgElement.children[0], evt.movementX, evt.movementY);
  this.moveLine("x2", "y2", this.svgElement.children[1], evt.movementX, evt.movementY);
  this.moveAnchor(anchor, evt.movementX, evt.movementY);
}

```

```

moveLine(ax, ay, line, dx, dy) {
  var x1 = +line.getAttribute(ax) + dx;
  var y1 = +line.getAttribute(ay) + dy;
  line.setAttribute(ax, x1);
  line.setAttribute(ay, y1);
}

```

Notice that we have to move both the transparent line and the visible line.

- At this point in the coding I learned about the **movementX** and **movementY** properties of the event, which if I'd known about earlier would have changed how I had implemented some of the other code!

The **moveAnchor** function will be common to all shapes, so it lives in the **SvgElement** base class:

[Hide](#) [Copy Code](#)

```

moveAnchor(anchor, dx, dy) {
  var tx = +anchor.getAttribute("tx") + dx;
  var ty = +anchor.getAttribute("ty") + dy;
  anchor.setAttribute("transform", "translate(" + tx + "," + ty + ")");
  anchor.setAttribute("tx", tx);
  anchor.setAttribute("ty", ty);
}

```

Next, we need an actual **Anchor** shape class:

[Hide](#) [Copy Code](#)

```

class Anchor extends SvgObject {
  constructor(anchorController, svgElement, onDrag) {
    super(anchorController, svgElement);
    this.wireUpEvents(svgElement);
    this.onDrag = onDrag;
  }

  wireUpEvents(svgElement) {
    // The mouse controller is actually the derived anchor controller.
    this.registerEventListener(svgElement, "mousedown", this.mouseController.onMouseDown,
this.mouseController);
    this.registerEventListener(svgElement, "mousemove", this.mouseController.onMouseMove,
this.mouseController);
    this.registerEventListener(svgElement, "mouseup", this.mouseController.onMouseUp,
this.mouseController);
    this.registerEventListener(svgElement, "mouseleave", this.mouseController.onMouseLeave,
this.mouseController);
  }
}

```

Again, in order to deal with mouse events being received by the surface when the anchor is dragged "too fast", the **AnchorController** spoofs the surface mouse controller into thinking (correctly so) that it's moving an anchor element:

[Hide](#) [Copy Code](#)

```

class AnchorController extends MouseController {
  constructor(mouseController) {
    super();
    // For handling dragging an anchor but the surface or shape gets the mousemove events.
    this.mouseController = mouseController;
  }

  onMouseDown(evt) {
    super.onMouseDown(evt);
    // For handling dragging an anchor but the surface or shape gets the mousemove events.
    this.mouseController.mouseDown = true;
    this.mouseController.activeController = this.activeController;
  }

  onMouseUp(evt) {

```

```

super.onMouseUp(evt);
// For handling dragging an anchor but the surface or shape gets the mousemove events.
this.mouseController.mouseDown = false;
this.mouseController.activeController = null;
}

// Ignore mouse Leave events when dragging an anchor.
onMouseLeave(evt) { }
}

```

- As a side note, this is getting annoying to have to implement and is indicative of a potential design flaw.

The real fun part is how the anchor controller, anchor shape, and drag event handlers are set up when the anchors are drawn. This is a change to the first rendition described earlier of the `showAnchors` function:

[Hide](#) [Copy Code](#)

```

showAnchors(anchors) {
  // not showing?
  if (this.anchors.length == 0) {
    var anchorGroup = getElement(ANCHORS_ID);
    // Reset any translation because the next mouse hover will set the anchors directly over the shape.
    anchorGroup.setAttributeNS(null, "transform", "translate(0, 0)");
    // We pass in the shape (which is also the surface) mouse controller so we can
    // handle when the shape or surface gets the mousemove event, which happens if
    // the user moves the mouse too quickly and the pointer leaves the anchor rectangle.
    this.anchorController = new AnchorController(this);

    anchors.map(anchorDefinition => {
      var anchor = anchorDefinition.anchor;
      // Note the additional translation attributes tx and ty which we use for convenience (so we don't
      have to parse the transform) when translating the anchor.
      var el = this.createElement("rect", {
        x: anchor.X - 5, y: anchor.Y - 5, tx: 0, ty: 0, width: 10, height: 10,
        fill: "#FFFFFF", stroke: "#808080", "stroke-width": 0.5 });
      // Create anchor shape, wire up anchor events, and attach it to the MouseController::AnchorController
      object.
      new Anchor(this.anchorController, el, this.partialCall(el, anchorDefinition.onDrag));
      anchorGroup.appendChild(el);
    });
  }
}

```

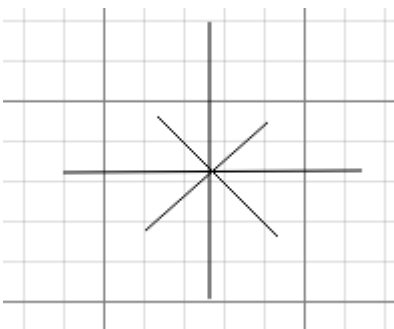
Notice the additional `tx` and `ty` attributes which are used to keep track of the anchor translation. The `partialCall` function lets us pass in the anchor element as part of the `onDrag` callback:

[Hide](#) [Copy Code](#)

```

// We need to set up a partial call so that we can include the anchor being dragged when we call
// the drag method for moving the shape's anchor. At that point we also pass in the event data.
partialCall(anchorElement, onDrag) {
  return (function (anchorElement, onDrag) {
    return function (evt) { onDrag(anchorElement, evt); }
  })(anchorElement, onDrag);
}

```



Anchor Drag Operations for Other Shapes

Circles and other shapes that need to maintain their aspect ratio are annoying because all the anchor points have to move! We have this issue with other shapes as well (such as resizing a rectangle or diamond), as sizing a shape changes other anchor locations. So another refactoring (not shown) passes in the entire anchors collection so shapes can translate the other anchors when one particular anchor is being dragged. We wouldn't have this problem if we removed all anchors but the anchor being moved, which is another possibility, but I don't like to rely on a UI behavior to control the internal logic of how objects are manipulated. So another refactoring to how anchors are created:

Hide Shrink ▲ Copy Code

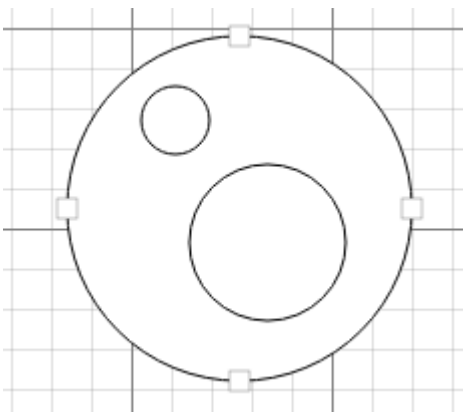
```
// We need to set up a partial call so that we can include the anchor being dragged when we call
// the drag method for moving the shape's anchor. At that point we also pass in the event data.
partialCall(anchors, anchorElement, onDrag) {
  return (function (anchors, anchorElement, onDrag) {
    return function (evt) { onDrag(anchors, anchorElement, evt); }
  })(anchors, anchorElement, onDrag);
}

showAnchors(anchors) {
  // not showing?
  if (this.anchors.length == 0) {
    var anchorGroup = getElement(ANCHORS_ID);
    // Reset any translation because the next mouse hover will set the anchors directly over the shape.
    anchorGroup.setAttributeNS(null, "transform", "translate(0, 0)");
    // We pass in the shape (which is also the surface) mouse controller so we can
    // handle when the shape or surface gets the mousemove event, which happens if
    // the user moves the mouse too quickly and the pointer leaves the anchor rectangle.
    this.anchorController = new AnchorController(this);
    var anchorElements = [];

    anchors.map(anchorDefinition => {
      var anchor = anchorDefinition.anchor;
      // Note the additional translation attributes tx and ty which we use for convenience
      // (so we don't have to parse the transform) when translating the anchor.
      var el = this.createElement("rect",
        { x: anchor.X - 5, y: anchor.Y - 5, tx: 0, ty: 0, width: 10, height: 10,
          fill: "#FFFFFF", stroke: "#808080", "stroke-width": 0.5 });
      anchorElements.push(el);
      anchorGroup.appendChild(el);
    });

    // Separate iterator so we can pass in all the anchor elements to the onDrag callback once they've been
    // accumulated.
    for (var i = 0; i < anchors.length; i++) {
      var anchorDefinition = anchors[i];
      var el = anchorElements[i];
      // Create anchor shape, wire up anchor events, and attach it to the MouseController::AnchorController
      // object.
      new Anchor(this.anchorController, el, this.partialCall(anchorElements, el, anchorDefinition.onDrag));
    }
  }
}
```

Circles



The circle anchor points are the top, bottom, middle and right:

[Hide](#) [Copy Code](#)

```
getAnchors() {
  var corners = this.getCorners();
  var middleTop = new Point((corners[0].X + corners[1].X) / 2, corners[0].Y);
  var middleBottom = new Point((corners[0].X + corners[1].X) / 2, corners[1].Y);
  var middleLeft = new Point(corners[0].X, (corners[0].Y + corners[1].Y) / 2);
  var middleRight = new Point(corners[1].X, (corners[0].Y + corners[1].Y) / 2);

  var anchors = [
    { anchor: middleTop, onDrag: this.topMove.bind(this) },
    { anchor: middleBottom, onDrag: this.bottomMove.bind(this) },
    { anchor: middleLeft, onDrag: this.leftMove.bind(this) },
    { anchor: middleRight, onDrag: this.rightMove.bind(this) }
  ];

  return anchors;
}
```

The adjustments to the circle radius and anchor points:

[Hide](#) [Shrink ▲](#) [Copy Code](#)

```
topMove(anchors, anchor, evt) {
  this.changeRadius(-evt.movementY);
  this.moveAnchor(anchors[0], 0, evt.movementY);
  this.moveAnchor(anchors[1], 0, -evt.movementY);
  this.moveAnchor(anchors[2], evt.movementY, 0);
  this.moveAnchor(anchors[3], -evt.movementY, 0);
}

bottomMove(anchors, anchor, evt) {
  this.changeRadius(evt.movementY);
  this.moveAnchor(anchors[0], 0, -evt.movementY);
  this.moveAnchor(anchors[1], 0, evt.movementY);
  this.moveAnchor(anchors[2], -evt.movementY, 0);
  this.moveAnchor(anchors[3], evt.movementY, 0);
}

leftMove(anchors, anchor, evt) {
  this.changeRadius(-evt.movementX);
  this.moveAnchor(anchors[0], 0, evt.movementX);
  this.moveAnchor(anchors[1], 0, -evt.movementX);
  this.moveAnchor(anchors[2], evt.movementX, 0);
  this.moveAnchor(anchors[3], -evt.movementX, 0);
}

rightMove(anchors, anchor, evt) {
  this.changeRadius(evt.movementX);
  this.moveAnchor(anchors[0], 0, -evt.movementX);
  this.moveAnchor(anchors[1], 0, evt.movementX);
  this.moveAnchor(anchors[2], -evt.movementX, 0);
}
```

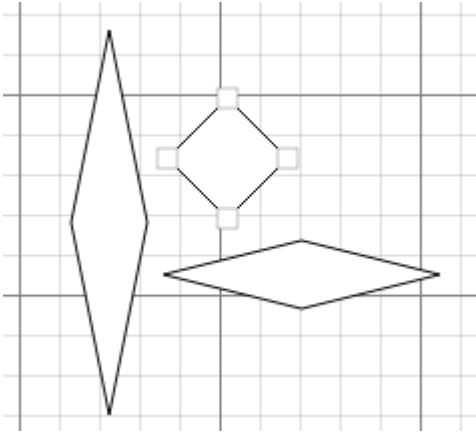
```

    this.moveAnchor(anchors[3], evt.movementX, 0);
}

changeRadius(amt) {
    var r = +this.svgElement.getAttribute("r") + amt;
    this.svgElement.setAttribute("r", r)
}

```

Diamonds



Diamonds are resized symmetrically top-bottom and left-right. This means that only the positions of the top-bottom or left-right anchors need to be updated when resizing vertically or horizontally, respectively. The most annoying problem is recalculating the path (for example: `d: "M 240 100 L 210 130 L 240 160 L 270 130 Z"`) as this isn't just setting (x, y) coordinates. Given that the bounding rectangle is the absolute coordinates on the SVG surface, we have to remove any translations (shape and surface) when setting the new path values:

[Hide](#) [Copy Code](#)

```

updatePath(ulCorner, lrCorner) {
    // example path: d: "M 240 100 L 210 130 L 240 160 L 270 130 Z"
    this.getRelativeLocation(ulCorner);
    this.getRelativeLocation(lrCorner);
    var mx = (ulCorner.X + lrCorner.X) / 2;
    var my = (ulCorner.Y + lrCorner.Y) / 2;
    var path = "M " + mx + " " + ulCorner.Y;
    path = path + " L " + ulCorner.X + " " + my;
    path = path + " L " + mx + " " + lrCorner.Y;
    path = path + " L " + lrCorner.X + " " + my;
    path = path + " Z";
    this.svgElement.setAttribute("d", path);
}

```

and in the `SvgElement` class:

[Hide](#) [Copy Code](#)

```

getRelativeLocation(p) {
    p.translate(-this.X, -this.Y);
    p.translate(-this.mouseController.surface.X, -this.mouseController.surface.Y);

    return p;
}

```

On a "move anchor" event, here's two of the four functions (the other two are identical except the signs are reversed):

[Hide](#) [Copy Code](#)

```

topMove(anchors, anchor, evt) {
    var ulCorner = this.getULCorner();
    var lrCorner = this.getLRCorner();
    this.changeHeight(ulCorner, lrCorner, -evt.movementY);
}

```



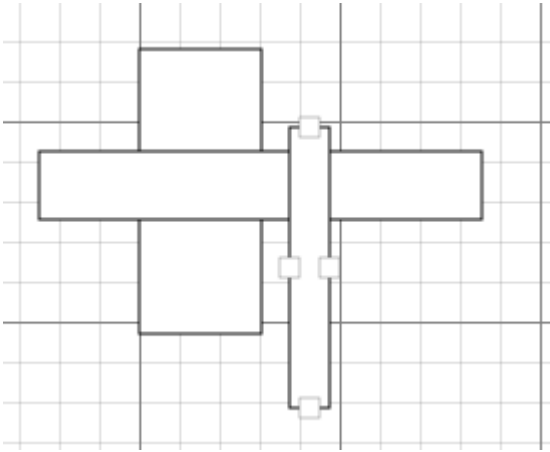
```

    this.moveAnchor(anchors[0], 0, evt.movementY); // top
    this.moveAnchor(anchors[1], 0, -evt.movementY); // bottom
}

leftMove(anchors, anchor, evt) {
    var ulCorner = this.getULCorner();
    var lrCorner = this.getLRCorner();
    this.changeWidth(ulCorner, lrCorner, -evt.movementX);
    this.moveAnchor(anchors[2], evt.movementX, 0);
    this.moveAnchor(anchors[3], -evt.movementX, 0);
}

```

Rectangles



For simplicity, we'll just use create the four anchors we've been using for circles and diamonds. Unlike diamonds, moving an anchor is *not* symmetrical, so in addition to the anchor itself, the diagonal anchors have to be updated as well. The only nuance here is in manipulating the (x, width) and (y, height) values. Again, illustrating only the code for top and left anchor moves (right and bottom sign changes and only adjust width and height):

[Hide](#) [Copy Code](#)

```

topMove(anchors, anchor, evt) {
    // Moving the top affects "y" and "height"
    var y = +this.svgElement.getAttribute("y") + evt.movementY;
    var height = +this.svgElement.getAttribute("height") - evt.movementY;
    this.svgElement.setAttribute("y", y);
    this.svgElement.setAttribute("height", height);
    this.moveAnchor(anchors[0], 0, evt.movementY);
    this.adjustAnchorY(anchors[2], evt.movementY/2);
    this.adjustAnchorY(anchors[3], evt.movementY / 2);
}

leftMove(anchors, anchor, evt) {
    // Moving the left affects "x" and "width"
    var x = +this.svgElement.getAttribute("x") + evt.movementX;
    var width = +this.svgElement.getAttribute("width") - evt.movementX;
    this.svgElement.setAttribute("x", x);
    this.svgElement.setAttribute("width", width);
    this.moveAnchor(anchors[2], evt.movementX, 0);
    this.adjustAnchorX(anchors[0], evt.movementX / 2);
    this.adjustAnchorX(anchors[1], evt.movementX / 2);
}

```

And there are a couple new helper functions in the SvgElement class:

[Hide](#) [Copy Code](#)

```

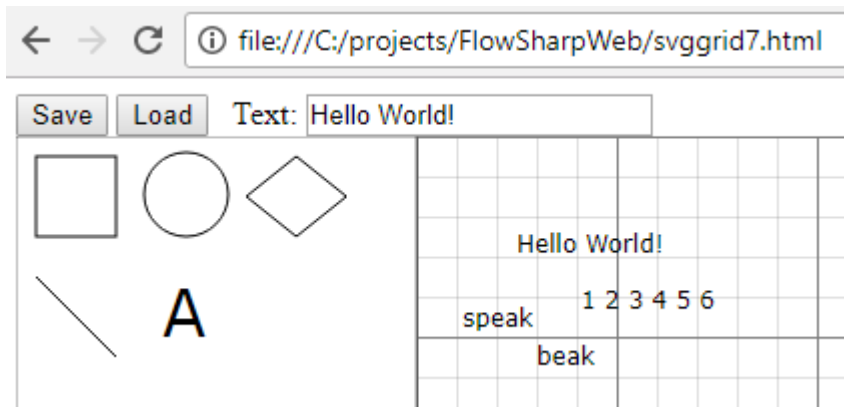
adjustAnchorX(anchor, dx) {
  var tx = +anchor.getAttribute("tx") + dx;
  var ty = +anchor.getAttribute("ty");
  anchor.setAttribute("transform", "translate(" + tx + "," + ty + ")");
  anchor.setAttribute("tx", tx);
  anchor.setAttribute("ty", ty);
}

adjustAnchorY(anchor, dy) {
  var tx = +anchor.getAttribute("tx");
  var ty = +anchor.getAttribute("ty") + dy;
  anchor.setAttribute("transform", "translate(" + tx + "," + ty + ")");
  anchor.setAttribute("tx", tx);
  anchor.setAttribute("ty", ty);
}

```

We can now resize and re-orient lines! One annoyance with the current implementation is that the anchor only appears when the mouse enters the shape. This leads to a minor but awkward mouse gesturing where the mouse has to be moved *into* the shape and then back *out* to the edge to select the anchor. This issue exists with connection points as well. One way to deal with this is that each shape needs to be in a group with a transparent but slightly larger mirror shape. It's on the todo list.

Text



One last thing in this article to make this application at least minimally useful - the ability to add some text to the diagram. At the moment, there's no fancy font, font size, alignment, or word wrapping features. Also, text is an independent shape -- if you overlay text on a rectangle, the text will not move when you move the rectangle. This is bare-bones functionality!

Adding the toolbox text shape to the toolbox group:

[Hide](#) [Copy Code](#)

```
<text id="toolboxText" x="73" y="100" font-size="32" font-family="Verdana">A</text>
```

There is also a supporting `Text` and `ToolboxClass` with typical implementation with one minor variation -- setting the inner HTML:

[Hide](#) [Copy Code](#)

```

createElement() {
  var el = super.createElement('text', { x: 240, y: 100, "font-size": 12, "font-family": "Verdana" });
  el.innerHTML = "[text]";

  return el;
}

```

Same with the `createElementAt` function.

I also don't want the cursor to change to an I-beam when the mouse moves over a text element, so our first (and only) CSS:

[Hide](#) [Copy Code](#)

```
<style>
  text {cursor:default}
</style>
```

Changing Text

As I mentioned earlier, I'm less interested in a fancy UI at the moment and more interested in getting the basic behaviors ironed out. So to set the text, one selects the shape and then enters the text in the edit box at the top of the diagram as shown in the screenshot at the start of this section. The implementation, in the `Text` class, is trivial:

[Hide](#) [Copy Code](#)

```
class Text extends SvgElement {
  constructor(mouseController, svgElement) {
    super(mouseController, svgElement);
    this.registerEventListener(svgElement, "mousedown", this.onMouseDown);
  }

  // Update the UI with the text associated with the shape.
  onMouseDown(evt) {
    var text = this.svgElement.innerHTML;
    document.getElementById("text").value = text;
  }

  setText(text) {
    this.svgElement.innerHTML = text;
  }
}
```

The only interesting thing to note here is that the `Text` class adds a second `mousedown` event handler so that it can set the text of text shape into the input box on the screen. When the text is changed in the input box, the selected shape's `setText` method is called:

[Hide](#) [Copy Code](#)

```
// Update the selected shape's text. Works only with text shapes right now.
function setText() {
  if (mouseController.selectedShape != null) {
    var text = document.getElementById("text").value;
    mouseController.selectedShape.setText(text);
  }
}
```

It's a bit kludgy, using the global `mouseController` and so forth, but we can expand upon this later to have all shapes include a text area.

Refactoring the Prototype to use an MVC Pattern

So far we've avoided having to maintain and persist a separate model. The shape classes `Rectangle`, `Text`, `Circle`, and so forth are more controllers than models, though there is a bit of entanglement evident in the `createShapeControllers` function, which is called when loading a diagram. This fragment:

[Hide](#) [Copy Code](#)

```
shape.X = +translate[0];
shape.Y = +translate[1];
```

is clue that we're entangling controller and model. Similarly, in the `Surface` class, the `serialize` / `deserialize` functions are another clue that controller and model are being entangled. The code for serializing the surface data is itself rather hacky. Reviewing it:

[Hide](#) [Copy Code](#)

```

serialize() {
  var el = document.createElement("surface");
  // DOM adds elements as lowercase, so let's just start with lowercase keys.
  var attributes = {x : this.X, y : this.Y, gridcellw : this.gridCellW, gridcellh : this.gridCellH, cellw :
this.cellW, cellh : this.cellH}
  Object.entries(attributes).map(([key, val]) => el.setAttribute(key, val));
  var serializer = new XMLSerializer();
  var xml = serializer.serializeToString(el);

  return xml;
}

```

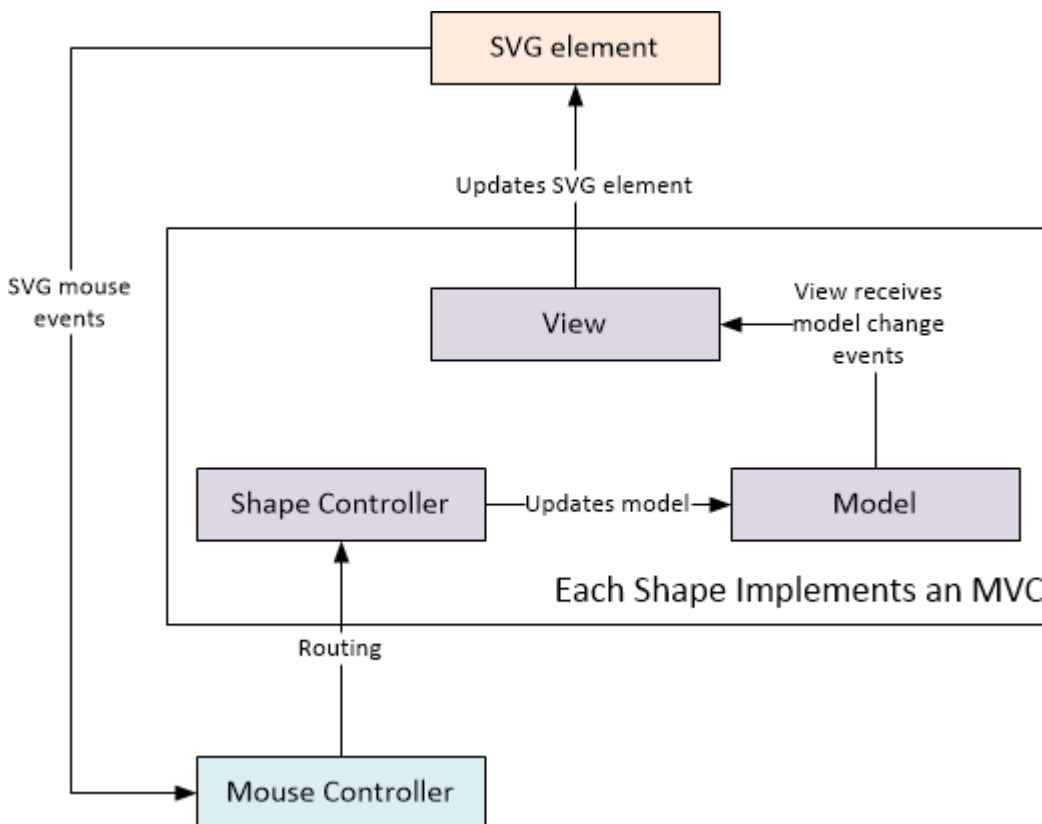
Besides the kludge of dealing with lowercase attributes, we also have the hack of serializing to XML to stay consistent with the actual SVG object graph serialization, which is also in XML. There are some options that should be considered.

- Serialize only the model and reconstruct the SVG object graph from the model. The drawback here is that the SVG, serialized as XML, is easily imported into some other application. Of course, we could add an "export" function if we want this behavior.
- Serialize the SVG object graph as XML and the shape models as JSON, which is more Javascript native. We either entangle XML and JSON in the same file or save them as separate files.

At the end of the day, I think the preferred approach is to serialize the *model* and reconstruct the SVG object graph *from the model*.

Models, Views, and Controllers

Here's the basic MVC pattern:



Each shape implements its own specific model, view, and controller. For example (because it's interesting) here are the MVC classes for the text shape:

The Text Model

[Hide](#) [Copy Code](#)

```

class TextModel extends ShapeModel {
  constructor() {

```

```

    super();
    this._x = 0;
    this._y = 0;
    this._text = "";
}

get x() { return this._x; }
get y() { return this._y; }
get text() { return this._text; }

set x(value) {
    this._x = value;
    this.propertyChanged("x", value);
}

set y(value) {
    this._y = value;
    this.propertyChanged("y", value);
}

set text(value) {
    this._text = value;
    this.propertyChanged("text", value);
}
}

```

The Text Controller

[Hide](#) [Copy Code](#)

```

class TextController extends Controller {
    constructor(mouseController, view, model) {
        super(mouseController, view, model);
    }

    // Update the UI with the text associated with the shape.
    onMouseDown(evt) {
        super.onMouseDown(evt);
        var text = this.model.text;
        document.getElementById("text").value = text;
        this.mouseController.selectedShapeController = this;
    }
}

```

The Text View

[Hide](#) [Copy Code](#)

```

class TextView extends View{
    constructor(svgElement, model) {
        super(svgElement, model);
    }

    // Custom handling for property "text"
    onPropertyChange(property, value) {
        if (property == "text") {
            this.svgElement.innerHTML = value;
        } else {
            super.onPropertyChange(property, value);
        }
    }
}

```

The Base Model

Every shape requires translation, so the base `Model` class handles these properties and provides some helper methods:

Hide Shrink ▲ Copy Code

```
class Model {
  constructor() {
    this.eventPropertyChanged = null;

    this._tx = 0;
    this._ty = 0;
  }

  get tx() { return this._tx; }
  get ty() { return this._ty; }

  propertyChanged(propertyName, value) {
    if (this.eventPropertyChanged != null) {
      this.eventPropertyChanged(propertyName, value);
    }
  }

  translate(x, y) {
    this._tx += x;
    this._ty += y;
    this.setTranslate(this._tx, this._ty);
  }

  // Update our internal translation and set the translation immediately.
  setTranslation(x, y) {
    this._tx = x;
    this._ty = y;
    this.setTranslate(x, y);
  }

  // Deferred translation -- this only updates _tx and _ty
  // Used when we want to internally maintain the true _tx and _ty
  // but set the translation to a modulus, as in when translating
  // the grid.
  updateTranslation(dx, dy) {
    this._tx += dx;
    this._ty += dy;
  }

  // Sets the "translate" portion of the "transform" property.
  // ALL models have a translation. Notice we do not use _tx, _ty here
  // nor do we set _tx, _ty to (x, y) because (x, y) might be mod'ed by
  // the grid (w, h). We want to use exactly the parameters passed in
  // without modifying our model.
  // See SurfaceController.onDrag and note how the translation is updated
  // but setTranslate is called with the mod'ed (x, y) coordinates.
  setTranslate(x, y) {
    this.translation = "translate(" + x + "," + y + ")";
    this.transform = this.translation;
  }

  // TODO: Later to be extended to build the transform so that it includes rotation and other things we can
  // do.
  set transform(value) {
    this._transform = value;
    this.propertyChanged("transform", value);
  }

  set tx(value) {
    this._tx = value;
    this.translation = "translate(" + this._tx + "," + this._ty + ")";
    this.transform = this.translation;
  }

  set ty(value) {
    this._ty = value;
  }
}
```

```

    this.translation = "translate(" + this._tx + "," + this._ty + ")";
    this.transform = this.translation;
  }
}

```

The Base View

The base **View** class has a helper function for acquiring the ID of the SVG element and sets the attribute of the associated SVG element:

[Hide](#) [Copy Code](#)

```

class View {
  constructor(svgElement, model) {
    this.svgElement = svgElement;
    model.eventPropertyChange = this.onPropertyChange.bind(this);
  }

  get id() {
    return this.svgElement.getAttribute("id");
  }

  onPropertyChange(property, value) {
    this.svgElement.setAttribute(property, value);
  }
}

```

Also notice that the constructor wires up the property changed "event" that the model fires.

Creating a Shape Programmatically

Given the new MVC architecture, here's how a shape is created programmatically. Notice that the model must be initialized to match the shape attribute values. Also notice that right now, our model doesn't handle other attributes such as fill, stroke, and stroke-width. We don't have UI support for that yet, so I haven't implemented those properties of the model.

[Hide](#) [Copy Code](#)

```

var rectEl = Helpers.createElement('rect', { x: 240, y: 100, width: 60, height: 60, fill: "#FFFFFF",
stroke: "black", "stroke-width": 1 });
var rectModel = new RectangleModel();
rectModel._x = 240;
rectModel._y = 100;
rectModel._width = 60;
rectModel._height = 60;
var rectView = new ShapeView(rectEl, rectModel);
var rectController = new RectangleController(mouseController, rectView, rectModel);
Helpers.getElement(Constants.SVG_OBJECTS_ID).appendChild(rectEl);
mouseController.attach(rectView, rectController);
// Most shapes also need an anchor controller
mouseController.attach(rectView, anchorGroupController);

```

Also notice how the mouse controller now supports multiple shape controllers!

Serialization

It's probably worth taking a quick look at how serialization works now:

[Hide](#) [Copy Code](#)

```

// Returns JSON of serialized models.
serialize() {
  var uberModel = [];
  var model = surfaceModel.serialize();
  model[Object.keys(model)[0]].id = Constants.SVG_SURFACE_ID;
  uberModel.push(model);
}

```

```

this.models.map(m => {
    var model = m.model.serialize();
    model[Object.keys(model)[0]].id = m.id;
    uberModel.push(model);
});

return JSON.stringify(uberModel);
}

```

The concrete model is responsible for serializing itself. The serializer tacks on the shape's ID which is actually not part of the model, it's part of the view! This code looks a bit weird because when a shape is dropped on to the surface, only the model and the shape's ID is registered in the diagram controller like this:

Hide Copy Code

```

addModel(model, id) {
    this.models.push({ model: model, id: id });
}

```

Hence this abomination of code: `model[Object.keys(model)[0]].id = m.id;`

The model dictionary has only one entry, the key is the shape name, and the value is the collection of attributes, to which we're adding "id". For example, a blank surface serializes like this:

Hide Copy Code

```

[{"Surface":{"tx":0,"ty":0,"gridCellW":100,"gridCellH":100,"cellW":20,"cellH":20,"id":"surface"}}]

```

Deserializing

Restoring the diagram is a bit more complicated because the appropriate model, view, and controller classes must be created as well as the SVG element. Deserializing the actual SVG element attributes is again left to the concrete model.

Hide Shrink ▲ Copy Code

```

// Creates an MVC for each model of the provided JSON.
deserialize(jsonString) {
    var models = JSON.parse(jsonString);
    var objectModels = [];
    surfaceModel.setTranslation(0, 0);
    objectsModel.setTranslation(0, 0);

    models.map(model => {
        var key = Object.keys(model)[0];
        var val = model[key];

        if (key == "Surface") {
            // Special handler for surface, we keep the existing MVC objects.
            // We set both the surface and objects translation, but the surface translation
            // is mod'd by the gridCellW/H.
            surfaceModel.deserialize(val);
            objectsModel.setTranslation(surfaceModel.tx, surfaceModel.ty);
        } else {
            var model = new this.mvc[key].model();
            objectModels.push(model);
            var el = this.mvc[key].creator();
            // Create the view first so it hooks into the model's property change event.
            var view = new this.mvc[key].view(el, model);
            model.deserialize(val, el);
            view.id = val.id;
            var controller = new this.mvc[key].controller(mouseController, view, model);

            // Update our diagram's model collection.
            this.models.push({ model: model, id: val.id });

            Helpers.getElement(Constants.SVG_OBJECTS_ID).appendChild(el);
        }
    });
}

```



```

    this.mouseController.attach(view, controller);

    // Most shapes also need an anchor controller. An exception is the Text shape, at least for now.
    if (controller.shouldShowAnchors) {
        this.mouseController.attach(view, anchorGroupController);
    }
}
});
}

```

The whole process is driven by a table that determines what actual MVC classes to instantiate, as well as any custom SVG element instantiation requirements:

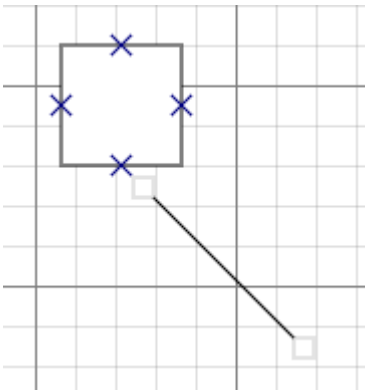
[Hide](#) [Copy Code](#)

```

this.mvc = {
  Rectangle: { model: RectangleModel, view: ShapeView, controller: RectangleController, creator : () =>
this.createElement("rect") },
  Circle: { model: CircleModel, view: ShapeView, controller: CircleController, creator: () =>
this.createElement("circle") },
  Diamond: { model: DiamondModel, view: ShapeView, controller: DiamondController, creator: () =>
this.createElement("path") },
  Line: { model: LineModel, view: LineView, controller: LineController, creator: () =>
this.createLineElement() },
  Text: { model: TextModel, view: TextView, controller: TextController, creator: () =>
this.createTextElement() },
};

```

Connecting Lines



Now that we have a solid MVC architecture, the additional glue required to manage connected lines (lines endpoints connected to shapes) can be accomplished. This involves:

- Connection points on the shape.
- Showing those connection points.
- Managing what line endpoint is connected to what shape's connection point.
 - Attaching endpoints
 - Detaching endpoints
 - Persisting connections
- Moving the lines when the shape moves.
- Moving the lines when the shape size changes.

That's a lot! Hopefully we can be efficient about how we implement all this.

Connection Points

For the moment, I'm going to keep connection points simple, meaning that for circles and diamonds, there's no 45 / 135 / 225 / 315 degree points (or even others.) For rectangles, there's no intermediate connection points between the edge and the midpoint. Connection points, like anchors, will for now be the cardinal compass points: N, S, E, W. However, we'll implement the structure so that it can be extended later on. As such, defining connection points looks very similar to defining anchors, except that there's no behavior associated with a connection point, it's just a visual aid. In the full blown implementation, connection points can be added, removed, moved around, and so forth, so what is important is that we have some way to associate a name (even if it's an auto-generated GUID) to a connection point so there is something concrete to use as a reference between the line endpoint and the shape connection point rather than just an index into an array.

As with anchors, we have a function that returns the connection points available to the shape. Custom connection points are currently not supported. I also compromise on the implementation in that instead of giving a connection point an actual ID, I'm "remembering" connection points by their index in the list of connection points. This is not ideal for the future where the user should be able to add/remove connection points on the shape. That said, here's how the connection points for the rectangle shape are defined (it will look very similar to anchors):

[Hide](#) [Copy Code](#)

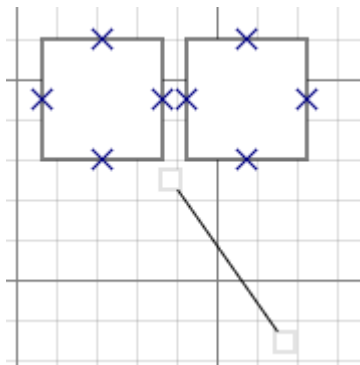
```
getConnectionPoints() {
  var corners = this.getCorners();
  var middleTop = new Point((corners[0].x + corners[1].x) / 2, corners[0].y);
  var middleBottom = new Point((corners[0].x + corners[1].x) / 2, corners[1].y);
  var middleLeft = new Point(corners[0].x, (corners[0].y + corners[1].y) / 2);
  var middleRight = new Point(corners[1].x, (corners[0].y + corners[1].y) / 2);

  var connectionPoints = [
    { connectionPoint: middleTop },
    { connectionPoint: middleBottom },
    { connectionPoint: middleLeft },
    { connectionPoint: middleRight }
  ];

  return connectionPoints;
}
```

The `connectionPoints` array is a dictionary of just one key-value pair -- this is overkill at the moment, but I suspect that like with anchors, some additional data will at some point be necessary.

Detecting Shapes That We're Near



The logical place to put the code that controls connecting / disconnecting from shapes is the `AnchorController`, which is created for each anchor when the mouse hovers over a shape. Also, only certain shapes (like lines) can connect to other shapes. So in the `AnchorController`, the `onDrag` function gets to also handle showing connection points of nearby shapes:

[Hide](#) [Copy Code](#)

```
onDrag(dx, dy) {
  // Call into the shape controller to handle
  // the specific anchor drag.
  this.fncDragAnchor(dx, dy);
  this.showAnyConnectionPoints();
}
```

Where the function `showAnyConnectionPoints` manages both the list of shapes currently displaying connection points as well as calling functions to show or remove the connection points of nearby shapes:

Hide Copy Code

```
showAnyConnectionPoints() {
  if (this.shapeController.canConnectToShapes) {
    var changes = this.getNewNearbyShapes(this.mouseController.x, this.mouseController.y);
    this.createConnectionPoints(changes.newShapes);

    // Other interesting approaches:
    // https://stackoverflow.com/questions/1885557/simplest-code-for-array-intersection-in-javascript
    // [...new Set(a)].filter(x => new Set(b).has(x));
    var currentShapesId = changes.newShapes.concat(changes.existingShapes).map(ns => ns.id);

    var noLongerNearShapes = this.shapeConnectionPoints.filter(s => currentShapesId.indexOf(s.id) < 0);
    this.removeExpiredShapeConnectionPoints(noLongerNearShapes);

    // Remove any shapes from the shapeConnectionPoints that do not exist anymore.
    var existingShapesId = changes.existingShapes.map(ns => ns.id);
    this.shapeConnectionPoints = this.shapeConnectionPoints.filter(s => existingShapesId.indexOf(s.id) >= 0);

    // Add in the new shapes.
    this.shapeConnectionPoints = this.shapeConnectionPoints.concat(changes.newShapes);

    console.log("scp: " + this.shapeConnectionPoints.length + ", new: " + changes.newShapes.length + ", existing: " + existingShapesId.length);
  }
}
```

This is really just a bunch of map and filter calls to add new shapes to the current shape connection points and remove old shapes that no longer should be showing connection points.

With `getNewNearbyShapes`, it's useful to return both new shapes to which we're near and any existing shapes to which we're still near:

Hide Copy Code

```
getNewNearbyShapes(x, y) {
  var newShapes = [];
  var existingShapes = [];
  var p = new Point(x, y);
  p = Helpers.translateToScreenCoordinate(p);
  var nearbyShapeEls = Helpers.getNearbyShapes(p);

  // Logging:
  // nearbyShapesEls.map(s => console.log(s.outerHTML.split(" ")[0].substring(1)));

  nearbyShapeEls.map(el => {
    var controllers = this.mouseController.getControllersByElement(el);
```

Hide Copy Code

```
    if (controllers) {
      controllers.map(ctrl => {
        if (ctrl.hasConnectionPoints) {
          var shapeId = ctrl.view.id;

          // If it already exists in the list, don't add it again.
          if (!this.shapeConnectionPoints.any(cp => cp.id == shapeId)) {
            var connectionPoints = ctrl.getConnectionPoints();
            newShapes.push({ id: shapeId, controller: ctrl, connectionPoints: connectionPoints });
          } else {
            existingShapes.push({ id: shapeId });
          }
        }
      });
    }
  });
}
```

```
});

return { newShapes : newShapes, existingShapes: existingShapes };
}
```

The salient part of this is that new shapes consist of the structure `{shape ID, controller, connection points}` and existing shapes is just the structure `{shape ID}`. In the previous function, these two lists are concatenated and the common shape ID is mapped into the collection of shapes currently showing connection points:

Hide Copy Code

```
var currentShapesId = changes.newShapes.concat(changes.existingShapes).map(ns => ns.id);
```

Showing Connection Points

Hide Copy Code

```
// "shapes" is a {id, controller, connectionPoints} structure
createConnectionPoints(shapes) {
  var cpGroup = Helpers.getElement(Constants.SVG_CONNECTION_POINTS_ID);

  shapes.map(shape => {
    shape.connectionPoints.map(cpStruct => {
      var cp = cpStruct.connectionPoint;
      var el = Helpers.createElement("g", { connectingToShapeId: shape.id });
      el.appendChild(Helpers.createElement("line",
        { x1: cp.x - 5, y1: cp.y - 5, x2: cp.x + 5, y2: cp.y + 5, fill: "#FFFFFF", stroke: "#000080",
"stroke-width": 1 }));
      el.appendChild(Helpers.createElement("line",
        { x1: cp.x + 5, y1: cp.y - 5, x2: cp.x - 5, y2: cp.y + 5, fill: "#FFFFFF", stroke: "#000080",
"stroke-width": 1 }));
      cpGroup.appendChild(el);
    });
  });
}
```

Any shapes for which we want to show connection points adds a group with two lines that form an X to the connection points group. Notice the attribute `connectingToShapeId` that sets the shape ID for the associated shape. We use this information next to remove connection points for a particular shape.

Removing Connection Points

Hide Copy Code

```
// "shapes" is a {id, controller, connectionPoints} structure
removeExpiredShapeConnectionPoints(shapes) {
  shapes.map(shape => {
    // <a
    href="https://stackoverflow.com/a/16775485/2276361">https://stackoverflow.com/a/16775485/2276361</a>
    var nodes = document.querySelectorAll('[connectingtoshapeid="' + shape.id + '"');
    // or: Array.from(nodes); <a
    href="https://stackoverflow.com/a/36249012/2276361">https://stackoverflow.com/a/36249012/2276361</a>
    // <a
    href="https://stackoverflow.com/a/33822526/2276361">https://stackoverflow.com/a/33822526/2276361</a>
    [...nodes].map(node => { node.parentNode.removeChild(node) });
  });
}
```

Removing connection points involves a document query to get all the connection point SVG groups associated with the shape and removing the child node.

Connecting to a Shape

Connecting to a shape involves finding the connection point (assuming we find only one) that the mouse is closest too and snapping the anchor point of the line to the connection point of the shape. We also tell the diagram model about the new connection. Here we see how the connection point index is used to track the actual connection point on the shape.

Hide Shrink ▲ Copy Code

```
connectIfCloseToShapeConnectionPoint() {
  var p = new Point(this.mouseController.x, this.mouseController.y);
  p = Helpers.translateToScreenCoordinate(p);

  var nearbyConnectionPoints = [];

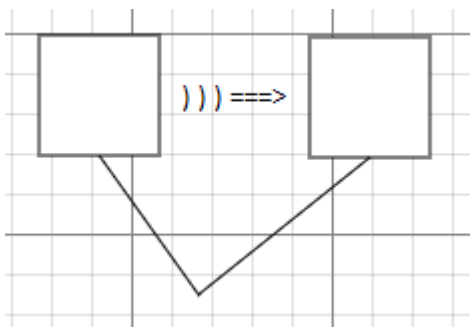
  this.shapeConnectionPoints.filter(scp => {
    for (var i = 0; i < scp.connectionPoints.length; i++) {
      var cpStruct = scp.connectionPoints[i];
      if (Helpers.isNear(cpStruct.connectionPoint, p, Constants.MAX_CP_NEAR)) {
        nearbyConnectionPoints.push({ shapeController: scp.controller, shapeCPIIdx : i, connectionPoint :
cpStruct.connectionPoint});
      }
    }
  });

  if (nearbyConnectionPoints.length == 1) {
    var ncp = nearbyConnectionPoints[0];

    // The location of the connection point of the shape to which we're connecting.
    var p = ncp.connectionPoint;
    // Physical location of endpoint is without line and surface translations.
    p = p.translate(-this.shapeController.model.tx, -this.shapeController.model.ty);
    p = p.translate(-surfaceModel.tx, - surfaceModel.ty);
    // Move the endpoint of the shape from which we're connecting (the line) to this point.
    this.shapeController.connect(this.anchorIdx, p);
    diagramModel.connect(ncp.shapeController.view.id, this.shapeController.view.id, ncp.shapeCPIIdx,
this.anchorIdx);
  }
}
```

A drawback with this approach is that it only works when dragging the endpoint anchor. If you're dragging the line, we're not detecting whether an endpoint is approaching another shape's connection point.

Updating Connections when the Shape is Moved



This was a simple addition to the **Controller** class **onDrag** function:

Hide Copy Code

```
// Default behavior
onDrag(dx, dy)
{
  this.model.translate(dx, dy);
  this.adjustConnections(dx, dy);
}

// Adjust all connectors connecting to this shape.
adjustConnections(dx, dy) {
```

```

var connections = diagramModel.connections.filter(c => c.shapeId == this.view.id);
connections.map(c => {
  // TODO: Sort of nasty assumption here that the first controller is the line controller
  var lineController = this.mouseController.getControllersById(c.lineId)[0];
  lineController.translateEndpoint(c.lineAnchorIdx, dx, dy);
});
}

```

Notice how `translateEndpoint` relies on the anchor index -- again, not ideal but quite sufficient for the current implementation:

[Hide](#) [Copy Code](#)

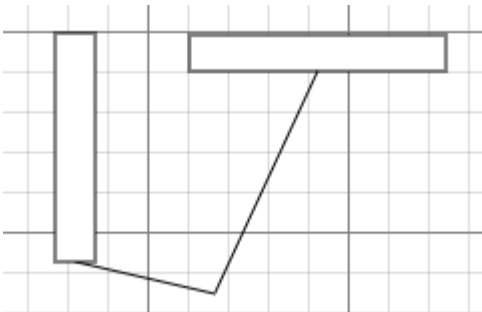
```

translateEndpoint(idx, dx, dy) {
  switch (idx) {
    case 0:
      var p = new Point(this.model.x1, this.model.y1);
      p = p.translate(dx, dy);
      this.model.x1 = p.x;
      this.model.y1 = p.y;
      break;
    case 1:
      var p = new Point(this.model.x2, this.model.y2);
      p = p.translate(dx, dy);
      this.model.x2 = p.x;
      this.model.y2 = p.y;
      break;
  }
}

```

Moving a line's endpoint is a simple matter of updating the endpoint based on the movement of the shape.

Updating Connections when the Shape is Resized



The **Controller** class implements the common function to translate line endpoints attached to the shape being resized:

[Hide](#) [Copy Code](#)

```

// Adjust the connectors connecting to this shape's connection point.
adjustConnectorsAttachedToConnectionPoint(dx, dy, cpIdx) {
  var connections = diagramModel.connections.filter(c => c.shapeId == this.view.id && c.shapeCPIIdx == cpIdx);
  connections.map(c => {
    // TODO: Sort of nasty assumption here that the first controller is the line controller
    var lineController = this.mouseController.getControllersById(c.lineId)[0];
    lineController.translateEndpoint(c.lineAnchorIdx, dx, dy);
  });
}

```

When the anchor point (which at the moment always has an associated connection point) is moved, the shape controller itself is responsible for calling the method to adjust any connections to that anchor/connection point. Here's an example of what happens when the top anchor of the rectangle shape is moved:

[Hide](#) [Copy Code](#)

```

topMove(anchors, anchor, dx, dy) {
  // Moving the top affects "y" and "height"
  var y = this.model.y + dy;
  var height = this.model.height - dy;
  this.model.y = y;
  this.model.height = height;
  this.moveAnchor(anchors[0], 0, dy);
  this.adjustAnchorY(anchors[2], dy / 2);
  this.adjustAnchorY(anchors[3], dy / 2);
  this.adjustConnectorsAttachedToConnectionPoint(0, dy, 0);
  this.adjustConnectorsAttachedToConnectionPoint(0, dy / 2, 2);
  this.adjustConnectorsAttachedToConnectionPoint(0, dy / 2, 3);
}

```

Certainly the code can be improved, the use of indices is annoying, and the switching of anchor-dx-dy and dx-dy-anchorIndex parameter order is also annoying. But it illustrates the point that each anchor "drag" function is responsible for figuring out how connection points (which also happen to be the anchor coordinates and in the same order) are adjusted.

Disconnecting Connections

When the entire line is moved, both endpoints are disconnected from any potential connections:

[Hide](#) [Copy Code](#)

```

onDrag(dx, dy) {
  super.onDrag(dx, dy);
  // When the entire line is being dragged, we disconnect any connections.
  diagramModel.disconnect(this.view.id, 0);
  diagramModel.disconnect(this.view.id, 1);
}

```

This is a simple filter operation in the diagram controller:

[Hide](#) [Copy Code](#)

```

// Disconnect any connections associated with the line and anchor index.
disconnect(lineId, lineAnchorIdx) {
  this.connections = this.connections.filter(c => !(c.lineId == lineId && c.lineAnchorIdx ==
lineAnchorIdx));
}

```

Similarly, whenever one of the endpoints of the line is moved, it is disconnected from any shape to which it might be connected. Note the use of indexing the connection point (aka the anchor index):

[Hide](#) [Copy Code](#)

```

// Move the (x1, y1) coordinate.
moveULCorner(anchors, anchor, dx, dy) {
  this.model.x1 = this.model.x1 + dx;
  this.model.y1 = this.model.y1 + dy;
  this.moveAnchor(anchor, dx, dy);
  diagramModel.disconnect(this.view.id, 0);
}

// Move the (x2, y2) coordinate.
moveLRCorner(anchors, anchor, dx, dy) {
  this.model.x2 = this.model.x2 + dx;
  this.model.y2 = this.model.y2 + dy;
  this.moveAnchor(anchor, dx, dy);
  diagramModel.disconnect(this.view.id, 1);
}

```

Removing A Shape

I almost forgot this! Removing a shape is an involved process of:

- detaching the shape from the mouse controller
- unwiring events
- removing anchors (as the shape is currently being hovered over)
- removing it from the model
- disconnecting any connections to the shape.
- removing it from the "objects" collection so it's gone from the diagram

Fortunately, these are mostly one-line calls into various controllers and models:

[Hide](#) [Copy Code](#)

```
...
case Constants.KEY_DELETE:
  // Mouse is "leaving" this control, this removes any anchors.
  this.currentHoverControllers.map(c => c.onMouseLeave());

  // Remove shape from diagram model, and all connections of this shape.
  diagramModel.removeShape(this.hoverShapeId);

  // Remove shape from mouse controller and detach events.
  this.destroyShapeById(this.hoverShapeId);

  // Remove from "objects" collection.
  var el = Helpers.getElement(this.hoverShapeId);
  el.parentNode.removeChild(el);

  // Cleanup.
  this.currentHoverControllers = [];
  this.hoverShapeId = null;
  handled = true;
  break;
...
```

Conclusion

There's so much more to do! Undo/redo, zooming, rotations, a "properties" window for setting colors and stroke widths, font and font sizes, arrow endpoints, smart line connectors, grouping, moving shapes up/down the hierarchy, shape templates, and so forth. To a large degree, these are all bells and whistles (with some complexity, particularly with regards to rotations and connection points) which I'll continue to add. What this article has presented is a good baseline for the core functionality of a web-based diagramming tool. Stay tuned for Part III!

One of the most annoying issues was dealing with mouse events not being received by the intended shape. For example, rapid mouse movement by the user causes the shape being moved to lag and the underlying SVG element to start receiving mouse movement events. Mouse up events exactly on the connection point results in the connection point receiving the event, which is why I moved connection points to be below anchors.

One thing I noticed was that once I implemented a true MVC pattern, a lot of the complexity of managing the mouse state went away. In fact, with the MVC pattern in place, adding the line connections and updating the persistence to include connections was a breeze.

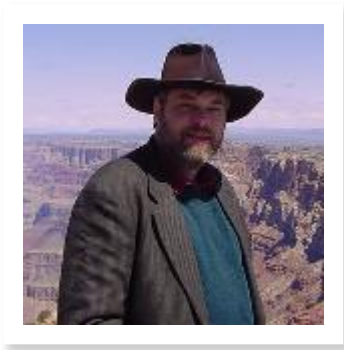
Lastly, this has been an incredibly useful exercise to learn about programmatically controlling SVG as well as learning some new things about Javascript. And while there's a lot more to do, I think I've laid an excellent foundation for continuing to enhance this application.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

Share

About the Author



Marc Clifton

U
n
i
t
e
d
S
t
a
t
e
s



Marc is the creator of two open source projects, [MyXaml](#), a declarative (XML) instantiation engine and [the Advanced Unit Testing framework](#), and [Interacx](#), a commercial n-tier RAD application suite. Visit his website, www.marcclifton.com, where you will find many of his articles and his [blog](#).

Marc lives in Philmont, NY.

You may also be interested in...

[The Offline-First Approach to Mobile App Development](#)

[Building Reactive Apps](#)

[JavaScript Front-End Web App Tutorial Part 1: Building a Minimal App in Seven Steps](#)

[MQTT Publication to Amazon Web Services \(AWS\)](#)

[Implementing a Flowchart with SVG and AngularJS](#)

[Using Intel® Math Kernel Library with Arduino Create](#)

Comments and Discussions

You must [Sign In](#) to use this message board.

**Canvas vs SVG** **Paulus Koshivi** 2hrs 38mins ago**TypeScript** **Paulus Koshivi** 3hrs 24mins ago

Re: TypeScript

Marc Clifton 3hrs 5mins ago

Re: TypeScript

Paulus Koshivi 2hrs 46mins ago

Re: TypeScript



Marc Clifton 2hrs ago

Refresh

1

[General](#) [News](#) [Suggestion](#) [Question](#) [Bug](#) [Answer](#) [Joke](#) [Praise](#) [Rant](#) [Admin](#)

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Terms of Use](#) | [Mobile](#)
Web01-2016 | 2.8.180326.1 | Last Updated 2 Apr 2018 Select Language 
Layout: [fixed](#) | [fluid](#)Article Copyright 2018 by Marc Clifton
Everything else Copyright © [CodeProject](#), 1999-2018