5
Getting Out of the Box

In this chapter we will cover:

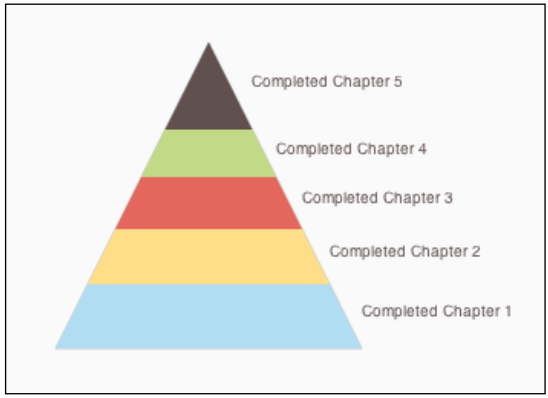
- Going though a funnel (a pyramid chart)
- Revisiting lines: making the line chart interactive
- Tree mapping and recursiveness
- Adding user interaction into tree mapping
- Making an interactive click meter

Introduction

We have covered our bases with the majority of the standard charts. At this stage, it's time for us to become more creative with our charts. From this chapter onwards, we will progress into more out-of-the-box, less commonly used charts and revisit some of our old charts to incorporate dynamic data into them or to change their layout.

Going through a funnel (a pyramid chart)

It's rare that you see a pyramid chart that is actually created dynamically. For the most part, they're designed and fleshed out creatively and turn into a `.jpg` file when they reach the web, and that's exactly why I wanted to start this chapter with this chart—it's not as complex as it might sound.



A pyramid chart is in essence a way for us to visualize changes in data that are quantitative by nature. They have a clear relationship between the lower layers and the higher layers. That sounded very vague, so let's explain it through an example.

Imagine that X amount of people complete their eighth year of school in a given year, if we follow the same group of people, how many of them would have completed their twelfth year of education four years later? Fair enough! We can't know the answer to that, but one thing we do know is that it can't be more than the initial X amount of people. The idea of a pyramid chart is exactly that of a body of data, of which less and less goes through the funnel as time or something else changes It's a really great chart to compare between levels of education, finance, involvement in politics, and so on.

Getting ready

Just as always, set up our HTML file logic. For a refresher on how to start up the HTML file, please go back to the *Graphics with 2D canvas* recipe in *Chapter 1*, *Drawing Shapes in Canvas*.

How to do it...

Beyond our standard HTML preparation, we need to come up with the data sources that we wish to showcase. Let's start building our pyramid. Go right into the JS file and let's start.

1. For our example, we will create a pyramid to find out how many people that read this book from chapter one through chapter five actually reach chapter five (this data is fake; I hope everyone that starts reading will get there!).

```
var layers = [{label:"Completed Chapter 1", amount:23},
  {label:"Completed Chapter 2", amount:15},
  {label:"Completed Chapter 3", amount:11},
  {label:"Completed Chapter 4", amount:7},
  {label:"Completed Chapter 5", amount:3} ];
```

2. Then, provide some charting and styling information.

```
var chartInfo= {height:200, width:200};

var s = { outlinePadding:4,
  barSize:16,
  font:"12pt Verdana, sans-serif",
  background:"eeeeee",
  stroke:"cccccc",
  text:"605050"
};
```

Note that, for the first time, we are differentiating between what we want the size of our canvas to be and the actual size of our chart (funnel/triangle). Another important thing to note is that, for our sample to work in its current format, our triangle height and width (base) must be the same.

3. Define a few global helper variables.

```
var wid;
var hei;
var totalPixels;
var totalData=0;
var pixelsPerData;
var currentTriangleHeight = chartInfo.height;
```

4. It's time for us to create our `init` function. This function will be doing most of the heavy lifting with the help of another function.

```
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  totalPixels = (chartInfo.height * chartInfo.width) / 2;
  for(var i in layers) totalData +=layers[i].amount;

  pixelsPerData = totalPixels/totalData;


  var context = can.getContext("2d");
  context.fillStyle = s.background;
  context.strokeStyle = s.stroke;

  context.translate(wid/2,hei/2 - chartInfo.height/2);

  context.moveTo(-chartInfo.width/2 , chartInfo.height);
  context.lineTo(chartInfo.width/2,chartInfo.height);
  context.lineTo(0,0);
  context.lineTo(-chartInfo.width/2 , chartInfo.height);

  for(i=0; i+1<layers.length; i++) findLine(context,
  layers[i].amount);

  context.stroke();
}
```

5. Our function performs the normal setup and executes the styling logic and then it creates a triangle, after which it finds the right points (by using the `findLine` function) at which we should cut the triangle:

```
function findLine(context,val){
  var newHeight = currentTriangleHeight;
  var pixels = pixelsPerData * val;
  var lines = parseInt(pixels/newHeight); //rounded

  pixels = lines*lines/2; //missing pixels

  newHeight-=lines;

  lines += parseInt(pixels/newHeight);
  currentTriangleHeight-=lines;

  context.moveTo(-currentTriangleHeight/2 ,
  currentTriangleHeight);
  context.lineTo(currentTriangleHeight/2,
  currentTriangleHeight);
}
```

This function finds the dots on our triangle based on the data of the current line. That's it; now its time to understand what we just did.

How it works...

After setting the code for lines in the `init` function, we are ready to start thinking about our triangle. First, we need to find out the total pixels that are within our triangle.

```
totalPixels = (chartInfo.height * chartInfo.width) / 2;
```

That is easy as we know our height and our width, so the formula is really simple. The next data point that is critical is the total amount of data. We can create a relationship between the pixels and the data.

```
for(var i in layers) totalData +=layers[i].amount;
```

As such, we loop through all the data layers and calculate the summary of all data points. At this stage, we are ready to find out the actual number of pixels. Each data element is equivalent to:

```
pixelsPerData = totalPixels/totalData;
```

After setting up the styles for our stroke and fill, we stop to think about the best translation that would help us build our triangle. For our triangle, I've picked the top edge to be the `0,0` point, after creating the triangle:

```
context.translate(wid/2,hei/2 - chartInfo.height/2);

context.moveTo(-chartInfo.width/2 , chartInfo.height);
context.lineTo(chartInfo.width/2,chartInfo.height);
context.lineTo(0,0);
context.lineTo(-chartInfo.width/2 , chartInfo.height);
```

The last two lines of our `init` function call the `findLine` method for each element in our `layers` array:

```
for(i=0; i+1<layers.length; i++) findLine(context, layers[i].amount);
context.stroke();
```

Time to dig into how the `findLine` function actually finds the points to create the lines. The idea is very simple. The basic idea is to try to find out how many lines it would take to complete the number of pixels in a triangle. As we are not building a math formula, we don't care if it's 100 percent accurate, but it should be accurate enough to work visually.

There's more...

Let's start with introducing color into our pallet.

```
var layers = [{label:"Completed Chapter 1", amount:23,
style:"#B1DDF3"},  {label:"Completed Chapter 2", amount:15,
style:"#FFDE89"},
  {label:"Completed Chapter 3", amount:11, style:"#E3675C"},
  {label:"Completed Chapter 4", amount:7, style:"#C2D985"},
  {label:"Completed Chapter 5", amount:3, style:"#999999"}];
```

OK, we are done with the easy part. Now, it's time to rework our logic.

Making findLine smarter

For us to be able to create a closed shape, we need to have a way to change the direction of the line drawn from right to left or from left to right and not have it go in one direction always. Beyond that, we are using `moveTo` right now and as such can never create a closed shape. What we actually want is to move our point and draw a line:

```
function findLine(context,val,isMove){
  var newHeight = currentTriangleHeight;
  var pixels = pixelsPerData * val;
  var lines = parseInt(pixels/newHeight); //rounded

  pixels = lines*lines/2; //missing pixels

  newHeight-=lines;

  lines += parseInt(pixels/newHeight);

  currentTriangleHeight-=lines;

  if(isMove){
    context.moveTo(currentTriangleHeight/2,
    currentTriangleHeight);
    context.lineTo(-currentTriangleHeight/2 ,
    currentTriangleHeight);
  }else{
    context.lineTo(-currentTriangleHeight/2 ,
    currentTriangleHeight);
    context.lineTo(currentTriangleHeight/2,
    currentTriangleHeight);
  }
}
```

Our next problem is that we don't want to change the actual triangle height as we will be calling this function more times than we did in the past. To come up with a plan for this problem, we need to extract some of the logic. We will return the new number of lines that were created, so that we can remove them externally from the triangle. This action enables us to have more finite control over visuals (a thing that will be important when we incorporate text).

```
function findLine(context,val,isMove){
  var newHeight = currentTriangleHeight;
  var pixels = pixelsPerData * val;
  var lines = parseInt(pixels/newHeight); //rounded

  pixels = lines*lines/2; //missing pixels

  newHeight-=lines;

  lines += parseInt(pixels/newHeight);

  newHeight = currentTriangleHeight-lines;

  if(isMove){
    context.moveTo(newHeight/2,newHeight);
    context.lineTo(-newHeight/2 , newHeight);
  }else{
    context.lineTo(-newHeight/2 , newHeight);
    context.lineTo(newHeight/2,newHeight);
  }


return lines;
}
```

At this stage, our `findLine` function is really smart and is capable of helping us to create closed shapes without controlling more than it needs to control (as it isn't changing any of our global data).

Changing the logic in init to create shapes

Now that we have a smart `findLine` function, it's time for us to rewrite our logic related to drawing lines in the `init` function.

```
var secHeight = 0;
  for(i=0;i<layers.length-1; i++){
    context.beginPath();
    findLine(context, 0,true);
    secHeight = findLine(context, layers[i].amount);
```

```
    currentTriangleHeight -= secHeight;
    context.fillStyle = layers[i].style;
    context.fill();
  }

  context.beginPath();
  findLine(context, 0,true);
  context.lineTo(0,0);
  context.fillStyle = layers[i].style;
  context.fill();
```

First, we draw all elements in our loop, minus the last one (as our last element is actually a triangle and not a line). Then, to help us hide our mathematical inaccuracy, we create a new path each time our loop starts and call our `findLine` function first with no new data (drawing a line in the last place where it drew a line as there is no data) and then drawing a second line, this time with the real new data.

Our exception to the rule is created out of the loop, and there, we just manually draw our shape, starting with the last line, and add the `0,0` point into it, over our triangle.

Adding text into our graph

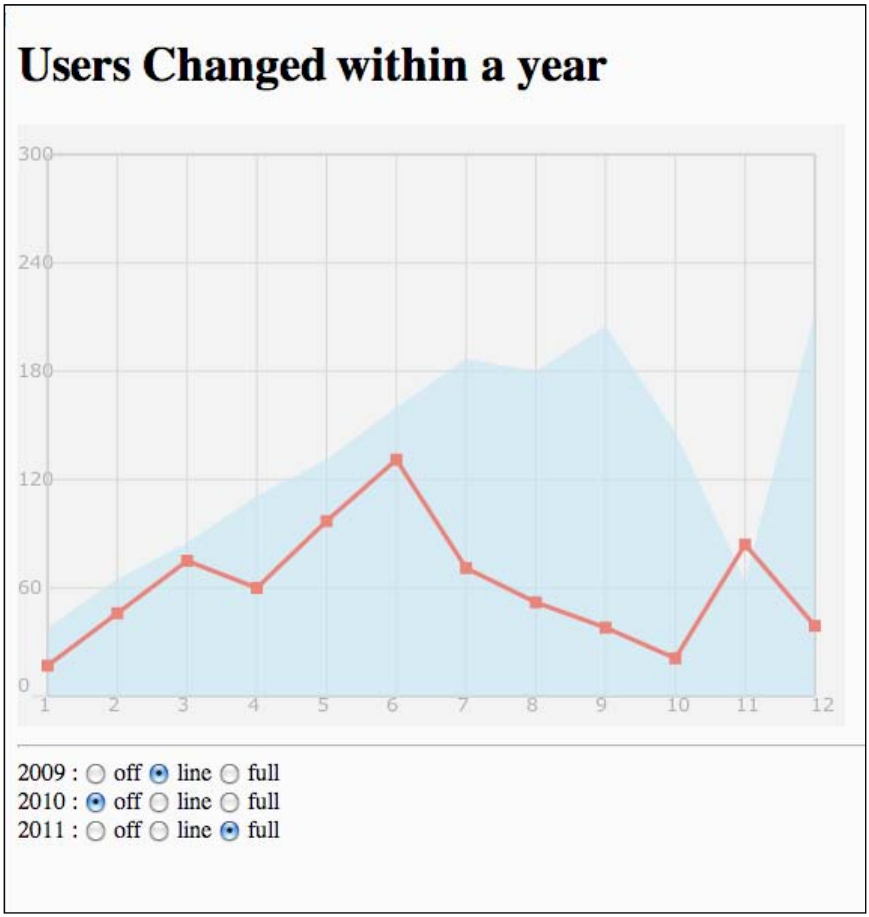
This one will be simple, as we are already getting back the line count before we resize our triangle. We can use this data to calculate where we want to position our textfield variable, so let's do it:

```
var secHeight = 0;
  for(i=0;i<layers.length-1; i++){
    context.beginPath();
    findLine(context, 0,true);
    secHeight = findLine(context, layers[i].amount);
    currentTriangleHeight -= secHeight;
    context.fillStyle = layers[i].style;
    context.fill();
    context.fillStyle = s.text;
    context.fillText(layers[i].label, currentTriangleHeight/2
    +secHeight/2, currentTriangleHeight+secHeight/2);
  }

  context.beginPath();
  findLine(context, 0,true);
  context.lineTo(0,0);
  context.fillStyle = layers[i].style;
  context.fill();
  context.fillStyle = s.text;
  context.fillText(layers[i].label, currentTriangleHeight/2 ,
  currentTriangleHeight/2);
```

Just see the difference between drawing our text in the loop and out of it. As we don't get new line data out of the loop, we need to change the point logic by using the total size of our leftover triangle.

Revisiting lines: making the line chart interactive

In this recipe, we will travel back in time to one of our earlier recipes, *Building line charts in Chapter 3*, *Creating Cartesian-based Graphs* and add some user control to it. This control enables the user to turn on and off the streams of data.



Getting ready

The first step that you will need to take is to grab the source code from *Chapter 3*, *Creating Cartesian-based Graphs*. We will rename `03.05.line-revamp.html` and `03.05.line-revamp.js` to `05.02.line-revisit`.

Now that we have our files up to date, add our HTML file—three radio groups to represent the three data sources (years 2009, 2010, and 2011).

```
<hr/>

  2009 : <input type="radio" name="i2009" value="-1" /> off
    <input type="radio" name="i2009" value="0" /> line
    <input type="radio" name="i2009" value="1" select="1" />
    full<br/>
  2010 : <input type="radio" name="i2010" value="-1" /> off
    <input type="radio" name="i2010" value="0" /> line
    <input type="radio" name="i2010" value="1" select="1" />
    full<br/>
   2011 : <input type="radio" name="i2011" value="-1" /> off
    <input type="radio" name="i2011" value="0" /> line
    <input type="radio" name="i2011" value="1" select="1" />
    full<br/>
```

Note that I've named each radio group with "i" added to the year and set the possible values to be `-1`, `0`, or `1`.

How to do it...

Perform the following steps:

1. Create a few constants (well, variables that are not going to change), and set the following three lines, now that the default values have already been assigned:

```
var HIDE_ELEMENT = -1;
var LINE_ELEMENT = 0;
var FILL_ELEMENT = 1;

var elementStatus={ i2009:FILL_ELEMENT,
  i2010:FILL_ELEMENT,
  i2011:FILL_ELEMENT};
```

2. Time to move the logic of creating the chart into a separate function. Everything after the initalization of our canvas is going to be moved out.

```
var context;

function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  context = can.getContext("2d");

  drawChart();
}
```

3. Update the radio boxes to highlight whatever is currently selected and to add the `onchange` events to all radio buttons.

```
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  context = can.getContext("2d");

  drawChart();

  var radios ;
    for(var id in elementStatus){
      radios = document.getElementsByName(id);
      for (var rid in radios){
      radios[rid].onchange = onChangedRadio;
      if(radios[rid].value == elementStatus[id] )
      radios[rid].checked = true;
    }
  }

}
```

4. Make some updates in our `drawChart` function. Our goal is to incorporate the new controller `elementStatus` into the drawing of lines.

```
function drawChart(){
  context.lineWidth = 1;
  context.fillStyle = "#eeeeee";
  context.strokeStyle = "#999999";
  context.fillRect(0,0,wid,hei);
```

```
  context.font = "10pt Verdana, sans-serif";
  context.fillStyle = "#999999";

  context.moveTo(CHART_PADDING,CHART_PADDING);
  context.rect(CHART_PADDING,CHART_PADDING,wid-
  CHART_PADDING*2,hei-CHART_PADDING*2);
  context.stroke();
  context.strokeStyle = "#cccccc";
  fillChart(context,chartInfo);


  if(elementStatus.i2011>-1)
  addLine(context,formatData(a2011,   "/2011","#B1DDF3"),"#B1DDF3"
,elementStatus.i2011==1);
  if(elementStatus.i2010>-1)
  addLine(context,formatData(a2010,
  "/2010","#FFDE89"),"#FFDE89",elementStatus.i2010==1);
  if(elementStatus.i2009>-1)
  addLine(context,formatData(a2009,
  "/2009","#E3675C"),"#E3675C",elementStatus.i2009==1);

}
```

5. Last but not least, let's add the logic into our `onChangedRadio` function.

```
function onChangedRadio(e){
  elementStatus[e.target.name] = e.target.value;
  context.clearRect(0,0,wid,hei);
  context.beginPath();
  drawChart();
}
```

That's it! We just added user interaction into our chart.

How it works...

We haven't planned for user interaction on this chart in advance. As such, we need to revisit it to change some of the logic. When Canvas draws something, that's it, it's there forever! We can't just delete an object, as there are no objects in Canvas, and as such, we need a way to redraw on demand. To accomplish that, we need to extract all the drawing logic from the `init` function and create the `drawChart` function. Besides adding our logic to the end of the function, we also need to add the start of the function:

```
context.lineWidth = 1;
```

Although we originally worked out the default to use as the width for our background, in a second redraw, our canvas would still have stored its last size (in our case it could be `3`), and as such, we reset it to the original value.

We are using an object called `elementStatus` to store the current status of each line on our chart. The values it can store are as follows:

- `-1`: Do not draw
- `0`: Draw a line with no fill
- `1`: Draw a fill

As such, we are adding the following logic into the end of our function:

```
if(elementStatus.i2011>-1) addLine(context,formatData(a2011, "/2011","
#B1DDF3"),"#B1DDF3",elementStatus.i2011==1);
```

As the logic repeats three times, let's just focus on one of them. If we want, we can use our constant variables to make the logic easier to view.

```
if(elementStatus.i2011!=HIDE_ELEMENT)
  addLine(context,formatData(a2011,
  "/2011","#B1DDF3"),"#B1DDF3",elementStatus.i2011==FILL_ELEMENT);
```

The logic breaks down into a first `if` statement, testing to see if our content should be hidden. If we establish that this line should be added, we draw it by sending into the fill/line parameter the outcome of comparing our current value to `FILL_ELEMENT`, resulting in two variations based on the outcome of this operation.

There's more...

Unfortunately, because we are not using any open source library, the built-in HTML capabilities don't allow us to set events to groups of radios, and as such, we need to find them all and add the `onchange` event to them using the IDs we are storing in our `elementStatus` controller.

```
var radios ;
  for(var id in elementStatus){
    radios = document.getElementsByName(id);
    for (var rid in radios){
      radios[rid].onchange = onChangedRadio;
      if(radios[rid].value == elementStatus[id] )
      radios[rid].checked = true;
    }

  }
```

Pay attention to the highlighted code. Here, we are checking to see whether our current radio button's value matches our element value in `elementStatus`. If it does, it means that the radio button will be selected.

Breaking down the logic of onChangedRadio

Let's take another peek at the logic in this function:

```
elementStatus[e.target.name] = e.target.value;
```

The first thing we do is save the newly selected value into our `elementStatus` controller.

```
context.clearRect(0,0,wid,hei);
```

We follow that by deleting everything from our canvas.

```
context.beginPath();
```

Next, wipe the slate clean and start with a new path.

```
drawChart();
```

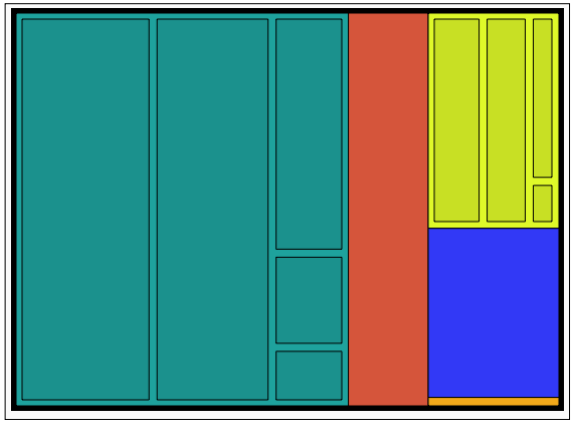
And... you guessed it... Then start drawing everything all over, and our new parameter in `elementStatus` will validate that the right things will be drawn.

See also

- The *Building line charts* recipe in *Chapter 3*, *Creating Cartesian-based Graphs*

Tree mapping and recursiveness

Tree mapping enables us to see in-depth data from a bird's-eye view. Contrary to comparative charts—such as most of the charts that we have created until now—tree mapping displays tree structured data as a set of nested rectangles, enabling us to visualize their quantitative nature and relationship.



Let's start with a tree mapping that showcases only one level of information.

Getting ready

We will start our application with the number of people in the world, in millions, divided by continent (based on public data from 2011).

```
var chartData = [
  {name: "Asia", value:4216},
  {name: "Africa",value:1051},
  {name: "The Americas and the Caribbean", value:942},
  {name: "Europe", value:740},
  {name: "Oceania", value:37}
];
```

We will update this data source later in our example, so keep in mind that this dataset is temporary.

How to do it...

We will start by creating a simple, working, flat tree chart. Let's jump right into it and figure out the steps involved in creating the tree map:

1. Let's add a few helper variables on top of our dataset.

```
var wid;
var hei;
var context;
var total=0;
```

2. Create the `init` function.

```
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  context = can.getContext("2d");

  for(var item in chartData) total +=
  chartData[item].value;

  context.fillRect(0,0,wid,hei);
  context.fillStyle = "RGB(255,255,255)";
  context.fillRect(5,5,wid-10,hei-10);
  context.translate(5,5);
  wid-=10;
  hei-=10;

  drawTreeMap(chartData);

}
```

3. Create the function `drawTreeMap`.

```
function drawTreeMap(infoArray){
  var percent=0;
  var cx=0;
  var rollingPercent = 0;
  for(var i=0; i<infoArray.length; i++){
    percent = infoArray[i].value/total;
    rollingPercent +=percent
    context.fillStyle =
    formatColorObject(getRandomColor(255));
    context.fillRect(cx,0 ,wid*percent,hei);
    cx+=wid*percent;
    if(rollingPercent > 0.7) break;

  }

  var leftOverPercent = 1-rollingPercent;
  var leftOverWidth = wid*leftOverPercent;
  var cy=0;
  for(i=i+1; i<infoArray.length; i++){
    percent = (infoArray[i].value/total)/leftOverPercent;
    context.fillStyle =
    formatColorObject(getRandomColor(255));
    context.fillRect(cx,cy ,leftOverWidth,hei*percent);
    cy+=hei*percent;
  }

}
```

4. Create a few formatting functions to help us create a random color for our tree map block.

```
function formatColorObject(o){
  return "rgb("+o.r+","+o.g+","+o.b+")";
}

function getRandomColor(val){
  return
  {r:getRandomInt(255),g:getRandomInt(255),
  b:getRandomInt(255)};
}

function getRandomInt(val){
  return parseInt(Math.random()*val)+1
}
```

There is a bit of overkill in the creation of so many formatting functions; their main goal is to help us when we are ready for the next step—to create more depth in our data (refer to the *There's more...* section in this recipe for more details).

How it works...

Let's start with the initial idea. Our goal is to create a map that will showcase the bigger volume areas inside our rectangular area and leave a strip on the side for the smaller areas. So, let's start with our `init` function. Our first task beyond our basic getting started work is to calculate the actual total. We do that by looping through our data source, thus:

```
for(var item in chartData) total += chartData[item].value;
```

We continued with some playing around with the design and making our work area 10 pixels smaller than our total canvas size.

```
CONTEXT.FILLRECT(0,0,WID,HEI);
CONTEXT.FILLSTYLE = "RGB(255,255,255)";
CONTEXT.FILLRECT(5,5,WID-10,HEI-10);
CONTEXT.TRANSLATE(5,5);
WID-=10;
HEI-=10;

drawTreeMap(chartData);
```

It's time to take a look into how our `drawTreeMap` function works. The first thing to notice is that we send in an array instead of working directly with our data source. We do that because we want to be open to the idea that this function will be re-used when we start building the inner depths of this visualization type.

```
function drawTreeMap(infoArray){...}
```

We start our function with a few helper variables (the `percent` variable will store the current `percent` value in a loop). The `cx` (the current x) position of our rectangle and `rollingPercent` will keep track of how much of our total chart has been completed.

```
var percent=0;
var cx=0;
var rollingPercent = 0;
```

Time to start looping through our data and drawing out the rectangles.

```
for(var i=0; i<infoArray.length; i++){
  percent = infoArray[i].value/total;
  rollingPercent +=percent
  context.fillStyle =
  formatColorObject(getRandomColor(255));
  context.fillRect(cx,0 ,wid*percent,hei);
  cx+=wid*percent;
```

Before we complete our first loop, we will test it to see when we cross our threshold (you are welcome to play with that value). When we reach it, we need to stop the loop, so that we can start drawing our rectangles by height instead of by width.

```
if(rollingPercent > 0.7) break;
}
```

Before we start working on our boxes, which take the full leftover width and expand to the height, we need a few helper variables.

```
var leftOverPercent = 1-rollingPercent;
var leftOverWidth = wid*leftOverPercent;
var cy=0;
```

As we need to calculate each element from now on based on the amount of space left, we will figure out the value (`leftOverPercent`), and then we will extract the remaining width of our shape and start up a new `cy` variable to store the current y position.

```
for(i=i+1; i<infoArray.length; i++){
  percent = (infoArray[i].value/total)/leftOverPercent;
  context.fillStyle = formatColorObject(getRandomColor(255));
  context.fillRect(cx,cy ,leftOverWidth,hei*percent);
  cy+=hei*percent;
}
```

We start our loop with one value higher than what we left off (as we broke out of our earlier loop before we had a chance to update its value and draw to the height of our remaining area.

Note that in both loops we are using `formatColorObject` and `getRandomColor`. The breakdown of these functions was created so that we can have an easier way to manipulate the colors returned in our next part.

There's more...

For our chart to really have that extra kick, we need to have a way to make it capable of showing data in at least a second lower-level details of data. To do that, we will revisit our data source and re-edit it:

```
var chartData = [
  {name: "Asia", data:[
    {name: "South Central",total:1800},
    {name: "East",total:1588},
    {name: "South East",total:602},
    {name: "Western",total:238},
    {name: "Northern",total:143}
  ]},
  {name: "Africa",total:1051},
```

```
    {name: "The Americas and the Caribbean", data:[
      {name: "South America",total:396},
      {name: "North America",total:346},
      {name: "Central America",total:158},
      {name: "Caribbean",total:42}
    ]},
    {name: "Europe", total:740},
    {name: "Oceania", total:37}
];
```

Now we have two regions of the world with a more in-depth view of their subregions. It's time for us to start modifying our code, so that it will work again with this new data.

Updating the init function – recalculating the total

The first step we need to carry out in the `init` function is to replace the current total loop with a new one that can dig deeper into elements to count the real total.

```
var val;
var i;
for(var item in chartData) {
  val = chartData[item];
  if(!val.total && val.data){
    val.total = 0;
    for( i=0; i<val.data.length; i++)
    val.total+=val.data[i].total;
  }

  total += val.total;
}
```

In essence, we are checking to see whether there is no total and whether there is a data source. If that is the case, we start a new loop to calculate the actual total for our elements—a good exercise for you now would be to try to make this logic into a recursive function (so that you can have more layers of data).

Next, we will change `drawTreeMap` and get it ready to become a recursive function. To make that happen, we need to extract the global variables from it and send them in as parameters of the function.

```
drawTreeMap(chartData,wid,hei,0,0,total);
```

Turning drawTreeMap into a recursive function

Let's update our function to enable recursive operations. We start by adding an extra new parameter to capture the latest color.

```
function drawTreeMap(infoArray,wid,hei,x,y,total,clr){
  var percent=0;
  var cx=x ;
  var cy=y;

  var pad = 0;
  var pad2 = 0;

  var rollingPercent = 0;
  var keepColor = false;
  if(clr){ //keep color and make darker
    keepColor = true;
    clr.r = parseInt(clr.r *.9);
    clr.g = parseInt(clr.g *.9);
    clr.b = parseInt(clr.b *.9);
    pad = PAD*2;
    pad2 = PAD2*2;
  }
```

If we pass a `clr` parameter, we need to keep that color throughout all the new rectangles that will be created, and we need to add a padding around the shapes so that it becomes easier to see them. We make the color a bit darker as well by subtracting 10 percent of its color on all its RGA properties.

The next stage is to add the padding and recursive logic.

```
for(var i=0; i<infoArray.length; i++){
  percent = infoArray[i].total/total;
  rollingPercent +=percent
  if(!keepColor){
    clr = getRandomColor(255);
  }

  context.fillStyle = formatColorObject(clr);
  context.fillRect(cx+pad ,cy+pad ,wid*percent - pad2,hei-pad2);
  context.strokeRect(cx+pad ,cy+pad ,wid*percent - pad2,hei-
  pad2);
  if(infoArray[i].data){
    drawTreeMap(infoArray[i].data,parseInt(wid*percent -
    PAD2),hei - PAD2,cx+ PAD,cy + PAD,infoArray[i].total,clr);
  }
```

```
    cx+=wid*percent;
    if(rollingPercent > 0.7) break;

  }
```

The same logic is then implemented on the second loop as well (to see it check the source files).

Turning the data and total to recursive data

Let's start by updating our tree data to be really recursive (for the full dataset please refer to the source code).

```
…
{name: "Asia", data:[
  {name: "South Central",total:1800},
  {name: "East",total:1588},
  {name: "South East",total:602},
  {name: "Western",total:238},
  {name: "Northern",data:[{name: "1",data:[
    {name: "2",total:30},
    {name: "2",total:30}
  ]},
  {name: "2",total:53},
  {name: "2",total:30}
]}  …
```

Now, with a tree map that has over four levels of information, we can revisit our code and finalize our last outstanding issue validating that our total is always up-to-date at all levels. To fix that, we will extract the logic of calculating the total into a new function and update the `total` line in the `init` function.

```
function init(){
  var can = document.getElementById("bar");

  wid = can.width;
  hei = can.height;
  context = can.getContext("2d");

  total = calculateTotal(chartData); //recursive function
...
```

Time to create this magical (recursive) function.

```
function calculateTotal(chartData){
  var total =0;
  var val;
  var i;
  for(var item in chartData) {
```

```
        val = chartData[item];
        if(!val.total && val.data)
          val.total = calculateTotal(val.data);

        total += val.total;
      }

  return total;

  }
```

The logic is really similar to what it was, with the exception that all the data entries are internal to the function, and each time there is a need to deal with another level of data, it's re-sent to the same function (in a recursive way) until all data is resolved—until it returns the total.

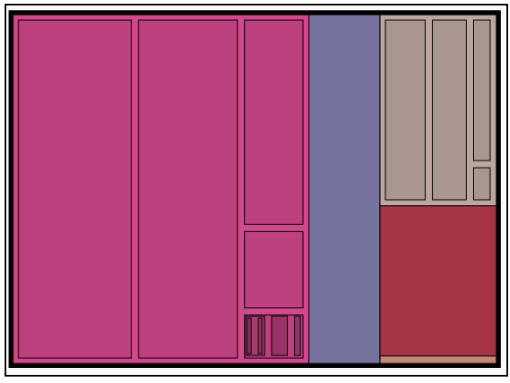
See also

- The *Adding user interaction into tree mapping* recipe

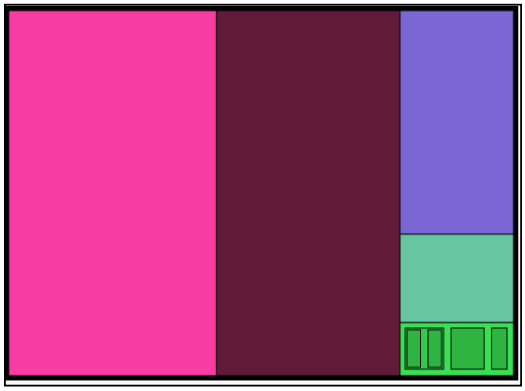
Adding user interaction into tree mapping

Until now, we have limited our user interaction with our samples. In one of our last samples, we added a controlled way to add and remove chart elements; in this one, we will enable the user to dig deeper into the chart and see more details by creating a truly endless experience (if we only had an endless amount of data to dig into).

In the following image, on the left-hand side, you can see the initial state and what happens after one click of the user (the chart redraws itself to showcase the area that was clicked on).



Consider the case when the user clicks on the chart (for example, the next picture is generated by clicking on the left-hand side rectangle—the tree map will update and zoom into that area).



Getting ready

To get this sample right, you will need to start from our last recipe, *Tree maping and recursiveness*, and adjust it to work for this sample.

How to do it...

This is our first sample where we make our canvas area interactive. In the next few steps, we will add some logic from the last sample into our recipe, to enable the user to zoom into or out of it:

1. Add a new global variable,

```
var currentDataset;
```

2. Store the current data that is sent to the tree mapping function.

```
currentDataset = chartData;
drawTreeMap(chartData,wid,hei,0,0,total);
```

3. Add a `click` event to our canvas area.

```
can.addEventListener('click', onTreeClicked, false);
```

4. Create the `onTreeClick` event.

```
function onTreeClick(e) {
  var box;
  for(var item in currentDataset){
    if(currentDataset[item].data){
      box = currentDataset[item].box;
```

```
            if(e.x>= box.x && e.y>= box.y &&
            e.x<= box.x2 && e.y<= box.y2){
              context.clearRect(0,0,wid,hei);
              drawTreeMap(currentDataset[item].data,wid,
              hei,0,0,currentDataset[item].total);
              currentDataset = currentDataset[item].data;

            break;
            }

        }
      }
    }
```

5. Draw a rectangle twice—within `drawTreemap`—for the first time in the first loop and again in the second loop. Let's replace it with an external function—replace both the `for` loop lines to draw a rectangle with:

```
drawRect(cx+pad ,cy+pad ,wid*percent – pad2,hei-
pad2,infoArray[i]);
```

6. Time to create the rectangle function.

```
function drawRect(x,y,wid,hei,dataSource){
  context.fillRect(x,y,wid,hei);
  context.strokeRect(x,y,wid,hei);
  dataSource.box = {x:x,y:y,x2:x+wid,y2:y+hei};

}
```

There you go! We have a fully functional, deep-level, endless interaction with the user (just dependent on how much data we have).

How it works...

The Canvas element doesn't currently support a smart way to interact with objects. As there are no objects in the canvas, as soon as you create the element it tunes into a bitmap and its information is removed from memory. Luckily for us, our sample is constructed out of rectangles, making it much easier to recognize when our element is clicked on. We will need to store in memory the current box location of each element that we draw.

As such, our first step of logic is the last thing that we did in our procedure (in step 6). We want to capture the points that construct our rectangles, so then in our `click` event we can figure out where our dot is in relation to the rectangle:

```
function onTreeClick(e) {
   var box;
  for(var item in currentDataset){
    if(currentDataset[item].data){
```

We loop through our data source (current one) and check to see whether the element we are currently in has a data source (that is, children); if it does, we continue, and if not, we will skip to the next element to test it as well.

Now that we know our element has children, we are ready to see if our dot is in the range of our element.

```
box = currentDataset[item].box;
if(e.x>= box.x && e.y>= box.y &&
   e.x<= box.x2 && e.y<= box.y2){
```

If it is, we are ready to redraw the tree map and replace our current dataset with the current deeper dataset.

```
context.clearRect(0,0,wid,hei);
drawTreeMap(currentDataset[item].data,wid,hei,0,0,currentDataset[it
em].total);
currentDataset = currentDataset[item].data;

break;
```

We then exit from the loop (by using a `break` statement). Please note that the last thing we do is update `currentDataset`, as we still need information from it to send the total data into `drawTreeMap`. When we have finished using it, we are ready to override it with the new dataset (what were the children before turn into our main players for the next round).

There's more...

Currently, there is just no way to get back without refreshing everything. So, let's add it to our logic that if the user clicks in an element with no children, we will revert to the original map.

Going back to the main treemap

Let's add the following code into the `click` event:

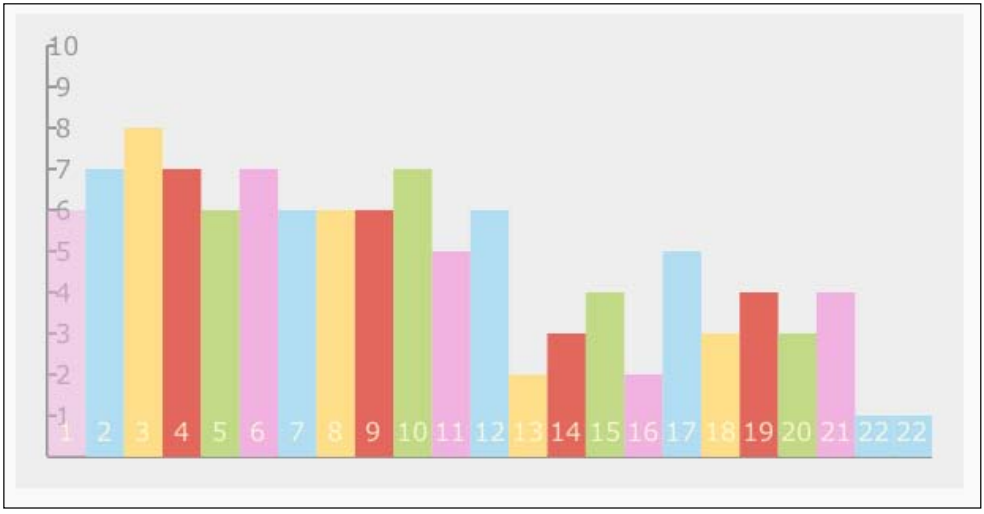
```
function onTreeClick(e) {
   var box;
  for(var item in currentDataset){
    if(currentDataset[item].data){
      box = currentDataset[item].box;
      if(e.x>= box.x && e.y>= box.y &&
      e.x<= box.x2 && e.y<= box.y2){
        context.clearRect(0,0,wid,hei);
        drawTreeMap(currentDataset[item].data,wid,
        hei,0,0,currentDataset[item].total);
        currentDataset = currentDataset[item].data;
```

```
            break;
            }

          }else{
            currentDataset = chartData;
            drawTreeMap(chartData,wid,hei,0,0,total);

          }
        }
    }
```

Fantastically done! We have just finished creating a fully interactive experience for our users, and now it's in your hands to make this look a bit better. Add some rollover labels and all the visualization that will make your chart visually pleasing and will help understanding.

Making an interactive click meter

In this next example, we will focus on one more powerful feature of any client-side programming—the ability to interact with the user and the ability to update data dynamically. To keep it simple, let's revisit an old chart—the bar chart from *Chapter 3*, *Creating Cartesian-based Graphs*—and integrate a counter that will count how many times a user clicks on an HTML document in any given second and update the chart accordingly.



How to do it...

Most of the steps are going to be familiar, if you have worked on the bar chart from *Chapter 3*, *Creating Cartesian-based Graphs*. So, let's run through them and then focus on the new logic:

1. Let's create some helper variables.

```
var currentObject = {label:1,
  value:0,
  style:"rgba(241, 178, 225, .5)"};
  var colorOptions = ["rgba(241, 178, 225,
  1)","#B1DDF3","#FFDE89","#E3675C","#C2D985"];

  var data = [];


var context;
var wid;
var hei;
```

2. Follow this with our `init` function.

```
function init(){

  var can = document.getElementById("bar");
  wid = can.width;
  hei = can.height;

  context = can.getContext("2d");


  document.addEventListener("click",onClick);
  interval = setInterval(onTimeReset,1000);
  refreshChart();
}
```

3. Now it's time to create the `onTimeReset` function.

```
function onTimeReset(){
  if(currentObject.value){
    data.push(currentObject);
    if(data.length>25) data = data.slice(1);
    refreshChart();
  }
  currentObject = {label:currentObject.label+1, value:0,
  style: colorOptions[currentObject.label%5]};

}
```

4. The next step is to create the `onClick` listener.

```
function onClick(e){
  currentObject.value++;
  refreshChart();
}
```

5. Now create the `refreshChart` function.

```
function refreshChart(){
  var newData = data.slice(0);
  newData.push(currentObject);

  drawChart(newData);
}
```

6. Last but not least, let's create `drawChart` (most of its logic is the same as for the `init` function discussed in *Chapter 3*, *Creating Cartesian-based Graphs*).

```
function drawChart(data){
  context.fillStyle = "#eeeeee";
  context.strokeStyle = "#999999";
  context.fillRect(0,0,wid,hei);

  var CHART_PADDING = 20;

  context.font = "12pt Verdana, sans-serif";
  context.fillStyle = "#999999";


  context.moveTo(CHART_PADDING,CHART_PADDING);
  context.lineTo(CHART_PADDING,hei-CHART_PADDING);
  context.lineTo(wid-CHART_PADDING,hei-CHART_PADDING);

  var stepSize = (hei - CHART_PADDING*2)/10;
  for(var i=0; i<10; i++){
    context.moveTo(CHART_PADDING, CHART_PADDING + i*
    stepSize);
    context.lineTo(CHART_PADDING*1.3,CHART_PADDING + i*
    stepSize);
    context.fillText(10-i, CHART_PADDING*1.5,
    CHART_PADDING + i*     stepSize + 6);
  }
  context.stroke();
```

```
    var elementWidth =(wid-CHART_PADDING*2)/ data.length;
    context.textAlign = "center";
    for(i=0; i<data.length; i++){
      context.fillStyle = data[i].style;
      context.fillRect(CHART_PADDING +elementWidth*i ,hei-
      CHART_PADDING - data[i].value*stepSize,
      elementWidth,data[i].value*stepSize);
      context.fillStyle = "rgba(255, 255, 225, 0.8)";
      context.fillText(data[i].label, CHART_PADDING
      +elementWidth*(i+.5), hei-CHART_PADDING*1.5);

    }
  }
```

That's it! We have an interactive chart that will be updated every second, depending on how many times you manage to click your mouse in 1 second—I assume no one can click more than 10 times a second but I've managed to get there (when using two hands).

How it works...

Let's focus on the breakdown of the data variables in *Chapter 3*, *Creating Cartesian-based Graphs*. We had all our data ready inside our data object. This time around, we are keeping the data object empty, and instead, we have one data line in a separate variable.

```
var currentObject = {label:1,
  value:0,
  style:"rgba(241, 178, 225, .5)"};
var data = [];
```

Each time the user clicks, we update the counter for `currentObject` and refresh the chart thus making the user experience more dynamic and live.

```
function onClick(e){
  currentObject.value++;
  refreshChart();
}
```

We set the interval in the `init` function as follows:

```
interval = setInterval(onTimeReset,1000);
```

Every time a second passes, the function checks whether the user had any clicks in that time interval, and if they did, it ensures that we push `currentObject` into the dataset. If the size of the dataset is greater than `25`, we cut the first item out of it and we refresh the chart. No matter what we create, a new empty object is labeled with a new label showing the current time in seconds.

```
function onTimeReset(){
  if(currentObject.value){
    data.push(currentObject);
    if(data.length>25) data = data.slice(1);
    refreshChart();
}
  currentObject = {label:currentObject.label+1, value:0, style:
  colorOptions[currentObject.label%5]};

}
```

One last thing that you should look at before we wrap this sample up is:

```
function refreshChart(){
  var newData = data.slice(0);
  newData.push(currentObject);

  drawChart(newData);

}
```

This part of our logic is really the glue that makes it possible for us to update our data every time a user clicks a button. The idea is we want to have a new array that will store the new data, but we do not want to the current element to be affected, so for that we are duplicating this data source by adding the new data object into it and then sending it off to create the chart.