Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 15

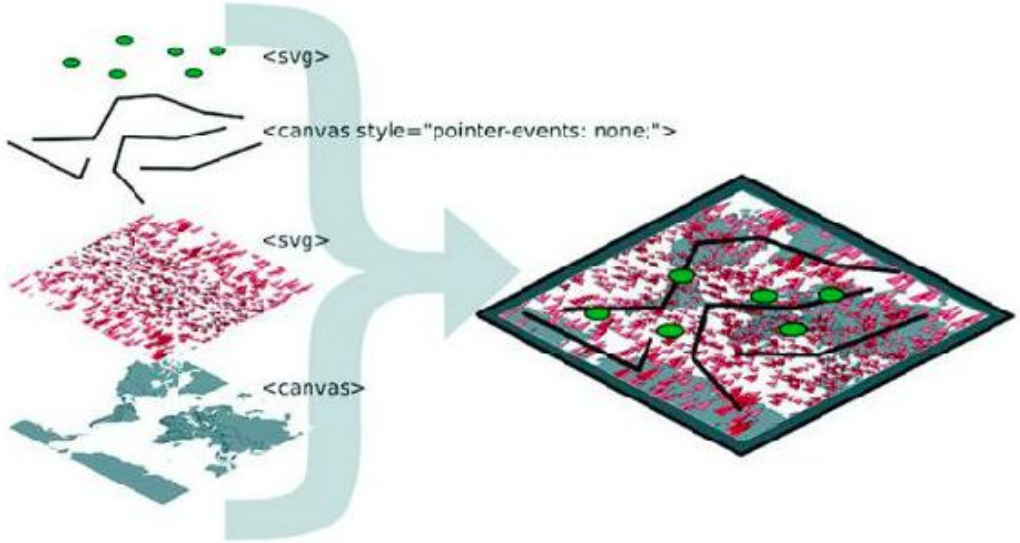
Mixed mode rendering

Overview

- Using built-in canvas rendering for D3 shapes
- Creating large random datasets of multiple types
- Using canvas drawing in conjunction with SVG to draw large datasets
- Optimizing geospatial, network, and traditional dataviz
- Working with quadtrees to enhance spatial search performance

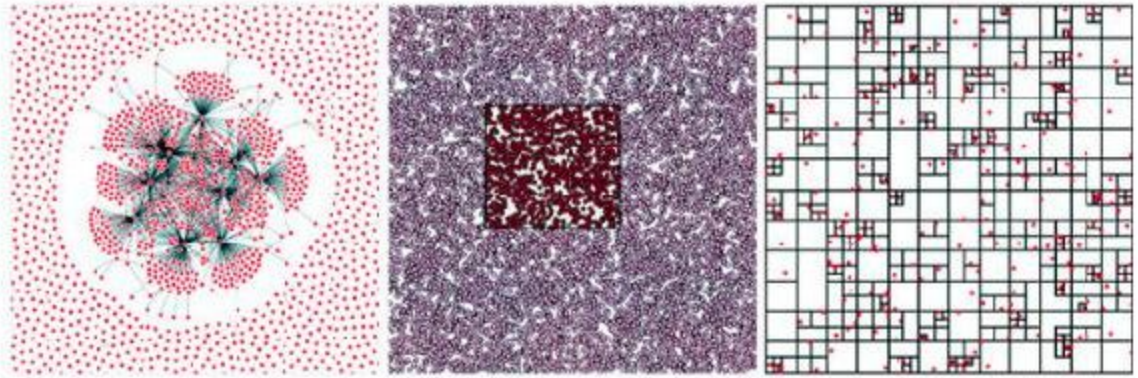
Overview

This lecture focuses on optimization techniques such as using canvas drawing to render large datasets in tandem with SVG for the interactive elements. This is demonstrated with maps, networks, and traditional xy data, which uses the D3 quadtree function.



Overview

This lecture focuses on optimization techniques such as using canvas drawing to render large datasets in tandem with SVG for the interactive elements. This is demonstrated with maps, networks, and traditional xy data, which uses the D3 quadtree function.



Built-in canvas rendering with d3-shape generators

bigdata.html

```
<!doctype html>
<html>
<head>
  <title>Big Data Visualization</title>
  <meta charset="utf-8" />
    <link type="text/css" rel="stylesheet" href="bigdata.css" />
</head>
<body>
<div>
<canvas height="500" width="500"></canvas>            1
   <div id="viz">
      <svg></svg>
</div>
</div>
<footer>
<script src="d3.v4.min.js"></script>
</footer>
</body>
</html>
```

1 Make sure to set the height and width attributes, not only the style attributes

Built-in canvas rendering with d3-shape generators

bigdata.css

```
body, html {
   margin: 0;
}
canvas {
  position: absolute;
  width: 500px;
  height: 500px;                1
}
svg {
  position: absolute;
  width:500px;
  height:500px;                 2
}
```

1 In this chapter we'll draw SVG over canvas, so the canvas element needs to have the same attributes as the SVG element
2 Likewise, identical settings for the SVG element

Built-in canvas rendering with d3-shape generators

```
path.country {
   fill: #C4B9AC;
   stroke-width: 1;
   stroke: #4F442B;
   opacity: .5;
}
path.sample {
   stroke: #41A368;
   stroke-width: 1px;
   fill: #93C464;
   fill-opacity: .5;
}
line.link {
   stroke-width: 1px;
   stroke: #4F442B;
   stroke-opacity: .5;
}
```

Built-in canvas rendering with d3-shape generators

```
circle.node {
  fill: #93C464;
  stroke: #EBD8C1;
  stroke-width: 1px;
}
circle.xy {
  fill: #FCBC34;
  stroke: #FE9922;
  stroke-width: 1px;
}
```

Built-in canvas rendering with d3-shape generators

Drawing violin plots on canvas

```
var fillScale = d3.scaleOrdinal().range(["#fcd88a", "#cf7c1c",
"#93c464"])

var normal = d3.randomNormal()
var sampleData1 = d3.range(100).map(d => normal())
var sampleData2 = d3.range(100).map(d => normal())
var sampleData3 = d3.range(100).map(d => normal())

var data = [sampleData1, sampleData2, sampleData3]

var histoChart = d3.histogram();

histoChart
  .domain([ -3, 3 ])
  .thresholds([ -3, -2.5, -2, -1.5, -1,
  -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3 ])
  .value(d => d)
var yScale = d3.scaleLinear().domain([ -3, 3 ]).range([ 400, 0 ])     1

var context = d3.select("canvas").node().getContext("2d")             2
```

1 Up until this point it’s all the same code
2 You need context to draw on canvas

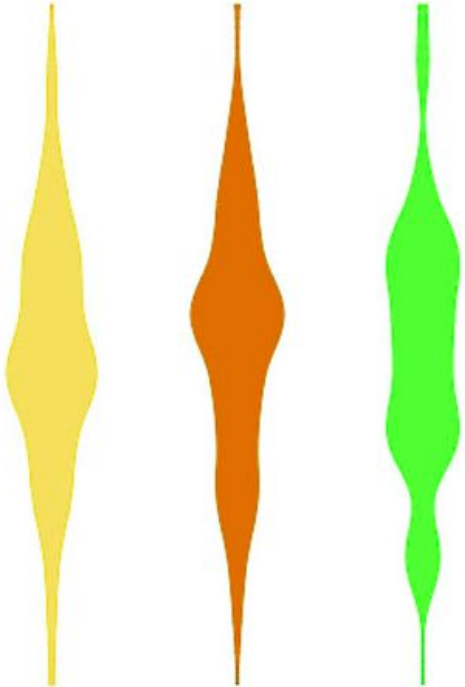
Built-in canvas rendering with d3-shape generators

```
 area = d3.area()
    .x0(d => -d.length)
    .x1(d => d.length)
    .y(d => yScale(d.x0))
    .curve(d3.curveCatmullRom)
    .context(context)                                    3

context.clearRect(0,0,500,500)                                  4
context.translate(0, 50)

    data.forEach((d, i) => {
    context.translate(100, 0)                              5
    context.strokeStyle = fillScale(i)
    context.fillStyle = d3.hsl(fillScale(i)).darker()
    context.lineWidth = "1px";
    context.beginPath()                                   6
    area(histoChart(d))                                  7
    context.stroke()                                    8
    context.fill()                                   8
})
```

3 Register the generator's .context with your context
4 This is one way to clear your canvas by blanking a rectangular section
5 Move the drawing start point with each shape
6 Start drawing
7 Run your generator with the appropriate data
8 Stroke and fill the shape you drew

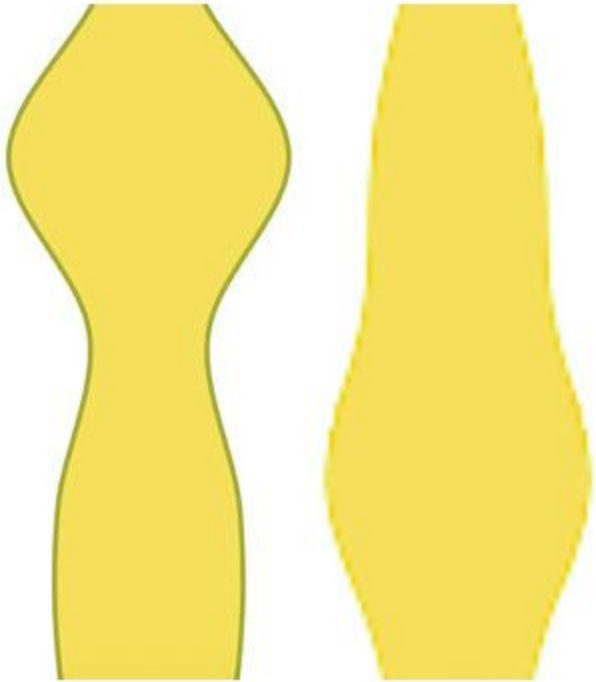
Built-in canvas rendering with d3-shape generators

Violin plots drawn using canvas. You can see that they’re more pixelated.



Built-in canvas rendering with d3-shape generators

Two zoomed-in shapes, one rendered in SVG (left) and one rendered with canvas (right)



Big geodata

Creating sample data

```
var sampleData = d3.range(1000).map(d => {              1
   var datapoint = {};                                  2
   datapoint.id = "Sample Feature " + d;
   datapoint.type = "Feature";
   datapoint.properties = {};
   datapoint.geometry = {};
   datapoint.geometry.type = "Polygon";
   datapoint.geometry.coordinates = randomCoords();
   return datapoint;
});
function randomCoords() {                               3
  var randX = (Math.random() * 350) - 175;
  var randY = (Math.random() * 170) - 85;
  return [[[randX - 5,randY],[randX,randY - 5],
        [randX - 10,randY - 5],[randX - 5,randY]]];
};
```

1 d3.range creates an array that we immediately map to an object array
2 Each datapoint is an object with the necessary attributes to be placed on a map
3 Draws a triangle around each random lat/long coordinate pair

Big geodata

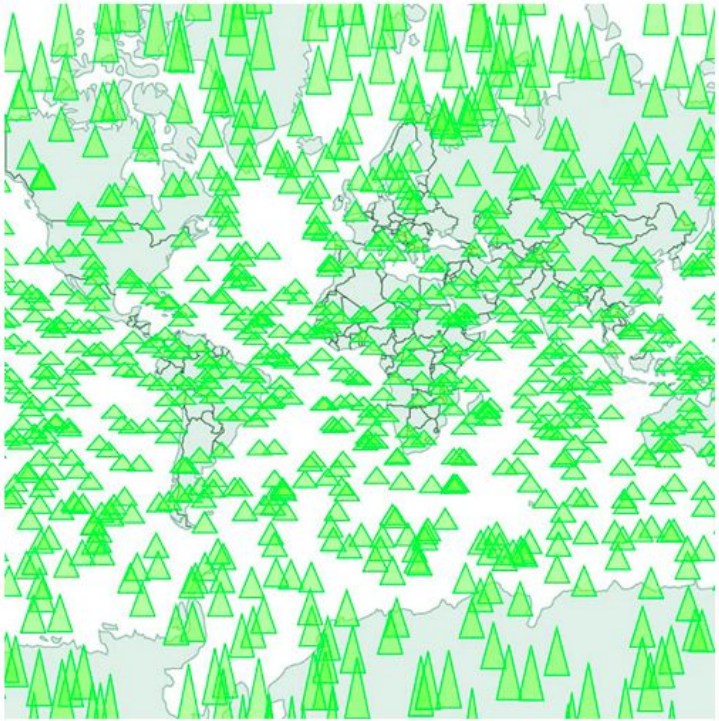
Drawing a map with our sample data on it

```
d3.json("world.geojson", data => {createMap(data)});
function createMap(countries) {
  var projection = d3.geoMercator()
      .scale(100).translate([250,250])                    1
  var geoPath = d3.geoPath().projection(projection);
  var g = d3.select("svg").append("g");
  g.selectAll("path.country")
      .data(countries.features)
      .enter()
      .append("path")
      .attr("d", geoPath)
      .attr("class", "country");
  g.selectAll("path.sample")
      .data(sampleData)
      .enter()
      .append("path")
      .attr("d", geoPath)
      .attr("class", "sample");
};
```

1 Adjusts the projection and translation of the projection rather than the <g> so we can use the projection later to draw to canvas

Big geodata

Drawing random triangles on a map entirely with SVG



Big geodata

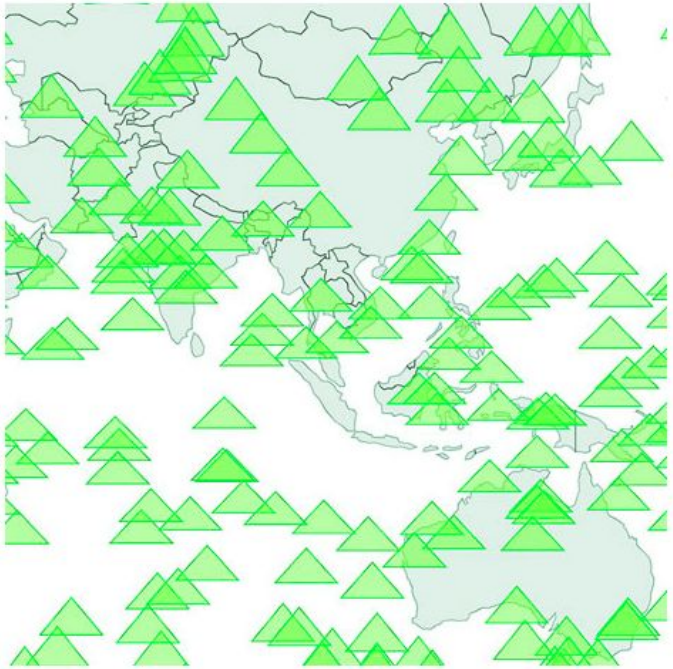
Adding zoom controls to a map

```
var mapZoom = d3.zoom()
     .on("zoom", zoomed);

  var zoomSettings = d3.zoomIdentity
      .translate(250, 250)
      .scale(100);

  d3.select("svg").call(mapZoom).call(mapZoom.transform,
zoomSettings);

  function zoomed() {
    var e = d3.event
      projection.translate([e.transform.x, e.transform.y])
        .scale(e.transform.k);                                    1
      d3.selectAll("path.country, path.sample").attr("d", geoPath)
  }
```

1 We use projection zoom in this example because it’ll be easier to draw canvas elements later

Big geodata

Zooming in on the sample geodata around East Asia and Oceania



Big geodata

Drawing the map with canvas

```
function createMap(countries) {
  var projection =
d3.geoMercator().scale(50).translate([150,100]);
  var geoPath = d3.geoPath().projection(projection);

  var mapZoom = d3.zoom()
     .on("zoom", zoomed)

  var zoomSettings = d3.zoomIdentity
      .translate(250, 250)
      .scale(100)

  d3.select("svg").call(mapZoom).call(mapZoom.transform,
zoomSettings)
  function zoomed() {
    var e = d3.event
      projection.translate([e.transform.x, e.transform.y])
        .scale(e.transform.k)
```

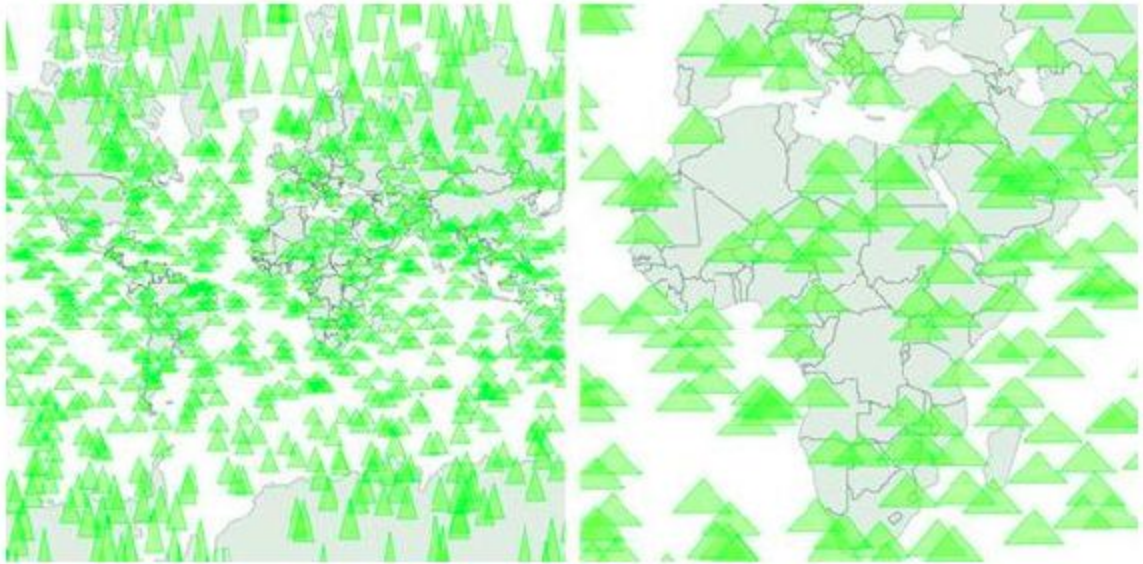
Big geodata

```
    var context = d3.select("canvas").node().getContext("2d")
    context.clearRect(0,0,500,500)                          1
    geoPath.context(context)                               2
    context.strokeStyle = "rgba(79,68,43,.5)"                 3
    context.fillStyle = "rgba(196,185,172,.5)"                3
    context.fillOpacity = 0.5
    context.lineWidth = "1px"
    for (var x in countries.features) {
      context.beginPath()
      geoPath(countries.features[x])                        4
      context.stroke()
      context.fill()
    }
    context.strokeStyle = "#41A368"
    context.fillStyle = "rgba(147,196,100,.5)";
    context.lineWidth = "1px"
    for (var x in sampleData) {
      context.beginPath()
      geoPath(sampleData[x])                                5
      context.stroke()
      context.fill()
    }
  }
}
```

1 Always clear the canvas before redrawing it if you're updating it
2 Switches geoPath to a context generator with our canvas context
3 Styles settings for countries
4 Draws each country feature to canvas
5 Draws each triangle to canvas

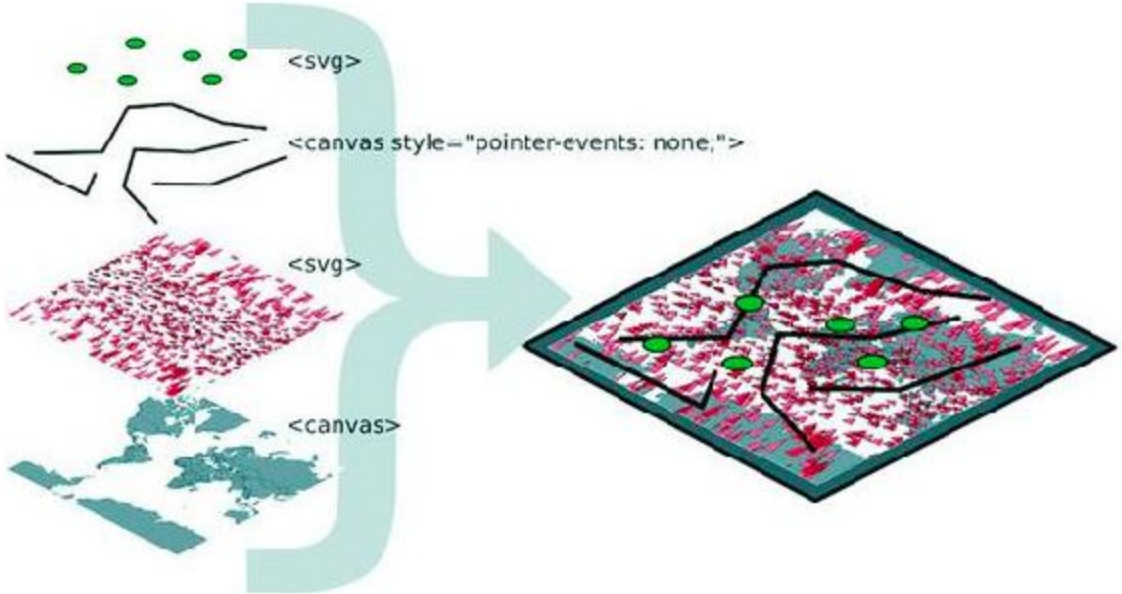
Big geodata

Drawing our map with canvas produces higher performance, but slightly less crisp graphics. On the left, it may seem like the triangles are as smoothly rendered as the earlier SVG triangles, but if you zoom in as we've done on the right, you can start to see clearly the slightly pixelated canvas rendering.



Big geodata

Placing interactive SVG elements below a <canvas> element requires that you set its pointer-events style to none, even if it has a transparent background, in order to register click events on the <svg> element underneath it.



Big geodata

Rendering SVG and canvas simultaneously

```
function createMap(countries) {
  var projection =
d3.geoMercator().scale(50).translate([150,100]);
  var geoPath = d3.geoPath().projection(projection);
  var svgPath = d3.geoPath().projection(projection);
1


  d3.select("svg")
    .selectAll("path.sample")
    .data(sampleData)
    .enter()
    .append("path")
    .attr("d", svgPath)
    .attr("class", "sample")
    .on("mouseover", function() {d3.select(this).style("fill",
"#75739F")});
```

1 We need to instantiate a different d3.geoPath for canvas and for SVG

Big geodata

```
var mapZoom = d3.zoom()
     .on("zoom", zoomed)

  var zoomSettings = d3.zoomIdentity
      .translate(250, 250)
      .scale(100)

  d3.select("svg").call(mapZoom).call(mapZoom.transform,
zoomSettings)
  function zoomed() {
    var zoomEvent = d3.event
      projection.translate([zoomEvent.transform.x,
zoomEvent.transform.y])
        .scale(zoomEvent.transform.k)
const featureOpacity = 0.5
```

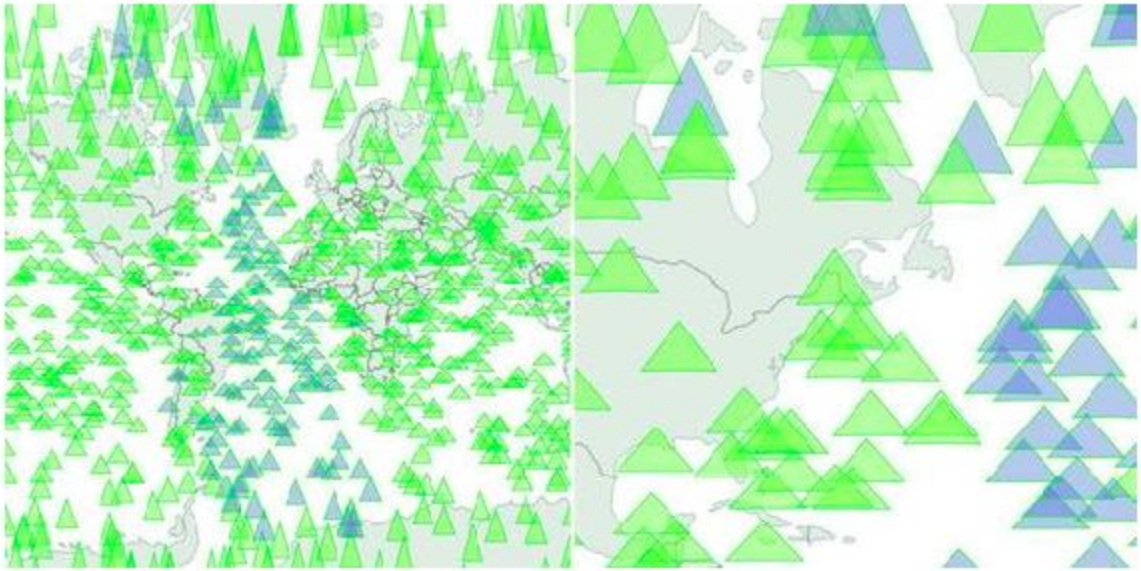
Big geodata

```
    var context = d3.select("canvas").node().getContext("2d");
    context.clearRect(0,0,500,500);
    geoPath.context(context);
    context.strokeStyle = `rgba(79,68,43,${featureOpacity})`;
    context.fillStyle = `rgba(196,185,172,${featureOpacity})`;
    context.lineWidth = "1px";
    countries.features.forEach(feature => {
      context.beginPath();
      geoPath(feature);                                    2
      context.stroke()
      context.fill();
    })

    d3.selectAll("path.sample").attr("d", svgPath);                3

  }
}
```

2 Draws canvas features with canvasPath
3 Draws SVG features with svgPath

Big geodata

Background countries are drawn with canvas, while foreground triangles are drawn with SVG to use interactivity. SVG graphics are individual elements in the DOM and are therefore amenable to having click, mouseover, and other event listeners attached to them.



Big geodata

Mixed rendering based on zoom interaction

```
...
    mapZoom = d3.zoom()
       .on("zoom", zoomed)                  1
       .on("start", zoomInitialized)        1
       .on("end", zoomFinished);              1
...
```

1 Assigns separate functions for each zoom state

Big geodata

Rendering action based on zoom event

Zoom event	Countries rendered as	Triangles rendered as
zoomed	Canvas	Canvas
zoomInitialized	Canvas	Hide SVG
zoomFinished	Canvas	SVG

Big geodata

Zoom functions for mixed rendering

```
var canvasPath = d3.geoPath().projection(projection);
--- Other code ----
function zoomed() {
  var e = d3.event
    projection.translate([e.transform.x, e.transform.y])
    .scale(e.transform.k)
  var context = d3.select("canvas").node().getContext("2d");
  context.clearRect(0,0,500,500);
  canvasPath.context(context);
  context.strokeStyle = "black";
  context.fillStyle = "gray";
  context.lineWidth = "1px";
  for (var x in countries.features) {
    context.beginPath();
    canvasPath(countries.features[x]);
    context.stroke()
    context.fill();
  }
```

Big geodata

```
  context.strokeStyle = "black";
  context.fillStyle = "rgba(255,0,0,.2)";
  context.lineWidth = 1;
  for (var x in sampleData) {
    context.beginPath();                          1
    canvasPath(sampleData[x]);
    context.stroke()
    context.fill();
  }
};
function zoomInitialized() {
  d3.selectAll("path.sample")
    .style("display", "none");                    2
  zoomed();                                  3
};
```

1 Draws all elements as canvas during zooming
2 Hides SVG elements when zooming starts
3 Calls zoomed to draw with canvas the SVG triangles we hid

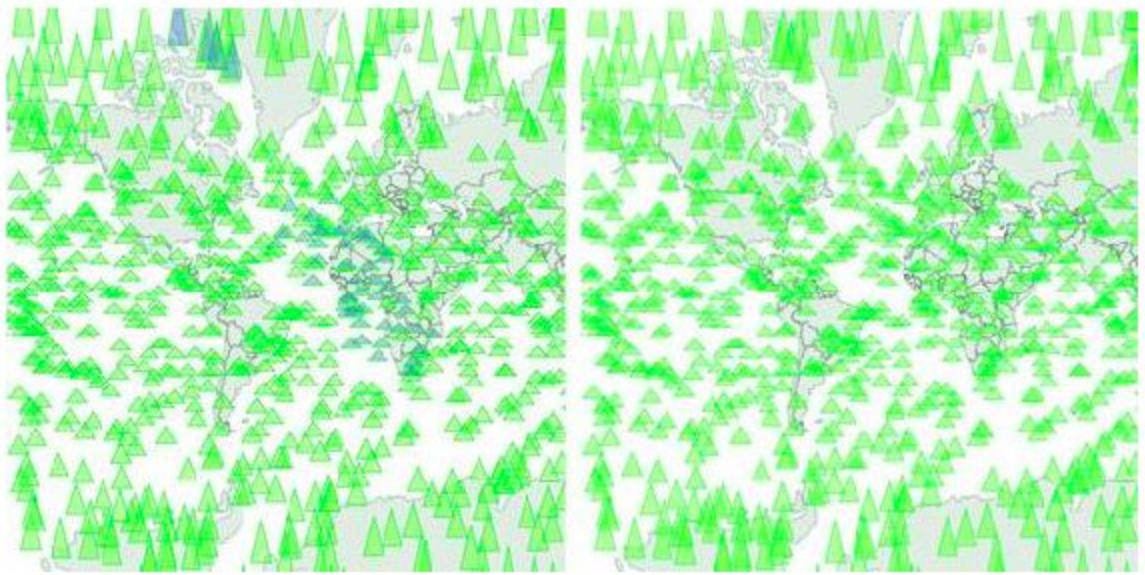
Big geodata

```
function zoomFinished() {
  var context = d3.select("canvas").node().getContext("2d");
  context.clearRect(0,0,500,500);
  canvasPath.context(context)
  context.strokeStyle = "black";
  context.fillStyle = "gray";
  context.lineWidth = "1px";
  for (var x in countries.features) {
    context.beginPath();
    canvasPath(countries.features[x]);                    4
    context.stroke()
    context.fill();
  }
  d3.selectAll("path.sample")
    .style("display", "block")                          5
    .attr("d", svgPath);                               6
};
```

4 Only draws countries with canvas at the end of the zoom
5 Shows SVG elements when zoom ends
6 Sets the new position of SVG elements

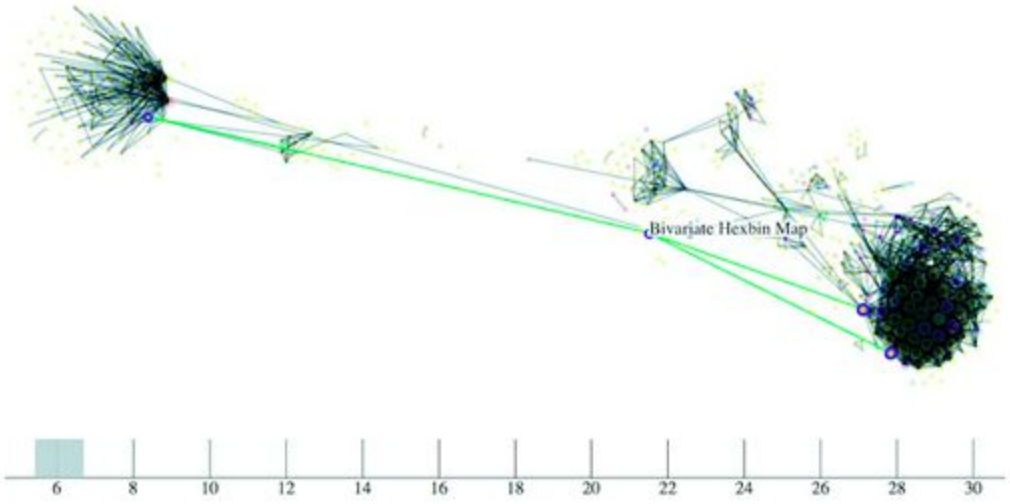
Big geodata

The same randomly generated triangles rendered in SVG while the map isn’t being zoomed or panned (left) and in canvas while the map is being zoomed or panned (right). Notice that only the SVG triangles have different fill values based on user interaction, because that isn’t factored into the canvas drawing code for the triangles on the right.



Big network data

A network of D3 examples hosted on gist.github.com that connects different examples to each other by shared functions. Here you can see that the example “Bivariate Hexbin Map” by Mike Bostock (http://bl.ocks.org/mbostock/4330486) shares functions in common with three different examples: Metropolitan Unemployment, Marey’s Trains II, and GitHub Users Worldwide. The brush and axis components allow you to filter the network by the number of connections from one block to another.



Big network data

Generating random network data

```
var linkScale = d3.scaleLinear()
        .domain([0,.9,.95,1]).range([0,10,100,1000]);              1
var sampleNodes = d3.range(3000).map(d => {
  var datapoint = {};
  datapoint.id = `Sample Node ${d}`;
  return datapoint;
})
var sampleLinks = [];
  var y = 0;
  while (y < 1000) {
    var randomSource = Math.floor(Math.random() * 1000);          2
    var randomTarget = Math.floor(linkScale(Math.random()));      3
    var linkObject = {source: sampleNodes[randomSource], target:
sampleNodes[randomTarget]}
    if (randomSource != randomTarget) {                           4
        sampleLinks.push(linkObject);
    }
    y++;
}
```

1 This scale makes 90% of the links to 1% of the nodes
2 The source of each link is purely random
3 The target is weighted toward popular nodes
4 Don’t keep any links that have the same source as target

Big network data

Force-directed layout

```
var force = d3.forceSimulation()
    .nodes(sampleNodes)
    .force("x", d3.forceX(250).strength(1.1))
    .force("y", d3.forceY(250).strength(1.1))
    .force("charge", d3.forceManyBody())
    .force("charge", d3.forceManyBody())
    .force("link", d3.forceLink())
    .on("tick", forceTick)                    1

force.force("link").links(sampleLinks)
```

1 This is all vanilla force-directed layout code like in lecture 6

Big network data

```
d3.select("svg")
   .selectAll("line.link")
   .data(sampleLinks)
   .enter()
   .append("line")
   .attr("class", "link");
d3.select("svg").selectAll("circle.node")
   .data(sampleNodes)
   .enter()
   .append("circle")
   .attr("r", 3)
   .attr("class", "node");
```

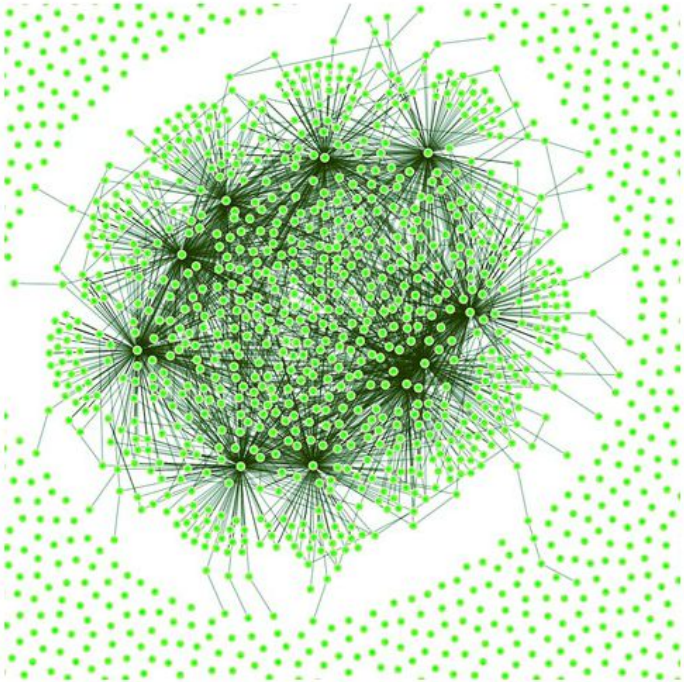
Big network data

```
function forceTick() {
  d3.selectAll("line.link")
     .attr("x1", d =>d.source.x)                2
     .attr("y1", d =>d.source.y)
     .attr("x2", d =>d.target.x)
     .attr("y2", d =>d.target.y);
  d3.selectAll("circle.node")
     .attr("cx", d =>d.x)
     .attr("cy", d =>d.y);
};
```

2 For our initial implementation, we render everything in SVG and update the SVG on every tick

Big network data

A randomly generated network with 3,000 nodes and 1,000 edges



Big network data

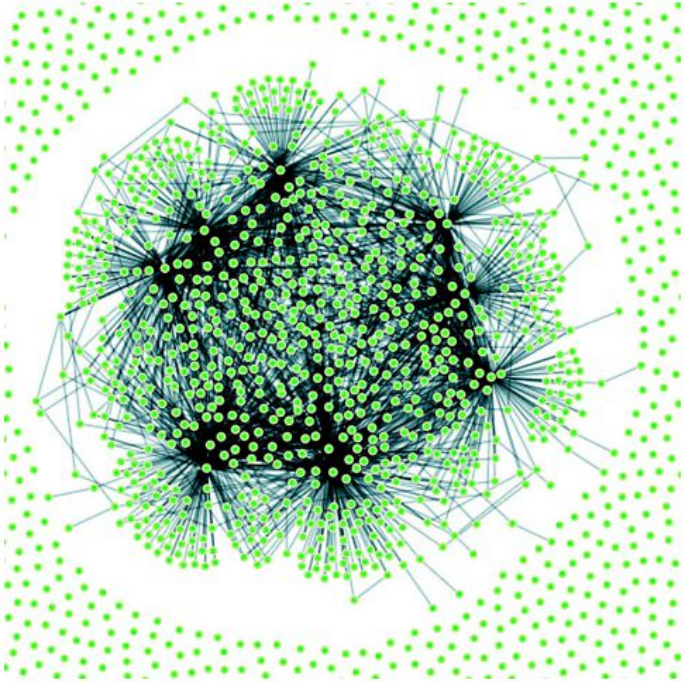
Mixed rendering network drawing

```
function forceTick() {
  var context = d3.select("canvas").node()
          .getContext("2d");
  context.clearRect(0,0,500,500);                    1
  context.lineWidth = 1;
  context.strokeStyle = "rgba(0, 0, 0, 0.5)";        2
  sampleLinks.forEach(function (link) {
    context.beginPath();
    context.moveTo(link.source.x,link.source.y)      3
    context.lineTo(link.target.x,link.target.y)      4
    context.stroke();
  });
  d3.selectAll("circle.node")                        5
    .attr("cx", d =>d.x)
    .attr("cy", d =>d.y)
};
```

1 Remember, you always need to clear your canvas
2 Draws links as 50% transparent black
3 Starts each line at the link source coordinates
4 Draws each link to the link target coordinates
5 Draws nodes as SVG

Big network data

A large network drawn with SVG nodes and canvas links



Optimizing xy data selection with quadtrees

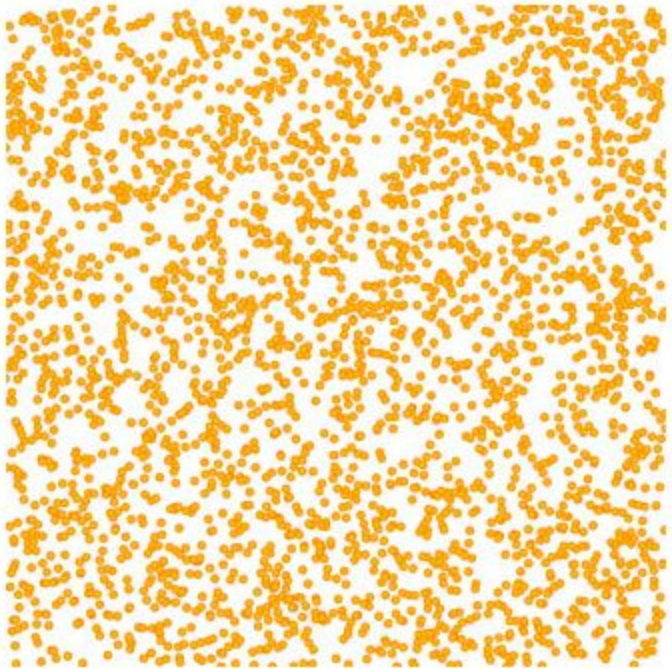
xy data generator

```
sampleData = d3.range(3000).map(function(d) {
  var datapoint = {};
  datapoint.id = `Sample Node ${d}`;
  datapoint.x = Math.random() * 500;
  datapoint.y = Math.random() * 500;              1
  return datapoint;
})
d3.select("svg").selectAll("circle")
  .data(sampleData)
  .enter()
  .append("circle")
  .attr("class", "xy")
  .attr("r", 3)
  .attr("cx", d => d.x)
  .attr("cy", d => d.y)
```

1 Because we know the fixed size of our canvas, we can hardwire this

Optimizing xy data selection with quadtrees

3,000 randomly placed points represented by orange SVG <circle> elements



Optimizing xy data selection with quadtrees

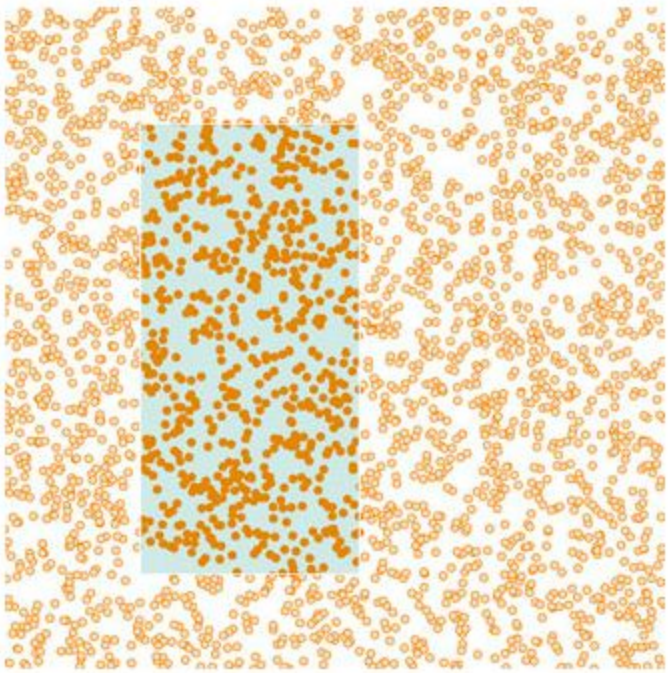
xy brushing

```
var brush = d3.brush()                                  1
  .extent([[0,0],[500,500]])
  .on("brush", brushed)

d3.select("svg").call(brush)

function brushed() {
  var e = d3.event.selection

  d3.selectAll("circle")
  .style("fill", d => {
    if (d.x >= e[0][0] && d.x <= e[1][0]
            && d.y >= e[0][1] && d.y <= e[1][1])     2
    {
      return "#FE9922"                                  3
    }
    else {
      return "#EBD8C1"                                  4
    }
  })
}
```

1 This brush gives us XY capability
2 Tests to see if the data is in our selected area
3 Colors the points in the selected
4 Colors the points outside the selected

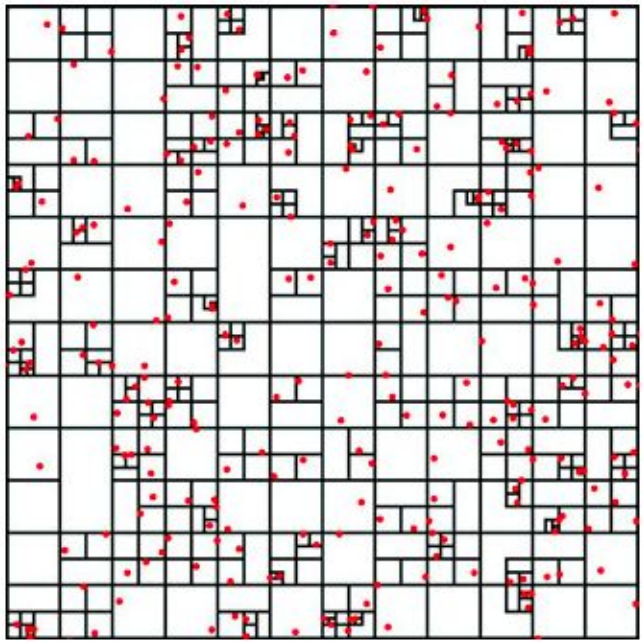
Optimizing xy data selection with quadtrees

Highlighting points in a selected region



Optimizing xy data selection with quadtrees

A quadtree for points shown in red with quadrant regions stroked in black. Notice how clusters of points correspond to subdivision of regions of the quadtree. Every point falls in only one region, but each region is nested in several levels of parent regions.



Optimizing xy data selection with quadtrees

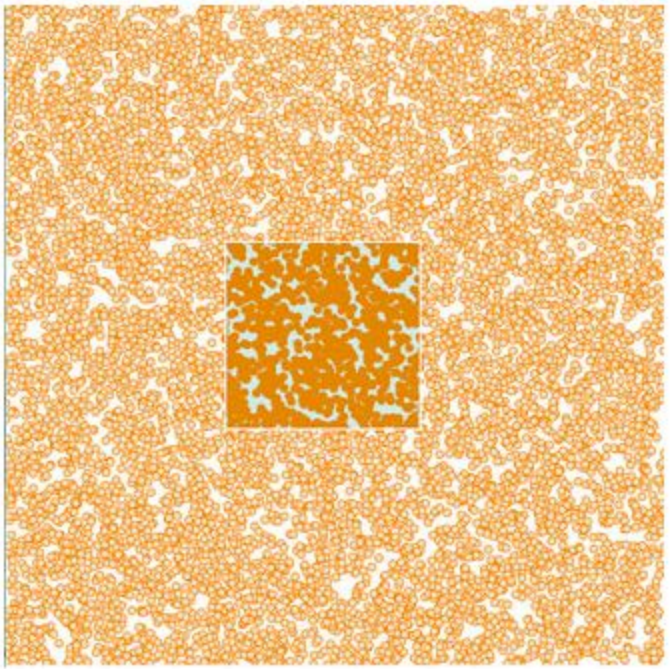
Creating a quadtree from xy data

```
var quadtree = d3.quadtree()
    .extent([[0,0], [500,500]])                                    1
 var quadIndex = quadtree(sampleData, d => d.x, d => d.y);         2
```

1 We need to define the bounding box of a quadtree as an array of upper-left and lower-right points
2 After creating a quadtree, we create the index by passing our dataset to it, along with the x and then y accessors

Optimizing xy data selection with quadtrees

Quadtree-optimized selection used with a dataset of 10,000 points



Optimizing xy data selection with quadtrees

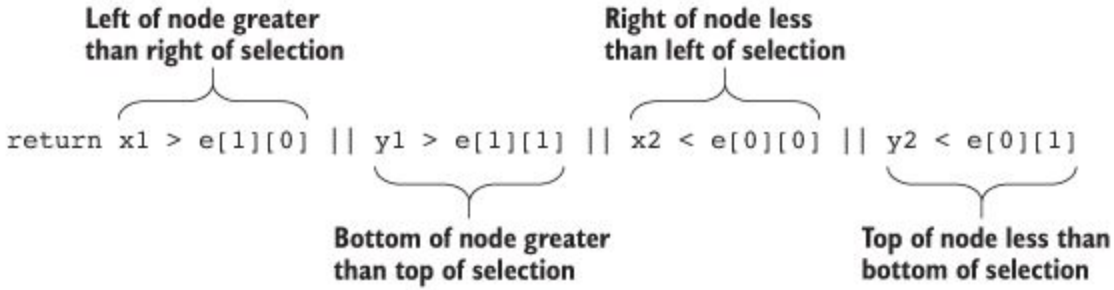
Quadtree-optimized xy brush selection

```
function brushed() {
  var e = d3.event.selection

  d3.selectAll("circle")
     .style("fill", "#EBD8C1")
     .each(d => {d.selected = false})                                    1
  quadIndex.visit(function(node,x0,y0,x1,y1) {

    if (node.data) {                                                    2
      if (node.data.x >= e[0][0] && node.data.x <= e[1][0]              3
          &&node.data.y >= e[0][1] && node.data.y <= e[1][1]) {
      node.data.selected = true;
    }
  }
  return x0 > e[1][0] || y0 > e[1][1] || x1 < e[0][0] || y1 < e[0][1]  4
  })

  d3.selectAll("circle")
      .filter(d => d.selected)
      .style("fill", "#FE9922")                                         5
}
```

1 Sets all circles to the unselected color and gives each a selected attribute to designate that's in our selection
2 Checks each node to see if it's a point or a container
3 Checks each point to see if it's inside our brush extent and sets selected to true if it is
4 Checks to see if this area of the quadtree falls outside our selection
5 Shows which points were selected

Optimizing xy data selection with quadtrees

The test to see whether a quadtree node is outside a brush selection involves four tests to see if it's above, left, right, or below the selection area. If it passes true for any of these tests, the quadtree will stop searching any child nodes.



More optimization techniques

Avoid general opacity, so instead of

```
d3.selectAll(elements).style("fill", "red").style("opacity", .5)
```

Do this:

```
d3.selectAll(elements).style("fill", "red").style("fill-opacity", .5)
```

More optimization techniques

Avoid general selections, so instead of

```
d3.selectAll("circle")
.style("fill", "#FE9922")
.each(d => {d.selected = false})
```

Do this:

```
d3.selectAll("circle")
.filter(d => d.selected})
.style("fill", "#FE9922")
.each(d => {d.selected = false})
```

More optimization techniques

Precalculate positions, So, instead of

```
d3.selectAll(elements)
.transition()
.duration(1000)
.attr("x", newComplexPosition);
```

Do this:

```
d3.selectAll(elements)
.each(function(d) {d.newX = newComplexPosition(d)});
d3.selectAll(elements)
.transition()
.duration(1000)
.attr("x", d => d.newX);
```