Introduction to Geocoding, Calculating Distances, and Map Making in R

Sarah Lotspeich, R Ladies Nashville

September 12, 2017

A few packages for spatial analysis and visualization in R

1. `ggmap`: A collection of functions to visualize spatial data and models on top of static maps from various online sources (e.g Google Maps and Stamen Maps). It includes tools common to those tasks, including functions for geolocation and routing.
2. `rgdal`: The Geospatial Data Abstraction Library, access to projection/ transformation operations.
3. `ggplot2`: A system for 'declaratively' creating graphics, based on "The Grammar of Graphics". You provide the data, tell 'ggplot2' how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.
4. `ggsn`: Adds north symbols (18 options) and scale bars in kilometers to maps in geographic or metric coordinates created with 'ggplot2' or 'ggmap'.
5. `broom`: The broom package takes the messy output of built-in functions in R, such as lm, nls, or t.test, and turns them into tidy data frames. We will use it here to turn imported shapefiles into data frames.
6. `geosphere`: Spherical trigonometry for geographic applications. That is, compute distances and related measures for angular (longitude/latitude) locations.
7. `gmapsdistance`: Get distance and travel time between two points from Google Maps. Four possible modes of transportation (bicycling, walking, driving and public transportation).

```
library(ggmap)
library(rgdal)
library(ggplot2)
library(ggsn)
library(broom)
library(geosphere)
library(gmapsdistance)
library(wesanderson)
```

Introduction to geocoding

In geospatial analyses, string addresses hold very little meaning just as in our everyday lives the precise latitude and longitude coordinates of our homes are not all that useful. That's where geocoding comes in. The basis of geocoding is the following (respectfully copy-pasted from Google about their Google Maps Geocoding API):

"Geocoding is the process of converting addresses (like a street address) into geographic coordinates (like latitude and longitude), which you can use to place markers on a map, or position the map."

Before we begin, here are a few "best practices" when preparing addresses to be geocoded (taken from the Harvard University Center for Geographic Analysis):

1. Remove suite/ apartment numbers as they will simply create "ties".
2. Be mindful of special characters like @, #, ?, ;, -

3. Buckle down and be prepared to spend a good long while "cleaning" addresses. I spent two days on my pharmacy addresses. I would highly recommend doing it with a quality latte (or perhaps something stronger) in hand.

Example 1: City of Austin Public Art Collection

Background: This file contains the names and address of artworks commissioned through the Austin Art in Public Places Program. Established by the City in 1985, the Art in Public Places (AIPP) program collaborates with local & nationally-known artists to include the history and values of our community into cultural landmarks that have become cornerstones of Austin's identity. The City of Austin was the first municipality in Texas to make a commitment to include works of art in construction projects (taken from here).

```
AustinPublicArt <- read.csv("https://query.data.world/s/93ujuqzxbfjkwnda3y2p8mgv9",
    header = TRUE, stringsAsFactors = FALSE)
colnames(AustinPublicArt)
```

```
## [1] "artist_full_name"            "art_title"
## [3] "art_location_name"           "art_location_street_address"
## [5] "art_location_city"           "art_location_state"
## [7] "art_location_zip"
```

```
AustinPublicArt$art_full_address <- paste(AustinPublicArt$art_location_street_address,
    AustinPublicArt$art_location_city, AustinPublicArt$art_location_state, AustinPublicArt$art_location_
    sep = " ")
unique(AustinPublicArt$art_full_address)  #let's see what we're working with here
```

```
##  [1] "Possum Point - North Bank at Shoal Creek; Austin TX 78701"
##  [2] "2800  Star of Texas Rd.; Austin TX 78719"
##  [3] "2505 Steck Ave.; Austin TX 78757"
##  [4] "1143 Northwestern Ave.; Austin TX 78702"
##  [5] "600 River Street; Austin TX 78701"
##  [6] "1163 Angelina St.; Austin TX 78702"
##  [7] "12500 Amherst Dr.; Austin TX 78727"
##  [8] "500 East Cesar Chavez St.; Austin TX 78701"
##  [9] "7201 Levander Loop; Austin TX 78702"
## [10] "2400 Holly St.; Austin TX 78702"
## [11] "1105 E. Cesar Chavez Street; Austin TX 78702"
## [12] "5803 Nuckols Crossing; Austin TX 78744"
## [13] "3600 Presidential Blvd.; Austin TX 78719"
## [14] "Webberville Road; Austin TX NA"
## [15] "808 Nile Rd.; Austin TX 78702"
## [16] "; Austin Texas 78702"
## [17] "Brush Square Park; 409 East Fifth Street Austin TX 78701"
## [18] "5th Street and Sabine Street; Austin TX 78701"
## [19] "625 East 10th Street; Austin TX 78701"
## [20] "7200 Berkman Drive; Austin TX NA"
## [21] "2220 Barton Springs Rd.; Austin TX 78746"
## [22] "1161 Angelina St.; Austin TX 78702"
## [23] "8011 Beckett Rd.; Austin TX 78749"
## [24] "2500 Exposition Blvd.; Austin TX 78703"
## [25] "2101 Jesse E. Segovia Street; Austin TX 78702"
## [26] "West side between 6th and 7th Street; Austin TX NA"
## [27] "3911 Manchaca Rd.; Austin TX 78704"
## [28] "300 N. Lamar Blvd.; Austin TX 78704"
## [29] "Cesar Chavez; East and West of S. First Street; Austin TX 78701"
## [30] "North Bank near Buford Fire Tower; Austin TX 78701"
## [31] "400 Ralph Ablanedo Drive; Austin TX NA"
## [32] "5801 Westminster Dr.; Austin TX 78723"
## [33] "8637 Spicewood Springs Rd.; Austin TX 78759"
## [34] "300 Cesar Chavez Street; Austin TX NA"
## [35] "5125 Convict Hill Rd.; Austin TX 78749"
## [36] "900 Barton Springs Road; Austin TX NA"
## [37] "2200 Hancock Dr.; Austin TX 78756"
## [38] "601 E 15th St.; 9th Floor Austin TX 78701"
## [39] "Corner of Blessing and Wheatly Ave.; Austin TX NA"
## [40] "Festival Beach Park along the; Nash Hernandez Sr. Road Austin TX 78702"
## [41] "300 South Congress Avenue; Austin TX 78704"
## [42] "7201 Colony Loop Drive; Austin TX 78724"
## [43] "5801 Ainez Dr.; Austin TX 78744"
## [44] "2nd Street & San Antonio; Austin TX 78701"
## [45] "7800 Johnny Morris Road; Austin TX 78724"
## [46] "4108 Todd Lane; Austin TX NA"
## [47] "2608 Gonzales St.; Austin TX 78702"
## [48] "1106 E. Rundberg Ln.; Austin TX 78753"
## [49] "; Austin Texas 78701"
## [50] "3635 RR 620 South; Austin TX 78738"
## [51] "2201 Barton Springs Rd.; Austin TX 78746"
## [52] "12400 Lamplight Village Ave.; Austin TX 78727"
## [53] "301 Nature Center Dr.; Austin TX 78746"
## [54] "34 Robert T. Martinez Jr. St.; Austin TX NA"
## [55] "5010 Old Manor Rd; Austin TX 78723"
## [56] "4411 Meinardus Road; Austin TX 78744"
## [57] "2100 E. 3rd St.; Austin TX 78702"
## [58] "3701 Grooms Street; Austin TX 78705"
## [59] "812 Springdale Road; Austin TX 78702"
## [60] "5811 Nuckols Crossing Rd.; Austin TX 78744"
## [61] "South Bank at South 1st &  Riverside Drive; Austin TX 78704"
## [62] "409 East Fifth St.; Austin TX 78701"
## [63] "651 North Pleasant Valley  Rd.; Austin TX 78702"
## [64] "8601 Tallwood Dr.; Austin TX 78759"
## [65] "1201 Webberville Rd.; Austin TX 78721"
## [66] "Veterans Drive; Austin TX NA"
## [67] "Veterans Drive to Shady Lane; Austin TX NA"
## [68] "South Bank at Barton Creek; Austin TX 78704"
## [69] "3810 Todd Lane; Austin TX NA"
## [70] "1600 Grove Blvd; Austin TX 78704"
## [71] "124 West Eighth St.; 3rd Floor; Austin TX 78701"
## [72] "1156 W. Cesar Chavez St.; Austin TX 78703"
## [73] "; Austin TX 78701"
## [74] "5400 Jimmy Clay Dr.; Austin TX 78744"
## [75] "1156 Hargrave St.; Austin TX 78702"
## [76] "11401 Escarpment Blvd.; Austin TX 78749"
## [77] "2201 Barton Springs Rd.; Austin TX 78704"
## [78] "4123 South First St.; Austin TX 78745"
## [79] "1200 Montopolis Drive; Austin TX NA"
## [80] "Roy Montelongo Scenic Overlook; Canterbury St and Pleasant Valley Rd Austin TX 78702"
```

Keys for cleaning addresses: all hail `sub()` and `gsub()`

If you just want to get rid of the first occurrence of something, use:

- Use `sub("What you don't want", "What you do want", "Where you want it")`

If you want to get rid of every occurrence of something, use:

- Use `gsub("What you don't want", "What you do want", "Where you want it")`

```
AustinPublicArt$art_full_address <- gsub(";", "", AustinPublicArt$art_full_address)  #get rid of specia
AustinPublicArt$art_full_address <- sub(".* - ", "", AustinPublicArt$art_full_address)  #get rid of 'Po
AustinPublicArt$art_full_address <- gsub("NA", "", AustinPublicArt$art_full_address)  #remove NA zipcod
```

Many geocoders exist today (both open-source and proprietary), but the basis is the same. The algorithm begins by taking a complete street address and breaking it down into its component parts. For example, if we wanted to geocode the Vanderbilt Biostatistics Department we would begin with

2525 West End Avenue, Nashville, TN 37203 → 2525 West End Avenue | Nashville | TN | 37203

The geocoder then works its way down the geographic hierarchy that you specify (depending on the scope of your data) to find increasingly more precise coordinates based on matching the address's:

1. State
2. City
3. Zip code
4. Street address

Fortunately, the R community has blessed us with yet another package: `ggmap`! `ggmap` contains a quick function called `geocode()` – tricky to remember, I know – that will take in a string address and output the latitude and longitude coordinates utilizing the Google Maps API (`source = "google"`) or the Data Science Toolkit online (`source = "dsk"`). Let's try this function out on the second artwork in the data set (the first cannot be geocoded).

```
# view second full address
AustinPublicArt$art_full_address[2]
```

```
## [1] "2800  Star of Texas Rd. Austin TX 78719"
```

```
# geocode second full address using Data Science Toolkit
geocode(AustinPublicArt$art_full_address[2], source = "dsk")
```

```
## Information from URL : http://www.datasciencetoolkit.org/maps/api/geocode/json?address=2800%20%20Sta
```

```
##         lon      lat
## 1 -97.81779 30.45109
```

```
# geocode second full address using Google Maps
geocode(AustinPublicArt$art_full_address[2], source = "google")
```

```
## Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=2800%20%20Star%20of%
```

```
##         lon      lat
## 1 -97.80663 30.26656
```

We can see that the geocoded addresses from each source are quite close. I have not (yet) found much information differentiating between the two, so for now I chose to proceed with Google Maps. It took around 3 minutes for the 168 addresses to geocode, so to save time an updated csv with the latitude and longitude can be found here.

```
# import completely geocoded AustinPublicArt dataset from the link above
AustinPublicArt.gc <- read.csv("https://raw.githubusercontent.com/sarahlotspeich/MappingInRTutorial/mas
    header = TRUE, stringsAsFactors = FALSE)
```

This is what is referred to as batch geocoding. Now, the unfortunate truth is that we will rarely be able to successfully geocode an entire set of addresses due to either unforgiveable type-os or missing data. In our case, we were unable to precisely pinpoint the location of 4% of the works of art. Excluding those that were unable to be geocoded leaves us with a subset of 161 works of art, and now that we have geocoded our addresses we are ready to build our first map!

It all starts with a shapefile

Now that we have data that we're interested in visualizing on a map, we need a blank canvas. One option for this blank canvas is called a "shapefile." This is the same file type used for Geographic Information Systems (GIS) software, so I prefer it because there is an abundance of data available in this format. The first place I look for maps of the United States is the U.S. Census Bureau.

Using the `readOGR()` function in the `rgdal` package, we can read in the shapefile we've downloaded as a usable spatial layer.

```
UrbanAreasUS.shp <- readOGR(dsn = path.expand("tl_2016_us_uac10/tl_2016_us_uac10.shp"),
    layer = "tl_2016_us_uac10")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "tl_2016_us_uac10/tl_2016_us_uac10.shp", layer: "tl_2016_us_uac10"
## with 3601 features
## It has 12 fields
## Integer64 fields read as strings:  ALAND10 AWATER10
```

```
class(UrbanAreasUS.shp)
```

```
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
```

What the heck is a `spatial polygons data frame`?

Classes of spatial data

1. `SpatialPoints`: simply describe locations (with no attributes)
2. `SpatialPointsDataFrame`: describes locations + attributes for them
3. `SpatialPolygonsDataFrame`: describes locations + shapes + attributes for them

We can access the familiar data frame component of the `UrbanAreas_US` spatial polygons data frame as follows:

```
#view the head of the UrbanAreas_US data frame
head(UrbanAreasUS.shp@data)
```

```
##   UACE10 GEOID10           NAME10                     NAMELSAD10 LSAD10
## 0  24310   24310        Dixon, IL        Dixon, IL Urban Cluster     76
## 1  27847   27847     Escanaba, MI     Escanaba, MI Urban Cluster     76
## 2  18100   18100 Clintonville, WI Clintonville, WI Urban Cluster     76
## 3  06166   06166      Bedford, IN      Bedford, IN Urban Cluster     76
## 4  75270   75270    Riverdale, CA    Riverdale, CA Urban Cluster     76
## 5  90946   90946      Visalia, CA     Visalia, CA Urbanized Area     75
##   MTFCC10 UATYP10 FUNCSTAT10   ALAND10 AWATER10   INTPTLAT10    INTPTLON10
## 0   G3500       C          S  25524689   938058 +41.8529507 -089.4817439
## 1   G3500       C          S  46488558   283456 +45.7274565 -087.0824457
## 2   G3500       C          S   5854721   502397 +44.6232203 -088.7611283
## 3   G3500       C          S  30403132     2314 +38.8566530 -086.5012383
## 4   G3500       C          S   2306823        0 +36.4310710 -119.8620544
## 5   G3500       U          S 164291020    39512 +36.2934877 -119.3006857
```

```r
head(data.frame(UrbanAreasUS.shp))
```

```
##   UACE10 GEOID10           NAME10                    NAMELSAD10 LSAD10
## 0  24310   24310        Dixon, IL        Dixon, IL Urban Cluster     76
## 1  27847   27847     Escanaba, MI     Escanaba, MI Urban Cluster     76
## 2  18100   18100 Clintonville, WI Clintonville, WI Urban Cluster     76
## 3  06166   06166      Bedford, IN      Bedford, IN Urban Cluster     76
## 4  75270   75270    Riverdale, CA    Riverdale, CA Urban Cluster     76
## 5  90946   90946      Visalia, CA     Visalia, CA Urbanized Area     75
##   MTFCC10 UATYP10 FUNCSTAT10   ALAND10 AWATER10   INTPTLAT10   INTPTLON10
## 0   G3500       C          S  25524689   938058 +41.8529507 -089.4817439
## 1   G3500       C          S  46488558   283456 +45.7274565 -087.0824457
## 2   G3500       C          S   5854721   502397 +44.6232203 -088.7611283
## 3   G3500       C          S  30403132     2314 +38.8566530 -086.5012383
## 4   G3500       C          S   2306823        0 +36.4310710 -119.8620544
## 5   G3500       U          S 164291020    39512 +36.2934877 -119.3006857
```

By taking advantage of this data frame, we can focus our map on the Austin, TX urban area using our usual data subsetting techniques. Be careful not to select a subset of only the data frame, because if you lose the spatial polygon part R will not plot your map properly (i.e. as a map).

```r
UrbanAreasUS.shp@data[which(UrbanAreasUS.shp@data$NAME10 == "Austin, TX"), ]
```

```
##      UACE10 GEOID10     NAME10                 NAMELSAD10 LSAD10 MTFCC10
## 3567  04384   04384 Austin, TX Austin, TX Urbanized Area     75   G3500
##      UATYP10 FUNCSTAT10    ALAND10 AWATER10   INTPTLAT10   INTPTLON10
## 3567       U          S 1356225919 11012063 +30.3585141 -097.7615330
```

```r
AustinTX.shp <- subset(UrbanAreasUS.shp, UrbanAreasUS.shp@data$NAME10 == "Austin, TX")
```

Spatial data in ggplot2

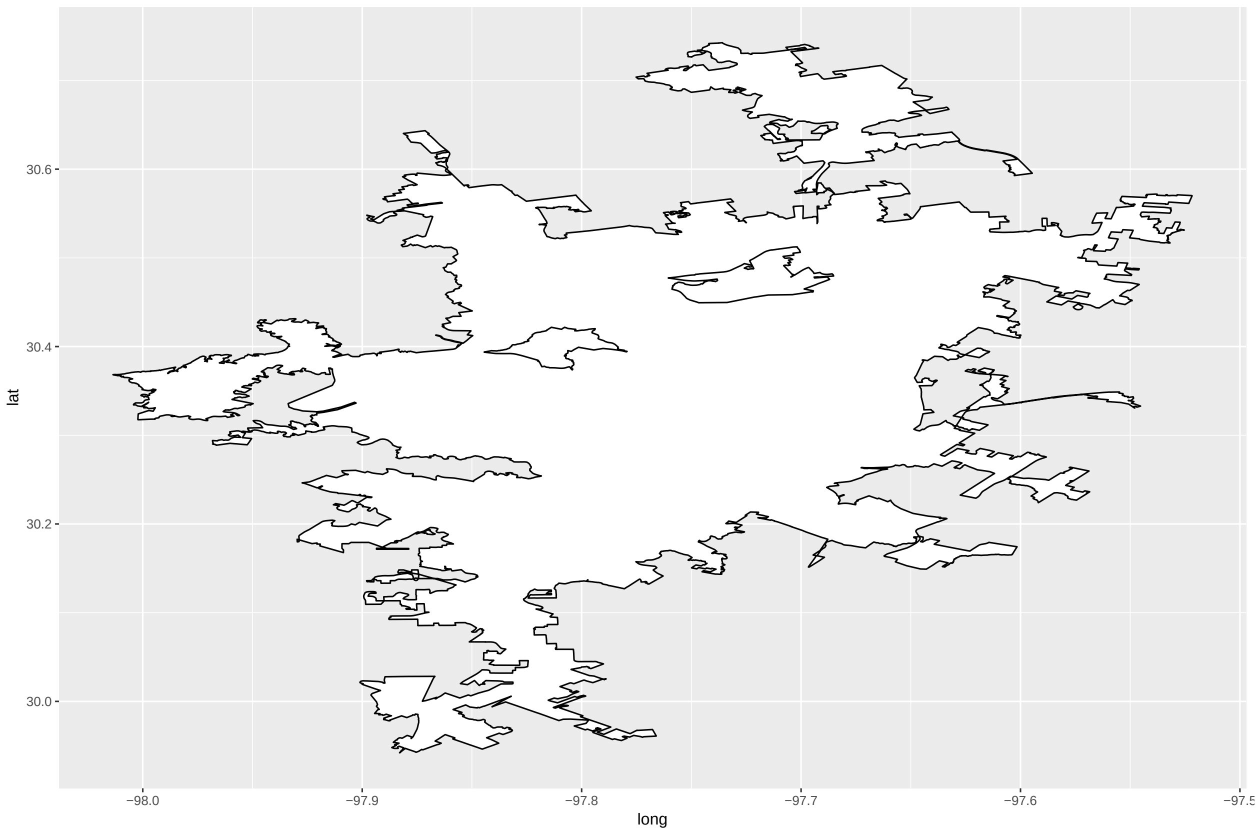
To map Austin in `ggplot2` we need to use the `tidy()` function in the `broom` package to convert the spatial polygon data frame to a traditional data frame. Previously, the `fortify()` function could be used for this, but this function is likely to become deprecated.

```r
AustinTX.df <- tidy(AustinTX.shp, region = "NAME10")
# or AustinTX.df <- fortify(AustinTX.shp, region='NAME10')
```

Creating Maps with ggplot2

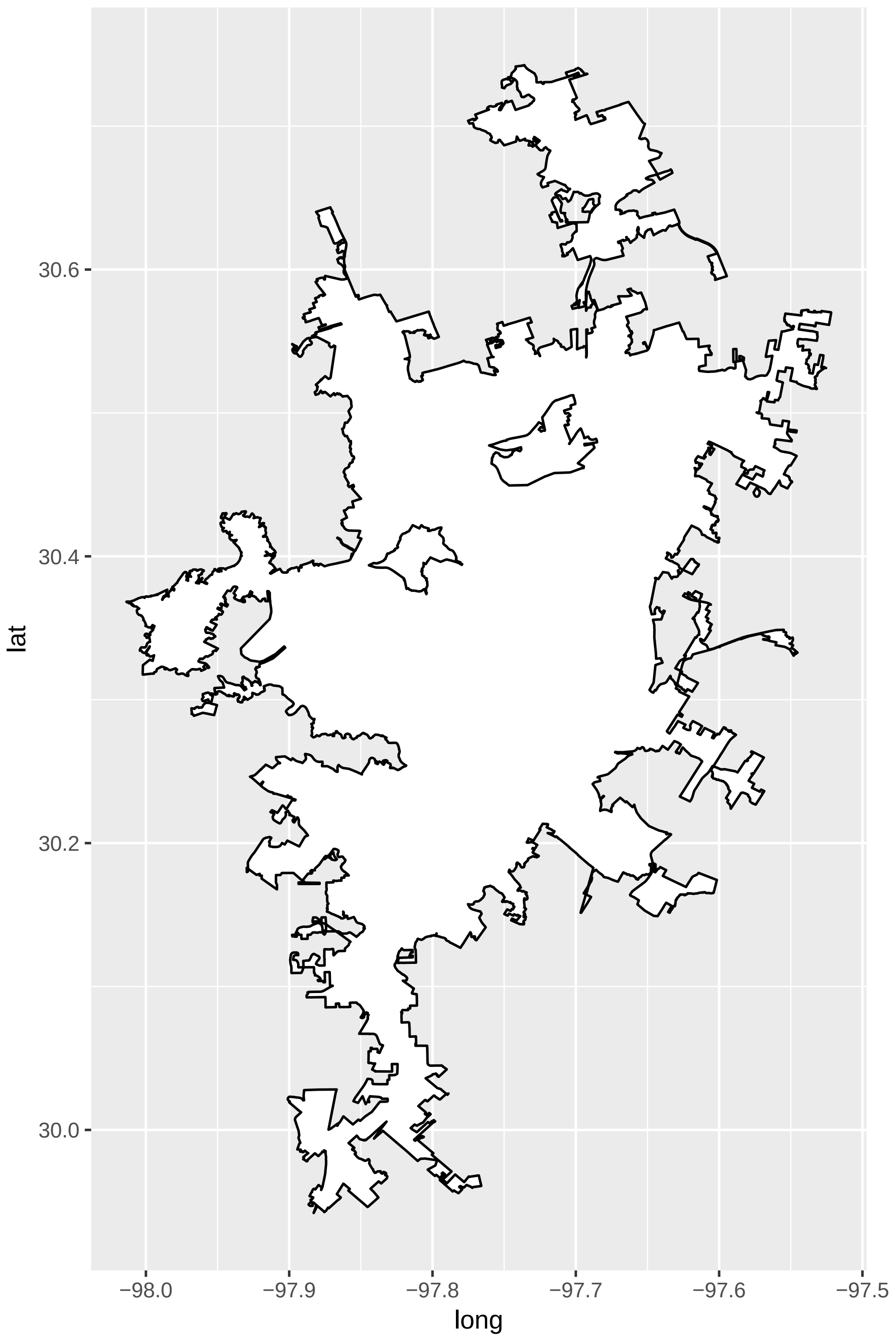
Once the shapefile has been formatted appropriately, we can utilize our knowledge of the "grammar of graphics" to map the urban area for Austin, TX. Begin with the `geom_polygon()` function to plot the latitude and longitude coordinates for the outline of the Austin, TX urban area (`aes(x = long, y = lat)`) and use `group = group` in your aesthetic to tell `ggplot2` to treat this area as one polygon. Feel free to select your own `fill` color, and outline color, `col`.

```r
(AustinTX.map <- ggplot() + geom_polygon(data = AustinTX.df, aes(x = long, y = lat,
    group = group), fill = "white", col = "black"))
```



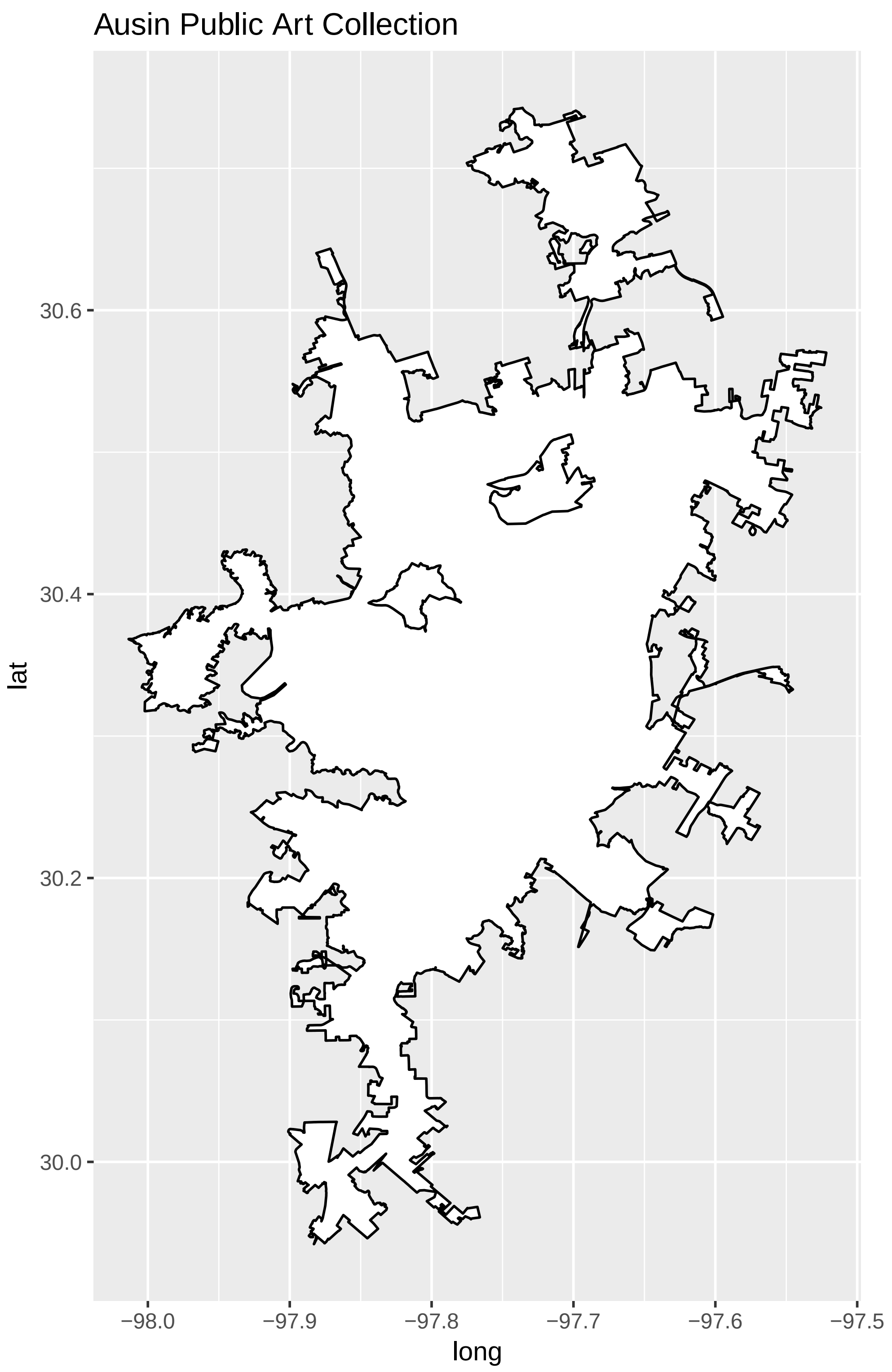
By adding `+ coord_equal(ratio = 1)` to the end, we can fix the skew of the plot.

```
(AustinTX.map <- AustinTX.map + coord_equal(ratio = 1))
```



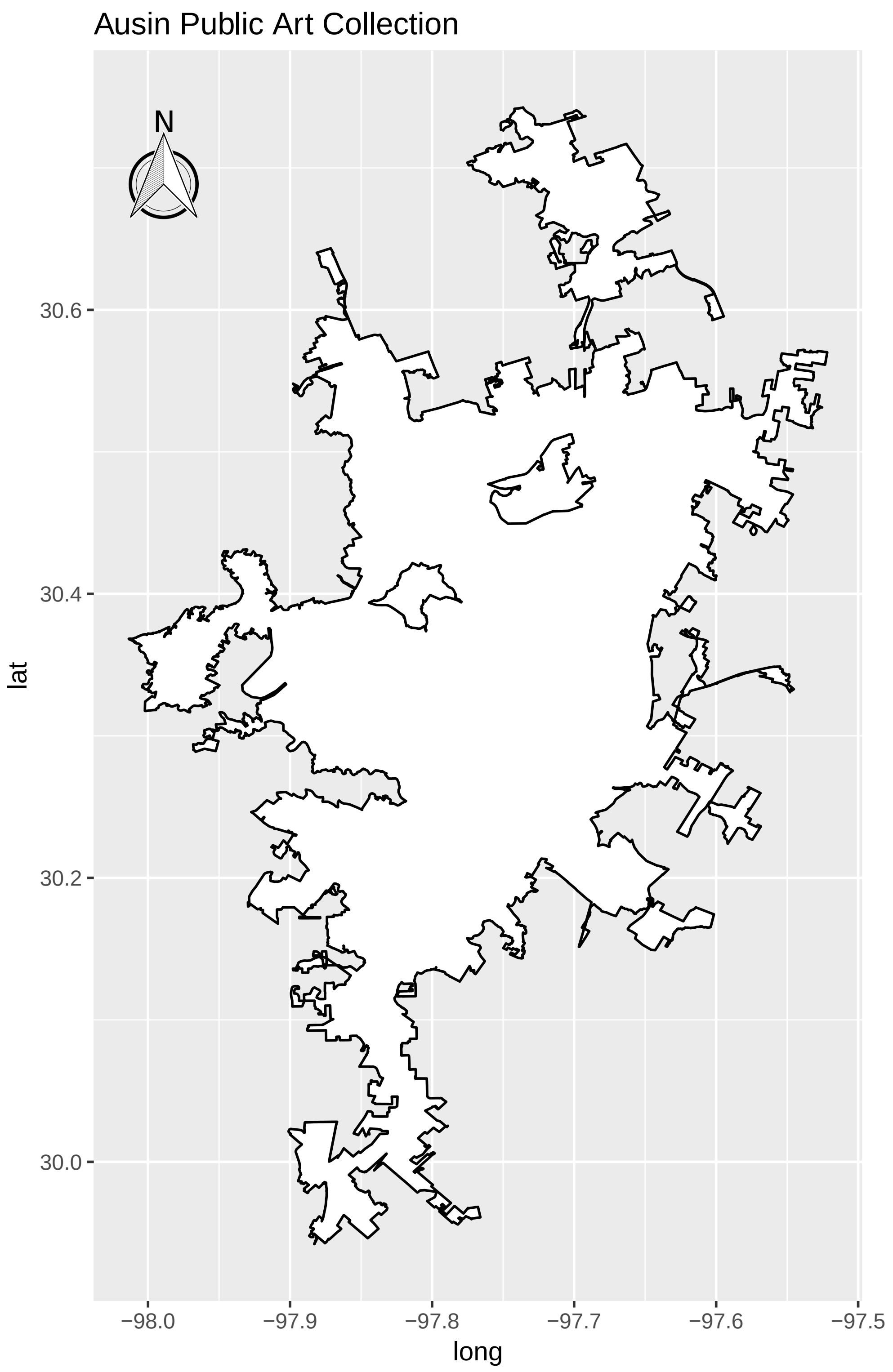
Use `ggtitle()` to add a main title describing your map.

```
(AustinTX.map <- AustinTX.map + ggtitle("Ausin Public Art Collection"))
```

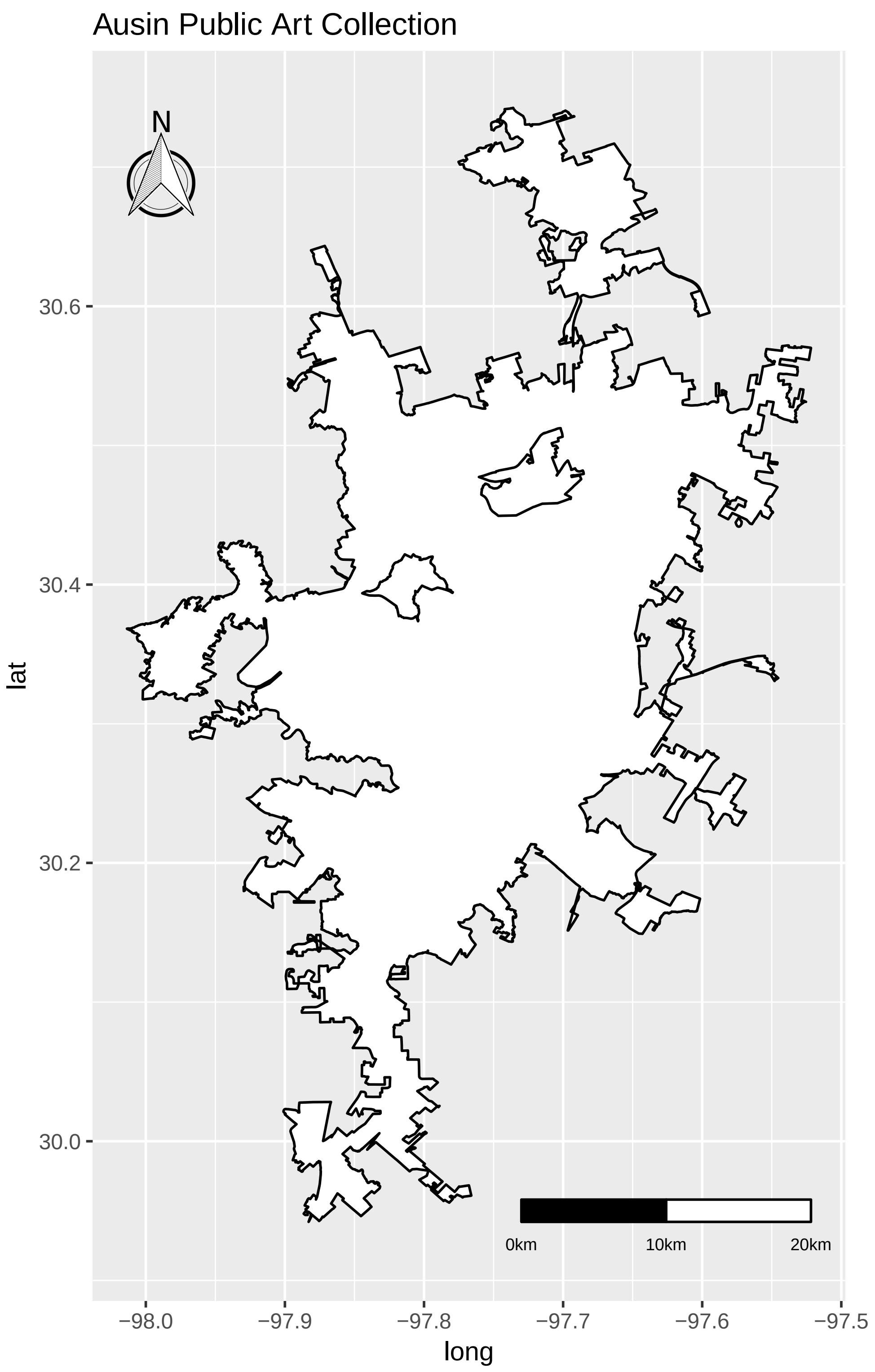


We will need the additional `ggsn` package to add north arrows and map scales for orientation. The `north()` function takes a `location` parameter for where to place it in the map. The `scalebar()` function also takes a `location` parameter, in addition to a `dist` input (distance in km per segment of the scale bar), `dd2km` (set to TRUE for a map in latitude and longitude, to FALSE for maps in meters), `model` (describing the projection used for the map, most commonly `= 'WGS84'`), and `st.size` which sets the text size for the map scale.

```r
(AustinTX.map <- AustinTX.map + north(AustinTX.df, location = "topleft"))
```



```r
(AustinTX.map <- AustinTX.map + scalebar(AustinTX.df, location = "bottomright",
    dist = 10, dd2km = TRUE, model = "WGS84", st.size = 2.5))
```

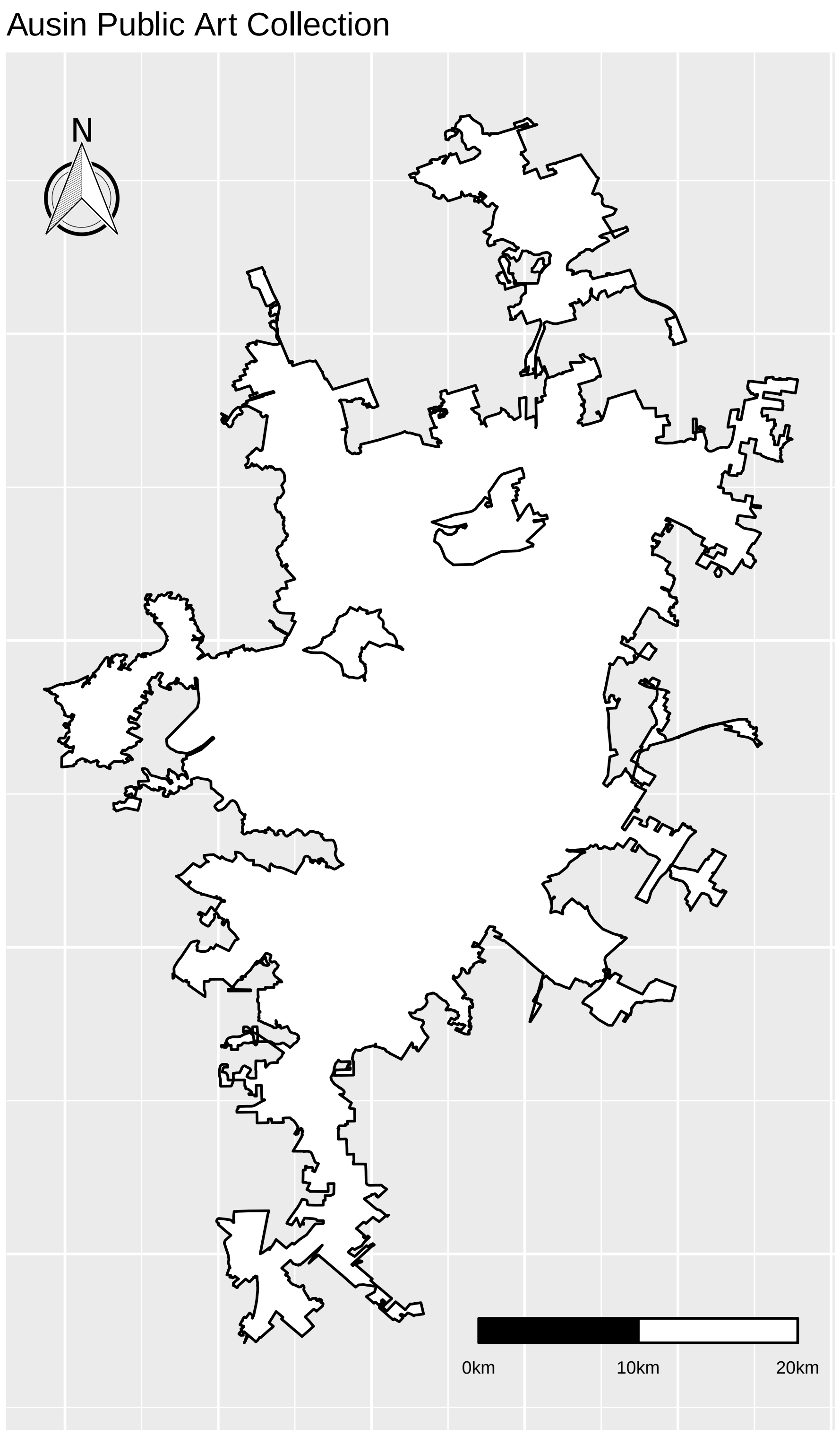


To make this look less like a plot and more like a traditional map, we can add the following chunk of to the end of our plot:

```
+ theme(axis.text.x=element_blank(), axis.text.y=element_blank(), axis.ticks=element_blank(),axis.title
```

to remove the x and y axis titles, tick lines, and values.

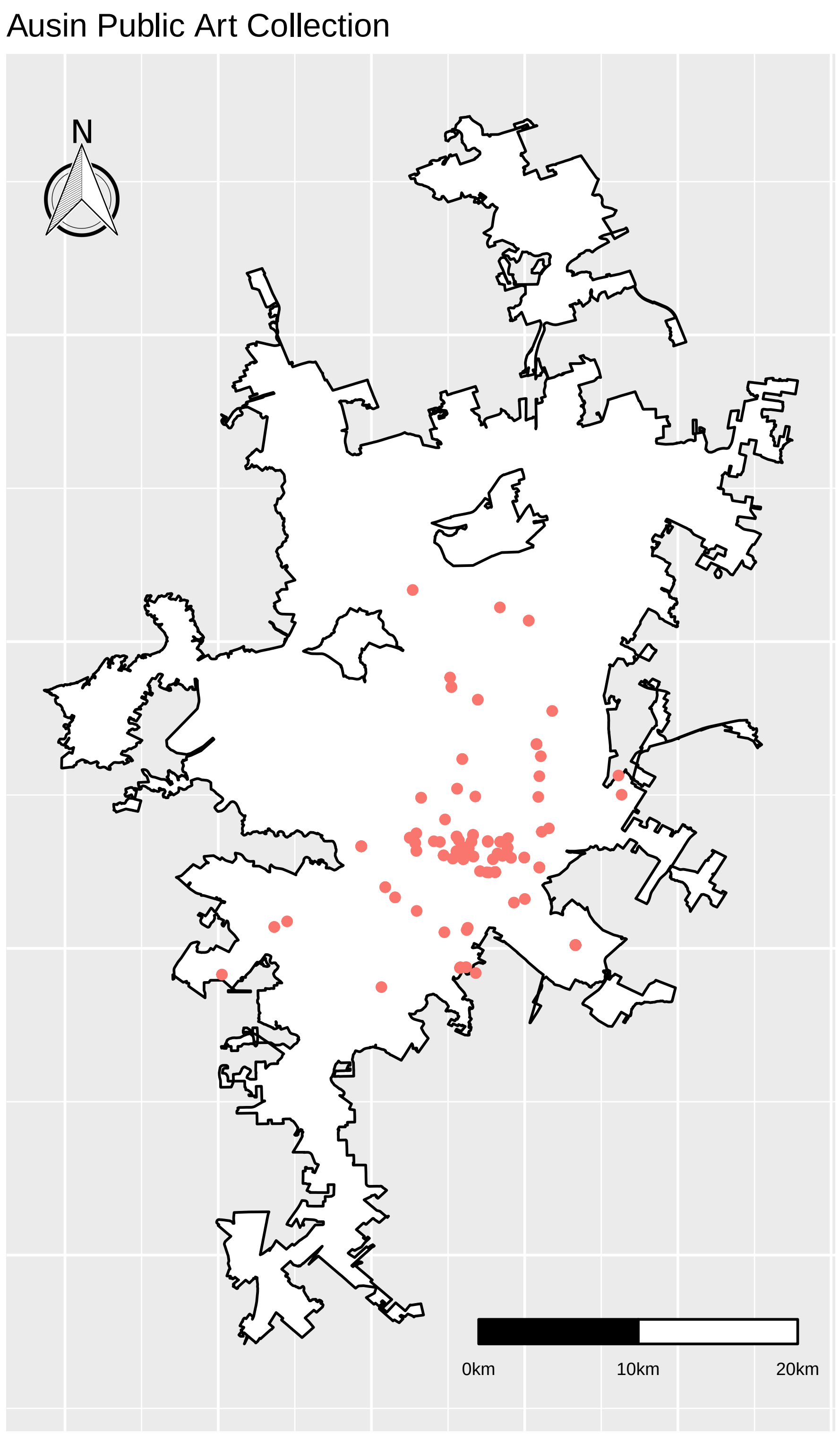
```
(AustinTX.map <- AustinTX.map + theme(axis.text.x = element_blank(), axis.text.y = element_blank(),
    axis.ticks = element_blank(), axis.title.x = element_blank(), axis.title.y = element_blank()))
```



Map of the City of Austin's Public Art Collection

Now we can begin with our base map from above, and overlay it with the locations of works of street art! We can use the `geom_point()` function with the latitude and longitude coordinates and the original `AustinPublicArt` data frame.

```
(AustinArtCollection.map <- AustinTX.map + geom_point(data = AustinPublicArt.gc,
    aes(x = art_location_long, y = art_location_lat, color = "orange")) + theme(legend.position = "none
```



From the map above, we should be able to make some preliminary observations about the spatial distribution of public artwork across the urban area for the city of Austin, Texas. Perhaps policy makers requested this analysis so that they could assess how the City of Austin Public Arts Collection is bringing enrichment to the people of Austin and if the collection is accessible to everyone. How would we answer their question?

How about estimating the average distance between works of art as a gauge of availability/ accessibility?

Calculating distances between places on a map

When we think about distances traveled, how do we measure them?

1. Distance (shortest route possible between two points)
   a. Haversine Formula: calculates the "great-circle" distance between two points (i.e. "as the crow flies"). Performs well even at short distances.

$$d(i,j) = 2r\text{arcsin}(\sin^2(\frac{\text{lat}_j - \text{lat}_i}{2}) + \cos(\text{lat}_i)\cos(\text{lat}_j)\sin^2(\frac{\text{long}_j - \text{long}_i}{2}))$$

   b. Spherical Law of Cosines: gives reliable estimates down to a few meters. Formula is simpler than Haversine Formula.

$$d(i,j) = r \cdot \text{arccos}(\sin(\text{lat}_i)\sin(\text{lat}_j) + \cos(\text{lat}_i)\cos(\text{lat}_j)\cos(\text{long}_j - \text{long}_j))$$

   c. Where $r = 3959$, the radius of the Earth in miles. If we were interested in calculating distances in kilometers or some other unit, we would replace r with the radius of the Earth in the desired output units.

2. Distance (incorporating transportation networks to measure true route between two points)
3. Travel time (incorporating transportation networks to measure true route between two points)

Which of these measures you choose might depend on the nature or scope of your project, but since our data set is reasonably manageable I thought we could experiment with all three.

Using the `geosphere` package for Haversine Formula and Spherical Law of Cosines

The `geosphere` package contains functions `disthaversine()` and `distcosine()` to calculate the distance between either two vectors of latitude/ longitude coordinates or matrices with two columns for latitude/ longitude, using the Haversine Formula and Spherical Law of Cosines, respectively.

```
# Recall how to access coordinates from a spatialpointsdataframe
head(AustinPublicArt.gc[, 9:10])
```

```
##   art_location_long art_location_lat
## 1         -97.74410         30.30413
## 2         -97.80663         30.26656
## 3         -97.73056         30.36215
## 4         -97.71594         30.26950
## 5         -97.74004         30.25816
## 6         -97.72410         30.27000
```

```
# Input: lat/long coordinates of the 1st and 2nd works of art Output:
# distance between 1st and 2nd works of art (in miles)
distHaversine(AustinPublicArt.gc[1, 9:10], AustinPublicArt.gc[2, 9:10], r = 3959)
```

```
## [1] 4.545254
```

```
# Input: lat/long coordinates of the 1st vs. all other works of art Output:
# distances between the 1st and all other works of art (in miles)
distHaversine(AustinPublicArt.gc[1, 9:10], AustinPublicArt.gc[-1, 9:10], r = 3959)
```

```
##   [1]  4.5452544  4.0900639  2.9236202  3.1855238  2.6424459  8.3366458
##   [7]  2.8108806  4.0900639  4.7795870  4.0382469  4.7795870  4.7795870
##  [13]  3.1199988  4.5452544  8.0510653  8.4214053  4.7795870  3.1855238
##  [19]  4.5452544  3.8249276  3.3059115  3.5019058  2.6897200  2.7250455
##  [25]  4.7795870  2.8108806  2.4492683  2.8108806  3.5709676  8.4214053
##  [31]  2.9376154  2.6718305  4.0382469  9.4629286  1.4592202  3.9405359
##  [37]  3.3173770  5.4605885  3.3059115  4.5452544  8.4214053  2.4904131
##  [43]  4.0382469  2.8156268  1.4626388  1.4592202  4.5452544  9.4127058
##  [49]  5.4605885  3.2511489  4.5452544  9.1190891  8.4214053  4.0382469
##  [55]  2.8533988  2.9236202  8.9203382  3.0497473  1.3516573  2.1713607
##  [61]  3.6857085  3.0497473  2.8108806  2.1713607  3.8202697  3.1546364
##  [67]  6.4127176  8.4214053  3.1855238  8.0518401  4.7795870  4.5452544
##  [73]  1.3516573  8.4214053  3.1199988  6.3099260  6.3784428  8.4214053
##  [79]  4.5452544  3.5181250  4.5452544  2.8533988  5.0963286  2.1562040
##  [85]  2.1562040  4.5853419  3.2166336  2.9376154  8.0711070  2.6718305
##  [91]  8.4214053  4.0382469  2.8769341  3.9706805  3.1830626  4.0382469
##  [97]  6.4864831  4.0382469  3.6857085  8.4214053  9.4629286  4.5853419
## [103]  8.0711070  2.8769341  3.4706668  4.0382469  0.7924867  4.0382469
## [109]  4.0581563  2.1562040  8.0767196  2.1713607  2.6897200  8.0518401
## [115]  4.0382469  3.1199988  3.3059115  2.6675452  3.7570021  5.0183079
## [121]  3.9959134  2.5622273  3.6857085  4.5452544  5.2459933  4.0581563
## [127]  2.8108806  6.2837867  5.5883827  2.3233835  8.0510653  4.0382469
## [133]  2.5364039  6.2837867  4.5452544  4.0581563  0.0000000  2.8533988
## [139]  2.8533988  3.6857085  5.2459933  2.1562040  4.0382469  4.7795870
## [145]  4.7795870  2.9236202  8.4214053  8.3382959  2.9802586  8.4214053
## [151] 12.4117367  3.5019058  2.8769341  8.0510653  3.2166336  5.7274785
## [157]  2.1562040  2.4492683  5.6249175  4.0382469
```

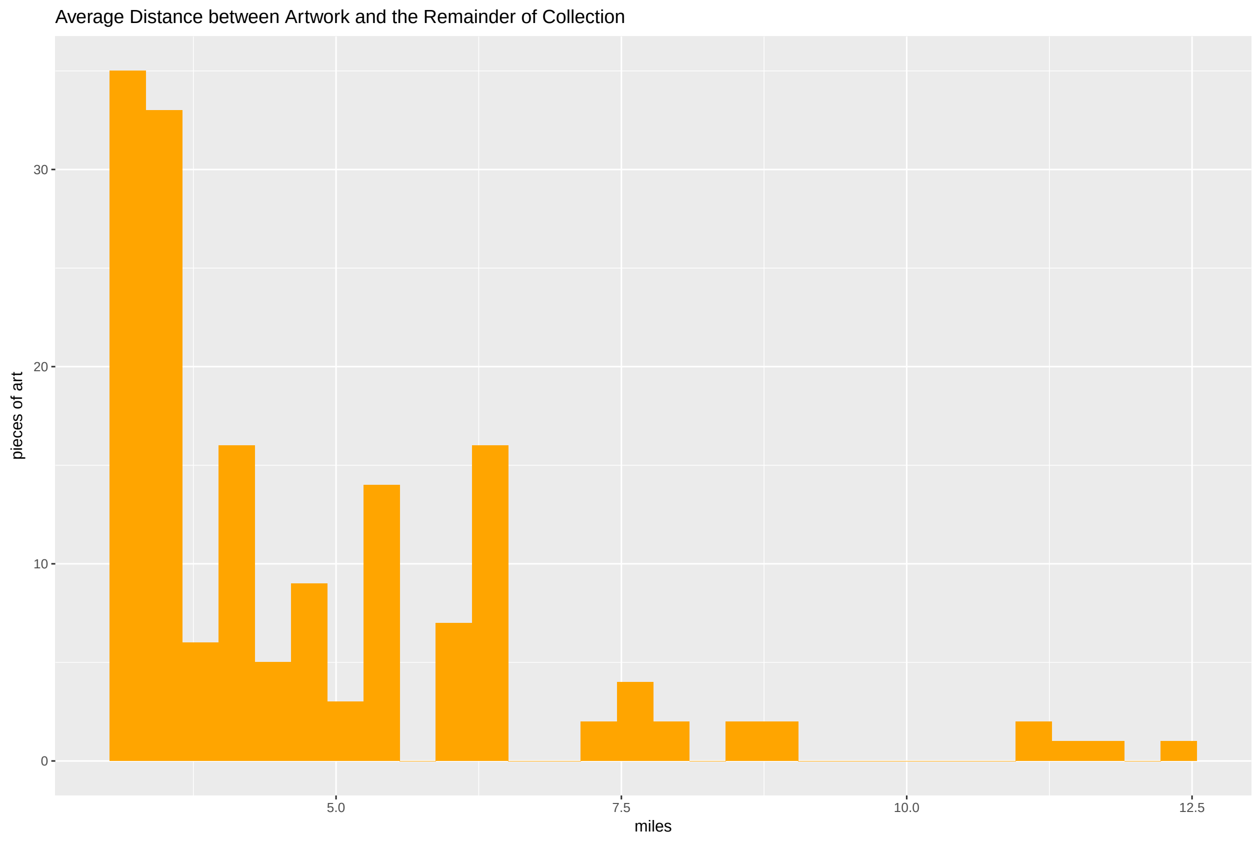
We can take the mean of this last command's output to calculate the average distance to other works of art for the first piece in the dataset (our proposed gauge of accessability). To find the average distance from each of the 168 works of art, we can construct a distance matrix using the `distm()` function in `geosphere()` and then take the `colmeans()` of that matrix.

```
DistanceMatrix.Haversine <- distm(AustinPublicArt.gc[, 9:10], AustinPublicArt.gc[,
    9:10], distHaversine) * 0.000621371  #convert km --> mi by multipying * 0.000621371
DistanceMatrix.Haversine[1, ]  #Look at the first row of distance matrix output
```

```
##   [1]  0.0000000  4.5500642  4.0943920  2.9267139  3.1888947  2.6452421
##   [7]  8.3454675  2.8138550  4.0943920  4.7846447  4.0425201  4.7846447
##  [13]  4.7846447  3.1233004  4.5500642  8.0595849  8.4303167  4.7846447
##  [19]  3.1888947  4.5500642  3.8289751  3.3094098  3.5056115  2.6925663
##  [25]  2.7279291  4.7846447  2.8138550  2.4518601  2.8138550  3.5747464
##  [31]  8.4303167  2.9407239  2.6746578  4.0425201  9.4729422  1.4607643
##  [37]  3.9447057  3.3208874  5.4663668  3.3094098  4.5500642  8.4303167
##  [43]  2.4930484  4.0425201  2.8186063  1.4641865  1.4607643  4.5500642
##  [49]  9.4226662  5.4663668  3.2545892  4.5500642  9.1287388  8.4303167
##  [55]  4.0425201  2.8564182  2.9267139  8.9297776  3.0529745  1.3530876
##  [61]  2.1736585  3.6896087  3.0529745  2.8138550  2.1736585  3.8243123
##  [67]  3.1579746  6.4195035  8.4303167  3.1888947  8.0603605  4.7846447
##  [73]  4.5500642  1.3530876  8.4303167  3.1233004  6.3166031  6.3851924
##  [79]  8.4303167  4.5500642  3.5218479  4.5500642  2.8564182  5.1017215
##  [85]  2.1584856  2.1584856  4.5901941  3.2200374  2.9407239  8.0796478
##  [91]  2.6746578  8.4303167  4.0425201  2.8799785  3.9748822  3.1864308
##  [97]  4.0425201  6.4933470  4.0425201  3.6896087  8.4303167  9.4729422
## [103]  4.5901941  8.0796478  2.8799785  3.4743394  4.0425201  0.7933253
## [109]  4.0425201  4.0624506  2.1584856  8.0852663  2.1736585  2.6925663
## [115]  8.0603605  4.0425201  3.1233004  3.3094098  2.6703680  3.7609777
## [121]  5.0236182  4.0001418  2.5649386  3.6896087  4.5500642  5.2515445
## [127]  4.0624506  2.8138550  6.2904361  5.5942962  2.3258421  8.0595849
## [133]  4.0425201  2.5390878  6.2904361  4.5500642  4.0624506  0.0000000
## [139]  2.8564182  2.8564182  3.6896087  5.2515445  2.1584856  4.0425201
## [145]  4.7846447  4.7846447  2.9267139  8.4303167  8.3471194  2.9834122
## [151]  8.4303167 12.4248707  3.5056115  2.8799785  8.0595849  3.2200374
## [157]  5.7335392  2.1584856  2.4518601  5.6308698  4.0425201
```

```
diag(DistanceMatrix.Haversine) <- NA  #remove 0 values for distance b/t a work of art and itself
AustinPublicArt.gc$avg_distance_haversine <- colMeans(DistanceMatrix.Haversine,
    na.rm = TRUE)
```

Just for fun, let's look at the histogram of average distance from each piece of art to the rest of the collection.



The inputs for the `distCosine()` function are the same, and the calculations will be similar for most reasonable distances. We could repeat all of the previous steps with the Spherical Law of Cosines simply by substituting `distCosine()` in for `distHaversine()`.

```
# Input: lat/long coordinates of the 1st and 2nd works of art Output:
# distance between 1st and 2nd works of art (in miles)
distCosine(AustinPublicArt.gc[1, 9:10], AustinPublicArt.gc[2, 9:10], r = 3959)
```

```
## [1] 4.545254
```

Using the `gmapsdistance` package for travel times and distances based on road networks

Depending on the focus of the spatial analysis, it may be more prudent to incorporate networks of roads and sidewalks to calculate distances or travel times directly rather than "as the crow flies" (the shortest possible distance between two points). With the gmapsdistance package this can be accomplished using the Google

Maps API directly in R to utilize Google Maps data to calculate travel distances (output in meters) and times (output in seconds) by a variety of modes (walking, biking, driving, or taking public transportation), at different times of day, and under different traffic conditions ("pessimistic", "optimistic", "best guess").

The inputs for the `gmapsdistance()` function are a little different. With this command, the origin and destination can either be named places or pairs of latitude and longitude coordinates of the forms `CITY+STATE` or `LATITUDE+LONGITUDE`, respectively. To simplify the code, I elected to use the `paste()` function to input coordinates in this format without altering the original spatial polygons data frame.

```
AustinPublicArt.gc[1, 9:10]  #access one pair of lat/long coordinates from the spatial polygons data fr
```

```
##   art_location_long art_location_lat
## 1          -97.7441         30.30413
```

```
paste(AustinPublicArt.gc[1, 10], AustinPublicArt.gc[1, 9], sep = "+")  #reformat this pair to be of the
```

```
## [1] "30.3041251+-97.744097"
```

Given what we've observed from the preliminary map of the dispersal of public art in Austin, it could be insightful to incorporate sidewalk and road networks to calculate distances for pedestrians rather than cars (as street art could be more easily enjoyed on foot). Let's begin by calculating the walking distance between the first and second works of art in the dataset.

```
gmapsdistance(origin = paste(AustinPublicArt.gc[1, 10], AustinPublicArt.gc[1,
    9], sep = "+"), destination = paste(AustinPublicArt.gc[2, 10], AustinPublicArt.gc[2,
    9], sep = "+"), mode = "walking")
```

```
## $Time
## [1] 7315
##
## $Distance
## [1] 9373
##
## $Status
## [1] "OK"
```

From this output, we see that there are two accesible objects: the `Time` and `Distance` between the `origin` and `destination`. Recall that `Time` is output in seconds and `Distance` in meters, so you will likely need to convert these. The following are additional function inputs that could be used to get more precise estimates of these: - `avoid`: when the `mode = "driving"`, we can specify routes avoiding `"tolls"`, `"highways"`, `"ferries"`, and `"indoor"` segments. - `traffic model`: when the `mode = "driving"`, we can specify the anticipated level of traffic using `"optimistic"`, `"pessimistic"`, or `"best_guess"`.

Example 2: Places of Worship

Background: The dataset contains locations and attributes of Places of Worship, created as part of the DC Geographic Information System (DC GIS) for the D.C. Office of the Chief Technology Officer (OCTO) and participating D.C. government agencies. Information provided by various sources identified Places of Worship such as churches and faith based organizations. DC GIS staff geo-processed this data from a variety of sources.

```
PlacesOfWorship.df <- read.csv("https://query.data.world/s/enqqu329pdhaq78usawjway3a",
    header = TRUE, stringsAsFactors = FALSE)
head(PlacesOfWorship.df)
```

```
##   objectid  gis_id                             name  religion
## 1        1   pow_1   Sixth Church of Christ, Scientist CHRISTIAN
## 2        2   pow_2 Seventh Church of Christ, Scientist CHRISTIAN
## 3       11  pow_12         Christian Tabernacle Church CHRISTIAN
## 4      101 pow_152  Ohev Sholom The National Synagogue    JEWISH
## 5      102 pow_153          Palisades Community Church CHRISTIAN
## 6      103 pow_154                   Paramount Baptist CHRISTIAN
##                        address latitude longitude
## 1 4601 MASSACHUSETTS AVENUE NW 38.94202 -77.09097
## 2           1204 47TH PLACE NE 38.90644 -76.93496
## 3          2033 11TH STREET NW 38.91792 -77.02679
## 4       1600 JONQUIL STREET NW 38.98411 -77.03693
## 5     5200 CATHEDRAL AVENUE NW 38.92996 -77.10583
## 6           3924 4TH STREET SE 38.83220 -77.00032
```

We begin by repeating the subset process to get our base layer shape for the state of Washington, D.C.. We can find this outline by downloading the States (and equivalent) shapefile from the United States Census Bureau. Helpful tidbit: the FIPS code for the District of Columbia is 11. By using the FIPS code instead of trying to use a string "District of Columbia" we can be careful about capitalizations or abbreviations.

```r
USStates.shp <- readOGR(path.expand("tl_2016_us_state/tl_2016_us_state.shp"),
    "tl_2016_us_state")
```

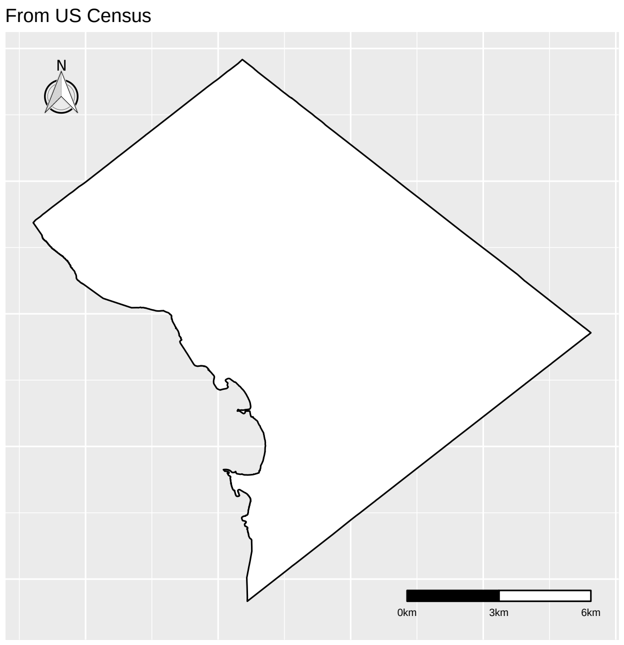
```
## OGR data source with driver: ESRI Shapefile
## Source: "tl_2016_us_state/tl_2016_us_state.shp", layer: "tl_2016_us_state"
## with 56 features
## It has 14 fields
## Integer64 fields read as strings:  ALAND AWATER
```

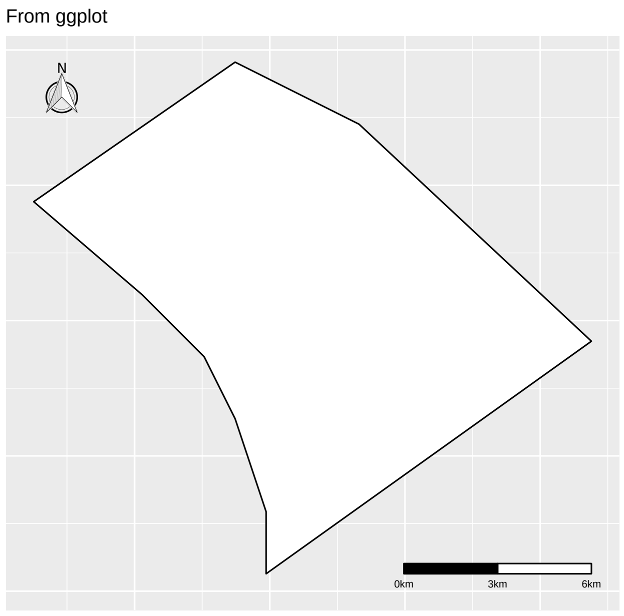
```r
WashingtonDC.shp <- subset(USStates.shp, USStates.shp@data$STATEFP == 11)
```

In `ggplot2`, we can plot it using either of the following commands. The first repeats the steps from the Austin, Texas example to `fortify()` the shapefile and then plot the latitude and longitude coordinates as a polygon using `geom_polygon()`. The second is new! Within `ggplot2`, there is a command called `map_data()` that allows you to directly read in common map shapes. We couldn't use this before because we were focusing on a small area, but now would be a fun time to explore it to plot Washington D.C.!

```r
WashingtonDC.df <- tidy(WashingtonDC.shp, region = "NAME")
WashingtonDC.gg <- subset(map_data("state"), map_data("state")$region == "district of columbia")
DCShapefile <- ggplot() + geom_polygon(data = WashingtonDC.df, aes(x = long,
    y = lat, group = group), fill = "white", col = "black") + coord_equal(ratio = 1) +
    theme(axis.text.x = element_blank(), axis.text.y = element_blank(), axis.ticks = element_blank(),
        axis.title.x = element_blank(), axis.title.y = element_blank()) + north(WashingtonDC.df,
    location = "topleft") + scalebar(WashingtonDC.df, location = "bottomright",
    dist = 3, dd2km = TRUE, model = "WGS84", st.size = 2.5)

DCMapData <- ggplot() + geom_polygon(data = WashingtonDC.gg, aes(x = long, y = lat,
    group = region), fill = "white", col = "black") + coord_equal(ratio = 1) +
    theme(axis.text.x = element_blank(), axis.text.y = element_blank(), axis.ticks = element_blank(),
        axis.title.x = element_blank(), axis.title.y = element_blank()) + north(WashingtonDC.gg,
    location = "topleft") + scalebar(WashingtonDC.gg, location = "bottomright",
    dist = 3, dd2km = TRUE, model = "WGS84", st.size = 2.5)
```



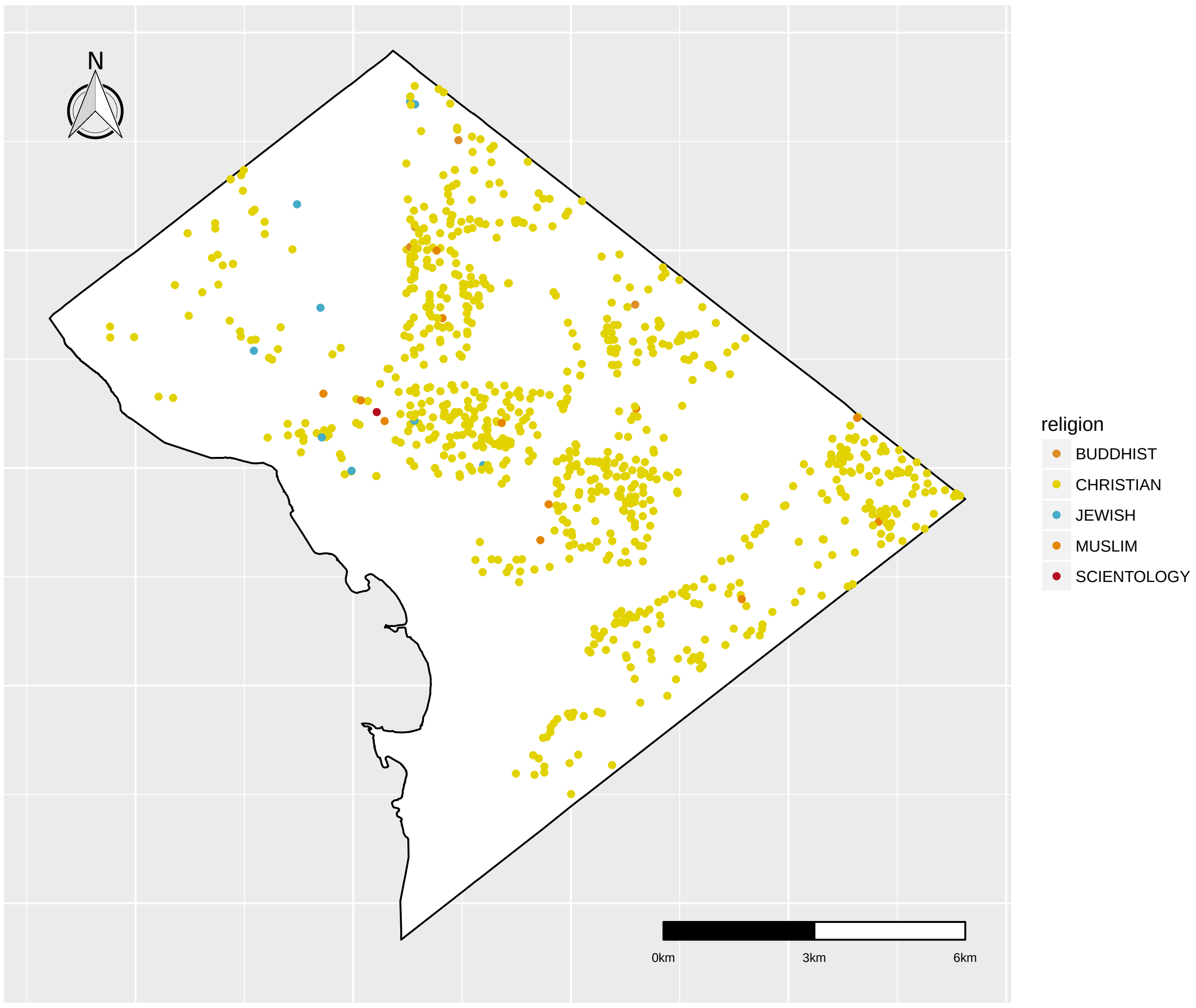


Personally, I prefer the precision of the shapefile to that of the `map_data()` plot. But for maps of larger areas (i.e. all of the states) this is definitely a great option!

Mapping location and attributes together

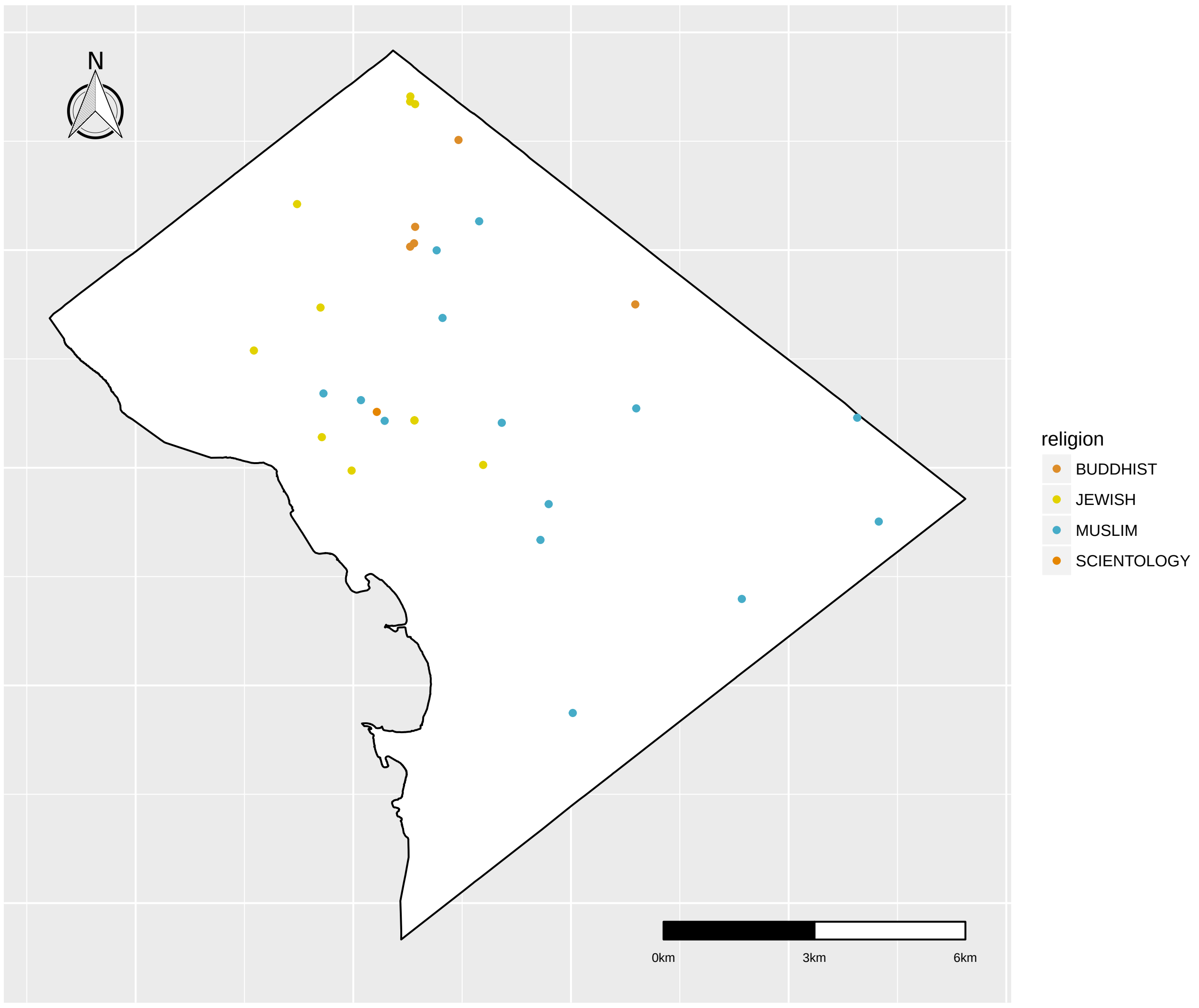
To illustrate incorporating attributes with spatial data, we will be creating a map of the places of worship in Washington, D.C. where the religion of each location can be identified from the unique color. To do so, instead of setting the `col` for the `geom_point()` statement we set `col = religion`. I was not a huge fan of the automatic color scheme, so I used `scale_color_manual()` with `values` from the `wesanderson` package color scheme `"Fantastic Fox"`.

```
(PlacesOfWorship.map <- DCShapefile + geom_point(data = PlacesOfWorship.df,
    aes(x = longitude, y = latitude, col = religion)) + scale_color_manual(values = wes_palette(n = 5,
    name = "FantasticFox")))
```



With a majority Christian locations, I elected to create a map with only non-Christian locations to provide more insight.

```
(PlacesOfWorship.map <- DCShapefile + geom_point(data = PlacesOfWorship.df[which(PlacesOfWorship.df$reli
    "CHRISTIAN"), ], aes(x = longitude, y = latitude, col = religion)) + scale_color_manual(values = wes
    name = "FantasticFox")))
```



Example 3: Claim-Based Medicare Spending by State

Claims-based Medicare spending: Price, age, sex and race-adjusted (20% sample for 2003-09: 100% sample for 2010-13). Dartmouth Atlas data have been used for many years to compare utilization and expenditures across hospital referral regions (HRRs).

```
Medicare <- read.csv("https://query.data.world/s/l6qwxpDqQw5RX8Urdo8G_hKq6P6i5a",
    header = TRUE, stringsAsFactors = FALSE)
head(Medicare)
```

```
##   StateNum StateName MedicareEnrollees
## 1        1   Alabama            494966
## 2        2    Alaska             56634
## 3       11   Georgia            720882
## 4       12    Hawaii             85577
## 5       13     Idaho            135364
## 6       14  Illinois           1188641
##   TotalMedicareReimbursementsPerEnrollee
## 1                                8680.93
## 2                                  8702.62
## 3                                  8760.11
## 4                                  7463.99
## 5                                  7946.36
## 6                                  9793.16
```

Getting colorful with choropleth maps

A choropleth is a map that shades geographic areas (counties, census blocks, urban areas, etc.) by their relative levels of a given attribute. Across the United States, we can use a choropleth to examine the relative per capita Medicare reimbursements by state. To do so, we will be go back to our US States shapefile and use the `tidy()` function to transform it to a `ggplot`-friendly data frame.

```
USStates.df <- tidy(USStates.shp, region = "NAME")
ContUSStates.df <- USStates.df[which(USStates.df$id != "Alaska"), ]
ContUSStates.df <- ContUSStates.df[which(ContUSStates.df$id != "Hawaii"), ]
ContUSStates.df <- ContUSStates.df[which(ContUSStates.df$id != "United States Virgin Islands"),
    ]
ContUSStates.df <- ContUSStates.df[which(ContUSStates.df$id != "Puerto Rico"),
    ]
ContUSStates.df <- ContUSStates.df[which(ContUSStates.df$id != "Guam"), ]
ContUSStates.df <- ContUSStates.df[which(ContUSStates.df$id != "Commonwealth of the Northern Mariana Isl
    ]
ContUSStates.df <- ContUSStates.df[which(ContUSStates.df$id != "American Samoa"),
    ]
ContUSStates.df <- ContUSStates.df[is.na(ContUSStates.df$long) == FALSE, ]
```

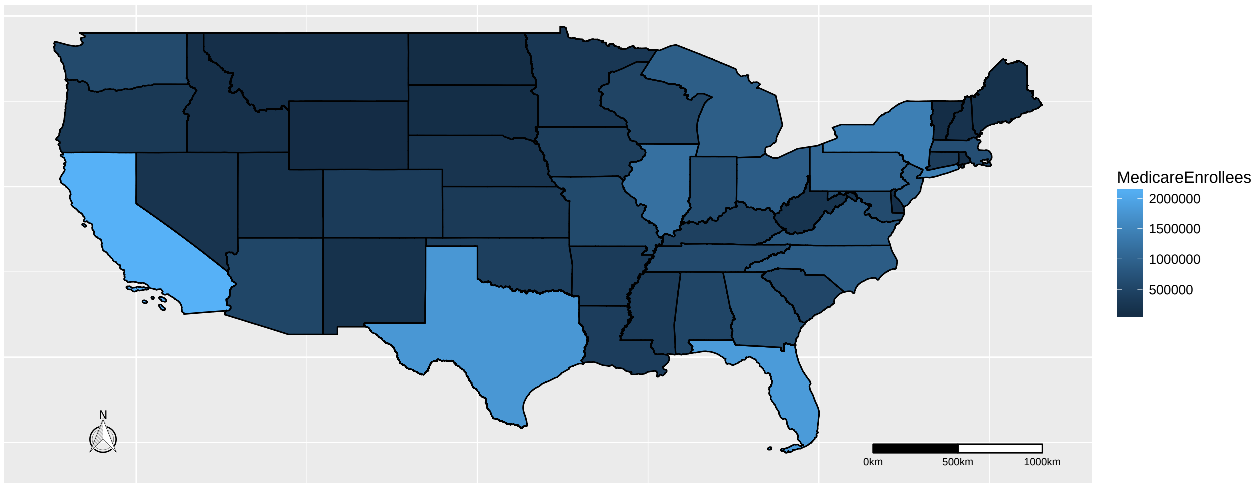
Now that we have state-level attributes (number of Medicare enrollees and total Medicare reimbursements per enrollee) and state-level polygons we need to `merge()` the two data frames.

```
MedicareStates <- merge(ContUSStates.df, Medicare, by.x = "id", by.y = "StateName",
    all.x = TRUE)
```

Now we can create a choropleth of the merged data frame `MedicareStates`.

```
(MedicareEnrollees.map <- ggplot() + geom_polygon(data = MedicareStates, aes(x = long,
    y = lat, group = group, fill = MedicareEnrollees), color = "black") + ggtitle("Number of Medicare E
    coord_equal(ratio = 1) + theme(axis.text.x = element_blank(), axis.text.y = element_blank(),
    axis.ticks = element_blank(), axis.title.x = element_blank(), axis.title.y = element_blank()) +
    north(MedicareStates, location = "bottomleft") + scalebar(MedicareStates,
    location = "bottomright", dist = 500, dd2km = TRUE, model = "WGS84", st.size = 2.5))
```

Number of Medicare Enrollees by U.S. State



Now go forth and map!