Maps in R

Michele Tobias, PhD

2026-02-19

Table of contents

Overview

Learning Goals

After completing this chapter, learners should be able to:

- Apply popular mapping workflows to create maps with spatial data
- Assess which workflow works best in different situations
- Apply cartographic best practices to a map composition in R

Workshop Description

In this workshop, participants will learn how to use several options for composing maps in R.

Software & Data

To fully participate in this workshop, learners will need to attend to the following before the workshop:

1. Download and install R
2. Download and install RStudio
3. Download the workshop data and, unzip it, and store it in a location they can find such as a folder dedicated to this workshop (we do not recommend leaving it in your Downloads folder). This is the same data we use in the Cartography for Academic Publications Workshop.

Prerequisites

For the best experience, we recommend learners have the following skills before taking this workshop:

- an introductory understanding of programming in R (for example, you are familiar with how to load libraries, load common data formats like a CSV, and are comfortable running basic commands). See DataLab's R Basics Workshop for help.
- an introductory understanding of spatial data (for example, you are familiar with the difference between raster and vector data). See DataLab's Spatial Data Formats Workshop for help.

- (recommended) review the concepts (first section) of DataLab's Cartography for Academic Publications Workshop
- (recommended) review the Davis R Users Group's Intro to GIS in R Workshop

1 Introduction

R is probably best used for cleaning and analyzing spatial data. Making good quality maps, however, takes a different set of skills.

1.1 Data Set

If you haven't already, download the workshop data from the online repository. If you downloaded a .zip file, unzip the data to a folder you can easily find.

The data we have to work with today is:

- **Lake Monsters** - LakeMonsters.gpkg - locations of lake monsters; global distribution
- **Lakes** - Lakes_GreatLakes-Area.gpkg - a clip of the Natural Earth Data lakes dataset for the Great Lakes and areas adjacent
- **States** - US_CAN_Admin1.gpkg - a clip of the Natural Earth Data states and provinces data for the US and Canada (I'm going to refer to these as "states" for simplicity, but I want to acknowledge that this includes Canadian provinces as well.)

Geopackage (.gpkg) is a single file, open vector format. We're using it today because it's one file per dataset (unlike Shapefile), which makes data management so much easier. See the README.txt file that comes with the data download for more details and sources of the data.

1.1.1 Data Processing

We'll be working with an international dataset of locations of lake monsters, the most famous of which is arguably Nessie who supposedly lives in Loch Ness in Scotland. This dataset was assembled from Wikipedia's List of Lake Monsters. The lake names were geocoded (you can find the R script that I wrote to process the data in the r_scripts folder of this repo, corrected, then exported to a geopackage file. Why did I process this data for you? It took a few hours to do and requires skills we are not focusing on in this workshop.

1.1.2 Recommended File Structure

This is the folder structure this workshop will assume you have:

Directory Structure

workshop_maps_in_r/
data/ dem.tif

```
LakeMonsters.gpkg

Lakes_GreatLakes-Area.gpkg

README.txt

US_CAN_Admin1.gpkg
```

output/

1.2 Map Details

Let's pretend we want to a map for a journal article about the underpinnings of the distribution of cryptozoology sightings in the northeastern US and southeastern Canada, focusing on lake monsters, creatures reported to live in lakes that are mainly known from folklore and typically take on the the shape of extinct or extraordinarlily large living reptiles. You want your readers to understand the relationship between the monster locations and also their location on the planet. In our imaginary scenario, we plan to submit our paper to one of the Nature journals.

What's the story? What should readers learn from this map? What data do you need to tell that story effectively?

The story I plan to tell is: where are these monsters reported to live in the US Great Lakes area? What are their names? For reference, what lakes and states or provinces are they in? The story you want to tell might be different, so feel free to make adjustments as needed.

1.3 Load Spatial Data

Open RStudio and create a new script file. Save it in your folder for this workshop.

Set your working directory to your workshop folder

```r
# set your working directory

setwd("path/to/your/workshop/folder/workshop_maps_in_r")
```

Load the libraries that we will need. If you have not installed these packages yet, you can do so with the command `install.packages("PackageName")`

```r
# load libraries

library(sf)     # for working with vector spatial data
library(terra)  # for working with raster spatial data
```

Let's load the data we'll need.

```r
# read spatial data

monsters <- st_read("data/LakeMonsters.gpkg")
```

```
Reading layer `LakeMonsters' from data source
  `C:\Users\mmtobias\Documents\GitHub\workshop_maps_in_R\data\LakeMonsters.gpkg'
  using driver `GPKG'
Simple feature collection with 55 features and 9 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: -123.4696 ymin: -41.00612 xmax: 144.3105 ymax: 65.35767
Geodetic CRS:  WGS 84
```

```r
lakes <- st_read("data/Lakes_GreatLakes-Area.gpkg")
```

```
Reading layer `Lakes_GreatLakes-Area' from data source
  `C:\Users\mmtobias\Documents\GitHub\workshop_maps_in_R\data\Lakes_GreatLakes-
Area.gpkg'
  using driver `GPKG'
Simple feature collection with 110 features and 10 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -101.2215 ymin: 35.93167 xmax: -64.94982 ymax: 54.41538
Geodetic CRS:  WGS 84
```

```r
states <- st_read("data/US_CAN_Admin1.gpkg")
```

```
Reading layer `US_CAN_Admin1' from data source
  `C:\Users\mmtobias\Documents\GitHub\workshop_maps_in_R\data\US_CAN_Admin1.gpkg'
  using driver `GPKG'
Simple feature collection with 64 features and 83 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -178.1945 ymin: 18.96391 xmax: -52.65365 ymax: 83.11611
Geodetic CRS:  WGS 84
```

```r
dem <- rast("data/dem.tif")
```

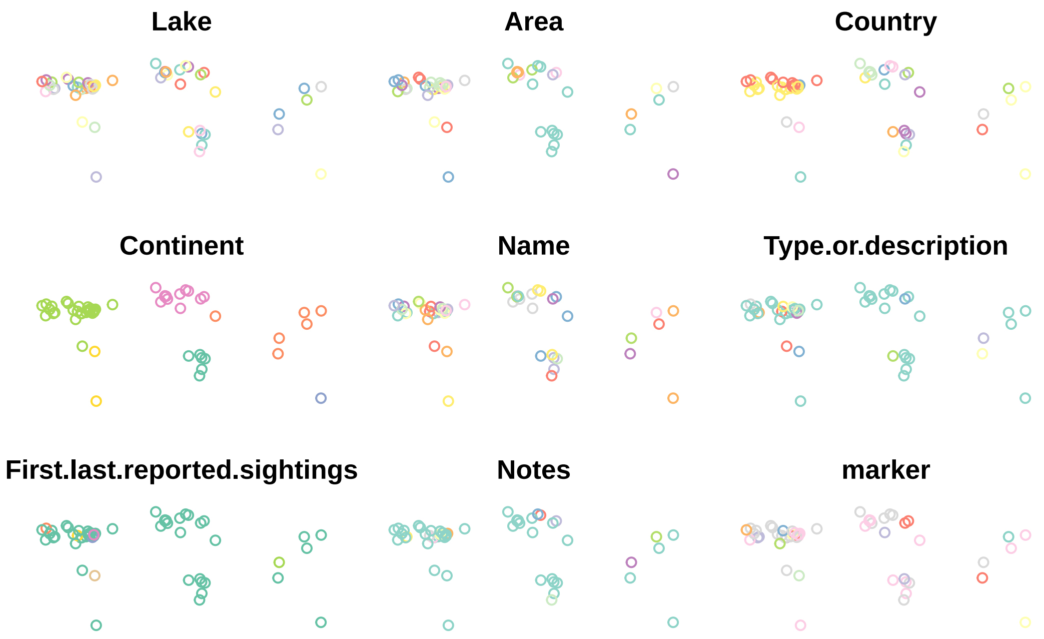
2 Base R

2.1 Intro to plot()

R comes with tools to create basic graphs like scatter plots. Many spatial data packages extend these tools to work with spatial data, allowing us to use the `plot()` function to visualize spatial data in map form. This is often a quick and convenient way to visualize your data to check results.

It seems pretty intuitive, but let's look at what happens if we use the plot command to plot our monsters data:

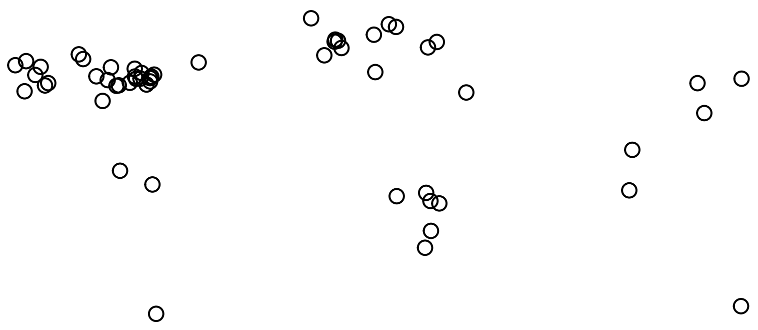
```
plot(monsters)
```



Whoops! What happened? If we don't specify which column in the attribute table to plot, `plot()` tries to plot all of them sequentially. If you have a large dataset, this can take a while and it's almost never what we actually want.

Let's just plot the shapes without any data. We can do this by specifying that we want to plot the geometry column. In this case, it's called "geom". We can also use the `st_geometry()` function to access the geometries.

```
plot(st_geometry(monsters))
```

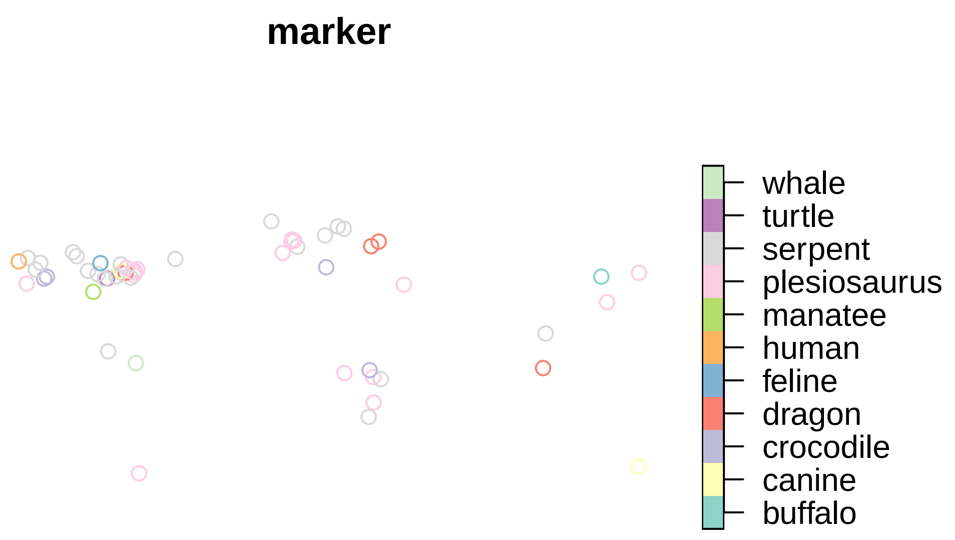


```
#Both of these work, but require you to know the name of the geometry column:
#plot(monsters$geom)
#plot(monsters["geom"])
```

We can see the distribution of the data and if you use your imagination, you might be able to make out the general shapes of the continents.

Let's plot them by marker type:

```
  plot(monsters["marker"])
```



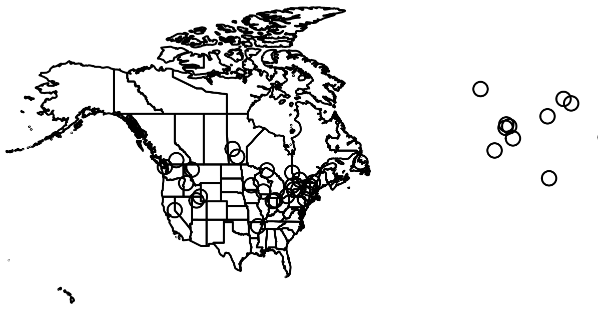
That's pretty rough, but it gives us an idea of how the marker (animal forms) are distributed in the dataset.

2.2 Multiple Layers

We've seen that using `plot()` to visualize a single layer can be a quick way to make sure a data set contains what we think it does, but sometimes we need more context.

You might have noticed that each time we call the `plot()` function, the previous data we plotted disappears. So how can we show more than one layer on the same plot? `plot()` has an argument called `add` that is boolean (either `TRUE` or `FALSE`) and the default is set to `FALSE`. If we specify `TRUE`, we can add multiple layers to the same plot. It draws the plots in order, so whatever we want to be the lowest layer, we need to plot that first. Let's try plotting the monsters (points) over the states layer (polygons):

```
plot(st_geometry(states))
plot(st_geometry(monsters), add=TRUE)
```



Ok! We've got two layers on the same set of axes! Yes, it's hard to read, but we can fix that next.

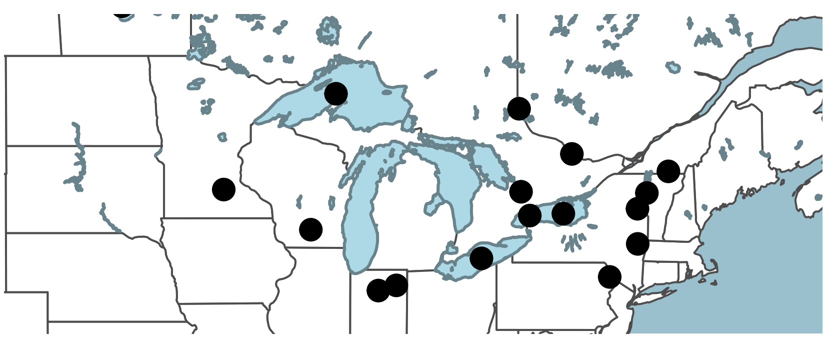
2.3 Styling

The goal of styling our data is to help it communicate better. Open circles on top of state outlines are definitely difficult to understand. Let's change the plotting arguments to make something more readable and to fit the story we want to tell.

How do we know what arguments are available to us? We can reference the documentation for the sf package's version of plot here or you can type `?plot` into your RStudio console and pick the sf version of the help for plot in the Help panel.

Most of the plotting arguments will need to be set in the first plot. The arguments in subsequent plots will just style the data.

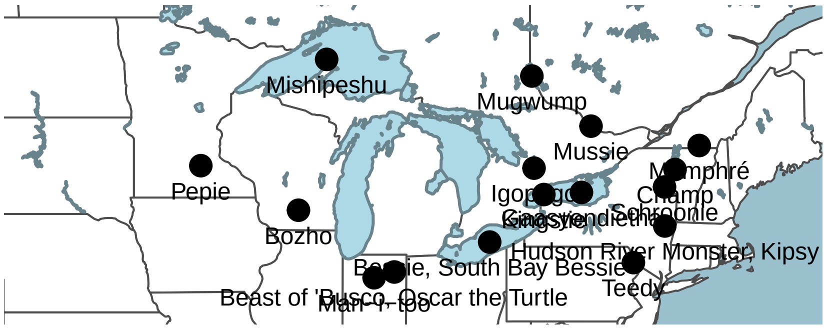
```
#create a bounding box object for the zoom area of our map (the Great Lakes region)
aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))


plot(
  st_geometry(states),        #plot the shape of the states
  col="white",                #fill color for the polygons (states)
  border="gray30",              #line color for the polygons (states)

  #xlim=c(-89, -80),            #set the limit of the x axis (longitude)
  #ylim=c(40, 49),              #set the limit of the y axis (latitude)
  extent=aoi,                   #set the map extent to our AOI bounding box
  #axes=TRUE,                   #makes ticks with lat/long coords... kinda messy
  bg="lightblue3"               #set the color behind the states (the ocean)
  )

plot(st_geometry(lakes),
     col="lightblue",
     border="lightblue4",
     lwd=1.5,                    #set line weight
     add=TRUE)


plot(st_geometry(monsters),
     pch = 19,                  #set the points to filled circles
     cex = 1.5,                  #set symbol size
     add=TRUE)
```



2.4 Add Text Labels

Let's add some text labels so we know what the names of each of the monsters are. We can keep the same code to make the plots, but we need to add a call to the function `text()` at the end to write the text labels on top of the rest of the map.

```r
#create a bounding box object for the zoom area of our map
aoi <- st_bbox(c(xmin = -89, xmax = -80, ymax = 49, ymin = 40), crs = st_crs(4326))


plot(
  st_geometry(states),
  col="white",
  border="gray30",
  extent=aoi,
  bg="lightblue3"
  )

plot(st_geometry(lakes),
     col="lightblue",
     border="lightblue4",
     lwd=1.5,
     add=TRUE)


plot(st_geometry(monsters),
     pch = 19,
     cex = 1.5,
     add=TRUE)

text(monsters,                #locations
     monsters$Name,           #labels
     cex = 0.75,              #scale for the characters (text)
     pos = 1)                 #text position: 1=below, 2=left, 3=above, 4=right
```



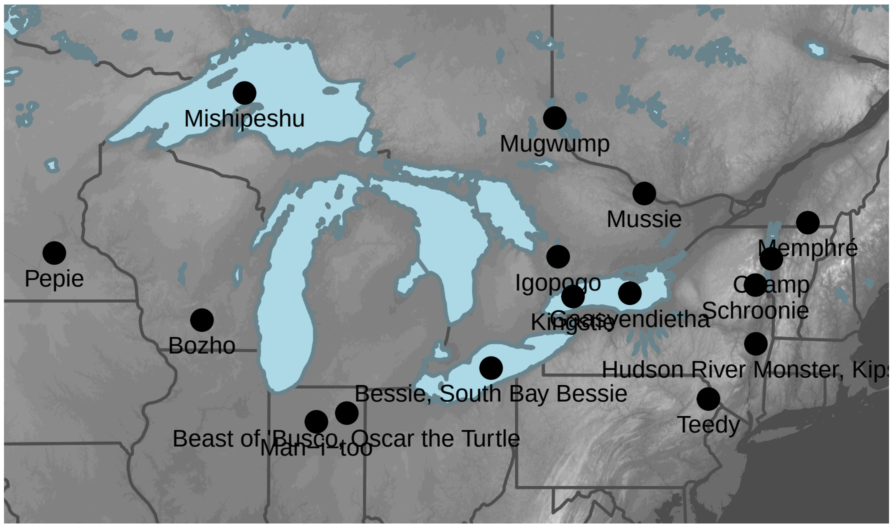
We don’t have much control over label placement with base `plot()` or the graphic’s final dimensions. If you want to edit this with more control, you could export a pdf file and edit it with a vector illustration software like Inkscape or Adobe Illustrator.

2.5 Add Raster Data

Having the state outlines as background information is clean and simple, but perhaps we want to include some information about the physical environment in which these monsters are found. A digital elevation model (DEM) will give us an indication of where the higher elevations are and the complexity of the landscape.

```
#create a bounding box object for the zoom area of our map
aoi <- st_bbox(c(xmin = -95, xmax = -70, ymax = 49.5, ymin = 39), crs = st_crs(4326))

 dem_aoi <- crop(dem, aoi) #crop the DEM to the AOI because you can't set the extent with

 options(terra.pal=gray.colors(20)) #set the colors for the raster

 plot(dem_aoi,                  #plot the cropped raster
    legend = FALSE,             #turn off the legend
    axes = FALSE,               #turn off the lat/long ticks
                                #set the break points on the color ramp: min, high water li
```

```
         breaks = c(
         -67.6842,
         5,
         seq(6, 1574, 88))
         )

    plot(
      st_geometry(states),
      #col="white",              #comment out fill color for transparent fill
      border="gray30",
      lwd = 1.5,
      bg="lightblue3",
      add=TRUE
    )

    plot(st_geometry(lakes),
         col="lightblue",
         border="lightblue4",
         lwd=2,
         add=TRUE)

    plot(st_geometry(monsters),
         pch = 19,
         cex = 1.5,
         add=TRUE)

    text(monsters,
         monsters$Name,
         cex = 0.75,
         pos = 1)
```



2.6 North Arrow & Scale Bar

The base `plot()` function isn't really designed to make publication-ready maps. Adding a scale bar and north arrow are not really in the scope of this particular work flow so we'll skip that here. However, the map we made probably doesn't need those things if our audience is familiar with the Great Lakes area because they map itself would convey a sense of scale and orientation.

2.7 Summary

We've seen that we can create a map quickly using `plot()` that communicates clearly what each layer contains. We have some basic options for controlling the style of the map, but to make a profession-quality map, we probably need a package specifically designed to make maps. We'll cover other map making packages in subsequent sections.

3 tmap

tmap is a popular package for making maps. It uses the concepts of a Grammar of Graphics to layer data as well as visualization rules. If you're familiar with the `ggplot2` package, this will feel similar. If you're not familiar with `ggplot2` (or not a fan), that's ok because `tmap` uses layers in much the same way we just built maps using the base R `plot()` function.

3.1 Simple Map

First, we load the libraries we'll need:

```r
#install.packages("tmap")
library("tmap")
```

```
Warning: package 'tmap' was built under R version 4.4.3
```

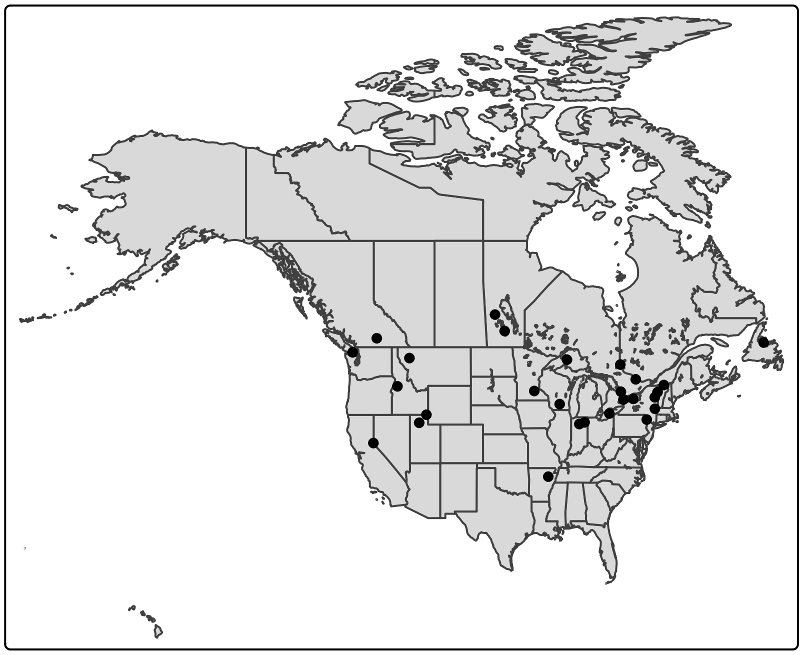
Grammar of Graphics tools typically follow a pattern: first you indicate which data you want to work with, then you indicate the way the data should be styled. Let's map the states data to see a basic no-frills example:

```r
tm_shape(states) +     #the data we want to map
tm_polygons()          #style the data using the defaults
```

3.2 Multiple Layers

With `tmap`, if we want to add multiple layers to the map, we can think of each layer as a separate map that we stack together. For our map, we'll first make the layer that contains the states, then we'll add a layer that contains the lakes, then a final layer that contains the monsters.

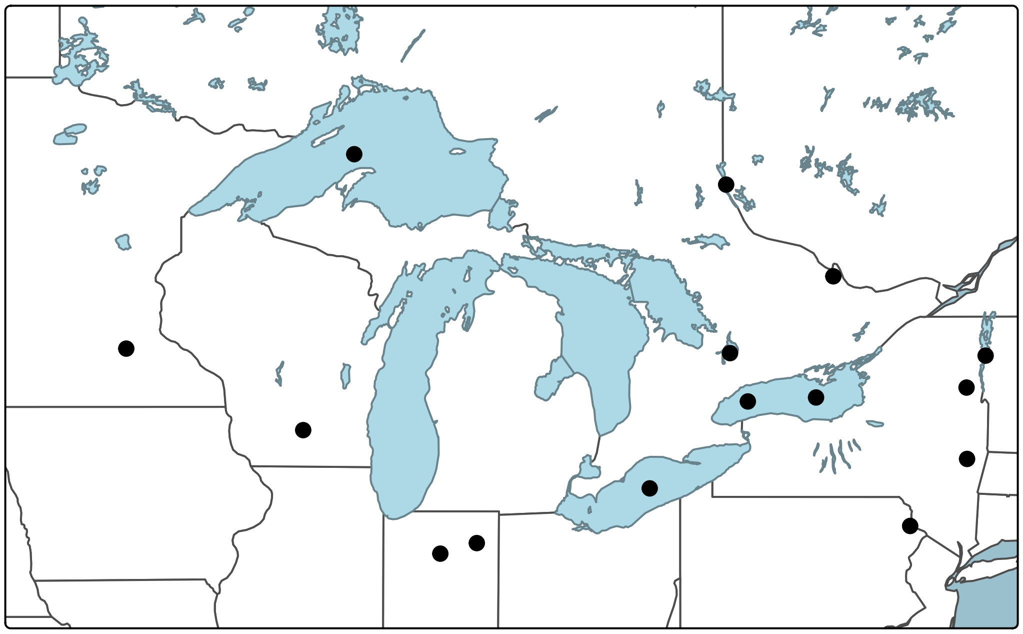
```r
map_states <- tm_shape(states) +
  tm_polygons()

map_lakes <- map_states +          #start with the states map
  tm_shape(lakes) +                #add the lakes data
  tm_polygons()                    #style the lakes data

map_monsters <- map_lakes +        #start with the lakes map
  tm_shape(monsters) +             #add the monsters data
  tm_dots()                        #style the monsters data

map_monsters                       #call the map variable to plot the map inside it
```



3.3 Styling

The goal of styling our data is to help it communicate better. Dots on top of state outlines are definitely difficult to understand. Let's change the plotting arguments to make something more readable and to fit the story we want to tell.

```
#create a bounding box object for the zoom area of our map
aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))


  map_states <-
    tm_shape(states,
       bbox = aoi) +                      #set the extend for the map

  tm_polygons(
    fill="white",                       #fill the polygons with white
    col="gray30") +                     #make the outline dark gray
  tm_layout(bg.color="lightblue3")      #set the background color with the layout options


  map_lakes <- map_states +
    tm_shape(lakes) +
    tm_polygons(
      fill = "lightblue",                   #fill the polygons with light blue
      col="lightblue4"                      #make the outline a darker blue
    )

map_monsters <- map_lakes +
  tm_shape(monsters) +
  tm_dots(size=.5)                        #set the size of the points

map_monsters
```

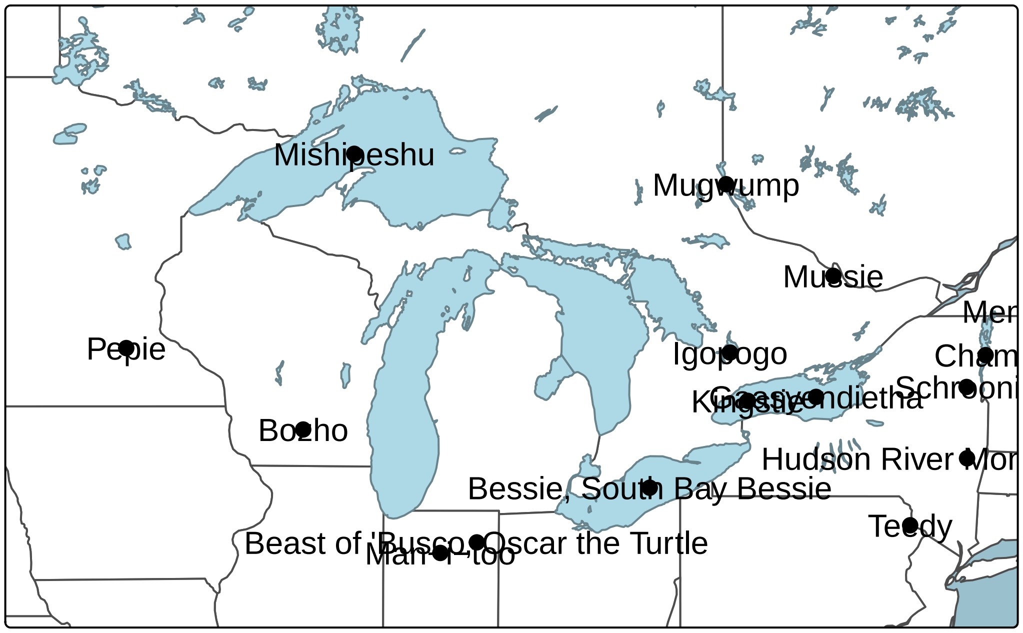


3.4 Add Text Labels

Let's add some text labels so we know what the names of each of the monsters are.

```
#create a bounding box object for the zoom area of our map
aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))


  map_states <-
```

```
    tm_shape(states,
       bbox = aoi) +

  tm_polygons(
    fill="white",
    col="gray30") +
  tm_layout(bg.color="lightblue3")


  map_lakes <- map_states +
  tm_shape(lakes) +
  tm_polygons(
    fill = "lightblue",
    col="lightblue4"
    )

map_monsters <- map_lakes +
  tm_shape(monsters) +
  tm_dots(size=.5) +

  tm_text("Name")              #create labels, specifying the column to use

map_monsters
```

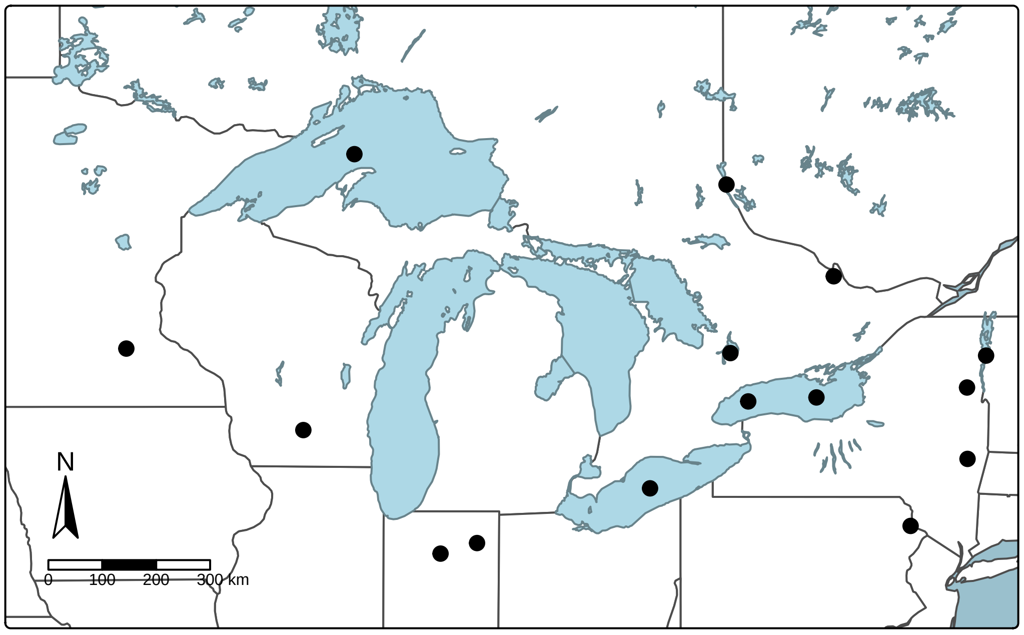


Labeling of points is challenging for `tmap`. It doesn't have a way to avoid label collisions (overlaps) and the placement options don't really do much to avoid labels running over the symbols. I also haven't found much guidance on placing the labels. Most of the tutorials avoid this problem by labeling polygons where the exact placement isn't a concern. Like the discussion of `plot()`, this map would benefit from moving the labels by hand. Export a pdf and edit it in a vector illustration software like Inkscape or Adobe Illustrator.

3.5 North Arrow & Scale Bar

One big benefit of the `tmap` workflow is the built in option to add norh arrows and scale bars. Adding them in is super simple and if you're happy with the defaults, it's pretty quick. We'll put ours in the lower left corner where they won't cover up our monster points. Also, we're adding these to the last layer of the map we built so that the arrow and scale bar don't get covered up by anything else. (Note I'm taking out the labels because they are messy.)

```
#create a bounding box object for the zoom area of our map
aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))


  map_states <-
    tm_shape(states,
       bbox = aoi) +
    tm_polygons(
      fill="white",
      col="gray30") +
    tm_layout(bg.color="lightblue3")


    map_lakes <- map_states +
    tm_shape(lakes) +
    tm_polygons(
      fill = "lightblue",
      col="lightblue4"
      )

map_monsters <- map_lakes +
  tm_shape(monsters) +
  tm_dots(size=.5) +

  tm_compass(position=c("left", "bottom")) +      #add the north arrow (compass rose)
  tm_scalebar(position=c("left", "bottom"))       #add the scale bar


map_monsters
```



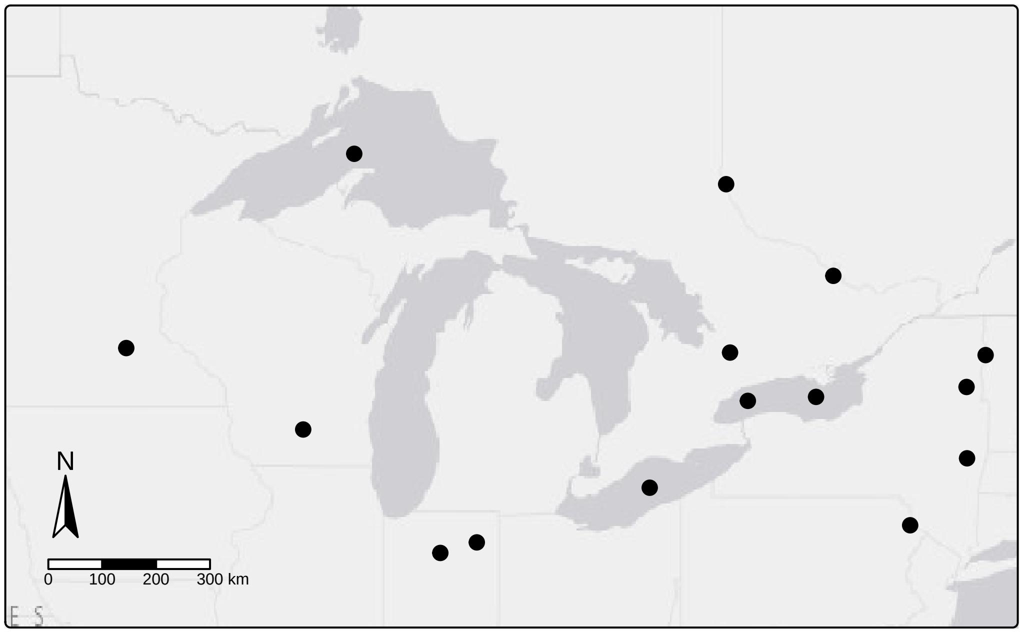
4 Ready-Made Tile Services

Another benefit of the `tmap` workflow is the easy availability of using tile services (commonly known as “basemaps” but people use that term to mean so many other things as well). Tile services are continuous map layers that are stored in square chunks (tiles) so you only need to load the part of the world in your area of interest. One tile service that many people are familiar with is Google Maps, but open data like OpenStreetMap is also available as a tile service.

Remember that even if a tile service makes a dataset easy to use, you still need to comply with the terms of service for that specific dataset.

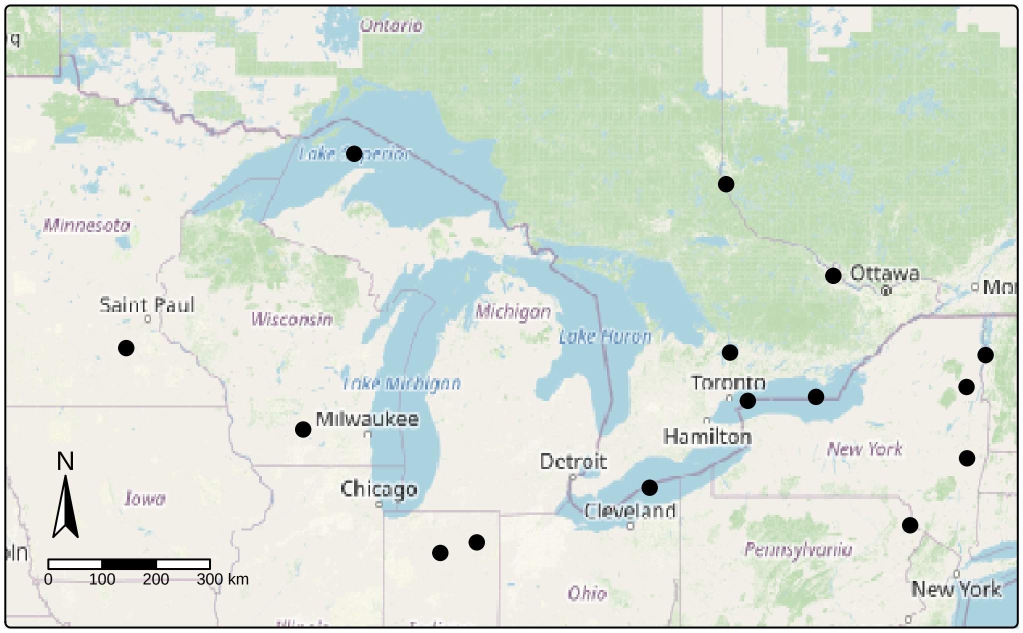
Let’s replace our base layers (states and lakes) with the default basemap.

```r
#create a bounding box object for the zoom area of our map
aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))


    map_monsters <-
      tm_basemap() +                                #start with the default base map
    tm_shape(monsters, bbox = aoi) +      #move the bounding box parameter here
    tm_dots(size=.5) +

    tm_compass(position=c("left", "bottom")) +
    tm_scalebar(position=c("left", "bottom"))


map_monsters
```



What about a base layer with some more color or labels? Let's add the OpenStreetMap tile service.

```
#create a bounding box object for the zoom area of our map
aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))


    map_monsters <-
      tm_basemap("OpenStreetMap") +                       #start with the default base map
    tm_shape(monsters, bbox = aoi) +
    tm_dots(size=.5) +

    tm_compass(position=c("left", "bottom")) +
    tm_scalebar(position=c("left", "bottom"))


map_monsters
```



4.1 Interactive Maps

Being able to pan and zoom around a plot is a useful tool for understanding your data or the results of an analysis. tmap makes this relatively easy with the `tmap_leaflet()` function. Let's plot our monster data on the default basemap. The pattern of the code should look familiar: set up the map, building up layers from the bottom, and then plot the map.

```
  #create a bounding box object for the zoom area of our map
  aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))

leaflet_map =
  tm_shape(monsters, bbox = aoi) +      #add the monster data
  tm_dots(size=.5)                      #style the monster data

tmap_leaflet(leaflet_map, show = TRUE)  #plot the monster data
```

Notice the source information for the basemap printed in the lower right corner of the map. If you want to publish this map, you'll need to check the Terms of Service for this data source.

Try zooming out and panning around the world to see more lake monster locations in context. You can change your basemap like we did earlier in the workshop with a parameter like `tm_basemap("OpenStreetMap")` when you define your map contents.

4.2 Summary

We've seen that we can build a map in layers using `tmap` in a similar manner to the workflow we used in the base `plot()` workflow. `tmap` has more built-in features for composing maps and finer control over some aspects of maps like north arrows and scale bars. It also has built-in base map tools which makes making a map quick. But `tmap` still doesn't have an easy way to deal with label placement.

5 ggplot2

As you have seen so far in this workshop, R has some useful tools for visualizing spatial data built in and there are packages that you can work with to make maps that are more refined. Next we'll talk about the ggplot2 package so you can compare and contrast three of the most popular ways to make maps in R.

ggplot2 (commonly referred to as "ggplot") is a popular package for plotting data in R. It has been extended to work with spatial data. Like `tmap`, it also uses the concepts of a Grammar of Graphics to layer data as well as visualization rules. ggplot is a big topic that could be its own workshop, so today we'll cover the basics so you can learn more on your own if this sparks your interest.

A common point of confusion is the difference between ggplot and ggmap. ggmap is another popular package for making maps in R that extends the `ggplot2` concept. ggmap assumes you want to use a basemap with every plot. We will focus on ggplot in this workshop.

5.1 Simple Map

First, we load the libraries we'll need:

```
#install.packages("ggplot2")
library("ggplot2")
```

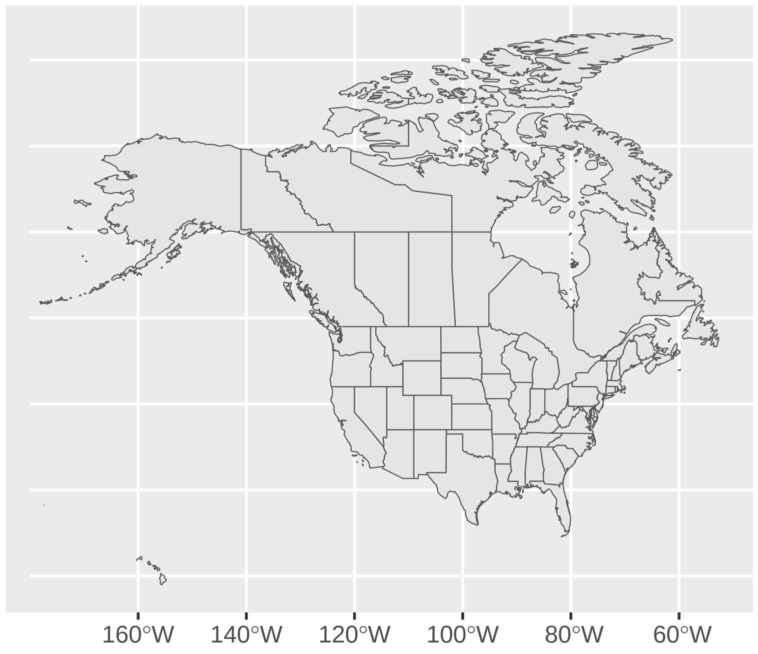
```
Warning: package 'ggplot2' was built under R version 4.4.3
```

```
library(ggspatial)
```

```
Warning: package 'ggspatial' was built under R version 4.4.3
```

Grammar of Graphics tools typically follow a pattern: first you indicate which data you want to work with, then you indicate the way the data should be styled. Let's map the states data to see a basic no-frills example:

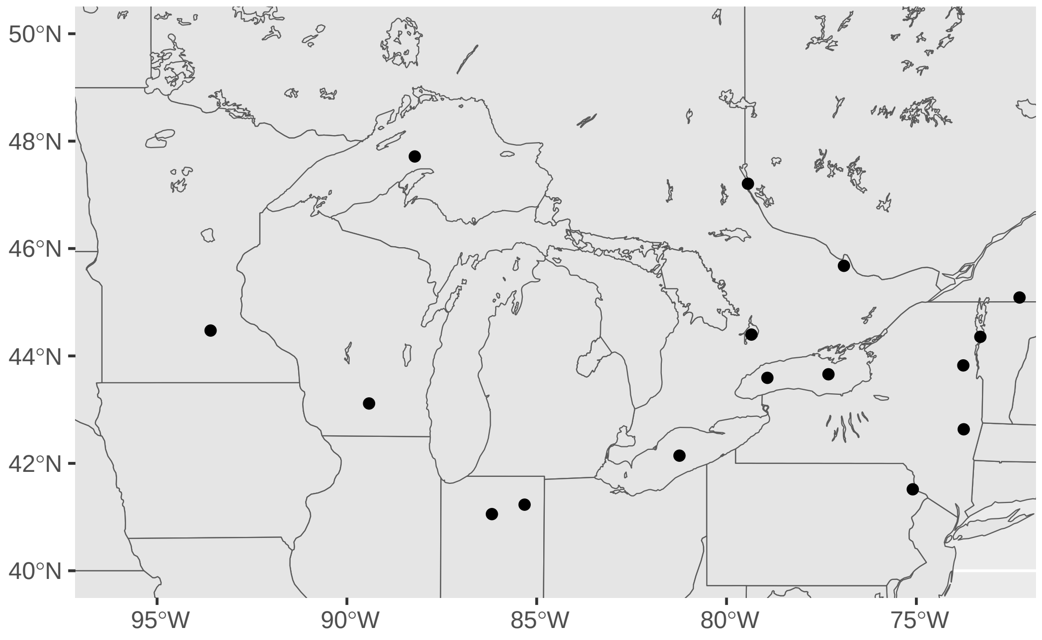
```r
ggplot(states) +        #indicate which data to plot
  geom_sf()             #use the default style for sf objects
```



5.2 Multiple Layers

With `ggplot2`, if we want to add multiple layers to the map, we can think of each layer as a separate map that we stack together. For our map, we'll first make the layer that contains the states, then we'll add a layer that contains the lakes, then a final layer that contains the monsters.

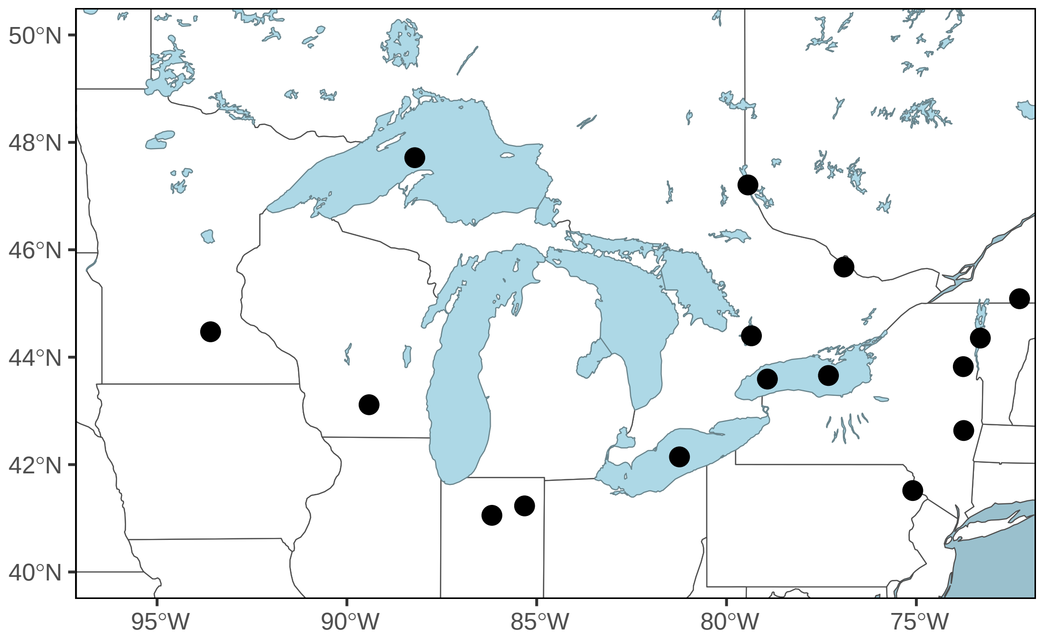
```r
aoi <- st_bbox(c(xmin = -96, xmax = -73, ymax = 50, ymin = 40), crs = st_crs(4326))

ggplot() +

  geom_sf(data = states) +          #plot the states

  geom_sf(data = lakes) +           #plot the lakes and make them blue

  geom_sf(data = monsters) +        #plot the monsters

  coord_sf(                         #set the zoom based on our AOI bounding box
     xlim=c(aoi[1], aoi[3]),
     ylim=c(aoi[2], aoi[4]))
```



5.3 Styling

The goal of styling our data is to help it communicate better. Dots on top of state outlines are definitely difficult to understand. Let's change the plotting arguments to make something more readable and to fit the story we want to tell.

```
gg_monsters <- ggplot() +      #assign the results of the plot to a variable

geom_sf(data = states,
        fill = "white",        #fill the states with white
        color = "gray30") +    #make the outline a dark gray

geom_sf(
  data = lakes,
  fill = "lightblue",          #fill the lakes with blue
  color = "lightblue4") +      #make the outline a gray-blue

geom_sf(data = monsters,
        size = 3                #make the points larger
        ) +

    coord_sf(
      xlim=c(aoi[1], aoi[3]),
      ylim=c(aoi[2], aoi[4]))

    ocean_color <- "lightblue3"     #pick a color for the ocean

    gg_monsters +                   #add theme parameters to the plot & remove lines
      theme(
        panel.background = element_rect(fill = ocean_color, color = ocean_color),
        panel.grid.major = element_line(color = ocean_color),
        panel.grid.minor = element_line(color = ocean_color),
        panel.border = element_rect(color = "black", fill = NA)   #add a black neatline
        )
```

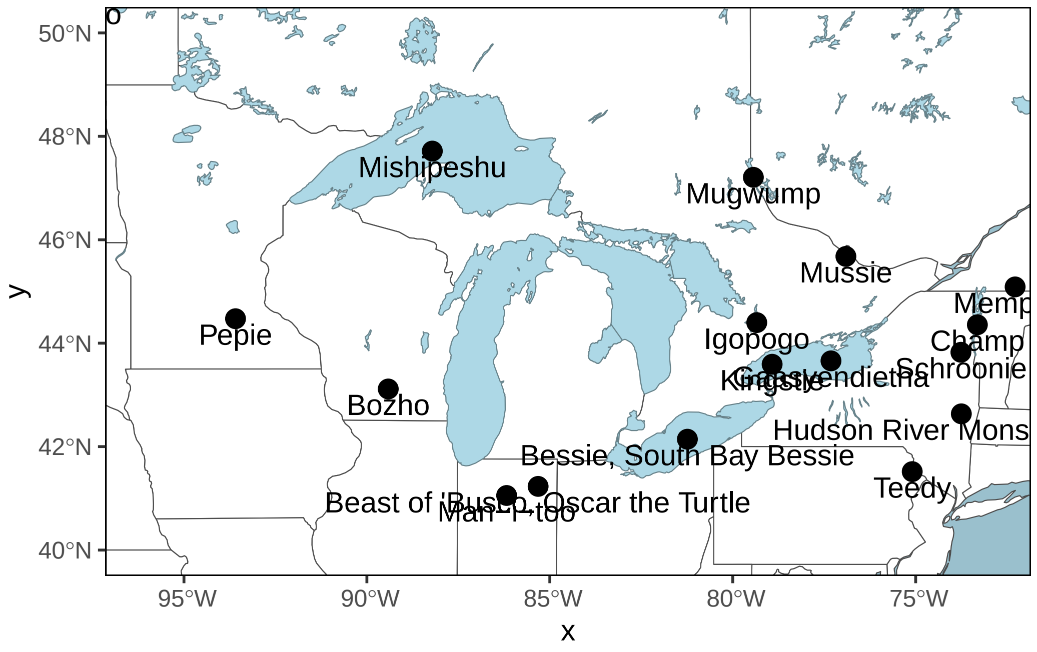


5.4 Add Text Labels

Let's add some text labels so we know what the names of each of the monsters are. We can just update the last line of the code we wrote above, adding the labeling details.

```
gg_monsters +
theme(
  panel.background = element_rect(fill = ocean_color, color = ocean_color),
  panel.grid.major = element_line(color = ocean_color),
  panel.grid.minor = element_line(color = ocean_color),
  panel.border = element_rect(color = "black", fill = NA)
) +
geom_sf_text(data=monsters, aes(label = Name), nudge_y = -.3) #add the label text and nudge
```

```
Warning in st_point_on_surface.sfc(sf::st_zm(x)): st_point_on_surface may not
give correct results for longitude/latitude data
```



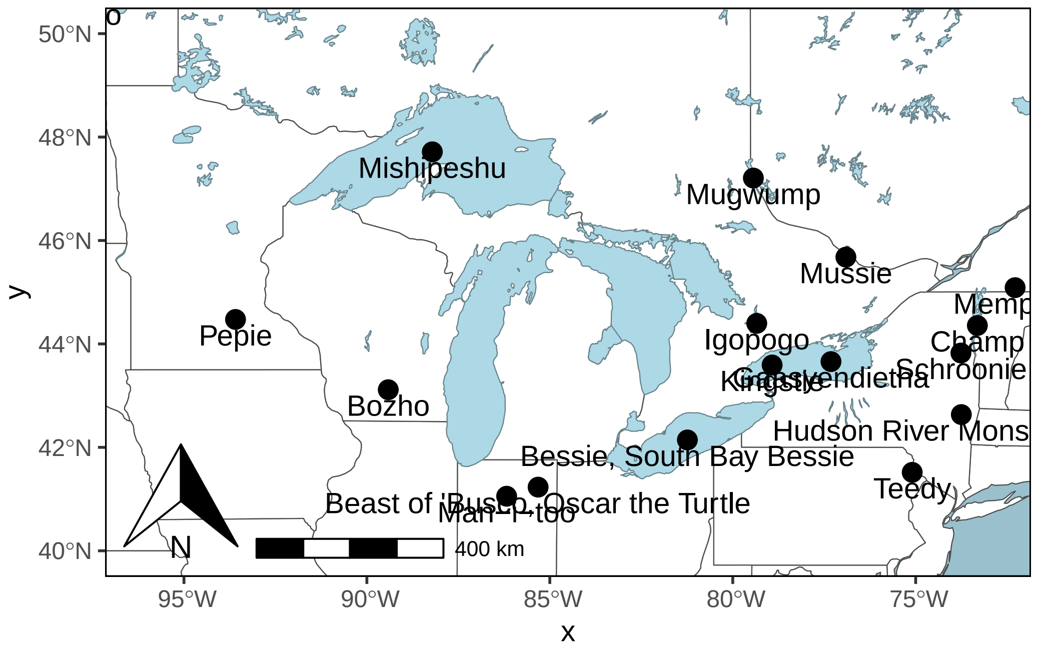
Labeling of points is challenging for all of the methods we've looked at. ggplot2 has some options to avoid label collisions (overlaps) that you you can explore more on your own. If you want finer label placement control, export a pdf and edit it in a vector illustration software like Inkscape or Adobe Illustrator.

5.5 North Arrow & Scale Bar

`ggplot2` does not come with the ability to add map elements like a north arrow or scale bar to your map, but we can add this capability with another package: ggspatial.

```
gg_monsters +                           #add theme parameters to the plot & remove lines
  theme(
    panel.background = element_rect(fill = ocean_color, color = ocean_color),
    panel.grid.major = element_line(color = ocean_color),
    panel.grid.minor = element_line(color = ocean_color),
    panel.border = element_rect(color = "black", fill = NA)
  ) +
  geom_sf_text(data=monsters, aes(label = Name), nudge_y = -.3)+
  annotation_scale(location = "bl", pad_x = unit(2, "cm"))+         #add the scale bar and mo
  annotation_north_arrow(location = "bl", which_north = "grid")     #add the north arrow
```

```
Warning in st_point_on_surface.sfc(sf::st_zm(x)): st_point_on_surface may not
give correct results for longitude/latitude data

Scale on map varies by more than 10%, scale bar may be inaccurate
```



This map is far from beautiful, but it's a good first draft. If you like ggplot2, there's a lot more you can learn about refining placement and style options, but you've seen the basics to know what's possible and you've got a starting point to learn more on your own.

6 Ready-Made Tile Services

ggmap is another popular package for making maps in R that extends the `ggplot2` concept. ggmap assumes you want to use a basemap with every plot.

Remember that even if a tile service makes a dataset easy to use, you still need to comply with the terms of service for that specific dataset.

Unfortunately, it seems that the current version of ggmap requires you to have a Google API Key to access even non-Google tiles. That's outside the scope of this workshop, but there is documentation on the internet for how to use this tool to build `ggplot2` maps on top of a basemaps.

6.1 Summary

We've seen that we can build a map in layers using `ggplot2` in a similar manner to the workflow we used in the base `plot()` and `tmap` workflow. `ggplot2` is an entire universe of plot creation with many options for fine control and attractive themes that you can explore. It has perhaps the steepest learning curve, but also has big rewards if you have the patience to stick with it.

7 Additional Resources

Below are some other materials we think you might find helpful either as a rereference or as a tool to learn more about this topic.

7.1 Working with Spatial Data

DataLab's Cartography for Academic Publications Workshop

DataLab's Spatial Data Formats Workshop

maptimeDavis - a welcoming community of practice focused on spatial data

7.2 Learning & Using R

DataLab's R Basics Workshop

Davis R Users Group - a welcoming community of practice focused on using the R programming language

7.3 Other Mapping in R Tutorials

Plotting Simple Features

NCEAS R Color Cheatsheet

Making beautiful inset maps in R using sf, ggplot2 and cowplot

tmap website

Elegant and informative maps with tmap

Spatial Data & R

Geocomputation with R: Making maps with R - a good summary of packages for composing maps

7.3.1 ggplot2 Tutorials

R as GIS for Economists

Making Maps with R

ggplot2: Elegant Graphics for Data Analysis