STATS 401 Lab 10 Tutorial: D3.js Interaction

Lab Goal

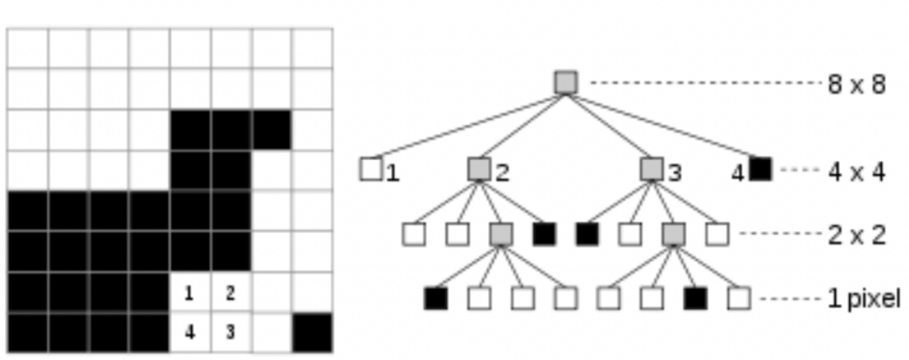
In this lab, we will learn to build interactions in D3.js through:

- Quadtree for picking small items easier
- Dragging for moving elements
- Brushing for selecting regions
- Zoom and pan

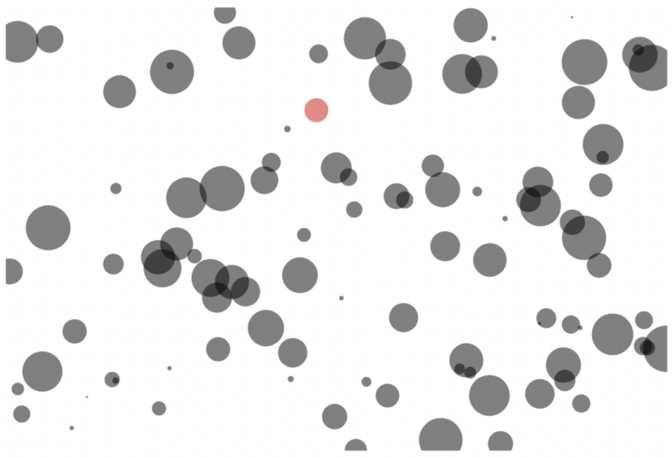
Code Instructions and Steps

Task 1: Quadtrees

A quadtree is a tree data structure that recursively divides an area into smaller and smaller areas and can make searching for items more efficient.



Line charts can also benefit from quadtrees when the user's task is to pick points along a line. One of the most useful D3 modules (especially when creating bar, line, and scatter charts) is the axis module which draws axes. If your chart contains items sized according to a data variable (such as a bubble chart), hovering over (or clicking) tiny items can be difficult. For instance, try hovering over the tiny circles (highlighted in red):



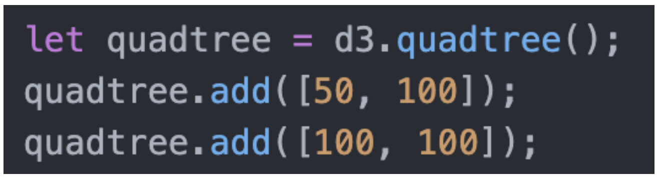
You can make picking small items easier by searching for the closest item to the mouse pointer each time the mouse is moved. This can be an expensive operation but can be made more efficient using D3's quadtree module.

You can directly use the main.css, and the CSS folder you created in Lab7 for this lab.

Create a new file, and name it area1.js script.
Create a new html file named quadtree.html, copy the code below to the file:

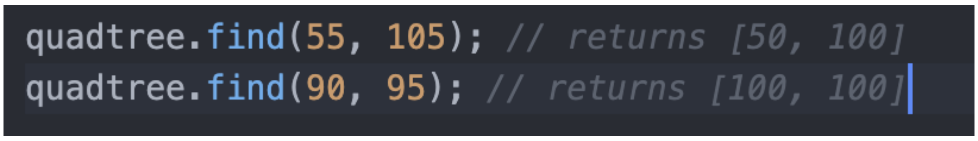
```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title></title>
   <script type="text/javascript" src="d3.js"></script>

 </head>
 <body>

  <svg  width="600" height="400"></svg>
    <script type="text/javascript", src = "area1.js">
</script>
 </body>
</html>
```

With D3's quadtree module you can create a quadtree, add some points to it, then find the closest point within the quadtree to a given coordinate. First, create a quadtree by calling `d3.quadtree()`. Then add single points to it using `.add`:



```
let quadtree = d3.quadtree();
quadtree.add([50, 100]);
quadtree.add([100, 100]);
```

Next, test it out by giving a coordinate (x, y) you can find the nearest point in the quadtree using `.find(x, y)`:



```
quadtree.find(55, 105); // returns [50, 100]
quadtree.find(90, 95); // returns [100, 100]
```

You can also add a distance (as the third argument) so that only points within that distance are returned:

```
quadtree.find(60, 100, 20); // returns [50, 100]
quadtree.find(60, 100, 5); // returns undefined
```

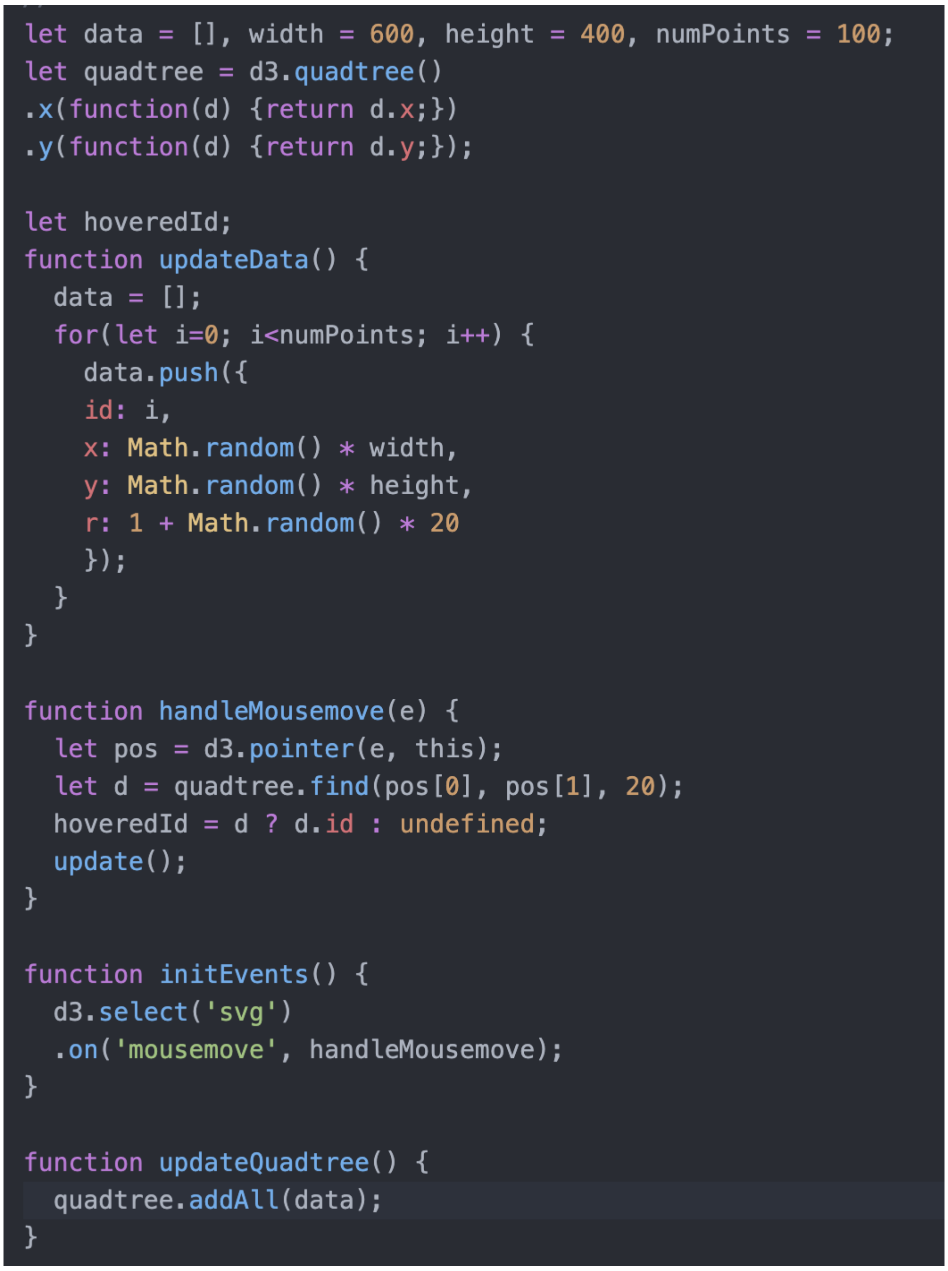
The first `.find` returns `[50, 100]` because this is the closest point and is within a distance of 20. The second .find returns undefined because the closest point `[50, 100]` is more than 5 away. This is useful for ensuring the returned point is close to the requested point. Without this constraint, outlier points can get selected even though the pointer isn't very close. OR, you can add an array of points using `.addAll`:

```
quadtree.addAll([[10, 50], [60, 30], [80, 20]]);
```

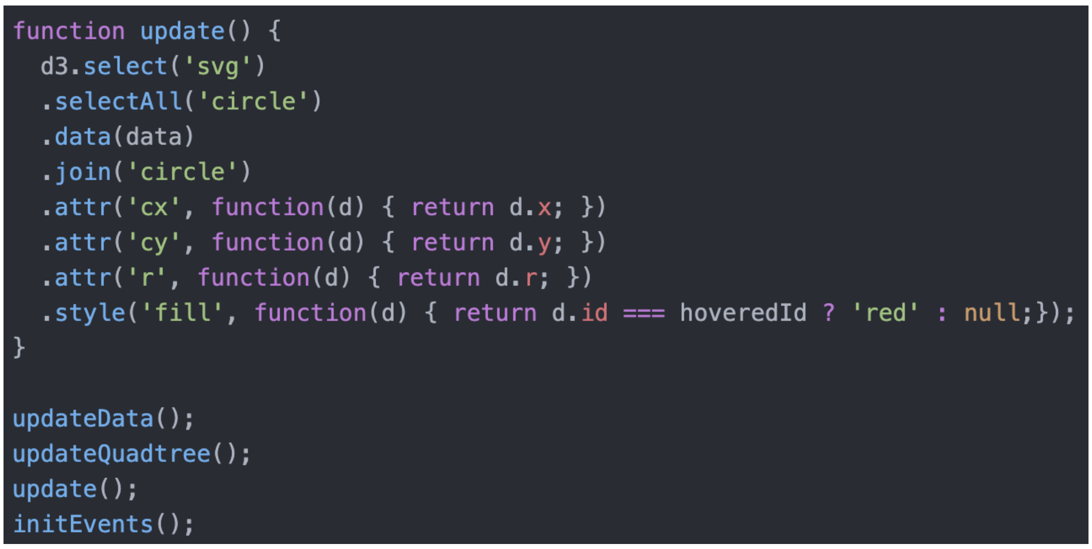
The `.add` and `.addAll` methods are cumulative i.e. the existing quadtree points persist. If you have an array of objects, you can specify accessor functions using the .x and .y methods:

```
let data = [
{ x: 50, y: 100 },
{ x: 100, y: 100 }
];
let quadtree = d3.quadtree()
.x(function(d) {return d.x;})
.y(function(d) {return d.y;});
quadtree.addAll(data);
quadtree.find(60, 100); // returns {x: 50, y: 100}
```

Example
Let's look at an example where we create some randomized points (`updateData`) and add them to a quadtree (`updateQuadtree`). We draw the points (update) and set up a mouse move event on the SVG element (`initEvents`). When the mouse moves (see handleMousemove) we search for the nearest point to the mouse pointer using the quadtree. We update hovered with the found point's id then call update again so that the hovered point is colored red:

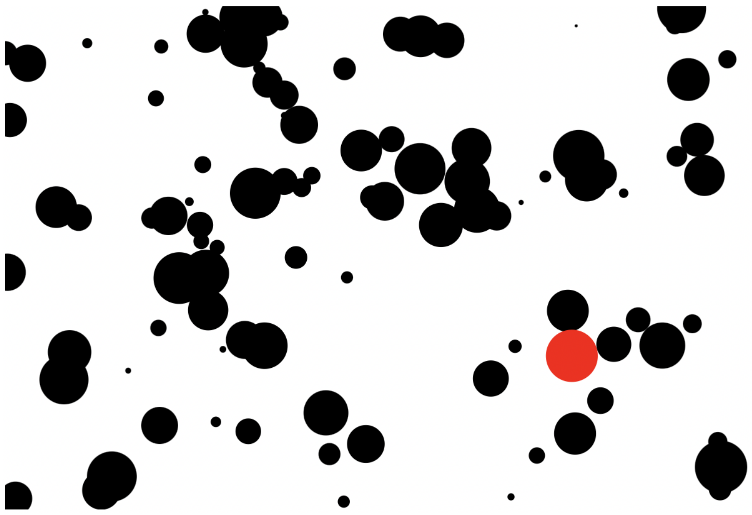


```
let data = [], width = 600, height = 400, numPoints = 100;
let quadtree = d3.quadtree()
.x(function(d) {return d.x;})
.y(function(d) {return d.y;});

let hoveredId;
function updateData() {
  data = [];
  for(let i=0; i<numPoints; i++) {
    data.push({
    id: i,
    x: Math.random() * width,
    y: Math.random() * height,
    r: 1 + Math.random() * 20
    });
  }
}

function handleMousemove(e) {
  let pos = d3.pointer(e, this);
  let d = quadtree.find(pos[0], pos[1], 20);
  hoveredId = d ? d.id : undefined;
  update();
}

function initEvents() {
  d3.select('svg')
  .on('mousemove', handleMousemove);
}

function updateQuadtree() {
  quadtree.addAll(data);
}
```



```
function update() {
  d3.select('svg')
  .selectAll('circle')
  .data(data)
  .join('circle')
  .attr('cx', function(d) { return d.x; })
  .attr('cy', function(d) { return d.y; })
  .attr('r', function(d) { return d.r; })
  .style('fill', function(d) { return d.id === hoveredId ? 'red' : null;});
}

updateData();
updateQuadtree();
update();
initEvents();
```

In `handleMousemove`, we use d3.pointer which gives the event object e and an HTML/SVG element and returns the mouse position relative to the HTML/SVG element. In our example, this is the SVG element because this is the element .on was called on (see initEvents). Now your mouse pointer only needs to be within 20 pixels of its nearest circle:



Task 2: Dragging

D3 has a module for adding drag behavior to elements. Dragging is where you hover over an element, press the mouse button, move the pointer, then release the mouse button, in order to move the element.
D3's drag module also supports touch gestures. There are three steps to making HTML/SVG elements draggable:

- call `d3.drag()` to create a drag behavior function
- add an event handler that's called when a drag event occurs. The event handler receives an event object with which you can update the position of the dragged element
- attach the drag behavior to the elements you want to make draggable

Create a new file, and name it area2.js script.
Create a new html file named dragging.html, copy the code below to the file:

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title></title>
   <script type="text/javascript" src="d3.js"></script>

 </head>
 <body>
```

```
  <svg  width="600" height="400"></svg>
    <script type="text/javascript", src = "area2.js">
</script>
 </body>
</html>
```

Calling `d3.drag()` creates a drag behavior:

```
let drag = d3.drag();
```

A drag behavior is a function that adds event listeners to elements. It also has methods such as .on defined on it. You can attach an event handler to the drag behavior by calling the .on method. This accepts two arguments:

- the event type ('drag', 'start' or 'end')
- the name of your event handler function

```
function handleDrag(e) {
// update the dragged element with its new position }
let drag = d3.drag()
.on('drag', handleDrag);
```

The event types are 'drag', 'start', and 'end'. 'drag' indicates a drag. 'start' indicates the start of the drag (e.g. the user has pressed the mouse button). 'end' indicates the end of the drag (e.g. the user has released the mouse button).
`handleDrag` receives a single parameter e which is an object representing the drag event. The drag event object has several properties, the most useful of which are:

Property name	Description
`.subject`	The joined data of the dragged element
`.x & .y`	The new coordinates of the dragged element
`.dx & .dy`	The new coordinates of the dragged element, relative to the previous coordinates

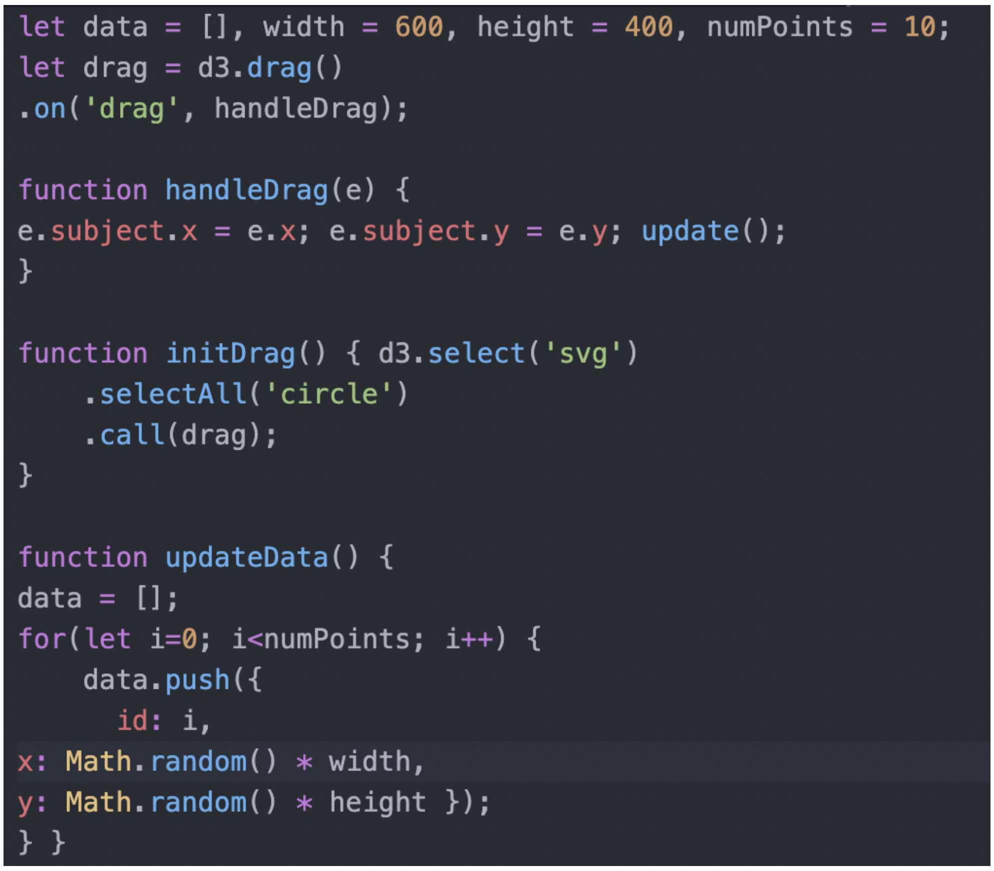
If the dragged element was created by a data join and the joined data has x and y properties, the x and y properties of the drag event object are computed such that the relative position of the element and pointer are maintained. (This prevents the element's center 'snapping' to the pointer position.) Otherwise, x and y are the pointer position relative to the dragged element's parent element.

You attach the drag behavior to elements by selecting the elements and passing the drag behavior into the `.call` method.
For example to add drag behavior to circle elements:

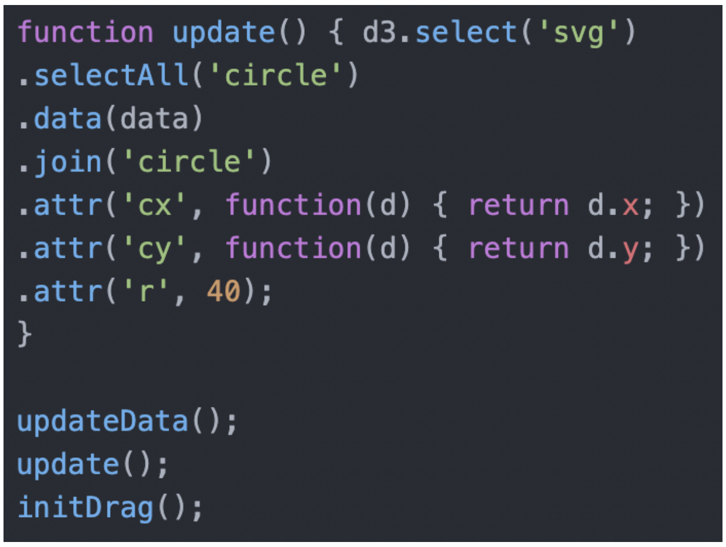
```
d3.select('svg')
.selectAll('circle')
.call(drag);
```

The drag behavior is a function that sets up event listeners on the selected elements (each circle element in the above example). When drag events occur the event handler (`handleDrag` in the above examples) is called.

Example
In the following code an array of random coordinates is joined to circle elements (`updateData` and `update`). A drag behavior is created using `d3.drag()` and attached to the circle elements (`initDrag`). When a circle is dragged, handleDrag gets called and an event object e is passed in as the first argument. `e.subject` represents the joined data of the dragged element. The x and y properties of the joined data are updated to `e.x` and `e.y.` `update` is then called to update the circle positions.

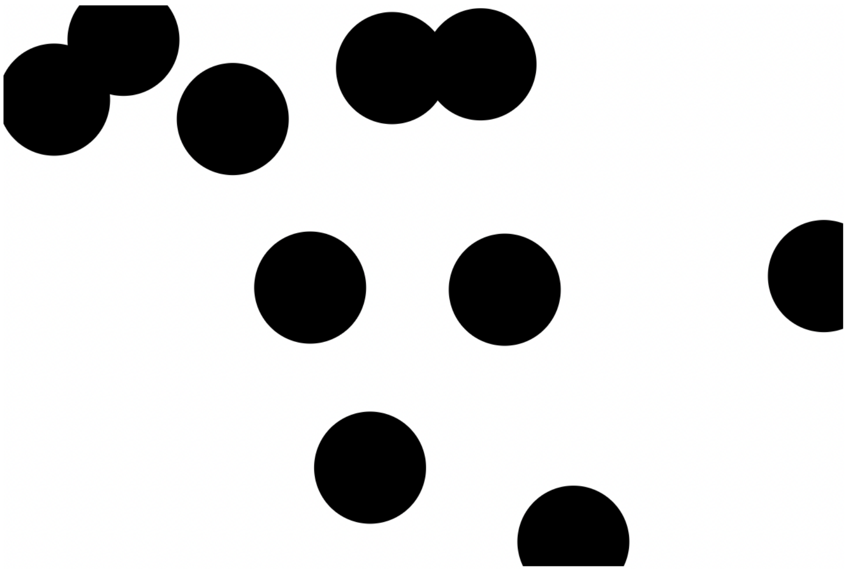


```
let data = [], width = 600, height = 400, numPoints = 10;
let drag = d3.drag()
.on('drag', handleDrag);

function handleDrag(e) {
e.subject.x = e.x; e.subject.y = e.y; update();
}

function initDrag() { d3.select('svg')
    .selectAll('circle')
    .call(drag);
}

function updateData() {
data = [];
for(let i=0; i<numPoints; i++) {
    data.push({
      id: i,
x: Math.random() * width,
y: Math.random() * height });
} }
```



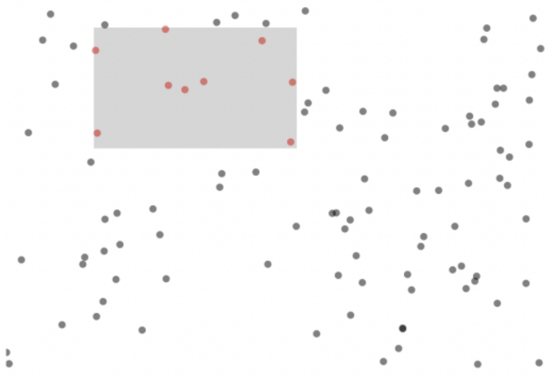
```
function update() { d3.select('svg')
.selectAll('circle')
.data(data)
.join('circle')
.attr('cx', function(d) { return d.x; })
.attr('cy', function(d) { return d.y; })
.attr('r', 40);
}

updateData();
update();
initDrag();
```

Notice that the relative position of the pointer and the dragged circle is maintained (try initiating a drag near the edge of a circle). Try it out in the browser!



Task 3: Brushing

Brushing lets you user specify an area (by pressing the mouse button, moving the mouse, then releasing) in order to, for example, select a group of elements. Try selecting circles by pressing the mouse button, dragging, then releasing the button (for your final code).



D3 has a module for adding brushing behavior to an element (or, less commonly, multiple elements).

Create a new file, and name it area3.js script.
Create a new HTML file named brushing.html, copy the code below to the file:

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title></title>
   <script type="text/javascript" src="d3.js"></script>

 </head>
 <body>

  <svg  width="600" height="400">
    <g></g>
  </svg>
    <script type="text/javascript", src = "area3.js">
</script>
 </body>
</html>
```

There's three steps to adding brush behaviour to an HTML or SVG element:

- call `d3.brush()` to create a brush behaviour function
- add an event handler that's called when a brush event occurs. The event handler receives the brush extent which can then be used to select elements, define a zoom area, etc.
- attach the brush behavior to an element (or elements)

Calling d3.brush() creates a brush behavior:

```
let brush = d3.brush();
```

A brush behavior is a function that has methods such as .on defined on it. The function itself adds event listeners to an element as well as additional elements (mainly `rect` elements) for rendering the brush extent.
You can attach an event handler to a brush behavior by calling the `.on` method. This accepts two arguments:

- the event type ('brush', 'start' or 'end')
- the name of your event handler function

```
function handleBrush(e) {
// get the brush extent and use it to, for example, select
elements }
```

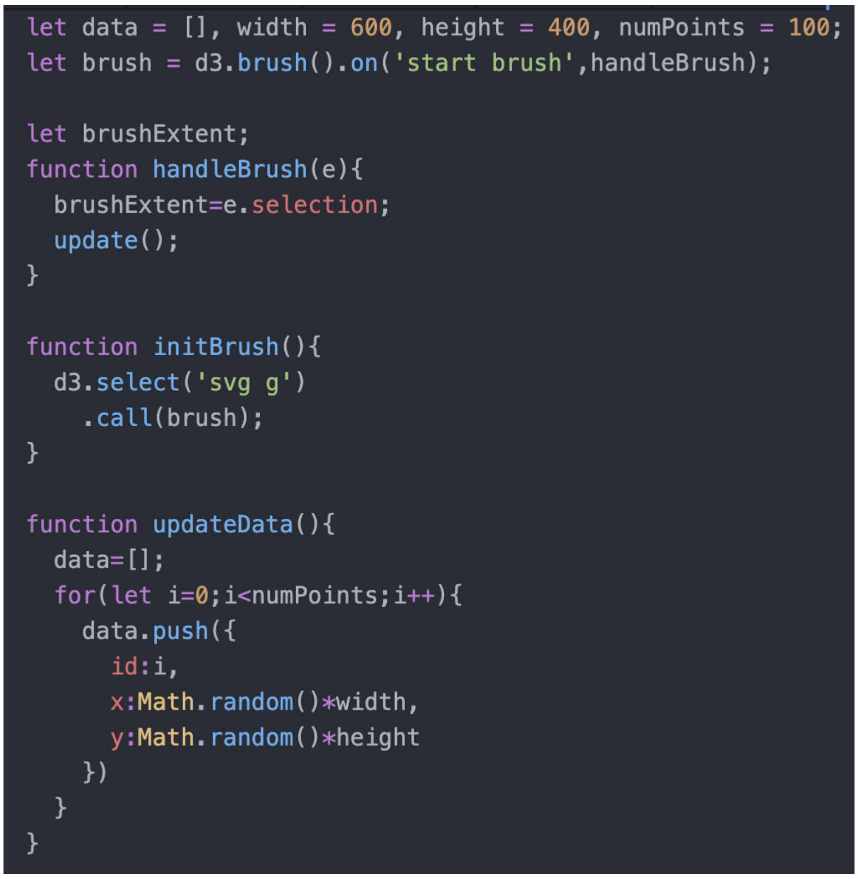
```
let brush = d3.brush()
.on('brush', handleBrush);
```

The event types are 'brush', 'start' and 'end'. 'brush' indicates that the brush extent has changed. 'start' indicates the brushing has started (e.g. the user has pressed the mouse button). 'end' indicates the end of brushing (e.g. the user has released the mouse button).
`handleBrush` receives a single parameter e which is an object representing the brushing event. The most useful property of the brushing event is `.selection` which represents the extent of the brush as an array `[[x0, y0], [x1, y1]]` where `x0, y0`, and `x1, y1` are the opposite corners of the brush. Typically `handleBrush` will compute which elements are within the brush extent and update them accordingly.
You attach the brush behavior to an element by selecting the element and passing the brush behavior into the `.call` method:

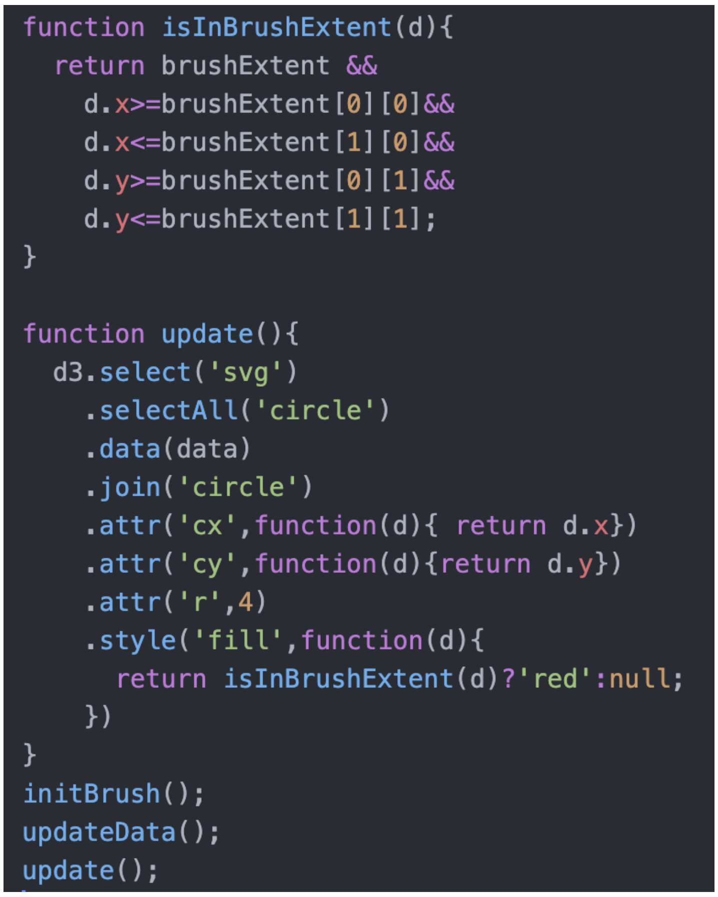
```
d3.select('svg')
.call(brush);
```

Examples
Here's a complete example where an array of random coordinates is joined to circle elements (`updateData` and `update`). When the brush is active, the circles within the brush extent are colored red.

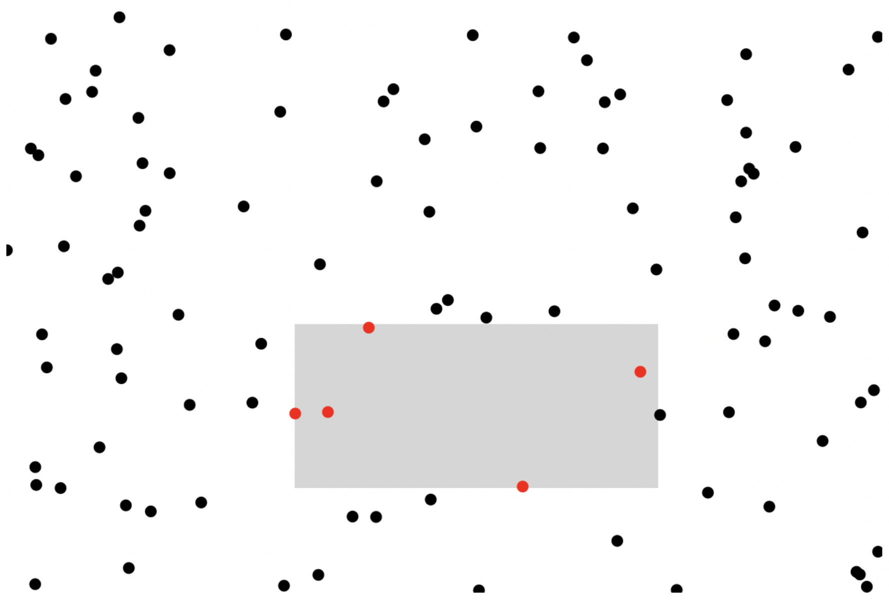
The brush is initialized in `initBrush`. (Note that it's attached to a g element within the SVG element, in order to keep the elements used to render the brush separate from the circles.) When brushing occurs, `handleBrush` is called. This receives a brush event object e which has a property selection that defines the extent of the brush. This is saved to the variable `brushExtent` and `update` is called. `update` performs the data join and colors circles red if they're within the extent defined by `brushExtent` (see `isInBrushExtent)`:



```
let data = [], width = 600, height = 400, numPoints = 100;
let brush = d3.brush().on('start brush',handleBrush);

let brushExtent;
function handleBrush(e){
  brushExtent=e.selection;
  update();
}

function initBrush(){
  d3.select('svg g')
    .call(brush);
}

function updateData(){
  data=[];
  for(let i=0;i<numPoints;i++){
    data.push({
      id:i,
      x:Math.random()*width,
      y:Math.random()*height
    })
  }
}
```



```
function isInBrushExtent(d){
  return brushExtent &&
    d.x>=brushExtent[0][0]&&
    d.x<=brushExtent[1][0]&&
    d.y>=brushExtent[0][1]&&
    d.y<=brushExtent[1][1];
}

function update(){
  d3.select('svg')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx',function(d){ return d.x})
    .attr('cy',function(d){return d.y})
    .attr('r',4)
    .style('fill',function(d){
      return isInBrushExtent(d)?'red':null;
    })
}
initBrush();
updateData();
update();
```



One-dimensional brushes

D3 also provides brushes `d3.brushX` and `d3.brushY` that constrain the brush to a single dimension. They work in a similar fashion to `d3.brush`, the main difference being the event object's `.selection` property which is an array of two numbers `[min, max]` which represent the extent of the brush.

Programmatic control of brushing

You can also set the brush extent programmatically. For example, you can create a button that sets the brush to maximum size. The brush behavior has two methods for setting the brush extent `.move` and `.clear`. The first sets the brush extent to `[[x0, y0], [x1, y1]]`, and the second clears the brush. `.move` and `.clear` should be called on the element that receives the brush gestures. For example:

```
d3.select('svg g')
.call(brush.move, [[50, 50], [100, 100]]);
```

Task4: Zoom and Pan

D3 provides a module that adds zoom and pan behaviour to an HTML or SVG element. In the following example, click and move the mouse to pan and use the mouse wheel to zoom. When zoom and pan gestures (such as dragging or a pinch gesture) occur, a transform (consisting of scale and translate) is computed by D3 and passed into an event handler. The event handler typically applies the transform to chart elements. There are three steps to add zoom and pan behavior to an element:

- call `d3.zoom()` to create a zoom behavior function

- add an event handler that gets called when a zoom or pan event occurs. The event handler receives a transform which can be applied to chart elements
- attach the zoom behavior to an element that receives the zoom and pan gestures

It's helpful to distinguish between the HTML or SVG element that receives the zoom and pan gestures and the elements that get zoomed and panned (the elements that get transformed). It's important that these elements are different, otherwise, the panning won't work properly. Calling `d3.zoom()` creates a zoom behavior:

```
let zoom = d3.zoom();
```

Although it's called d3.zoom, this module handles zoom and pan events. A zoom behavior is a function that adds event handlers (for drags, mouse wheel events and touch events, etc.) to an element. It also has methods such as .on defined on it. You can attach an event handler to your zoom behavior by calling the `.on` method. This accepts two arguments:
- the event type ('zoom', 'start' or 'end')
- the name of your event handler function

```
function handleZoom(e) {
// apply transform to the chart }
let zoom = d3.zoom()
.on('zoom', handleZoom);
```

The event types are `'zoom'`, `'start'` and `'end'`. `'zoom'` indicates a change of transform (e.g. the user has zoomed or panned). `'start'` indicates the start of the zoom or pan (e.g. the user has pressed the mouse button). `'end'` indicates the end of the zoom or pan (e.g. the user has released the mouse button).

`handleZoom` receives a single parameter e which is an object representing the zoom event. The most useful property of this object is transformed. This is an object that represents the latest zoom transform and is typically applied to the chart element(s):

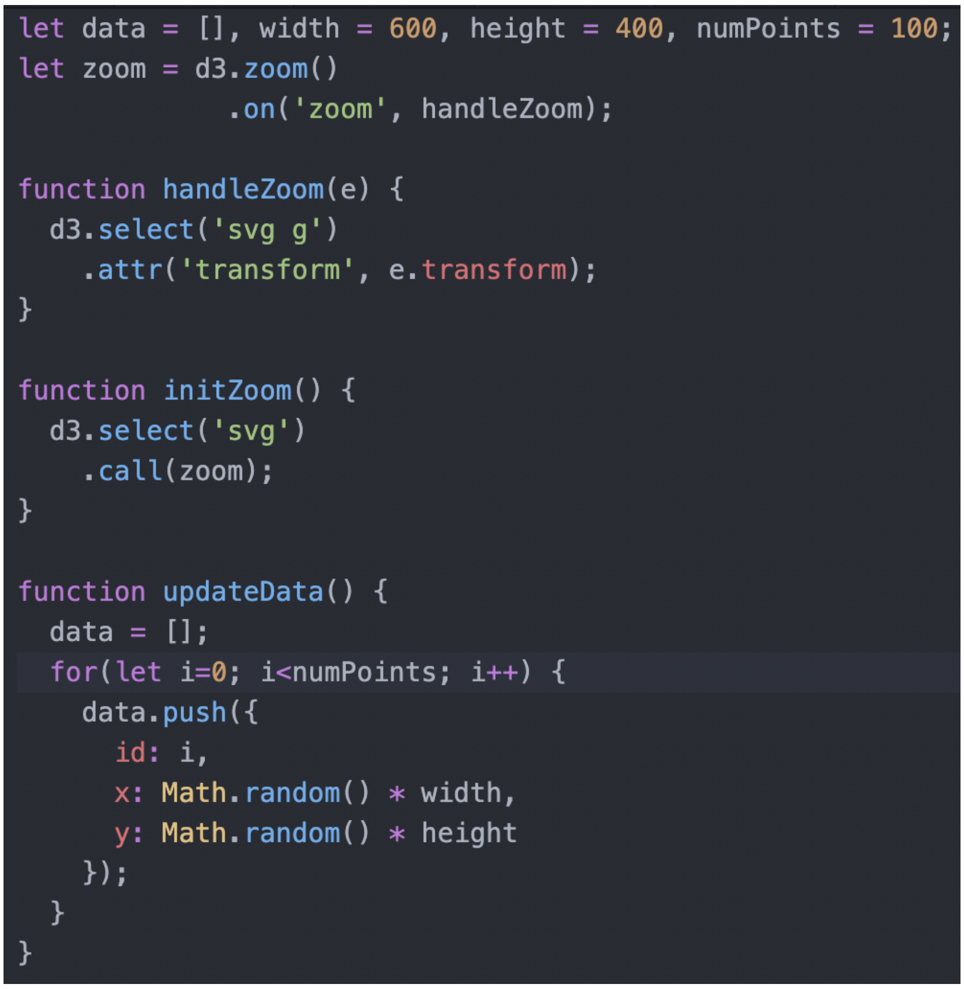
```
function handleZoom(e) {
     d3.select('g.chart')
     .attr('transform', e.transform);
}
```
`e.transform` has three properties x, y and k. x and y specify the translate transform and k represents the scale factor. It also has a `.toString` method which generates a string such as "`translate(38.9,-4.1) scale(1.3)`". This means you can pass e.transform directly into `.attr`. You attach the zoom behavior to an

element by selecting the element and passing the zoom behavior into the `.call` method:

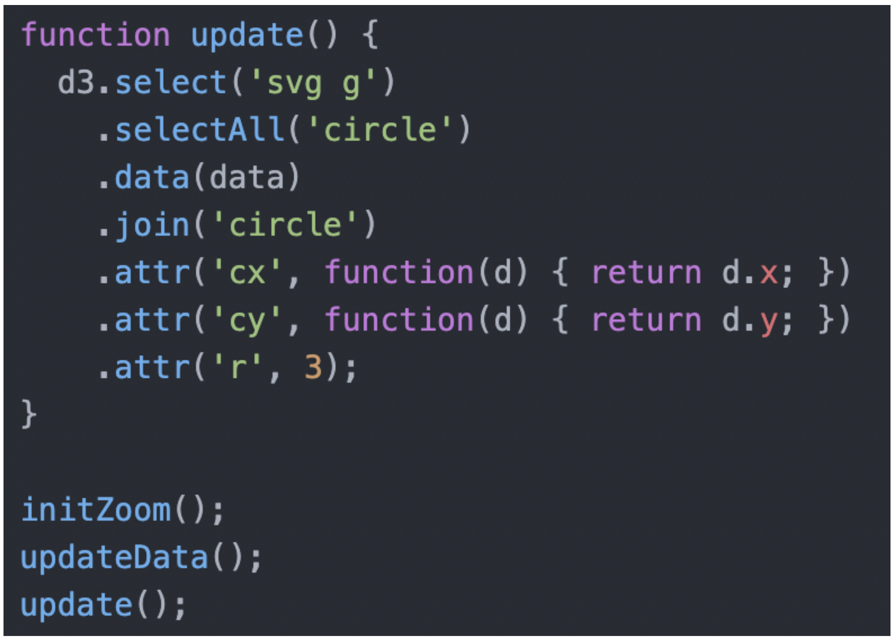
```
d3.select('svg')
.call(zoom);
```

The zoom behavior is a function that sets up event listeners on the selected element (`svg` in the above example). When zoom and pan events occur, a transform is computed and passed into the event handler (`handleZoom` in the above examples).

Example
Here’s a full example where an array of random coordinates is joined to circle elements:



```
let data = [], width = 600, height = 400, numPoints = 100;
let zoom = d3.zoom()
             .on('zoom', handleZoom);

function handleZoom(e) {
  d3.select('svg g')
    .attr('transform', e.transform);
}

function initZoom() {
  d3.select('svg')
    .call(zoom);
}

function updateData() {
  data = [];
  for(let i=0; i<numPoints; i++) {
    data.push({
      id: i,
      x: Math.random() * width,
      y: Math.random() * height
    });
  }
}
```



```
function update() {
  d3.select('svg g')
    .selectAll('circle')
    .data(data)
    .join('circle')
    .attr('cx', function(d) { return d.x; })
    .attr('cy', function(d) { return d.y; })
    .attr('r', 3);
}

initZoom();
updateData();
update();
```