Introduction

In this session we are going to unleash the true power of the simple features. You will see how fast `sf` really is and learn about

- making points, lines and polygons,
- type transformations,
- topology,
- topological relations,
- geometric operations and buffers,
- spatial joins and aggregation, and
- solve some cool examples.

At the end you will be able to do 90% of the vector operations you will need in your research and have the tools to solve the remaining 10% on your own.

Revisting simple feature classes

An `sfg` class object is a *geometry*

- geometry of a single feature, seven geometry types
- vector, matrix, or list of matrices of coordinates with defined dimension and type of geometry

An `sfc` class object is a *geometry collection*

- list of `sfg` objects, seven sub-classes
- coordinate reference system through `crs` attribute

An `sf` class object is a geometry collection with *attributes*

- data frame with geometry column of class `sfc`
- sticky geometry column through `sf_column` attribute

Understanding sfg and sfc objects

```
# Create sfg point objects of some cities
h_sfg <- st_point(c(9.7320, 52.3759))
g_sfg <- st_point(c(9.9158, 51.5413))
w_sfg <- st_point(c(10.7865, 52.4227))
class(w_sfg)

## [1] "XY"    "POINT" "sfg"
```

```
# Create sfc object with multiple sfg objects
points_sfc <- st_sfc(h_sfg, g_sfg, w_sfg, crs = 4326)
class(points_sfc)

## [1] "sfc_POINT" "sfc"
```

Understanding sf objects

```r
# Create a simple feature object
tbl <- tibble(name = c("Hannover", "Göttingen",
                       "Wolfsburg"))
points_sf <- st_sf(tbl, geometry = points_sfc)
class(points_sf)

## [1] "sf"         "data.frame"
```

```r
points_sf %>% glimpse(width=50)

## Observations: 3
## Variables: 2
## $ name     <chr> "Hannover", "Göttingen", "Wo...
## $ geometry <POINT [°]> POINT (9.732 52.3759),...
```

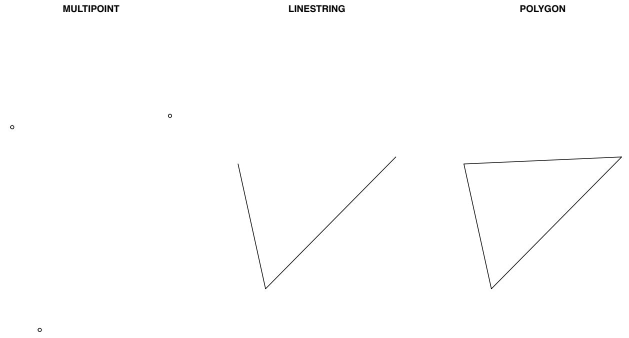
Making lines and polygons

```r
# Create line from points
line_sfc <- st_linestring(rbind(h_sfg, g_sfg, w_sfg))
print(line_sfc, width = 50)

## LINESTRING (9.732 52.3759, 9.9158 51.5413, 10.7...
```

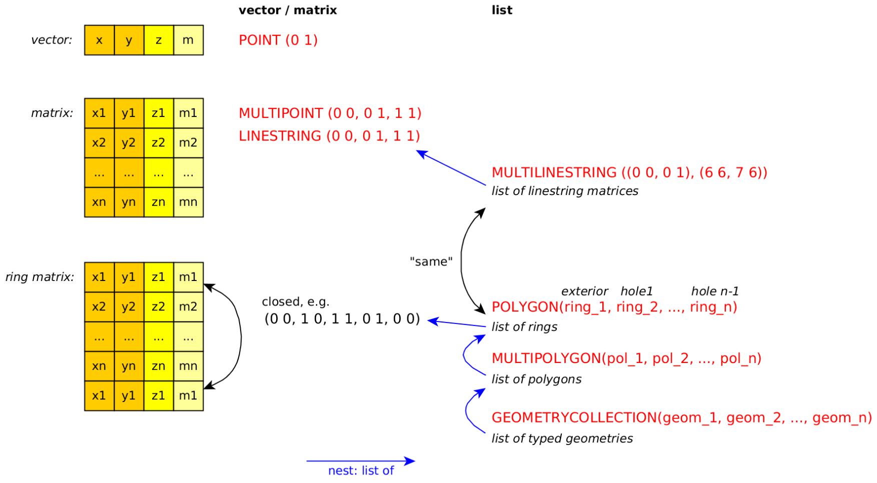
```r
# Create poly from points
poly_sfc <- st_polygon(list(rbind(h_sfg, g_sfg,
                                  w_sfg, h_sfg)))
print(poly_sfc, width = 50)

## POLYGON ((9.732 52.3759, 9.9158 51.5413, 10.786...
```

Plotting basic geometries

```r
par(mfrow = c(1, 3))
plot(points_sfc, main = "MULTIPOINT")
plot(line_sfc, main = "LINESTRING")
plot(poly_sfc, main = "POLYGON")
```



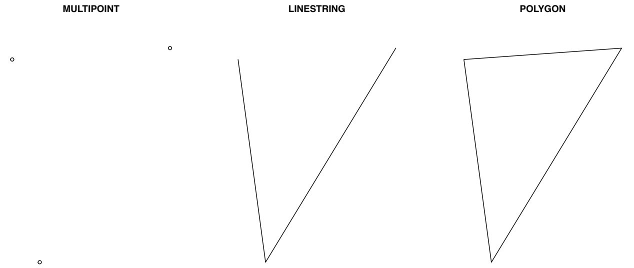
Inside an `sfg` object



Type transformations

Using st_cast(), we can cast objects from one type to another. This conversion is often very useful. Careful: casting works differently on sfg objects, sfg columns and sf objects.

```r
multip_sfc <- st_combine(points_sfc)
linest_sfc <- st_cast(multip_sfc, "LINESTRING")
polygo_sfc <- st_cast(linest_sfc, "POLYGON")
```



An aside on WKT & WKB

Well-known text (WKT) is the standard encoding for simple feature geometries.

```
st_as_text(h_sfg)

## [1] "POINT (9.732 52.3759)"
```

Well-known binary (WKB) are hex strings readable by computers.

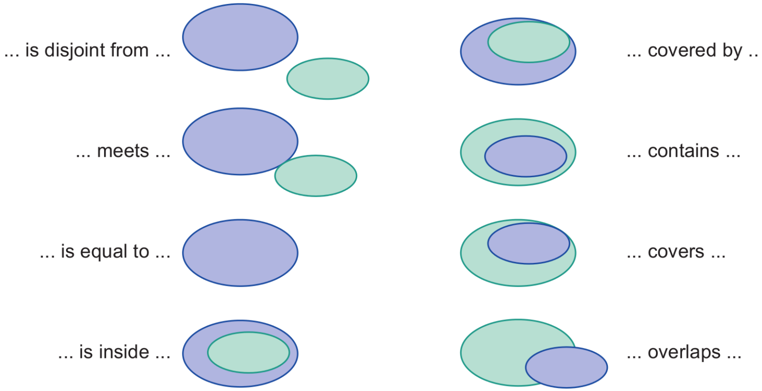
```
st_as_binary(h_sfg)

##  [1] 01 01 00 00 00 10 58 39 b4 c8 76 23 40 80 48
## [16] bf 7d 1d 30 4a 40
```

Topological relations I

Topological relations are binary logical operations on spatial objects (e.g. intersects, covers, touches, contains, and more).

- Equals(a,b): a = b use `st_equals()`
- Disjoint(a,b): $a \cap b = \varnothing$ use `st_disjoint()`
- Intersects(a,b): $a \cap b \neq \varnothing$ use `st_intersects`
- Touches/ meets(a,b): $(a \cap b \neq \varnothing) \vee (a^0 \cap b^0 = \varnothing)$ use `st_touches`
- Contains(a,b): $a \cap b = b$ use `st_contains`
- Covers(a,b): $a^0 \cap b = b$ use `st_covers`
- CoveredBy(a,b): $b^0 \cap a = a$ use `st_covered_by`
- Overlaps(a,b): like intersects without touches. use `st_overlaps`
- Within/ inside(a,b): $a \cap b = a$ use `st_within`

Topological relations II



An example

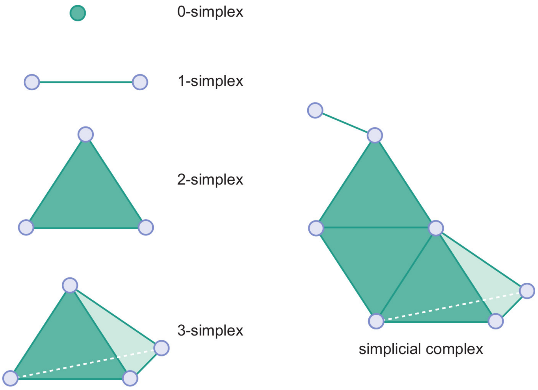
Let's stick with the Hannover-Göttingen-Wolfsburg polygon and ask whether it contains Hildesheim:

```
heim_sfg <- st_point(c(9.9580, 52.1548))
st_contains(poly_sfc, heim_sfg)[1]

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

```
st_contains(poly_sfc, heim_sfg, sparse = F)

##      [,1]
## [1,] TRUE
```

Topology of simplices

Basic simplices are well-understood. We can break down complex topological relationships into their basic forms.

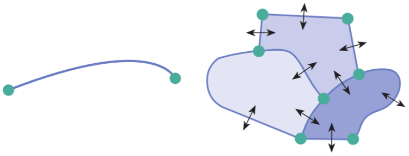
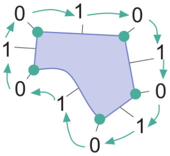
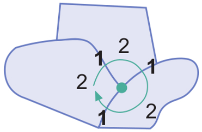


Topological consistency

Five rules of topological consistency of polygons in 2d space:

1. Every 1-simplex (‘arc’) must be bounded by two 0-simplices (‘nodes’ or ‘points’, namely its begin and end node).
2. Every 1-simplex borders two 2-simplices (‘polygons’, namely its ‘left’ and ‘right’ polygons).
3. Every 2-simplex has a closed boundary consisting of an alternating (and cyclic) sequence of 0- and 1-simplices.
4. Around every 0-simplex exists an alternating (and cyclic) sequence of 1- and 2-simplices.
5. 1-simplices only intersect at their (bounding) nodes.

Consistency illustrated

Topological errors

The most common errors:

- Overshooting occurs when two lines cross, which creates dangles, undershooting when they do not touch.
- Rule 2 fails: *slivers* occur when two polygons do not touch. *Overlaps* when they intersect.
- Breaking rule 5 leads to notorious and simply evil *self-intersections*.

Most GIS, like `sf` in *R*, are not fully topologically aware. They might create these errors during spatial operations.

All of these errors are hard to fix automatically (use `lwgeom`). It is often best to edit the polygon manually in QGIS, after flagging the problems using topology checkers.

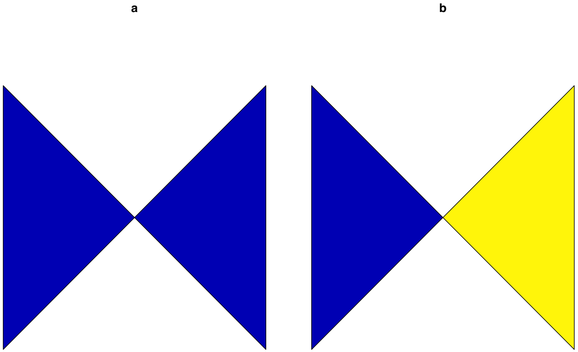
Self-intersections illustrated I

```r
# Create a self-intersecting polygon:
p1_wkt <- "POLYGON((0 0, 0 10, 10 0, 10 10, 0 0))"
p1 <- st_as_sfc(p1_wkt, crs=4326)
st_is_valid(p1, reason = T)

## [1] "Self-intersection[5 5]"
```

```r
# Repair using lwgeom, cast to single polygons
require(lwgeom)
p1_fixed <- st_make_valid(p1) %>% st_cast("POLYGON")
st_is_valid(p1_fixed, reason = T)

## [1] "Valid Geometry" "Valid Geometry"
```

Self-intersections illustrated II

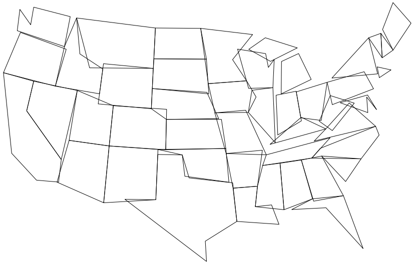
```r
# Plot both polygons
par(mfrow=c(1,2))
plot(st_sf(a = 1, p1), reset=F)
plot(st_sf(b = 1:2, p1_fixed), key.pos=NULL, reset=F)
```



Simplifiying polygons and lines I

Simplification is the process of reducing the vertice count. `st_simplify` implements the Douglas-Peucker algorithm, which is not topologically aware. Only use it on line features.

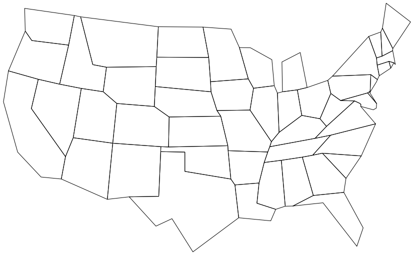
```r
usa_ea <- st_transform(us_states, 2163) #from spData
usa_sf <- st_simplify(usa_ea, dTolerance = 1e+05)
plot(usa_sf[1], col=NA, main="")
```



Simplifiying polygons and lines II

The rmapshaper package was written to provide topologically aware simplification using the Visvalingam algorithm.

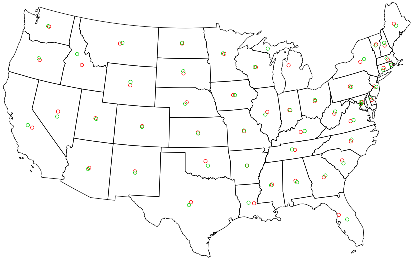
```
require(rmapshaper)
usa_ea$AREA = as.numeric(usa_ea$AREA) # kill units
usa_ms <- ms_simplify(usa_ea, keep = 0.01)
plot(usa_ms[1], col=NA, main="")
```



Centroids

Geographic centroids are the centers of mass of a given geometry. `st_point_on_surface()` always returns a point on the geometry, but it does not necessarily coincide with the centroid.

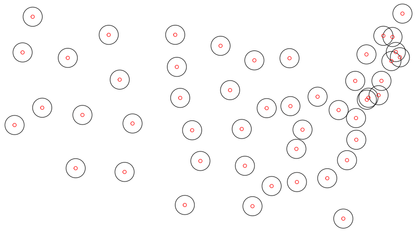
```r
plot(usa_ea[1], col=NA, main="", reset=F)
plot(st_centroid(usa_ea), col=2, add=T)
plot(st_point_on_surface(usa_ea), col=3, add=T)
```



Buffers

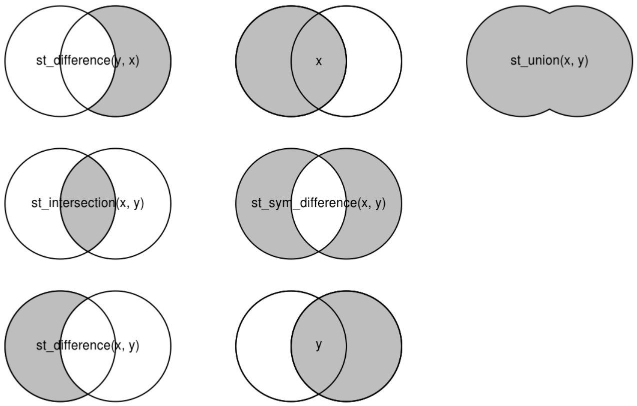
A buffer expands the geometry outward by a specified distance. Works a bit differently for points and lines than for polygons. Only makes sense using projected coordinates (fix area or distance).

```r
buff_sf <- st_centroid(usa_ea) %>% st_buffer(dist = 1e5)
plot(buff_sf[1], col=NA, reset=F, main="")
plot(st_centroid(usa_ea)[1], col=2, add=T)
```



Clipping

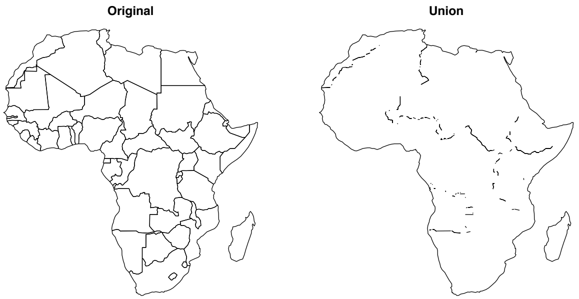
Clipping describes a set of operations where the underlying geometry (column) is changed. Think of these operations as Venn diagrams:



Unions (dissolve)

Unions create slivers when the underlying geometries are imperfect.

```
africa_sf <- st_read("./data/africa_scale.shp", quiet = T)
par(mfrow=c(1,2))
plot(africa_sf[1], col=NA, main="Original", reset=F)
plot(st_union(africa_sf), col=NA, main="Union")
```



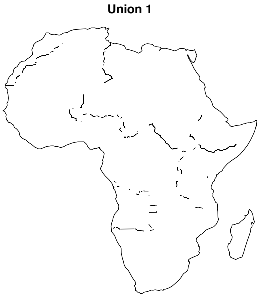
Use group and summarize for unions

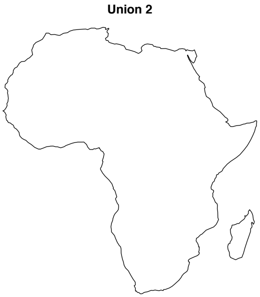
```r
# Transform to Mollweide
africa_sf <- st_transform(africa_sf, "+proj=moll")

# Group_by and summarize does unions
first_sf <- africa_sf %>%
  group_by(continent) %>% summarize()

# Snapping the object to itself by 1m kills slivers
second_sf <- africa_sf %>%
  st_snap(africa_sf, tolerance =  1) %>%
  group_by(continent) %>% summarize()

# Transform back to longlat
first_sf <- first_sf %>% st_transform(4326)
second_sf <- second_sf %>% st_transform(4326)
```

Unions without slivers

... and voilà, gone are the slivers.

```r
par(mfrow=c(1,2))
plot(first_sf[1], col=NA, main="Union 1", reset=F)
plot(second_sf[1], col=NA, main="Union 2", reset=F)
```





Cropping and bounding boxes

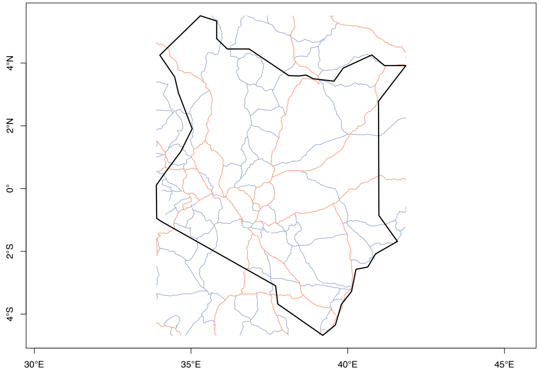
The bounding rectangle of any geometry is given by `st_bbox()`. Geometries can be clipped by this bounding box via `st_crop()`.

```
# get some data and subset boundaries to Kenya
afr_rds <- st_read("./data/africa_roads.shp", quiet=T)
kenya <- africa_sf %>% st_transform(4326) %>%
  filter(adm0_a3 == "KEN")
st_bbox(kenya)

##     xmin     ymin     xmax     ymax
## 33.89357 -4.67677 41.85508  5.50600
```

```
# calls st_bbox automatically
kenya_rds <- st_crop(afr_rds, kenya)
```

Cropped roads and Kenyan boundaries

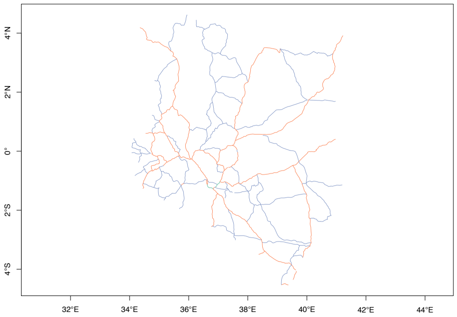
```r
par(mfrow=c(1,1))
plot(kenya_rds[1], main="", reset=F,
     key.pos = NULL, axes=T)
plot(kenya[1], col=NA, lwd=2, add=T)
```



Spatial subsetting

We can select features of a spatial object based on whether or not they in some way relate in space to *another* object. You can use any topological relationship.

```r
plot(kenya_rds[kenya, "type", op = st_within],
     main="", key.pos=NULL, axes=T)
```



Spatial joins with topological operations

Matching spatial data is easy: `st_join()` performs a left join and uses `st_intersects()` by default, but it can perform inner joins and use any topological relationship as well.

Let's work through a tricky example with our African cities and boundaries data:

```r
afr_ctys <- st_read("./data/africa_ctys.shp", quiet=T,
                    stringsAsFactors = F) %>%
  select(name, iso3v10)
afr_sf <- st_read("./data/africa_bnds.shp", quiet = T,
                    stringsAsFactors = F)
# join
afr_ctys_1 <- afr_ctys %>% st_join(afr_sf)
```

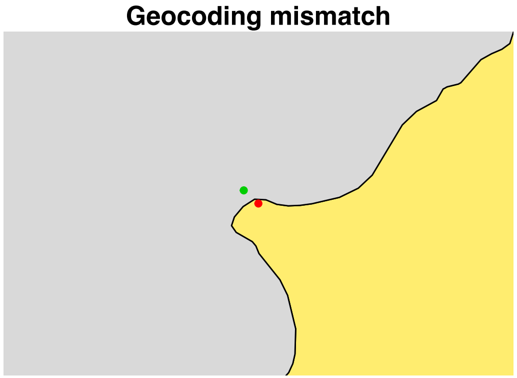
Example continued

```
# filter out disagreements
afr_ctys_1 <- afr_ctys_1 %>% filter(iso3v10!=adm0_a3)
# all mismatches other than in MAR
glimpse(afr_ctys_1 %>% filter(admin != "Morocco"))

## Observations: 24
## Variables: 5
## $ name     <chr> "Lagouira", "Bangui", "Goma"...
## $ iso3v10  <chr> "ESH", "CAF", "COD", "GNQ", ...
## $ admin    <chr> "Western Sahara", "Democrati...
## $ adm0_a3  <chr> "SAH", "COD", "RWA", "GAB", ...
## $ geometry <POINT [°]> POINT (-17.08969 20.83...
```

```
# let's focus on Bangui
bangui <- afr_ctys_1 %>% filter(name == "Bangui")
```

What's going on?

```r
plot(bangui[1], reset=F, main="Geocoding mismatch")
plot(afr_sf[1], add=T)
plot(bangui[1], col=2, pch=20, add=T)
plot(st_point(c(18.558,4.395)), col=3, pch=20, add=T)
```



Intersections on tibbles and data frames

Most of the previous operations can be solved with `st_intersection()`, which automatically computes the *intersection* of the input geometries and merges their data frames.

It will automatically *split* lines and polygons in the process.

Let's revisit the cities:

```r
# st_intersection (same as inner join)
afr_ctys_2 <- afr_ctys %>% st_intersection(afr_sf)
# kill disagreements
afr_ctys_2 <- afr_ctys_2 %>% filter(iso3v10!=adm0_a3)
# compare to st_join after filtering
nrow(afr_ctys_1) == nrow(afr_ctys_2)

## [1] TRUE
```

Non-overlapping joins

Sometimes we want to spatially join units that have no topological relationship.

For example, we might have city coordinates (with names) and unnamed polygons of urban areas. We would like to match the polygons to all city points within 3 km.

We can—in principle—do something like this:

```r
st_join(urbanareas, citypoints,
        st_is_within_distance, dist = 3000)
```

I would urge you do this more explicit with a buffer, an intersection and then some post filtering. That's much clearer and you control the details of the process.

A note on spatial aggregation

Simple features includes a convenience function for aggregating spatial units by another: `aggregate()`.

The function is clunky though; it treats all variables in the same way, unless you change their aggregation type.

There are better alternatives:

1. Intersect $\longrightarrow$ Group $\longrightarrow$ Summarize, or
2. Group $\longrightarrow$ Summarize $\longrightarrow$ Join.

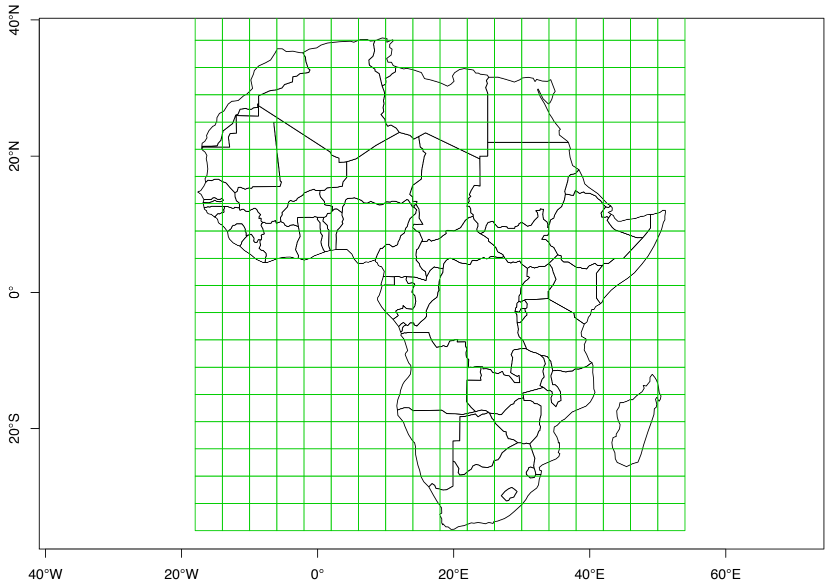
They force you to think explicitly about what you are aggregating and how you are doing it.

Creating grids

Country borders are endogenous (although less so in Africa). We may instead want to analyze data based on equal-sized grid cells. How to create a "fishnet"? Easy ...

```r
# back to degrees
africa_sf <- africa_sf %>% st_transform(4326)

# make grid of 4° cells
grid_sf <- africa_sf %>% st_make_grid(cellsize = 4,
                          offset = c(-18,-35),
                          what="polygons")

# add IDs to grid, make sf
grid_sf <- st_sf(id = 1:length(grid_sf),
                 geometry = grid_sf)
```

Our grid and Africa



Miscellaneous: Planar coordinates redux

```
# polygons crossing the poles create problems
polepol <- list(rbind(c(0,80), c(120,80),
                      c(240,80), c(0,80))) %>%
  st_polygon() %>% st_sfc(crs = 4326)
pole <- st_point(c(0,90)) %>% st_sfc(crs = 4326)
# the n. pole should really be inside this poly
st_intersects(pole, polepol, sparse=F)[1]

## [1] FALSE
```

```
# worse, the centroid is totally off
st_coordinates(st_centroid(polepol))

##     X  Y
## 1 120 80
```