More Shiny

INFO 201

Joel Ross

Winter 2017

Deadlines This Week

- Thu 02/23: Project Proposal
- Fri 02/24: First Peer Evaluation
- Tue 02/28: Assignment 8 **(individual)**
- Thu 03/09: Project

Today's Objectives

By the end of class, you should be able to

- Be comfortable creating **Shiny** apps with a **UI** and **Server**
- Develop apps with **multiple files**
- **Publish** Shiny apps to the web
- Create **interactive visualizations**

Shiny Apps

Changes to **control widgets** update the value in `input`, which ***notifies*** the **reactive expression** to update the value in `output`, which is displayed by **reactive outputs**.

(1) user gives value to widget

(3) new output is shown

(2) change reruns the expression

```
ui <- fluidPage(
  textInput('user.name'),

  textOutput('message')
)


server <- function(input, output) {

  output$message <- renderText({

      my.message <- input$user.name
      return(my.message)
    })
}
```

Sidebar Layouts

UI layouts can include *other layouts* as content elements, allowing you to create complex page organizations.

```
ui <- fluidPage(    # UI is a fluid page

  # include panel with the title (also sets browser title)
  titlePanel("My Title"),

  # layout the page in two columns
  sidebarLayout(

    sidebarPanel(  # specify content for the "sidebar" column
      p("sidebar panel content goes here")
    ),

    mainPanel(     # specify content for the "main" column
      p("main panel content goes here")
    )
  )
)
```

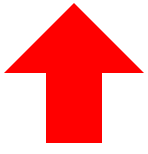
Module 15 exercise-2

Publishing Shiny Apps

Because Shiny apps require an R interpreter session to run the server, we can't just publish on GitHub Pages.

Instead, we can use free app hosting through

https://www.shinyapps.io/



Will need to

1. sign up for an account
2. install software
3. setup access token
4. publish app!

Practice:
Publish your App!

Why would we want to split a script into **multiple files**?

Multiple Scripts

"Include" one script inside another using the `source()` function.

```
### in helper.R ###

my.variable <- "Hello world!"

my.function <- function() {
  print("I said hello world!")
}
```

```
### in script.R ###

# "source" the other script
# this will run the code in that script
source('relative/path/to/helper.R')

print(my.variable)  # created by other script
my.function()  # created by other script
```

Multiple Scripts

"Include" one script inside another using the `source()` function.

```
### in my_ui.R ###

my.ui <- fluidPage(  # define UI
  # ...
)
```

```
### in my_server.R ###

my.server <- function(input, output) {  # define server
  # ...
}
```

```
### in app.R ###

# load the UI and Server
source('my_ui.R')
source('my_server.R')

shinyApp(ui = my.ui, server = my.server)
```

Multi-File Apps

We can also define a Shiny UI/Server directly using multiple files: `ui.R` and `server.R`

Put these files **in the same folder** to run them as an app!

```
### in ui.R ###

my.ui <- fluidPage(
  # ...
)

# create the UI in this file
shinyUI(my.ui)
```

```
### in server.R ###

my.server <- function(input, output) {
  # ...
}

# create the server in this file
shinyServer(my.server)
```

Interactive Visualizations

Interactive Visualizations

Shiny is able to track mouse interactions with a rendered plot (including when using **ggplot2**). Mouse **events** store values in the `input` like any other control widget

```
ui <- fluidPage(
  plotOutput('my_plot', click='my_click_key'),
  verbatimTextOutput('info')
)

server <- function(input, output){
  output$my_plot <- renderPlot({
    ggplot(data = mpg) +  # plot the mpg data
      geom_point(aes(x = displ, y = hwy, color=class))
  })

  output$info <- renderPrint({
    return(input$my_click_key)
  })
}
```

key to save click location in

information about the click (where)

Plot Interactions

Shiny current supports four different kinds of mouse inputs.

```
plotOutput("plot",

  # click on the plot, gets location of click
  click = "plot_click",

  # double-click on the plot, gets location of click
  dblclick = "plot_dblclick",

  # hover over the plot (300ms default), gets location of mouse
  hover = "plot_hover",

  # select area of the plot, gets (x,y) corners of the box
  brush = "plot_brush"
)
```

Selecting Points

Shiny provides helper functions **nearPoints()** and **brushedPoints()** that allow you to easily look up the rows that correspond to the ***points*** selected.

```
output$info <- renderPrint({
  nearPoints(mpg, input$my_click_key)
})
```

data frame to look up rows in

mouse event info

implied return() call!

Dynamic Plots

http://shiny.rstudio.com/gallery/plot-interaction-exclude.html

Reactive Values

In order for a variable (e.g., the data being plotted) to change, it needs to be a **reactive value**, which can be stored in a `reactiveValue()` list.

```
server <- function(input, output) {

   values <- reactiveValues()  # like a list
   values$a <- 4   # assign reactive value

}
```

like *input*, will cause expressions to re-run on change

https://shiny.rstudio.com/articles/reactivity-overview.html

Observing Events

Use **observeEvent()** to specify a function that should be run ***in response*** to an interaction. This is like using **reactive()**, except that declaring what to *do* on change.

respond to change in value

```
observeEvent(input$plot_click, {
  print("Plot was clicked")

  # can update a reactive value,
  # which will trigger render functions
  # that depend on it
  values$a <- 4
})
```

(nameless) function to run

Updating Widgets

Can use **update** methods to change the value of a UI **control widget** from the server:

```
ui <- fluidPage(
  textInput('username', label="Name"),
  actionButton('clear', label="Clear")  # a button
)


server <- function(input, output, session) {

  # respond to button press
  observeEvent(input$clear, {

    # respond change TextInput
    updateTextInput(session, 'username', value='')
  })
}
```

Module 15 exercise-4

Designing Interactive Visualizations

Information Seeking Mantra

Overview first, zoom and filter, then details on demand

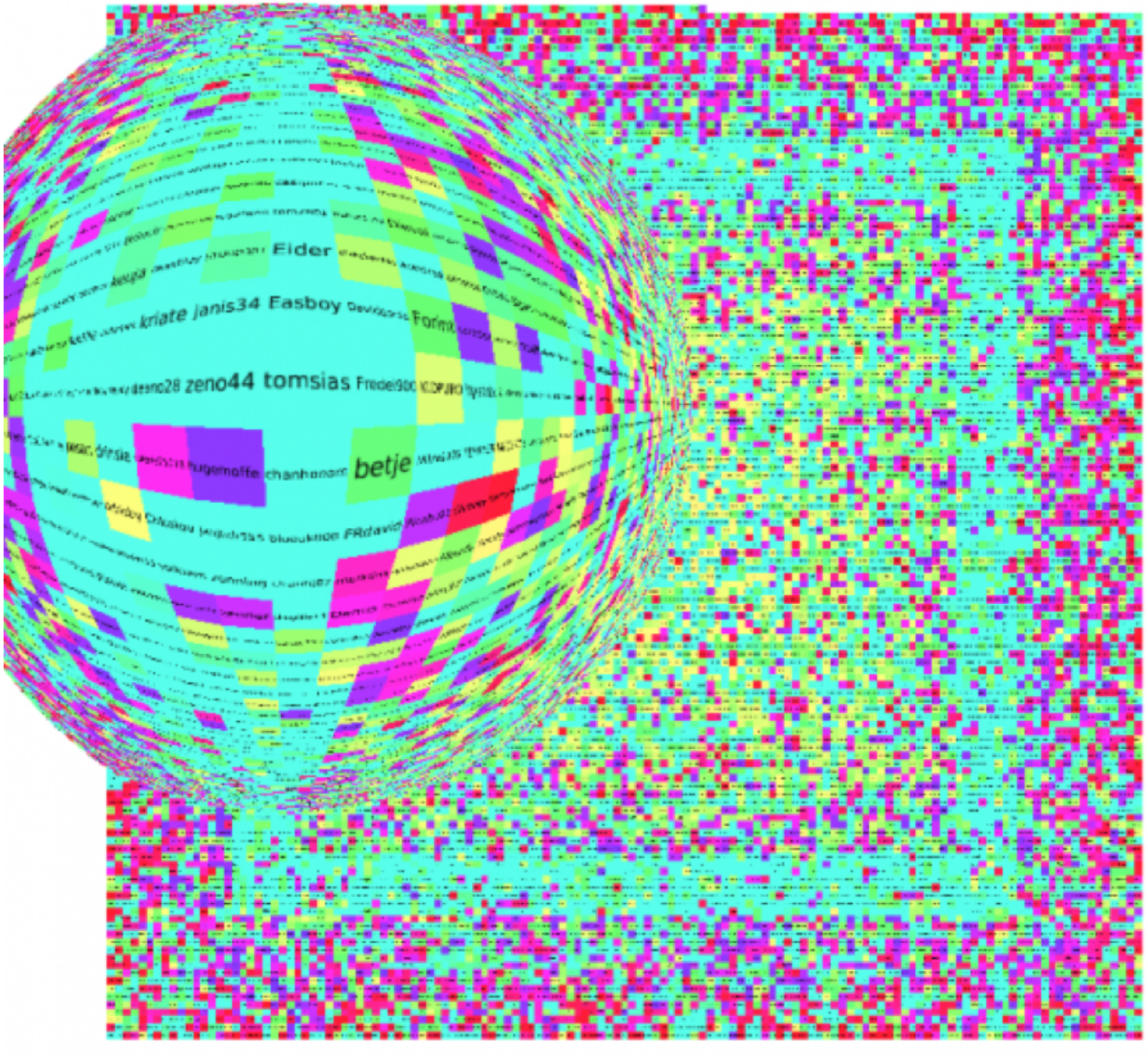
There are many visual design guidelines but the basic principle might be summarized as the Visual Information Seeking Mantra:

Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand
Overview first, zoom and filter, then details-on-demand

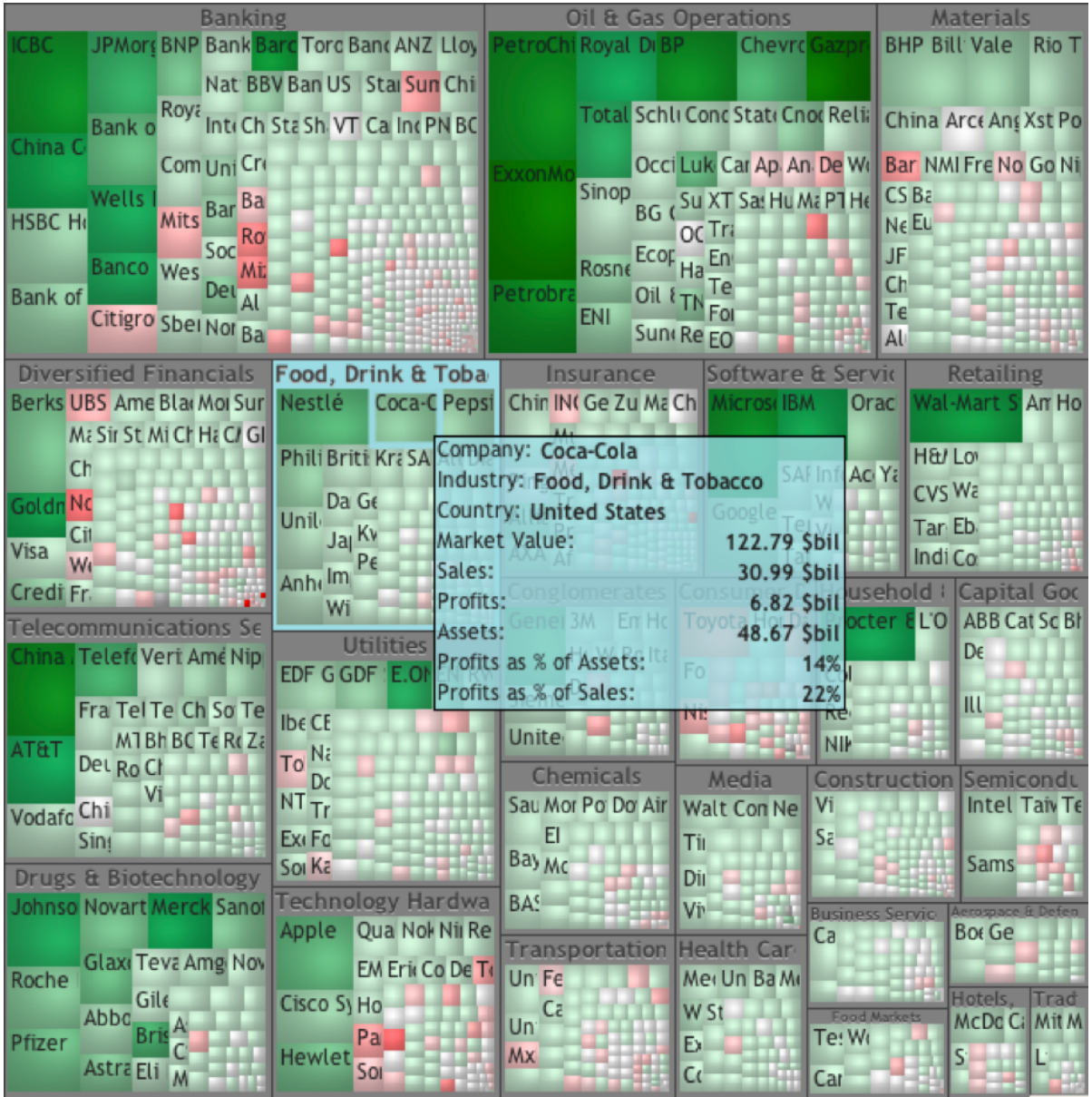
Each line represents one project in which I found myself rediscovering this principle and therefore wrote it down it as a reminder. It proved to be only a starting point in trying to characterize the multiple information-visualization innovations occurring at university, government, and industry research labs.

Overview First

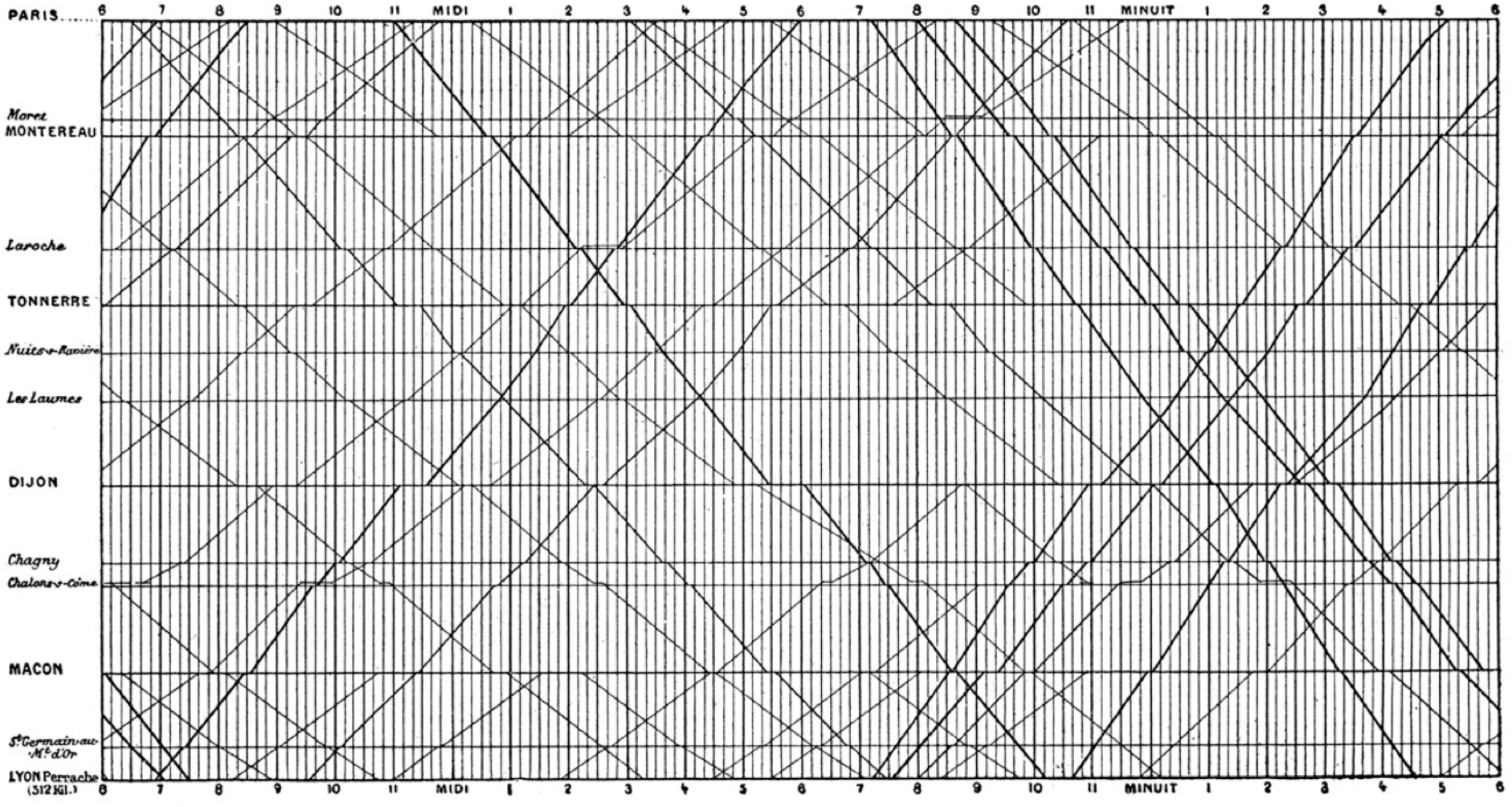
Zoom and Filter



Details on Demand



Is this interactive?



E. J. Marey, *La Méthode Graphique* (Paris, 1885), p. 20. The method is attributed to the French engineer, Ibry.

Action Items!

- Be comfortable with **module 15**
- Deadlines: Project Proposal, Peer Evaluation, Assignment 8

Tuesday: Some statistical modeling (?)