**Session 2 - Using data in visualization**

This session shows how data can be manipulated in order to represent it in a Processing sketch. It shows what types of data are most easily visualized, and introduces some simple data processing required to convert data into a format usable by Processing.

**By the end of this session you should be able to:**

* produce a sketch that displays data stored in a separate file
* find and use sources of information for getting Processing to do something
* create variables of type float, String and PImage
* link data and classes to your sketches by dragging them into Processing
* use the Table class to read and extract values from a tabular text file
* produce a simple map from a file containing locations and attributes
* acquire data from a source on the web

**Contents**

1. [2.1 Introduction](http://moodle.city.ac.uk/mod/page/view.php?id=580556#s2.1)
2. [2.2 Using image data](http://moodle.city.ac.uk/mod/page/view.php?id=580556#s2.2)
3. [2.3 Using tabular data](http://moodle.city.ac.uk/mod/page/view.php?id=580556#s2.3)
4. [2.4 Using service APIs](http://moodle.city.ac.uk/mod/page/view.php?id=580556#s2.4)
5. [2.5 Conclusions](http://moodle.city.ac.uk/mod/page/view.php?id=580556#s2.5)

**2.1 Introduction**

Before reading on, you must make sure you have completed the previous exercises and checked you have met the learning outcomes from the last session. You should also read the [feedback on the Session 1 exercises](http://moodle.city.ac.uk/mod/page/view.php?id=580540) if you have not done so already. Each week we will be building upon ideas and skills developed in the previous sessions so it is important not to fall behind.

We saw last week that one of the important motivations for data visualization was to *start with some data to answer a question* (Fry) and to consider first *what* it is we wish to visualize (Munzner). We also saw that the 7-stage data visualization workflow starts with the *acquire* stage where we obtain the data we wish to visualize. So the emphasis of this session will be on the data acquisition process considering the types of data sources available to us, how we go about getting data in a format we can process and how the characteristics of those data will influence the visualization approach we take. Most of the session therefore fits into the '*what?*' part of Munzner's visualization framework, and in particular the data abstraction process (see Munzner, 2015, Ch.2 for an introduction).

To illustrate the diversity of data-driven questions for which visualization might offer a solution, let's first consider a couple of contrasting domains and tasks.

**Patterns of Collaborative Software Development**

How do people collaborate when working on large open source programming projects? This is an important question when considering how to mange collaborative software engineering. Large projects can involve hundreds of contributors over many years contributing and editing many millions of lines of code. The [code\_swarm](http://code.google.com/p/codeswarm/) project is itself an open source project that uses Processing to visualize contributions to software projects. The data used to generate the visualization are provided by the *revision control software* (Git, SVN etc.) that keeps track of all source code and documentation changes. By showing these changes visually and animating changes over time, we get to see the complex patterns of collaboration.

The example above shows 10 years of development of the software *Eclipse*.

Does this visualization work? Would it help answer questions about the nature of software collaboration? Can you detect trends? outliers? make comparisons? What elements contribute to its success or otherwise? What kinds of other collaborative activity might benefit from such visualization?

**Global Development and Health**

Hans Rosling has made a name for himself and his [GapMinder](http://www.gapminder.org/) project by using data visualization to show how global patterns of development and health have changed over time. The example below (and the one made famous by his [2006 TED talk](http://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen.html)) uses a simple scatterplot to relate economic development and health, but importantly uses animation to show how this relationship varies over space and time.

This data-rich example shows data visualization used for communication rather than exploration, helped in part by the dynamic presentation of Rosling. But the data are sufficiently rich to invite further questions (e.g. what's causing those apparent rapid fluctuations in life expectancy for some countries?). You can explore the data directly at [GapMinder World](http://www.gapminder.org/world/). We will also explore one of the datasets later on in this session.

**2.2 Using Image Data**

Not surprisingly when constructing graphical representations of data, images of various kinds form an important part of the user's experience. Sometimes those images are constructed directly out of numeric data and user interactions (as we saw last week). Alternatively, existing image data may be incorporated into the visualization. These might be photographs or pre-compiled graphics.

Processing makes it very easy to load and display image data in the standard .png, .jpg and .gif formats. Two commands in particular are useful, [loadImage()](http://processing.org/reference/loadImage_.html) for retrieving image data from a file or the web and [image()](http://processing.org/reference/image_.html) for displaying an image in a sketch.

Here's a simple Processing sketch to load an image from the web and display it.

// Draws an image in a sketch.

// Jo Wood, 26th January, 2016

PImage img; // This is a variable holding an entire image.

void setup()

{

size(400,150);

// Load the image from a remote URL.

img = loadImage("http://max-planck-research-networks.net/img/gallery/a01.jpg");

}

void draw()

{

background(255);

// Draw the image to fit inside the sketch.

image(img,0,0,width,height);

}

As we have seen previously, the sketch consists of a setup() method that does the once-only initialisation of the sketch and a repeat many times draw() method that does the drawing of the sketch. In this example setting up involves loading an image from a remote URL and storing the loaded image inside a variable that we've named img. This variable can be treated just like we did for the numeric float variables we saw last week, but in this case instead of a float type, the variable is a PImage type.

We draw the contents of the PImage variable inside the draw() method using the command image(). This command can take 5 parameters. The first is the variable storing the image to draw (img in this example). The next four are *x,y,w,h*where *x* and *y* are the pixel coordinates of the top-left position in which the image is to be drawn and the optional *w* and*h* define the size at which you wish the image to appear. In other words, the last four parameters are just like the ones we used last week to draw a rectangle with the rect() command.

In this example we draw the image starting at the top left corner (0,0) and set *w* and *h* to be the sketch's own width and height to ensure it takes up the full window size.

Note that because we put the image into the img variable inside the setup() method and we draw it in the draw()method, img needs to be visible to both methods. We therefore declare it at the top of the code outside the braces of both methods.

[](http://staff.city.ac.uk/~jwo/datavis/session02/images/imageSketch1.jpg)

Output from the sketch (click to enlarge).

**Variable Scope**   
  
A variable's *scope* is the degree to which its name and contents are visible within your sketch. There are two common scopes used in Processing sketches - *method variables* and *sketch variables*. Scope is controlled by where in your sketch the variable is *declared*. That is where the line of code that first names the variable and its type (e.g. float radius orPImage img) is placed.   
  
A *method variable*, as the name suggests is one that is declared inside the curly braces that define a method such as setup() or draw(). It can only be used inside the same method in which it was declared. If you try to use it outside of the method, Processing will complain that it cannot find the variable. Try it and see.   
  
In contrast, a *sketch variable* is 'global' in the sense that it can be used at any point in your sketch. To create a sketch variable, simply declare it outside of any methods, usually near the top of your sketch code, such as PImage img; in the example above.   
  
As a general rule, you should make all variables method variables unless you have good reason to allow them to be shared by more than one method, in which case you declare them at the top of your code as sketch variables.

We can treat an image variable just like any other drawn element. So for example, we could move an image around our sketch according to the current mouse position. The example below shows a simple example.

// Draws a small image at the current mouse position.

// Jo Wood, 26th January, 2016

PImage img;

void setup()

{

size(400,320);

// Load the image from a local file.

img = loadImage("sphere.png");

// Position image so it is centred at the given coordinates.

imageMode(CENTER);

}

void draw()

{

background(255);

// Draw the image centred on the current mouse position.

image(img,mouseX,mouseY);

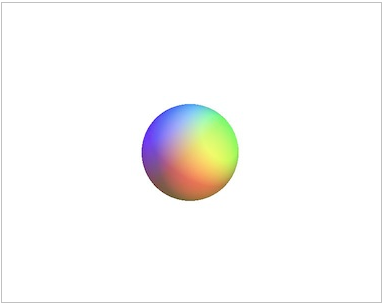
}

In this example, we use a slightly shorter version of image() which instead of drawing the image at (0,0) [top-left of sketch], we draw it at the current mouse location as stored in the variables mouseX and mouseY. Without specifying a width and height, Processing will draw the image at its natural scale.

Note also that in this case we load the image not from a remote URL but a local file called sphere.png. Where does Processing look to find such files? The answer is that it will always look for a folder called data in the same location as the sketch code. An easy way to place data in the data folder, simply drag the file into your sketch code in Processing.

Have a go creating the example sketch above. Start Processing and copy the code above into it. Save the sketch as ImageSketch somewhere on your computer. Now download [sphere.png](http://staff.city.ac.uk/~jwo/datavis/session02/imageSketch2/data/sphere.png) (right-click on this link and Save as...). Try dragging the saved image file into Processing which should create the data folder and place a copy of the image inside it. You should now be able to run the sketch and see the moving sphere image.

You can find out more about using images in Processing sketches by consulting the **image** section of the [Processing Reference](http://processing.org/reference/).

[](http://staff.city.ac.uk/~jwo/datavis/session02/imageSketch2/javascript/index.html)

Output from the sketch (click to activate).

**2.3 Using Tabular Data**

A large amount of data that you may wish to visualize will be stored in tabular format. Spreadsheets and relational databases can be used easily to output tables of numeric and text data. For example, the [Guardian Data Blog](http://www.theguardian.com/news/datablog/interactive/2013/jan/14/all-our-datasets-index) releases tabular spreadsheet data on a range of topics from government cuts, gun crime, disease pandemics, award and honours nominations to textual analysis of political speeches.

We will use [GapMinder's tabular data](http://www.gapminder.org/data/) on global development and health to illustrate how we can use Processing to handle numeric and text data in tables but the same approach could be used with any other tabular data you have access to. The approach described here follows that of [Fry (2008), Chapter 3.](http://moodle.city.ac.uk/mod/page/view.php?id=580556#references)

So let's start with some data and a question. The data we shall explore are on global dental health as indicated by the average number of bad teeth per 12 year old child in 2004. The data were retrieved from [GapMinder](http://www.gapminder.org/data/) and can be viewed in[tab-separated text format](http://staff.city.ac.uk/~jwo/datavis/session02/map2/data/badTeeth.tsv). The first few rows of the table are shown below:

|  |  |
| --- | --- |
| **CountryName** | **NumBadTeeth** |
| Afghanistan | 2.90 |
| Albania | 3.02 |
| Algeria | 2.30 |
| Angola | 1.70 |
| Anguilla | 2.50 |
| Antigua and Barbuda | 0.70 |
| Argentina | 3.40 |
| *:* | *:* |

The question we wish to ask of the data is, *does the global pattern of dental health reflect the general patterns of health and economic development in 2004? If not, what are the patterns of difference?*For the moment, we will take 'general patterns of health and economic development' to be your general knowledge of global development. But we could design a sketch that explicitly compared data about dental health with data about general health and development.

Our visualization task will be to map the bad teeth data so we can examine the global patterns in dental health. To do this we need a global map we can use as a backdrop. We've created a map image called [worldCountries.png](http://staff.city.ac.uk/~jwo/datavis/session02/map0/data/worldCountries.png) from data provided by the [Natural Earth](http://www.naturalearthdata.com/downloads/110m-cultural-vectors/) web site. As we will be overlaying additional data on top of this background map, the landmass has been set to light grey with white outlines to provide better contrast with the data we will display.

The code to display the backdrop map is straightforward, being essentially identical to the simple image drawing sketch shown in Section 2.2:

// Draws a simple global map with country outlines

// Version 1.2, 29th January 2016

PImage mapImage; // Background map image.

// Start the sketch and load the map image

void setup()

{

size(1000,550);

mapImage = loadImage("worldCountries.png");

}

// Plots the data over the background map.

void draw()

{

background(255);

// Draw the background image taking up the full width and height of sketch.

image(mapImage,0,0,width,height);

}

As before, to get Processing to know where to look for the image, simply drag a copy of it into the Processing window. This will force it to create a data/ folder and place a copy of [worldcountries.png](http://staff.city.ac.uk/~jwo/datavis/session02/map0/data/worldCountries.png) within it.

[](http://staff.city.ac.uk/~jwo/datavis/session02/images/map0.png)

Sketch showing world map background image (click to enlarge).

To plot data items on this background map, we are going to need access to another table that relates country names to their latitude/longitude locations. This is provided by [Ohad Eder-Pressman](http://ohadpr.com/2010/04/countries-approximate-lat-lon-and-iso-3166-1-alpha-2/) as a simple table that relates country name to a latitude/longitude pair. Note the convention when using latitude and longitude is that the northern hemisphere has latitudes between 0 (equator) and 90 (north pole), and the southern hemisphere latitudes between 0 (equator) and -90 (south pole). Longitudes vary from 0 (Greenwich meridian) to -180 going west and +180 going east of the Greenwich meridian.

Some of the names were 'cleaned' so they match the GapMinder country names, and the complete table of country locations can be found in the tab-separated values file [countryLocations.tsv](http://staff.city.ac.uk/~jwo/datavis/session02/map1/data/countryLocations.tsv). The first few rows are shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **CountryName** | **CountryCode** | **Latitude** | **Longitude** |
| Afghanistan | AF | 33.94 | 65.71 |
| Albania | AL | 41.15 | 20.17 |
| Algeria | DZ | 28.03 | 1.66 |
| Angola | AO | -11.20 | 17.87 |
| Anguilla | AI | 18.22 | -63.07 |
| Antigua and Barbuda | AG | 17.06 | -61.80 |
| Argentina | AR | -38.42 | -63.62 |
| *:* | *:* | *:* | *:* |

The following uses a special kind of *class* built into processing called [Table](http://processing.org/reference/Table.html). This provides the functionality for reading, querying and writing tabular data and is very useful for many data visualization sketches.   
  
The Table class was introduced into Processing in 2013 after Ben Fry's 2008 book *Visualizing Data* had been written. If you are working your way though his book, you will find he uses his own similar, but not quite identical Table class.

Processing's Table is an example of a *class*. A class is simply a self-contained block of code that can store information and has methods within it for processing that information. Details of what the Table class can do can be found in the[Processing Reference](http://processing.org/reference/Table.html).

We are going to build up the bad teeth map step by step, so firstly let's assemble the data and some rudimentary code in the same sketch.   
  
Create a new sketch in Processing and save it as **Map**. Copy the code from the example above (the one that starts with PImage mapImage) into your sketch. Add [worldCountries.png](http://staff.city.ac.uk/~jwo/datavis/session02/map1/data/worldCountries.png) to it by saving a local copy of the image and dragging it into your sketch. Do the same with the data file [countryLocations.tsv](http://staff.city.ac.uk/~jwo/datavis/session02/map1/data/countryLocations.tsv).

Let's look at the code that makes use of the Table class to read the country location table:

// Draws a global map showing country locations.

// Version 1.5, 29th January, 2016

// Author Ben Fry with modifications by Jo Wood, giCentre.

PImage mapImage; // Background map image.

Table locationTable; // Table storing country locations.

// Start the sketch and load the data (an image and a table).

void setup()

{

size(1000,550);

// Load the background map.

mapImage = loadImage("worldCountries.png");

// Load data from table

locationTable = loadTable("countryLocations.tsv","header,tsv");

}

// Plots the data over the background map.

void draw()

{

background(255);

// Draw the background image taking up the full width and height of sketch.

image(mapImage,0,0,width,height);

// Set the appearance of the data circles.

fill(192,0,0);

noStroke();

// Go through each row in table drawing each data item as a circle.

for (int row=0; row<locationTable.getRowCount(); row++)

{

float latitude = locationTable.getFloat(row,"latitude");

float longitude = locationTable.getFloat(row,"longitude");

// Scale each location so it uses screen coordinates, not geographic coordinates.

float x = map(longitude,-180,180,0,width);

float y = map(latitude,-60,85,height,0);

ellipse(x,y,5,5);

}

}

The purpose of this stage of the sketch is to read the country locations from the tabular file, draw the background map, and overlay a point representing each country at its correct location.

The first new thing we do is to create a new variable we've chosen to call locationTable, but instead of being a float orPImage, it is of type Table. We initialise the locationTable with the line:

locationTable = loadTable("countryLocations.tsv","header,tsv");

which uses Processing's [loadTable()](http://processing.org/reference/loadTable_.html) command to read the data from the file countryLocations.tsv and store the results so we can use the numbers and text in our sketch. The first parameter is the name of the file to read and the second specifies that the file has a single line header naming each column and that it is in tab separated values (TSV) format. After calling this command, the entire table is stored in the variable locationTable just like we might store a single number inside a float variable.

The first half of draw() is the same as the image drawing sketch we considered above. The new code starts with the line

for (int row=0; row<locationTable.getRowCount(); row++)

This is an example of a *loop* - a block of code that repeats some task a fixed number of times. In this case, we create a new number variable called row which starts at 0 and will increase by 1 every time the loop repeats while the value of row is less than the number of rows in the table. The block of code inside the curly braces below is the code that gets repeated; once for each row in the table.

The Table class has a command built into it called getFloat(row,columnName) which will extract a number from the table at the given row and the column with the given name. In our sketch, we wish to extract just the latitude and longitude values from the table which were labeled as such in the [first header row of the TSV file](http://staff.city.ac.uk/~jwo/datavis/session02/map1/data/countryLocations.tsv). The results of this extraction are stored in two float variables we've named as latitude and longitude.

The final part of our draw() method converts the latitude, longitude pair into screen coordinates at which we draw a small circle. Because longitude varies from -180 (left of our map) to 180 (right of our map) but the screen coordinates vary from 0 (left) and width (right), we need to convert each number before we can draw the ellipse. This is achieved simply with the Processing command [map()](http://processing.org/reference/map_.html) which is used to scale one range of numbers (longitude) to another (screen x-coordinates). We then do the same converting latitude to screen y-coordinates. Note that in the case of latitude, we also flip the vertical direction so that the smallest latitude (-60 degrees south in our map) has the largest y-coordinate (heightat the bottom of the sketch).

The 'for loop' as used in the example above is one of a number of ways we can get sections of program code to repeat. The general structure of a for loop is: 

for (initialise ; condition; change)

{

// Code to be repeated

}

The for line itself consists to three sections separated by semicolons. The first initialises a counting variable. This is the value that the counting variable has when the loop starts. The second bit tests a *condition* that while true, will force the loop to repeat. The final section (*change*) does something to the counting variable each time the loop repeats. For example: 

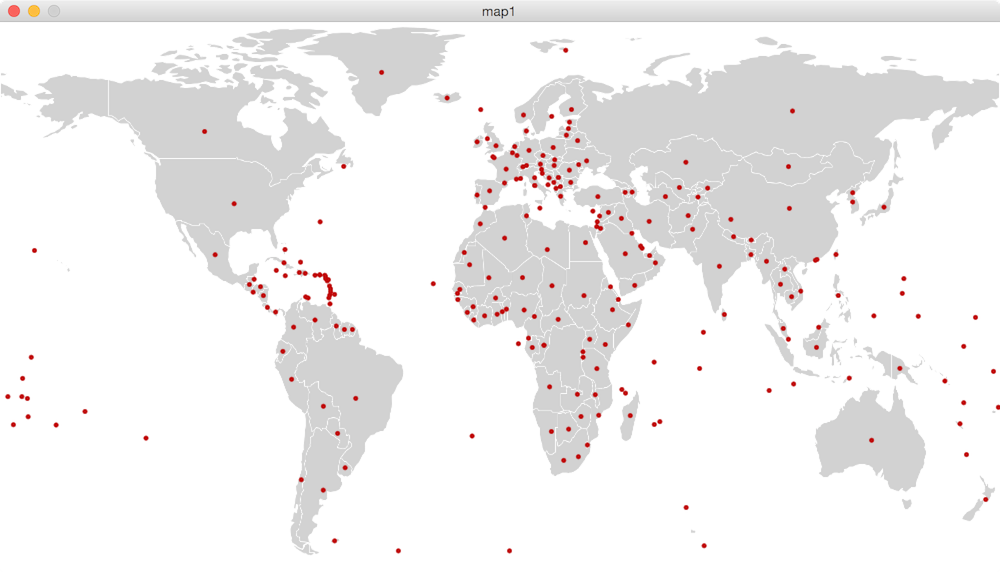
for (int count=0 ; count<100; count++)

{

// Any code here will be repeated 100 times

}

The expression count++ simply adds 1 to the variable count each time the loop is repeated.   
  
Looping, or more generally *iteration* is a very common way to define repetitive tasks in Processing. For some further examples, see the **Control** section of the [Processing Examples](http://processing.org/learning/basics/) documentation.

[](http://staff.city.ac.uk/~jwo/datavis/session02/images/map1.png)

Sketch showing world map with country locations (click to enlarge).

So far we have managed to display a background image and read a table of locations to provide coordinates of the country circles to plot. What we haven't done is say anything about dental health. To do this, we need to read in a second table containing the bad teeth data.

Make sure your sketch has access to the file [badTeeth.tsv](http://staff.city.ac.uk/~jwo/datavis/session02/map2/data/badTeeth.tsv) by saving a local copy and dragging it into your sketch.

We will move on to the final stage by modifying the sketch to read in the second table and link it to the first.

// Draws a global map showing country locations.

// Version 1.7, 29th January, 2016

// Author Ben Fry with modifications by Jo Wood, giCentre.

PImage mapImage; // Background map image.

Table locationTable; // Table storing country locations.

Table dataTable; // Table storing dental health data.

float dataMin = MAX\_FLOAT; // Used to store min and max data values.

float dataMax = MIN\_FLOAT;

// Start the sketch and load the data (image and two tables).

void setup()

{

size(1000,550);

// Load the background map.

mapImage = loadImage("worldCountries.png");

// Load data from tables

locationTable = loadTable("countryLocations.tsv","header,tsv");

dataTable = loadTable("badTeeth.tsv","header,tsv");

// Find the minimum and maximum values

for (int row=0; row<dataTable.getRowCount(); row++)

{

dataMin = min(dataMin,dataTable.getFloat(row,"NumBadTeeth"));

dataMax = max(dataMax,dataTable.getFloat(row,"NumBadTeeth"));

}

}

// Plots the data over the background map.

void draw()

{

background(255);

// Draw the background image taking up the full width and height of sketch.

image(mapImage,0,0,width,height);

// Set the appearance of the data circles.

fill(192,0,0,80);

stroke(255);

strokeWeight(0.5);

// Go through each row in table drawing each data item as a circle.

for (int row=0; row<dataTable.getRowCount(); row++)

{

// Get the bad teeth data for the country.

String countryName = dataTable.getString(row,"CountryName");

float badTeeth = dataTable.getFloat(row,"NumBadTeeth");

float circleSize = map(badTeeth,dataMin,dataMax,1,30);

// Find the row in the location table that matches the country in the data table.

TableRow countryRow = locationTable.findRow(countryName,"CountryName");

// It is possible that we cannot find countryName (from the dataTable) in the

// location table, so only attempt to plot the circle if country is found in both.

if (countryRow != null)

{

// Extract the latitude and longitude from the row we have found.

float latitude = countryRow.getFloat("latitude");

float longitude = countryRow.getFloat("longitude");

// Scale the latitude and longitude to fit in the sketch.

float x = map(longitude,-180,180,0,width);

float y = map(latitude,-60,85,height,0);

ellipse(x,y,circleSize,circleSize);

}

}

}

Our modified map sketch now reads two tables of data - one (locationTable) containing the country locations as before, and one (dataTable) containing the bad teeth data. The only other part of setup() that is new is that we also find the minimum and maximum values in our bad teeth dataset. This will be necessary because when it comes to drawing the data we will scale the size of each circle to be somewhere between these minimum and maximum values.

To find the minimum and maximum values in a range of numbers we create another for loop in setup() that extracts each number from the table. We use the Processing command [min()](http://processing.org/reference/min_.html) to find the smaller of the two numbers dataMinand dataTable.getFloat(row,"NumBadTeeth". In other words each number in the table is compared with the current minimum value, and if the table value is smaller than the current minimum, the current minimum is updated with this new smaller number. We do the same with the maximum value so that be the end of the loop we have stored the smallest and largest numbers in the table.

The main effort of the draw() method is to create another loop that looks at each row of the badTeeth table (dataTable) in turn. We extract the name of each country from the relevant column in the row and store that in a new type of variable known as a [String](http://processing.org/reference/String.html). A string is simply a variable that can store text rather than numbers, images or tables. In this example, we name that variable countryName. We then extract the dental health measure from the column that was labeled "NumBadTeeth" in the TSV file and store that in the float variable badTeeth.

Because we do not know whether the badTeeth value for each country will also happen to be a number that corresponds to the number of pixels we wish our data circle to be, we scale the value to fit between the range 1 (smallest data value) and 30 (largest data value). We do that using the same map command we used when scaling latitude/longitude to screen coordinates. This guarantees that our smallest circle will be 1 pixel wide and our largest circle will be 30 pixels wide regardless of the data values we read from the table.

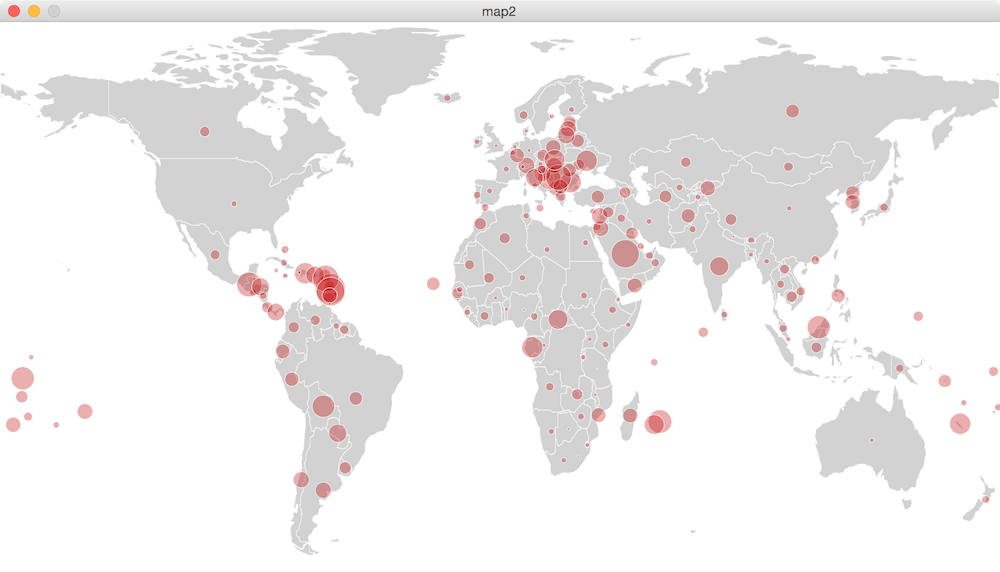
We next extract the country locations and store the values in the variables latitude and longitude. Unlike the previous version of this sketch, this time instead of providing a table row number to do this (getFloat(row,"latitude")), we use the country name to identify the correct row in the locationTable. The correct row is found by calling the Table method[findRow()](http://processing.org/reference/Table_findRow_.html) (highlighted in the code above). This will search the location table looking in the column "CountryName" for the row that contains the actual country stored insude the String variable countryName. This is the key line that allows us to relate the two tables of data together using each country name as the common key to both. The result of the match is stored in another new type of variable known as a [TableRow](http://processing.org/reference/TableRow.html).

If findRow() does not manage to find a row in the bad teeth table that matches the country name we have provided, it will report a special value - null - indicating that there is no table row with the requested match. We can use a new command - [if](http://processing.org/reference/if.html) - to make the drawing of the ellipse dependent on the matched row not being null. Only if the DataRow is not null will the code between the curly braces immediately following the if statement be executed. This will mean our sketch will work even if some of the countries named in the data table are not present in our location table.

This may seem like a rather complicated way of getting our map data, but it means our sketch will work even if we do not have dental records for all countries in our database. It also makes it much easier to replace the dental record table with other data tables storing different types of information associated with countries around the world.

The final bit of our draw() method takes the circle size, latitude and longitude extracted from the two tables and plots the result as a transparent red circle, sized by the number of bad teeth per 12 year old child, and located in each country's midpoint. The larger the circle, the poorer the dental health for the country at that location.

Going back to our original question, does this map now help us answer the question *does the global pattern of dental health reflect the general patterns of health and economic development in 2004?*. Could you add any further design improvements to assist in analysing the data?

[](http://staff.city.ac.uk/~jwo/datavis/session02/images/map2.png)

Sketch showing number of bad teeth per child (click to enlarge).

**2.4 Using Service APIs**

We have seen two categories of data that we can use in our data visualization sketches - images for importing graphical data and tables for importing structured numeric and textual data. A third important type of data source is offered by *APIs*or *Application Programming Interfaces*. These are services that are designed specifically for programmers to access data from some source. Many services and organisations provide such APIs. Some examples include:

* [Flickr API](http://www.flickr.com/services/api/) for access to the Flickr Photo Archive.
* [Twitter API](http://dev.twitter.com/) for access to tweets around the world.
* [Facebook API](http://developers.facebook.com/) for access to facebook social network database.
* [Transport for London API](http://www.tfl.gov.uk/businessandpartners/syndication/default.aspx) for cycle hire, oystercard, tube and travel usage and live updates.
* [TheyWorkForYou API](http://www.theyworkforyou.com/api/) for access to all parliamentary speeches and voting records.

There are many hundreds of more APIs available that may provide useful data feeds for your visualization. The formats of the data they provide will vary between APIs, some will be tabular and so can use the approach described above, while some will be in XML format or use standard protocols such as [JSON](http://www.json.org/). We cannot cover all these formats in theses sessions, but in Session 4 we will look at *parsing* data in a range of formats and use additional libraries to make using external data sources much easier. For now though, it is worth doing some searching of APIs in the areas of interest to you so you get a better understanding of the types of data that you may be able to incorporate into your data visualization.

**2.5 Conclusions.**

This session has provided some examples of data visualizations by considering some contrasting sources of data that may be represented visually. The first stage of any successful data visualization will be a question and some data that will help answer it. We have seen how Processing can be used to acquire two categories of data - images and tables, which together form a major part of many datavis applications. We have seen how dragging data files into a Processing sketch allows it to have easy access to those data and how classes can be used by your sketch to deal with some of the complexities of data handling.

This session has also introduced a few new programming concepts, which if you are not familiar with Java may take a little getting used to. In particular we have now created variables out of the int type for whole numbers, the String type for storing text, the PImage type for storing images and the Table TableRow classes for storing tables of numbers and text. We have used *loops* for repeating sections of code a fixed number of times (e.g. iterating through each row in a table) and some Processing methods min(), max() and map() for analysing and scaling numbers. As you practice creating new sketches, using these techniques will become more familiar to you and your sketches better able to use data to answer a question.

**Recommended Reading**

*Chapter 2 of Munzner (2015) introduces the idea of data abstraction as a process of generalising the 'what' can be visualized? design question*

**Munzner, T.** (2015) Chapter 2: What: Data Abstraction, pp.21-40 in *Visualization Analysis and Design*, CRC Press

*Two chapters in Fry (2008) will help to consolidate the ideas in this session. Chapter 3 goes through a similar mapping/table example and develops some of the representation ideas a bit further. Chapter 9 gives an overview of some of the other issues involved in acquiring data for your data visualizations.*

**Fry, B.** (2008) Chapter 3: Mapping, pp.31-53 in *Visualizing Data*, O'Reilly

**Fry, B.** (2008) Chapter 9: Acquiring data, pp.264-295 in *Visualizing Data*, O'Reilly

Last modified: Friday, 29 January 2016, 10:53 PM