Biost 563: Computing & Research Tools for Data Visualization

Ali Shojaie (and others)

Seattle, Summer 2015

Announcements

- Next week (last lecture): How To Design and Make a Poster (Leila Zelnick)

- Your posters should be *very close* to completion

- Please see the material for the next lecture on Canvas to get started on your poster.

Data Visualization

Data visualization is a critical component of Data Science and is used in two main settings

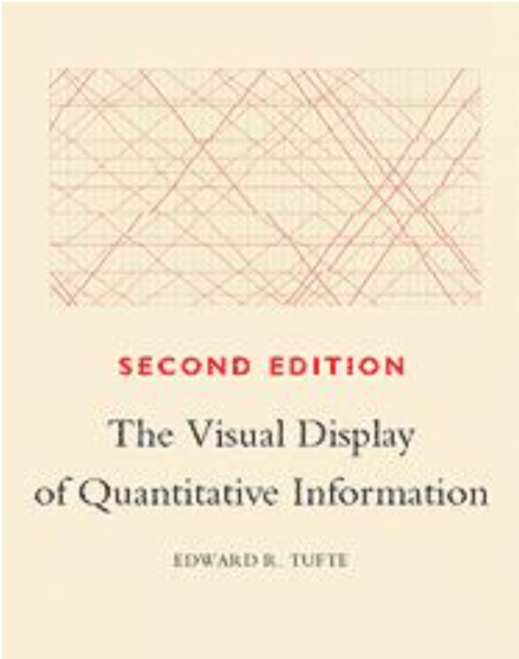
- visualization for explaining
- visualization for exploring

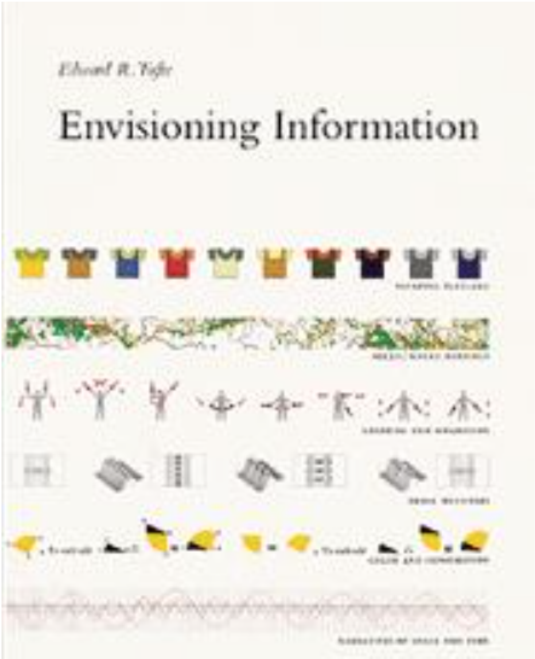
While common principals apply to both of the above, the aims of the two are different.

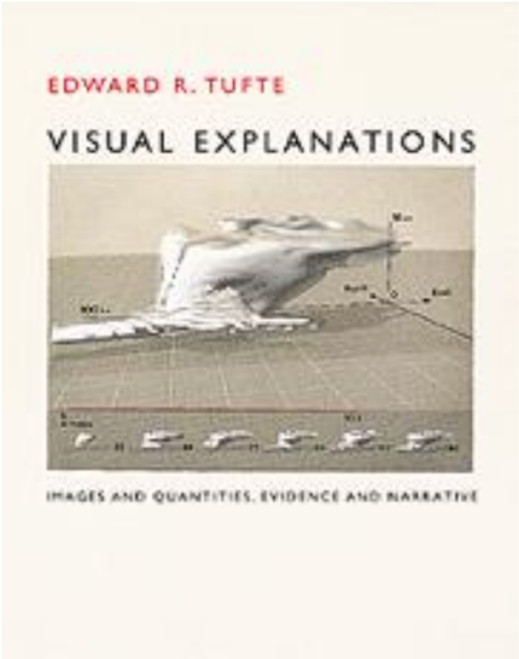
In this lecture we focus primarily on the second type of data visualization, and will introduce tools for exploratory data analysis in RR.

Communicating Graphically

- Data visualization is a lot *science* and a little art







- See Ken's slides on How To Communicate Graphically – I highly recommend them!

Communicating Graphically

- We won't talk about effective communication with graphics, which is critical for data visualization for explanation (I refer again to Ken's slides for this). Instead we focus on new *tools* for exploratory data visualization

- But, I should at least mention these principles from Tufte about data visualization
  - serve a reasonably clear purpose
  - show the data
  - avoid distorting what the data have to say
  - encourage the eye to compare different pieces of data

- We will discuss two recent R-packages
  - `ggplot2`
  - `shiny`

ggplot

- `ggplot`* is based on The Grammar of Graphics by Leland Wilkinson, and the `lattice` package

- `ggplot` is designed to work in a *layered fashion*, starting with a layer showing the raw data then adding layers of annotation and statistical summaries†

- The idea is to make the nice features of `lattice` available in a simpler way, and also make it easier to add additional components to the plot (as *layers*, which we talk about later)

*`ggplot` is a function in the `ggplot2` package, but I use them interchangebly!
†H. Wickham (2010) A Layered Grammar of Graphics, JCGS

ggplot

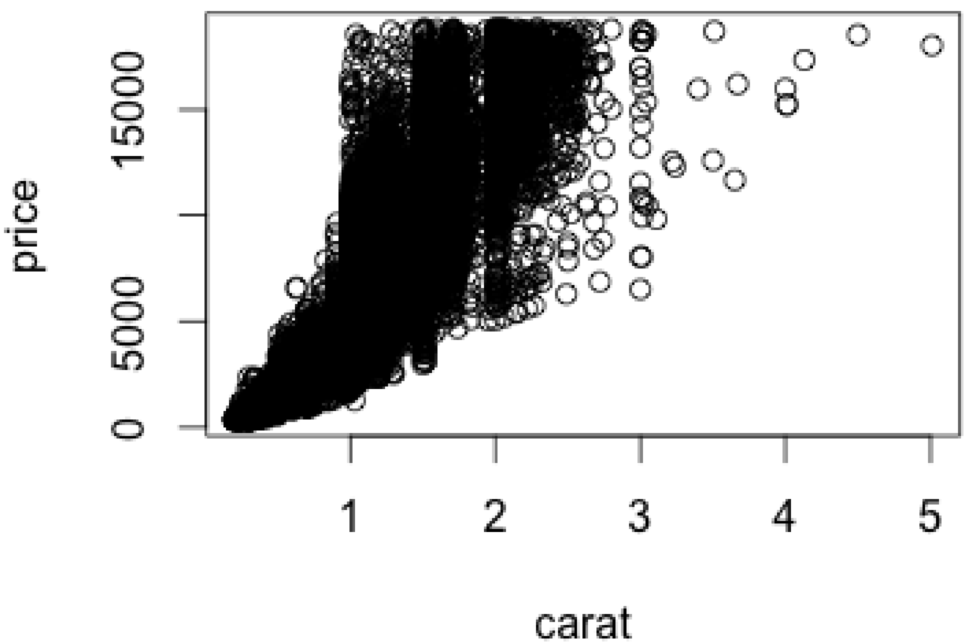
Let's look at an example `diamonds` data in R

- ∼54,000 round diamonds from http://www.diamondse.info/
- Variables:
  - carat, colour, clarity, cut
  - total depth, table, depth, width, height
  - price
- Is there a relationship between `carat` and `price`?

ggplot

Using the default settings in `plot()`

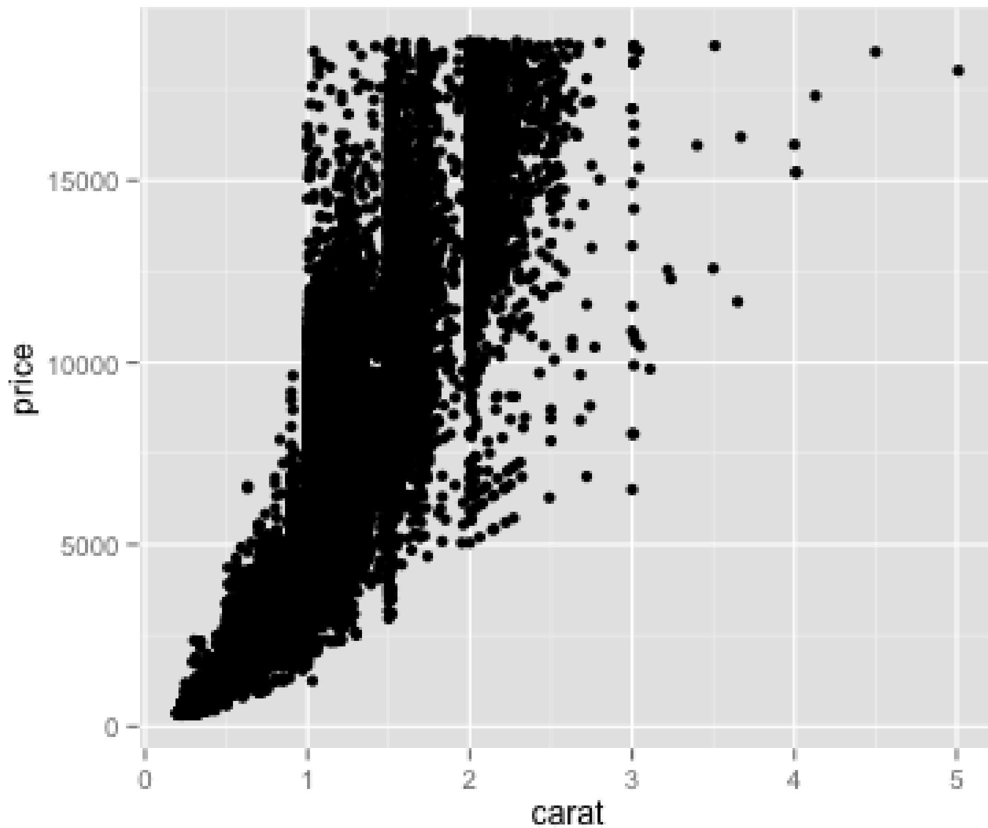
```
plot(price~carat, data=diamonds)
```



ggplot

Using the default settings in `ggplot()`

```
ggplot(diamonds, aes(carat, price)) + geom_point()
```



`ggplot` vs. R – 1

A first take

- The *default* option in `ggplot` looks nicer!
- `ggplot`'s syntax looks weird, especially if you're not very familiar with `lattice`
- `ggplot` may be slower than base R `plot()`
- You can clearly manipulate the `plot()` option to get the same plot (and even better plots), but that would require some extra coding
- This is OK (and perhaps what you should do) in the *final version* of the paper/report/slides/poster, but becomes laborious if you want to do RR
- The `ggplot` syntax also makes plotting more *structured* and easier to update in RR

What about the `ggplot` syntax?

```
ggplot(diamonds, aes(carat, price)) + geom_point()
```

What about the `ggplot` syntax?

```
ggplot(diamonds, aes(carat, price)) + geom_point()
```

The basic concept of a `ggplot` graphic is to combine different elements into layers. Each layer of a `ggplot` graphic must have a data set and aesthetic mappings

- `data`: for `ggplot()`, this must be a data frame!
- `aes`: a mapping from the data to the plot; basically the x and y-axes

What about the `ggplot` syntax?

```
ggplot(diamonds, aes(carat, price)) + geom_point()
```

Layers also have

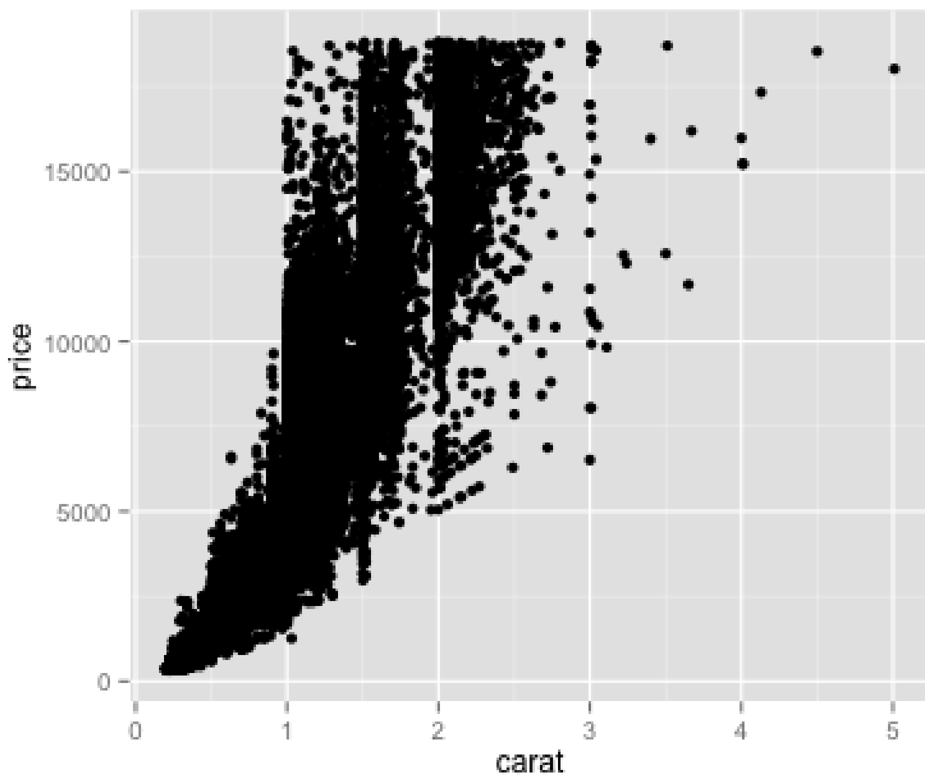
- a `geom`, or a geometric object: defines the overall look of the layer – is it bars, points, or lines?
- a `stat`, or a statistical summary: how should the data be summarized (e.g., binning for histograms, or smoothing to draw regression lines, etc).
- a `position`: how to handle overlapping points

When not specified, the defaults are used...

What about the ggplot syntax?

There are actually many ways to get the same plot!

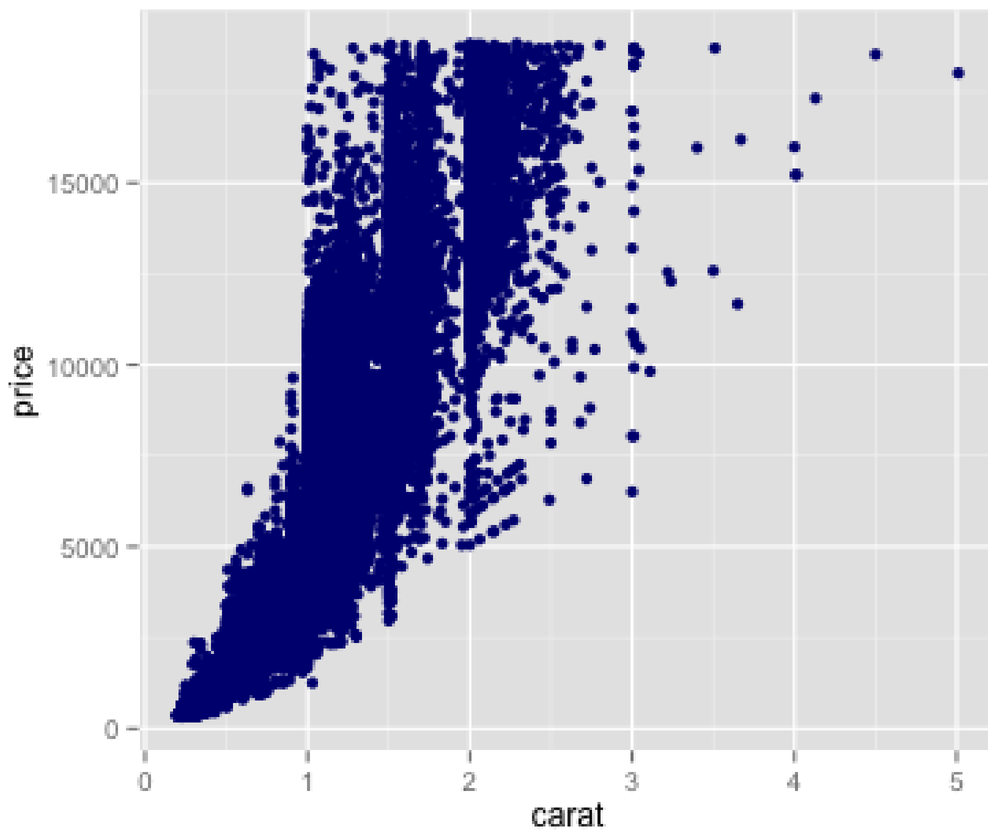
```
ggplot(diamonds, aes(price, carat)) + geom_point()
ggplot() + geom_point(aes(price, carat), diamonds)
ggplot(diamonds) + geom_point(aes(price, carat))
ggplot(diamonds, aes(price)) + geom_point(aes(y = carat))
```



Changing the aesthetics

You can control the aesthetics of each layer, e.g. `colour`, `size`, `shape`, `alpha` (opacity) etc.

```
ggplot(diamonds, aes(carat, price)) + geom_point(colour = "blue")
```



Changing the aesthetics

And here are more examples

```
ggplot(diamonds, aes(carat, price)) + geom_point(alpha = 0.2)
ggplot(diamonds, aes(carat, price)) + geom_point(size = 0.2)
ggplot(diamonds, aes(carat, price)) + geom_point(shape = 1)
```

Changing the aesthetics

And here are more examples

```
ggplot(diamonds, aes(carat, price)) + geom_point(alpha = 0.2)
ggplot(diamonds, aes(carat, price)) + geom_point(size = 0.2)
ggplot(diamonds, aes(carat, price)) + geom_point(shape = 1)
```

But be careful about the syntax

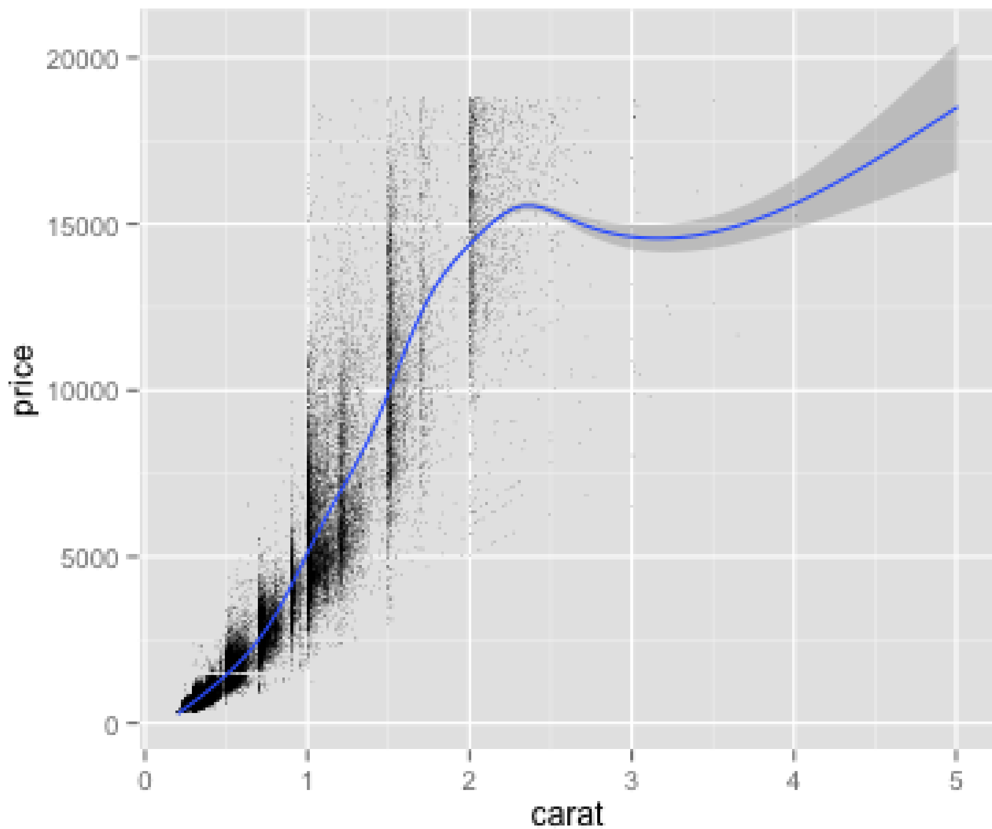
```
ggplot(diamonds, aes(carat, price)) + geom_point(aes(colour = 'blue'))
```

wouldn't work! (try it)

Combining layers

The real power of `ggplot` is its ability to combine layers

```
ggplot(diamonds, aes(carat, price)) + geom_point(size = 0.2) + geom_smooth()
```



This gives you a warning that you better change the defaults

Transformations and more

But before spending time to find a good smoother, we should *think* about what we're plotting...is this really what we want

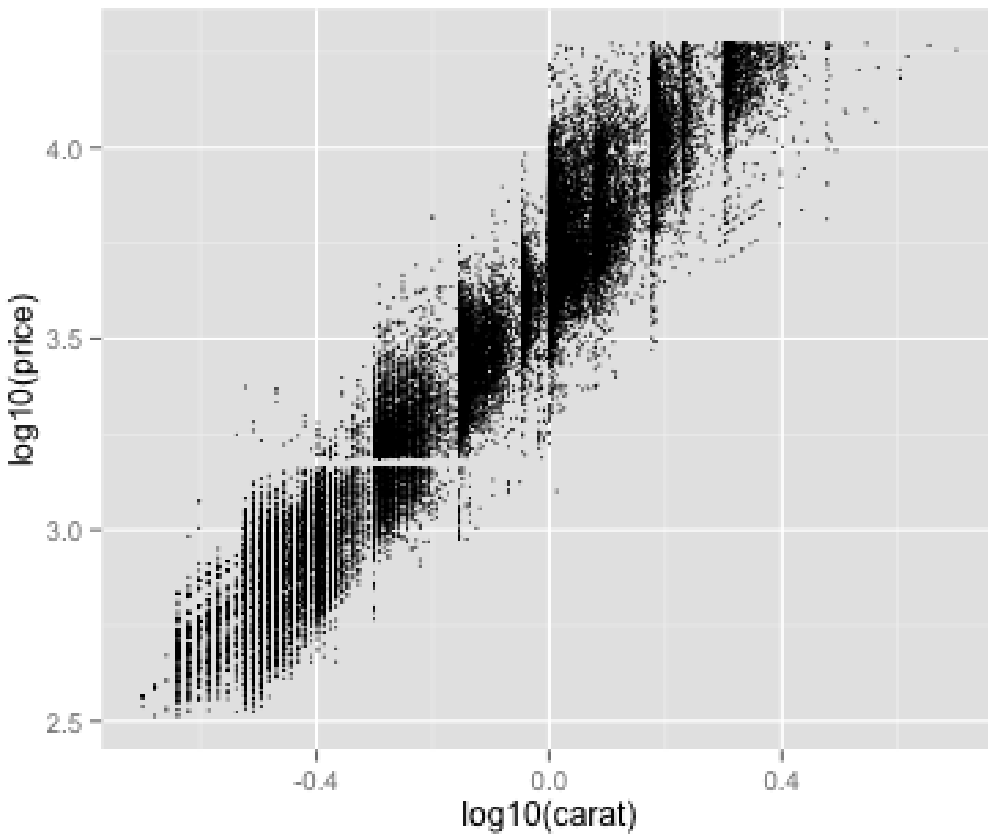
Remember

- serve a reasonably clear purpose
- show the data
- avoid distorting what the data have to say
- encourage the eye to compare different pieces of data

Transformations and more

In this case (and many other situations) a log transformation may make things clearer

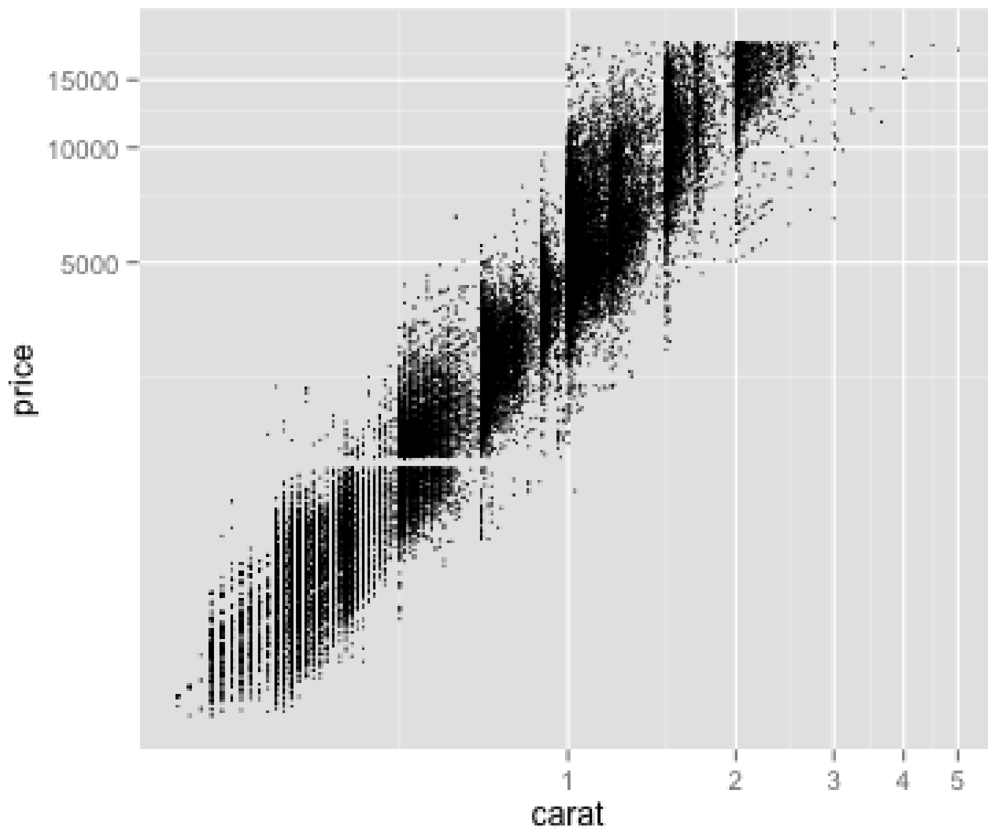
```
ggplot(diamonds, aes(log10(carat), log10(price))) + geom_point(size = 0.2)
```



Transformations and more

A better way to do this is to use `coord_trans()`

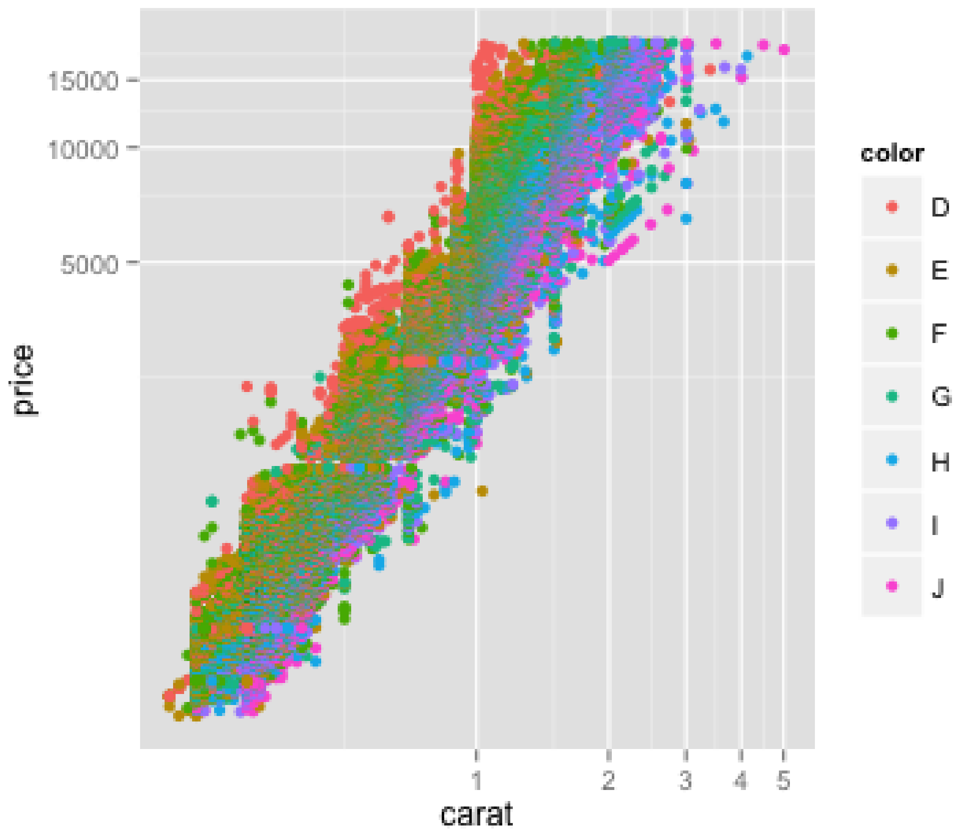
```
ggplot(diamonds, aes(carat, price)) + geom_point(size = 0.5)
    + coord_trans(x = "log10", y = "log10")
```



Adding information for a third variable

We can color by a *factor* variable (not that it's useful here!)

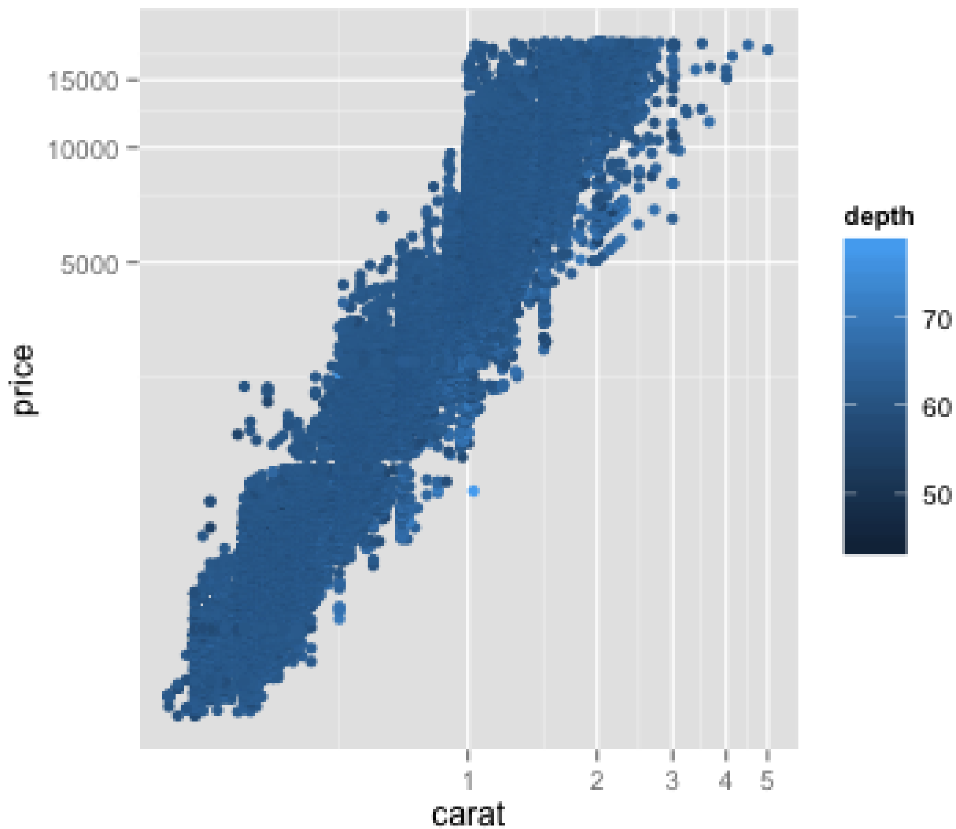
```
ggplot(diamonds, aes(carat, price, colour=color)) + geom_point() +
    coord_trans(x = "log10", y = "log10")
```



Adding information for a third variable

Can also color by a *continuous* variable (not really useful either!)

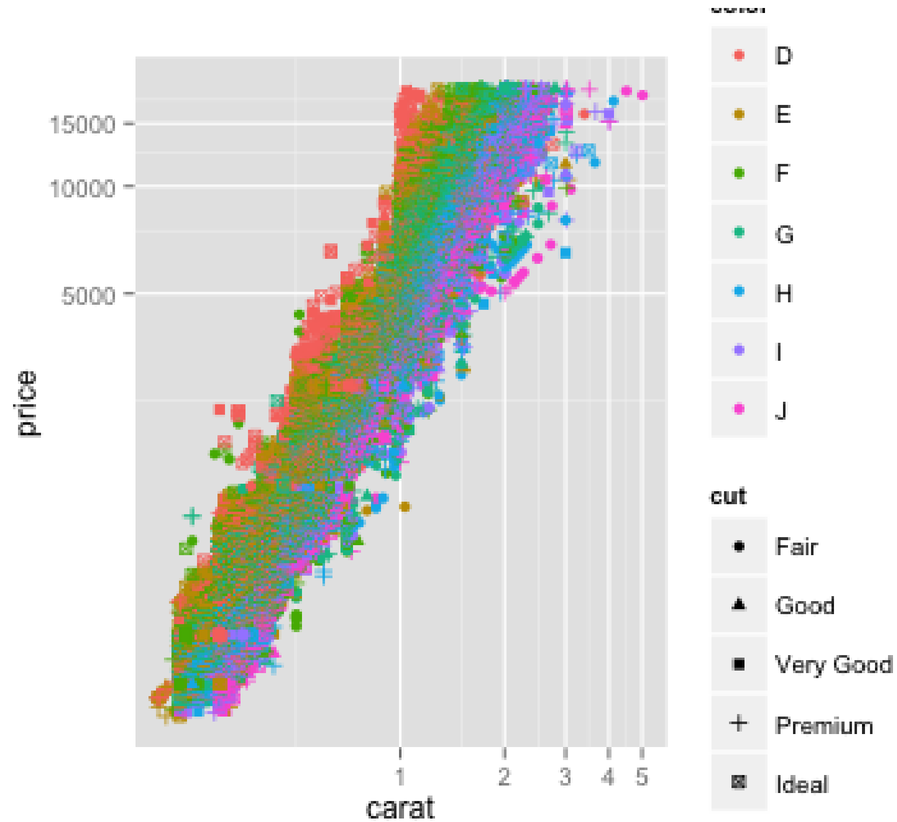
```
ggplot(diamonds, aes(carat, price, colour=depth)) + geom_point() +
    coord_trans(x = "log10", y = "log10")
```



Adding information for a third variable

And, as I mentioned, you can include information on more variables (though I really don't recommend this one!)[‡]

```
ggplot(diamonds, aes(carat, price, shape=cut, colour=color)) +
    geom_point() + coord_trans(x = "log10", y = "log10")
```

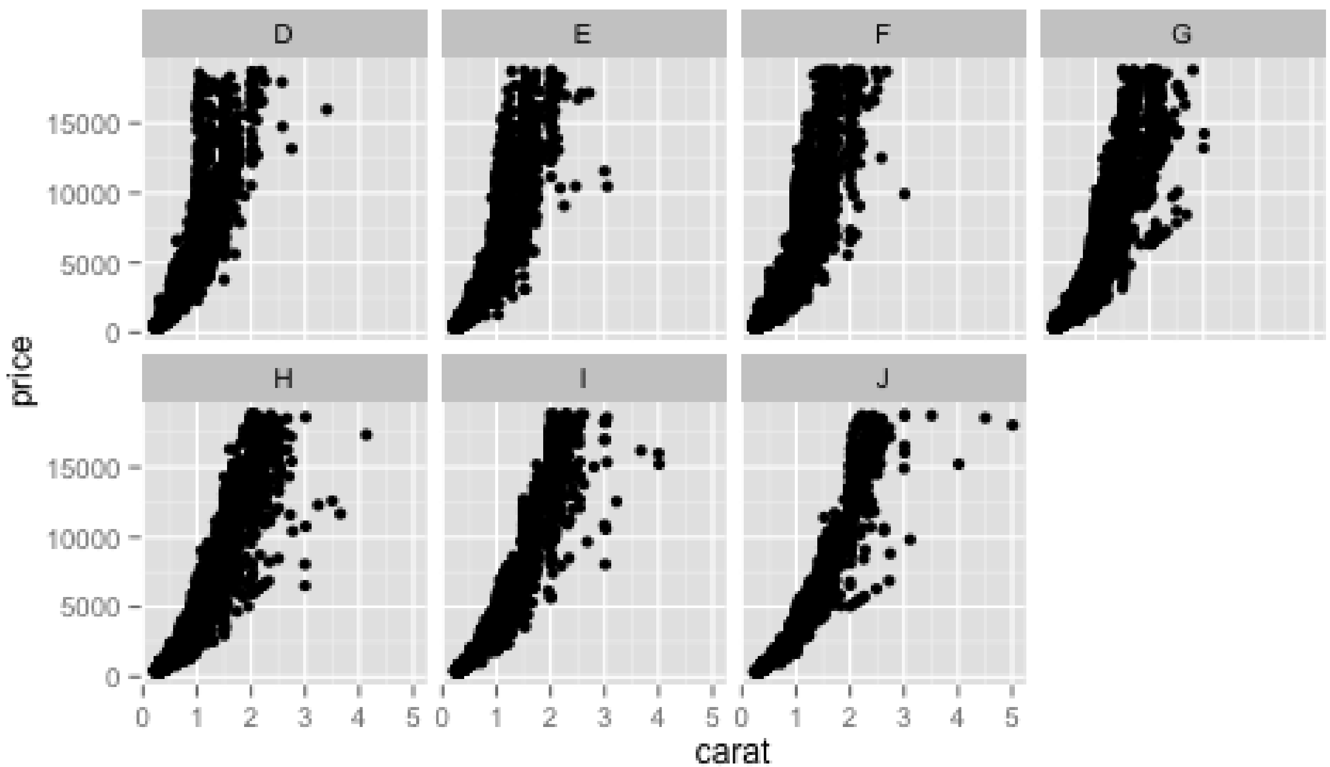


[‡]And, you can see that now the legends are cut!

Adding information for a third variable

In some cases, it may be more useful to get separate plots for each category of the third variable, to understand conditional relationships

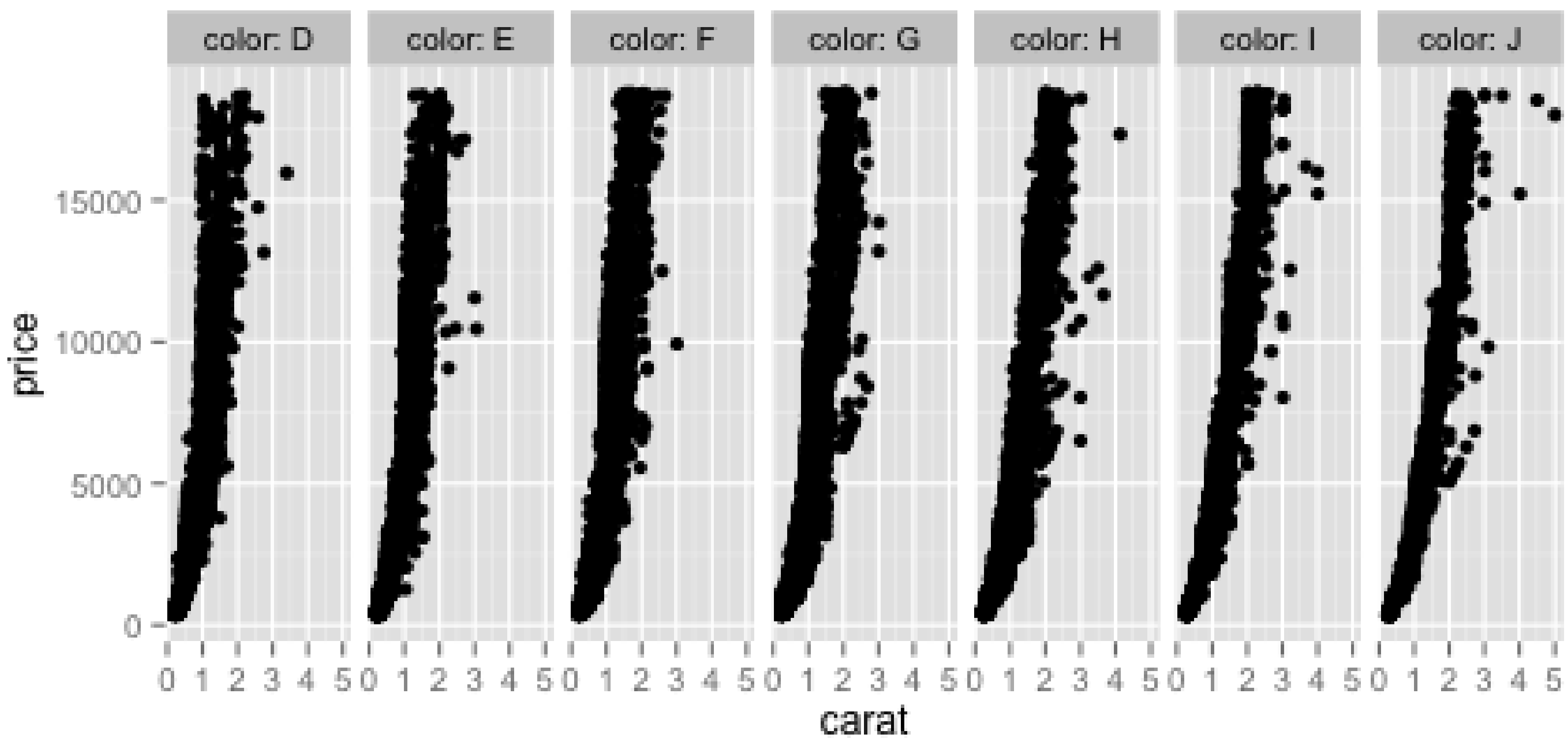
```
ggplot(diamonds, aes(carat, price)) + geom_point() +
    facet_wrap(~color, ncol=4)}
```



Adding information for a third variable

Alternatively, you can use the `facet_grid`, which also allows more than 1 conditioning variable (tables of plots)

```
ggplot(diamonds, aes(carat, price)) + geom_point() +
    facet_grid(~color, labeller=label_both)
```



Some Comments

Notice that in both cases,

```
ggplot(diamonds, aes(carat, price)) + geom_point() +
    facet_grid(~color, labeller=label_both)
ggplot(diamonds, aes(carat, price)) + geom_point() +
    facet_wrap(~color, ncol=4)}
```

the plotting part is the same!

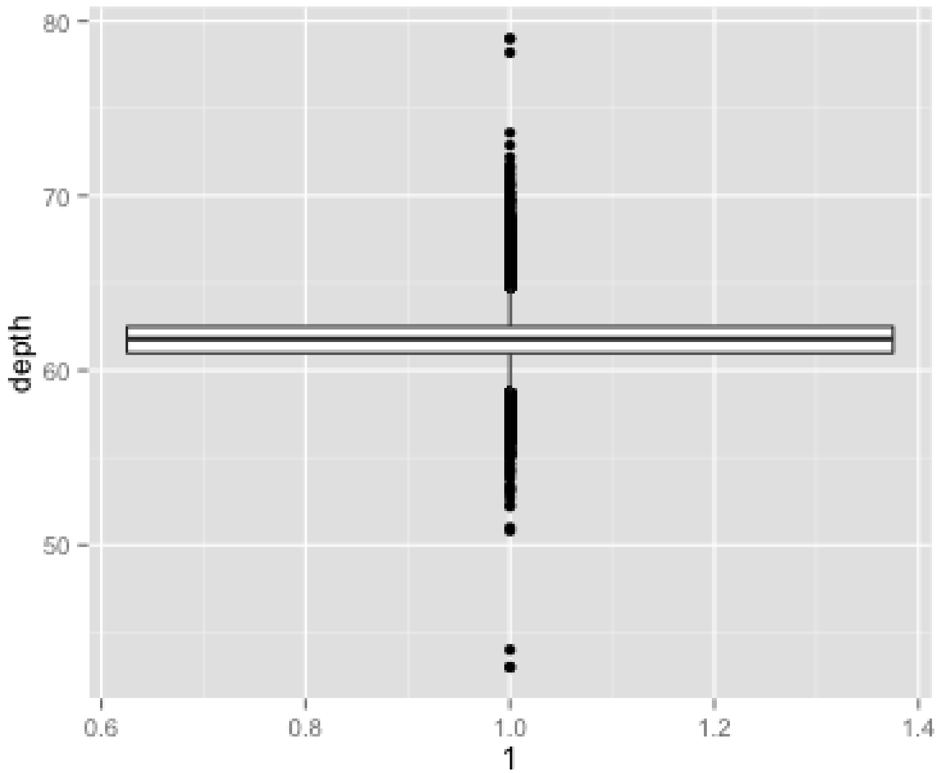
This means that you can save some typing and make your code more readable too

```
myplot <- ggplot(diamonds, aes(carat, price)) + geom_point()
myplot + facet_grid(~ color, labeller=label_both)
myplot + facet_wrap(~ color, ncol=4)}
```

Other plots

We can summarize univariate distributions using boxplots

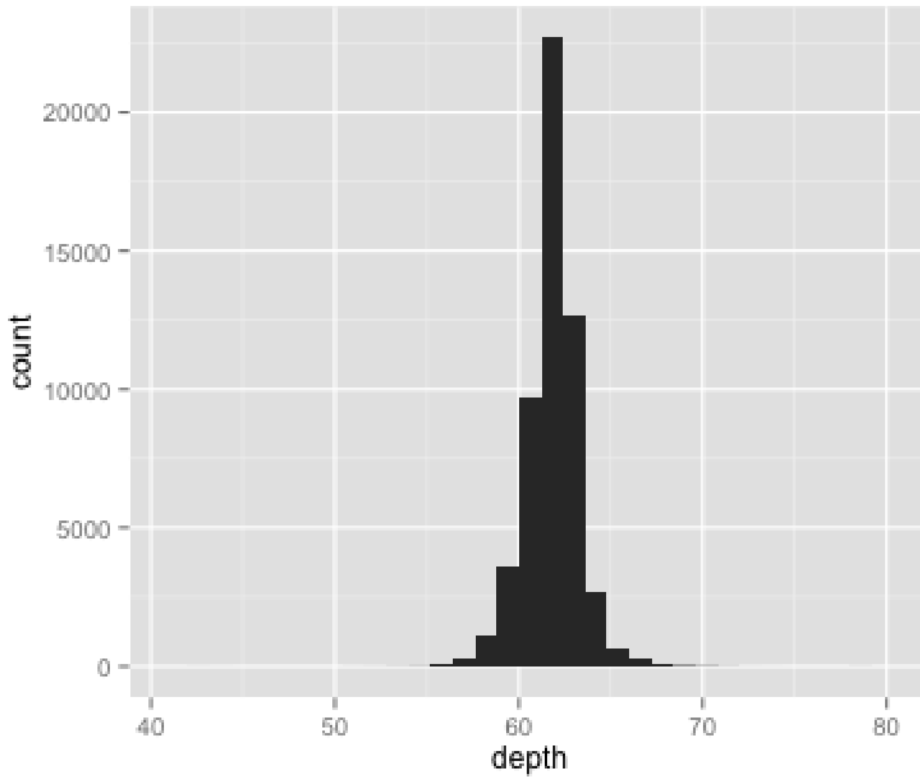
```
ggplot(diamonds, aes(1, depth)) + geom_boxplot()
```



Other plots

However, a histogram would be a better choice here

```
ggplot(diamonds, aes(depth)) + geom_histogram()
```

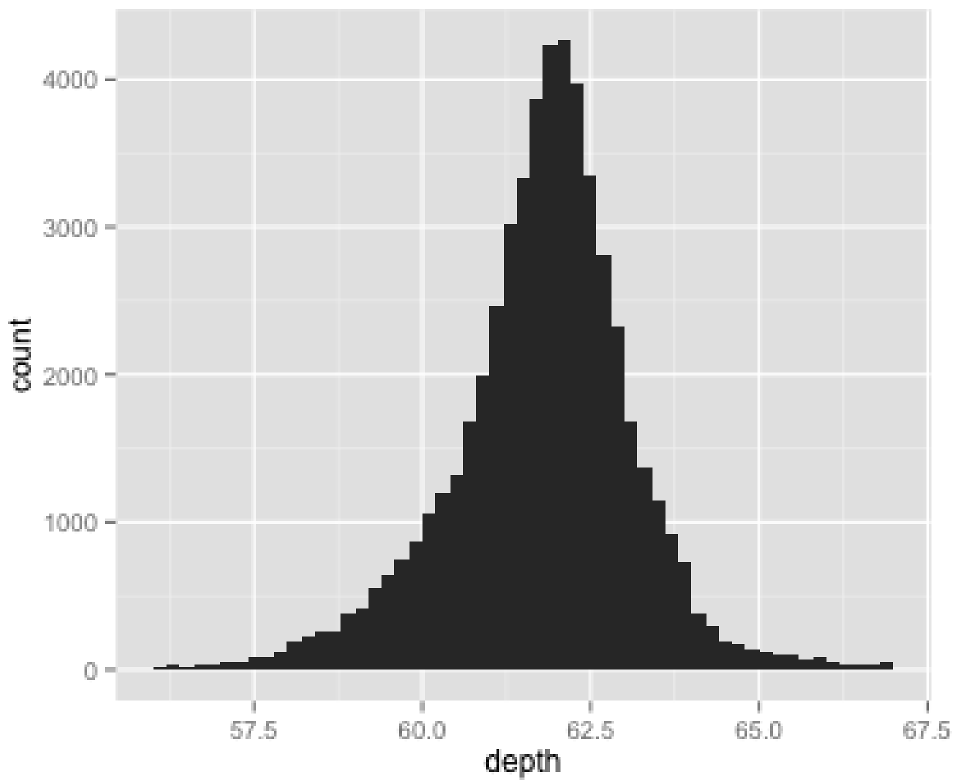


Notice the difference in the `aes` call; `boxplot` is really designed for multiple categories!

Other plots

Unfortunately, the default options in **`histogram`** may not be sensible, and you often need to adjust the **`binwidth`** and **`xlim`**

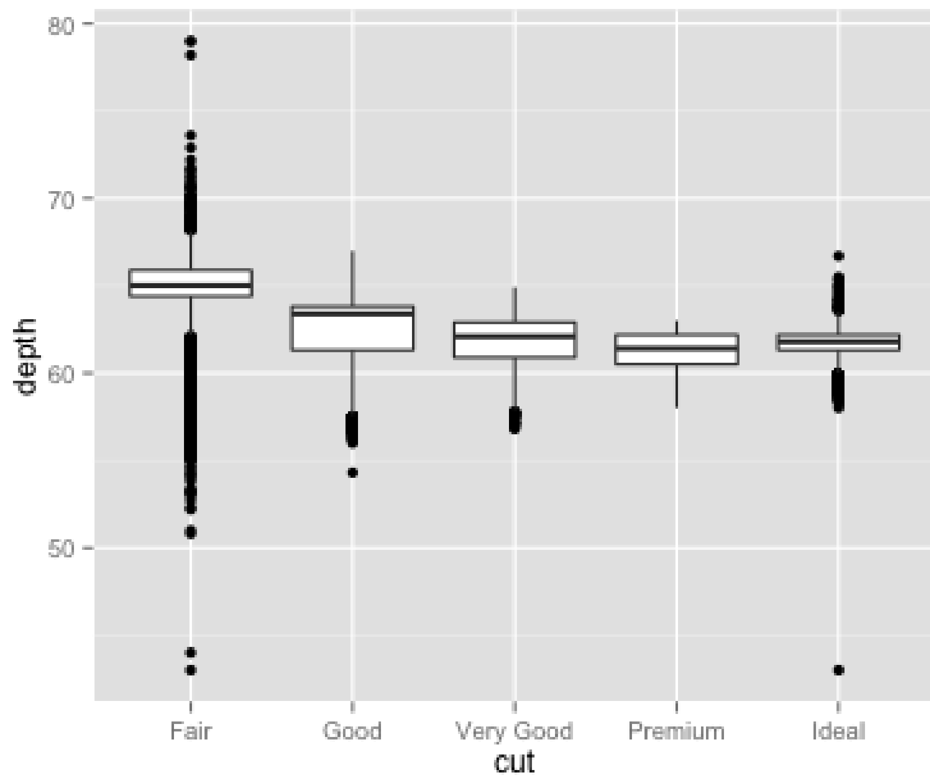
```
ggplot(diamonds, aes(depth)) + geom_histogram(binwidth=0.2) + xlim(56,67)
```



Other plots

A better use of `boxplot` is when we want to compare distributions of a quantitative variable across categories of a factor variable

```
ggplot(diamonds, aes(cut, depth)) + geom_boxplot()
```

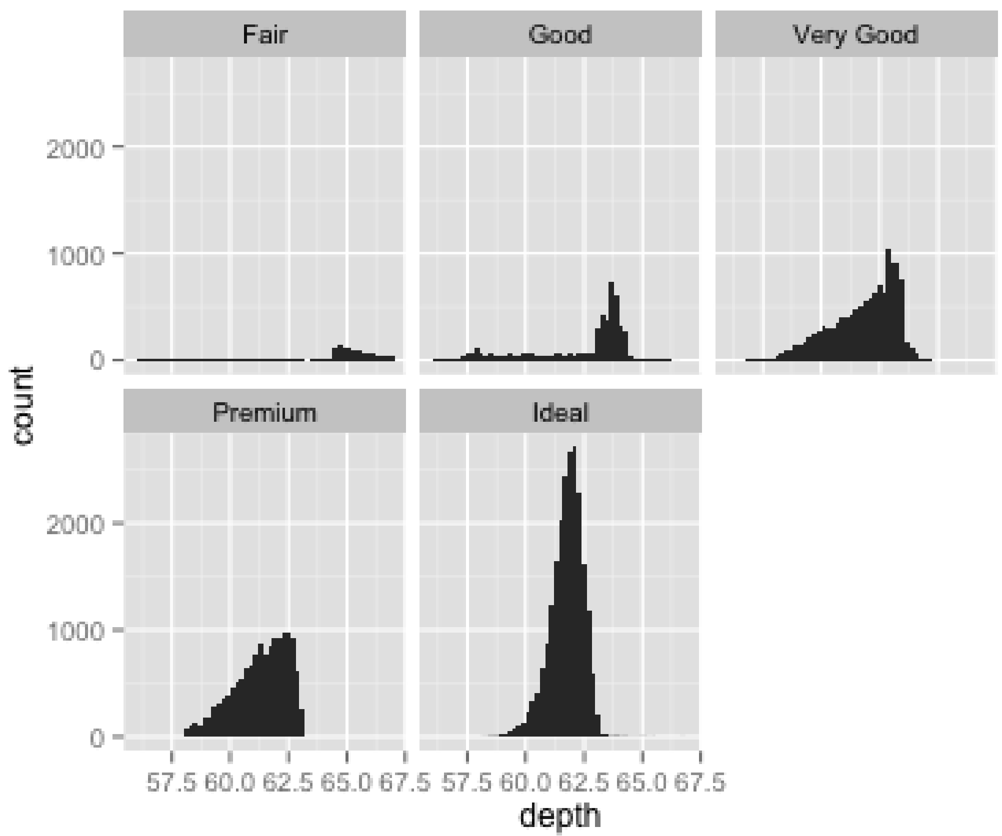


Notice the change in the `aes` call again.

Other plots

We can also get multiple histograms, though we need to either display them separately (less useful when comparing)

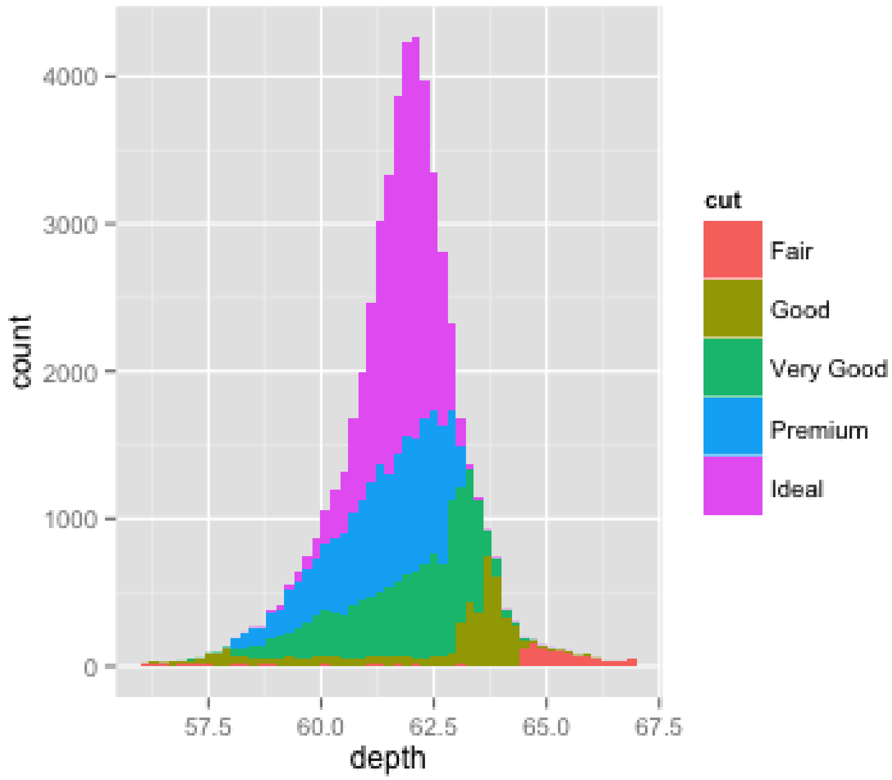
```
ggplot(diamonds, aes(depth)) + geom_histogram(binwidth = 0.2) +
    facet_wrap(~cut) + xlim(56, 67)
```



Other plots

Or, overlay them

```
ggplot(diamonds, aes(depth, fill=cut)) +
    geom_histogram(binwidth=0.2) + xlim(56,67)
```

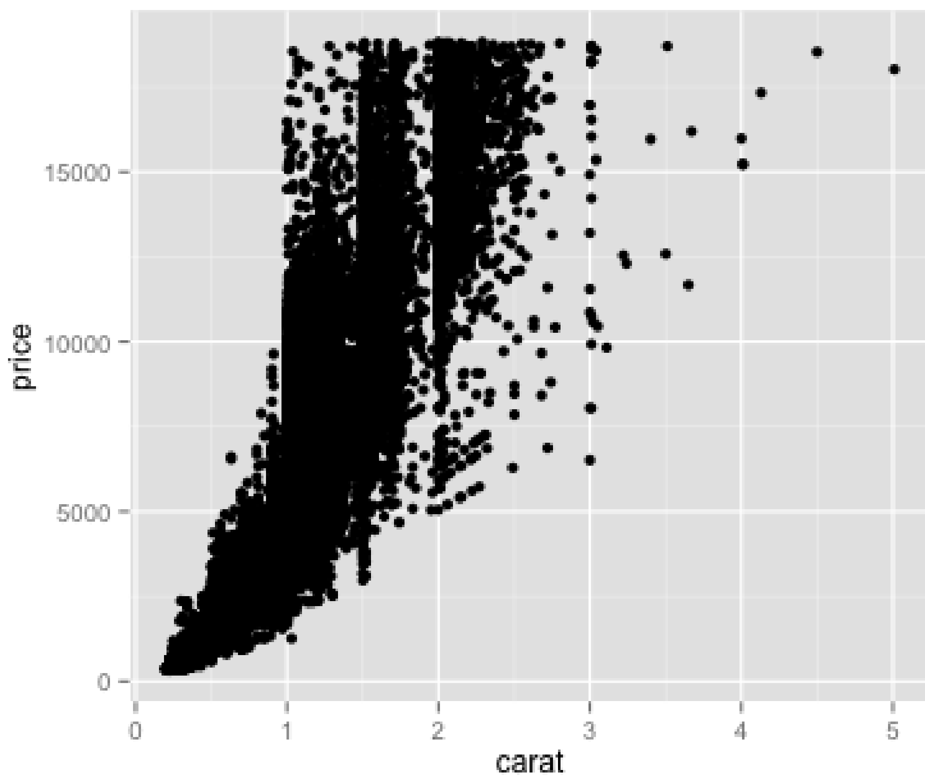


This is a particularly nice feature of `ggplot`

Finally, if you don't like the syntax...

the function `qplot` in `ggplot2` gives you a plot identical to `ggplot`, but with a syntax that is very similar to `plot()`!

```
qplot(carat, price, data=diamonds)
```



But I think this masks the logic of `ggplot`.

`ggplot` vs. R – 2

Some more reasons to use `ggplot`

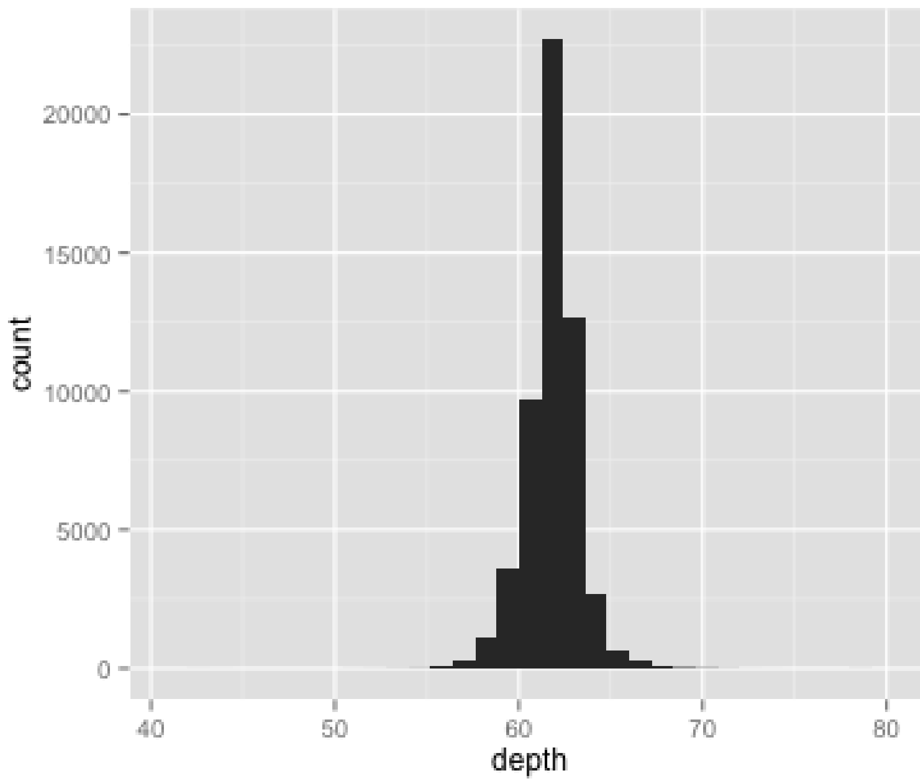
1. It's easy to add additional layers to the plot
2. Margins are handled better – don't get `margins too wide` error messages, or axes that are cut off
3. Effective faceting and coloring for exploring > 2 variables (though keep in mind that it is basically impossible to include info on > 4 variables on a plot)
4. Nice and intuitive defaults, both for basic plots and also for more complicated once (e.g. coloring by a continuous variable)
5. Some things are much easier, e.g., getting overlaid histograms

That's it for `ggplot`! I think you should know what you need to get started...

A (very short) introduction to `shiny`!

Remember our not-very-good-looking histogram

```
ggplot(diamonds, aes(depth)) + geom_histogram()
```



How do we choose the `binwidth`?

Only if there was a way to choose the `binwidth` dynamically!

A (very short) introduction to `shiny`!

Here comes `shiny`

- `shiny` is a web-based graphics interface that allows users to change graphic inputs dynamically
- This allows users to
  - modify and explore the graph
  - pose new (constrained) questions within the range of parameters allowed
- Each `shiny` app has two components
  - `server.R`: the plotting commands, wrapped in `renderPlot`
  - `ui.R`: the user interface
- The `shiny` app can then be run by calling

```
runApp(shinyApp(ui, server))
```

- Alternatively (and I suggest doing this instead), you can put your `server.R` and `hi.R` into a folder, say `myshinyfolder`, and then run the app by calling

```
runApp("myshinyfolder")
```

A (very short) introduction to shiny!

Here's my server.R[§]

```
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  output$distPlot <- renderPlot({
    x    <- diamonds[, 5]  # diamonds -- depth
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white', main='depth')
  })

})
```

Note this is using the base hist() function in R. You can do this with ggplot too.

[§]Based on one of the worked examples in the package

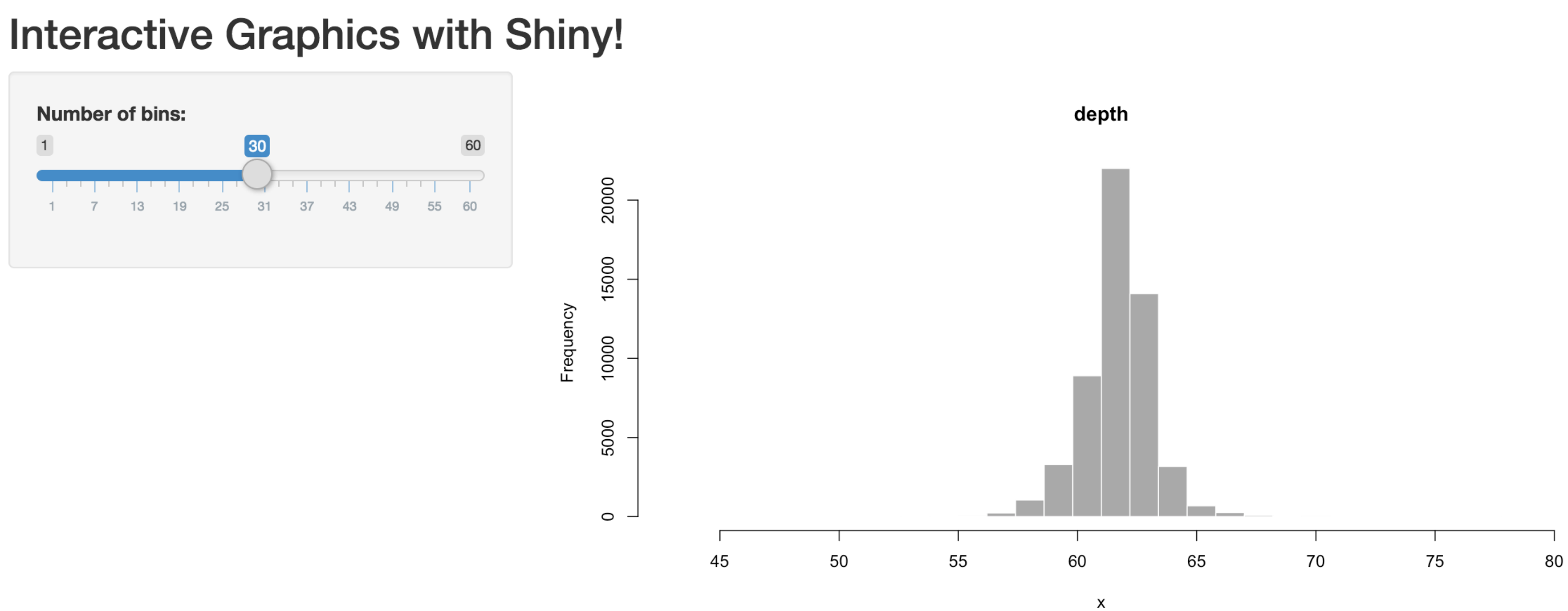
A (very short) introduction to `shiny`!

And my `ui.R`

```
# Define UI for application that draws a histogram
shinyUI(fluidPage(

  # Application title
  titlePanel("Interactive Graphics with Shiny!"),

  # Sidebar with a slider input for the number of bins
  sidebarLayout(
    sidebarPanel(
      sliderInput("bins",
                  "Number of bins:",
                  min = 1,
                  max = 60,
                  value = 30)
    ),

    # Show a plot of the generated distribution
    mainPanel(
      plotOutput("distPlot")
    )
  )
))
```

A (very short) introduction to `shiny`!

And this is what you get

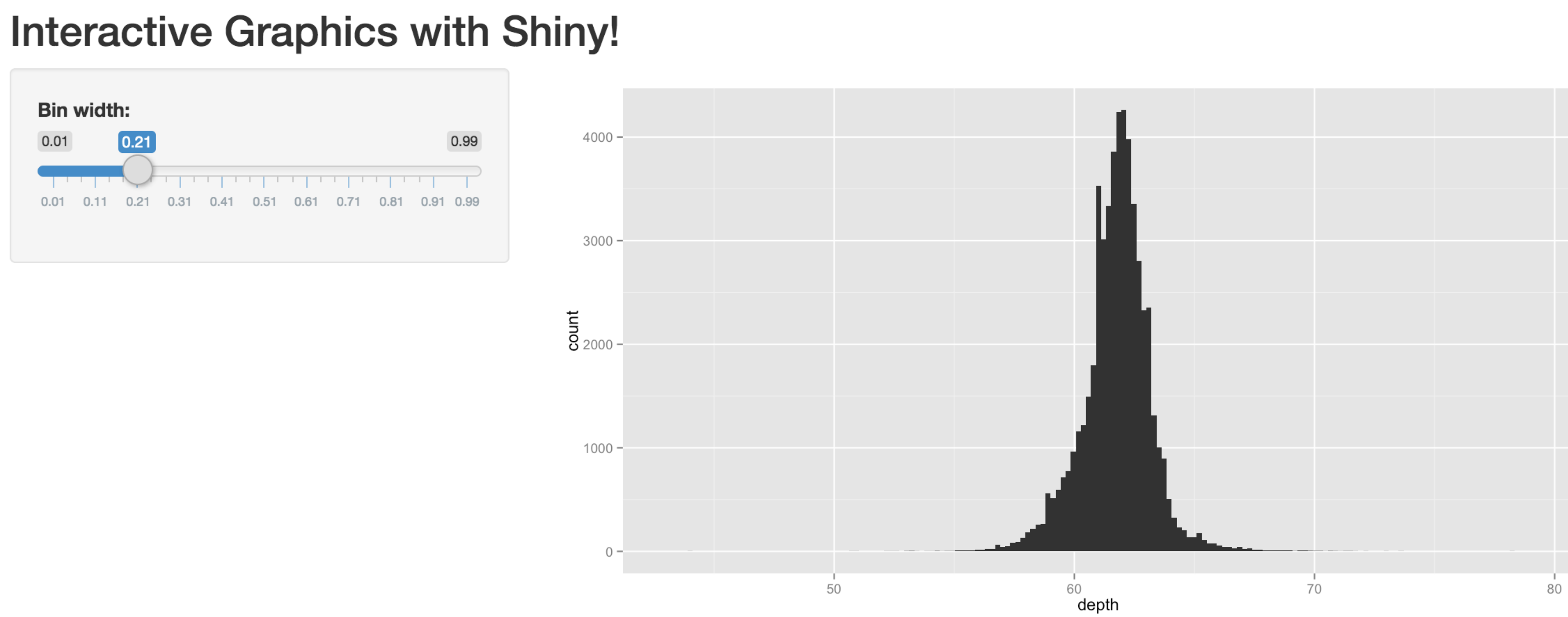


Which is a live web-based dynamic histogram

Note that for this to work, your R session has to be running! Alternatively, you can *publish* your shiny app on shinyapps.io which allows you to use the RStudio servers to host your code.

A (very short) introduction to `shiny`!

You can also do this with `ggplot`



and you get a similar web-based dynamic histogram, but again, your R session has to be running.

This was just a short intro; see shiny.rstudio for good tutorials and more details.

Next time

Last lecture: Leila Zelnick will talk about making posters

This means that the material for your poster needs to be ready by next week!