We love making maps, We also love making gifs.  
In this short post I make an animated gif of parts of a map moving. In this case the parts of the map only move in the xy direction

Today I show you how I made a part of the Netherlands ‘float away’. It is part of a larger nonsense project (I have many silly projects), and mostly just to document my path to learning about spatial analytics.

Manipulating Simple Feature Geometrics

Type transformations

This sections discusses how simple feature geometries of one type can be converted to another. For converting lines to polygons, see also st\_polygonize below.

For single geometries

For single geometries, st\_cast will

1. convert from XX to MULTIXX, e.g. LINESTRING to MULTILINESTRING
2. convert from MULTIXX to XX if MULTIXX has length one (else, it will still convert but warn about loss of information)
3. convert from MULTIXX to XX if MULTIXX does not have length one, but it will warn about the loss of information
4. convert GEOMETRYCOLLECTION of length one to its component if

Examples of the first three types are:

library(sf)

## Linking to GEOS 3.10.1, GDAL 3.3.3, PROJ 8.2.0; sf\_use\_s2() is TRUE

suppressPackageStartupMessages(library(dplyr))

st\_point(c(1,1)) %>% st\_cast("MULTIPOINT")

## MULTIPOINT ((1 1))

st\_multipoint(rbind(c(1,1))) %>% st\_cast("POINT")

## Warning in st\_cast.MULTIPOINT(., "POINT"): point from first coordinate only

## POINT (1 1)

st\_multipoint(rbind(c(1,1),c(2,2))) %>% st\_cast("POINT")

## Warning in st\_cast.MULTIPOINT(., "POINT"): point from first coordinate only

## POINT (1 1)

Examples of the fourth type are:

st\_geometrycollection(list(st\_point(c(1,1)))) %>% st\_cast("POINT")

## POINT (1 1)

For collections of geometry (sfc) and simple feature collections (sf)

It should be noted here that when reading geometries using st\_read, the type argument can be used to control the class of the returned geometry:

shp = system.file("shape/nc.shp", package="sf")

class(st\_geometry(st\_read(shp, quiet = TRUE)))

## [1] "sfc\_MULTIPOLYGON" "sfc"

class(st\_geometry(st\_read(shp, quiet = TRUE, type = 3)))

## [1] "sfc\_POLYGON" "sfc"

class(st\_geometry(st\_read(shp, quiet = TRUE, type = 1)))

## [1] "sfc\_GEOMETRY" "sfc"

This option is handled by the GDAL library; in case of failure to convert to the target type, the original types are returned, which in this case is a mix of POLYGON and MULTIPOLYGON geometries, leading to a GEOMETRY as superclass. When we try to read multipolygons as polygons, all secondary rings of multipolygons get lost.

When functions return objects with mixed geometry type (GEOMETRY), downstream functions such as st\_write may have difficulty handling them. For some of these cases, st\_cast may help modify their type. For sets of geometry objects (sfc) and simple feature sets (sf),st\_cast` can be used by specifying the target type, or without specifying it.

ls <- st\_linestring(rbind(c(0,0),c(1,1),c(2,1)))

mls <- st\_multilinestring(list(rbind(c(2,2),c(1,3)), rbind(c(0,0),c(1,1),c(2,1))))

(sfc <- st\_sfc(ls,mls))

## Geometry set for 2 features

## Geometry type: GEOMETRY

## Dimension: XY

## Bounding box: xmin: 0 ymin: 0 xmax: 2 ymax: 3

## CRS: NA

## LINESTRING (0 0, 1 1, 2 1)

## MULTILINESTRING ((2 2, 1 3), (0 0, 1 1, 2 1))

st\_cast(sfc, "MULTILINESTRING")

## Geometry set for 2 features

## Geometry type: MULTILINESTRING

## Dimension: XY

## Bounding box: xmin: 0 ymin: 0 xmax: 2 ymax: 3

## CRS: NA

## MULTILINESTRING ((0 0, 1 1, 2 1))

## MULTILINESTRING ((2 2, 1 3), (0 0, 1 1, 2 1))

sf <- st\_sf(a = 5:4, geom = sfc)

st\_cast(sf, "MULTILINESTRING")

## Simple feature collection with 2 features and 1 field

## Geometry type: MULTILINESTRING

## Dimension: XY

## Bounding box: xmin: 0 ymin: 0 xmax: 2 ymax: 3

## CRS: NA

## a geom

## 1 5 MULTILINESTRING ((0 0, 1 1,...

## 2 4 MULTILINESTRING ((2 2, 1 3)...

When no target type is given, st\_cast tries to be smart for two cases:

1. if the class of the object is GEOMETRY, and all elements are of identical type, and
2. if all elements are length-one GEOMETRYCOLLECTION objects, in which case GEOMETRYCOLLECTION objects are replaced by their content (which may be a GEOMETRY mix again)

Examples are:

ls <- st\_linestring(rbind(c(0,0),c(1,1),c(2,1)))

mls1 <- st\_multilinestring(list(rbind(c(2,2),c(1,3)), rbind(c(0,0),c(1,1),c(2,1))))

mls2 <- st\_multilinestring(list(rbind(c(4,4),c(4,3)), rbind(c(2,2),c(2,1),c(3,1))))

(sfc <- st\_sfc(ls,mls1,mls2))

## Geometry set for 3 features

## Geometry type: GEOMETRY

## Dimension: XY

## Bounding box: xmin: 0 ymin: 0 xmax: 4 ymax: 4

## CRS: NA

## LINESTRING (0 0, 1 1, 2 1)

## MULTILINESTRING ((2 2, 1 3), (0 0, 1 1, 2 1))

## MULTILINESTRING ((4 4, 4 3), (2 2, 2 1, 3 1))

class(sfc[2:3])

## [1] "sfc\_MULTILINESTRING" "sfc"

class(st\_cast(sfc[2:3]))

## [1] "sfc\_MULTILINESTRING" "sfc"

gc1 <- st\_geometrycollection(list(st\_linestring(rbind(c(0,0),c(1,1),c(2,1)))))

gc2 <- st\_geometrycollection(list(st\_multilinestring(list(rbind(c(2,2),c(1,3)), rbind(c(0,0),c(1,1),c(2,1))))))

gc3 <- st\_geometrycollection(list(st\_multilinestring(list(rbind(c(4,4),c(4,3)), rbind(c(2,2),c(2,1),c(3,1))))))

(sfc <- st\_sfc(gc1,gc2,gc3))

## Geometry set for 3 features

## Geometry type: GEOMETRYCOLLECTION

## Dimension: XY

## Bounding box: xmin: 0 ymin: 0 xmax: 4 ymax: 4

## CRS: NA

## GEOMETRYCOLLECTION (LINESTRING (0 0, 1 1, 2 1))

## GEOMETRYCOLLECTION (MULTILINESTRING ((2 2, 1 3)...

## GEOMETRYCOLLECTION (MULTILINESTRING ((4 4, 4 3)...

class(st\_cast(sfc))

## [1] "sfc\_GEOMETRY" "sfc"

class(st\_cast(st\_cast(sfc), "MULTILINESTRING"))

## [1] "sfc\_MULTILINESTRING" "sfc"

Affine transformations

Affine transformations are transformations of the type f(x)=xA+bf(x)=xA+b, where matrix AA is used to flatten, scale and/or rotate, and bb to translate xx. Low-level examples are:

(p = st\_point(c(0,2)))

## POINT (0 2)

p + 1

## POINT (1 3)

p + c(1,2)

## POINT (1 4)

p + p

## POINT (0 4)

p \* p

## POINT (0 4)

rot = function(a) matrix(c(cos(a), sin(a), -sin(a), cos(a)), 2, 2)

p \* rot(pi/4)

## POINT (1.414214 1.414214)

p \* rot(pi/2)

## POINT (2 1.224647e-16)

p \* rot(pi)

## POINT (2.449294e-16 -2)

Just to make the point, we can for instance rotate the counties of North Carolina 90 degrees clockwise around their centroid, and shrink them to 75% of their original size:

nc = st\_read(system.file("shape/nc.shp", package="sf"), quiet = TRUE)

ncg = st\_geometry(nc)

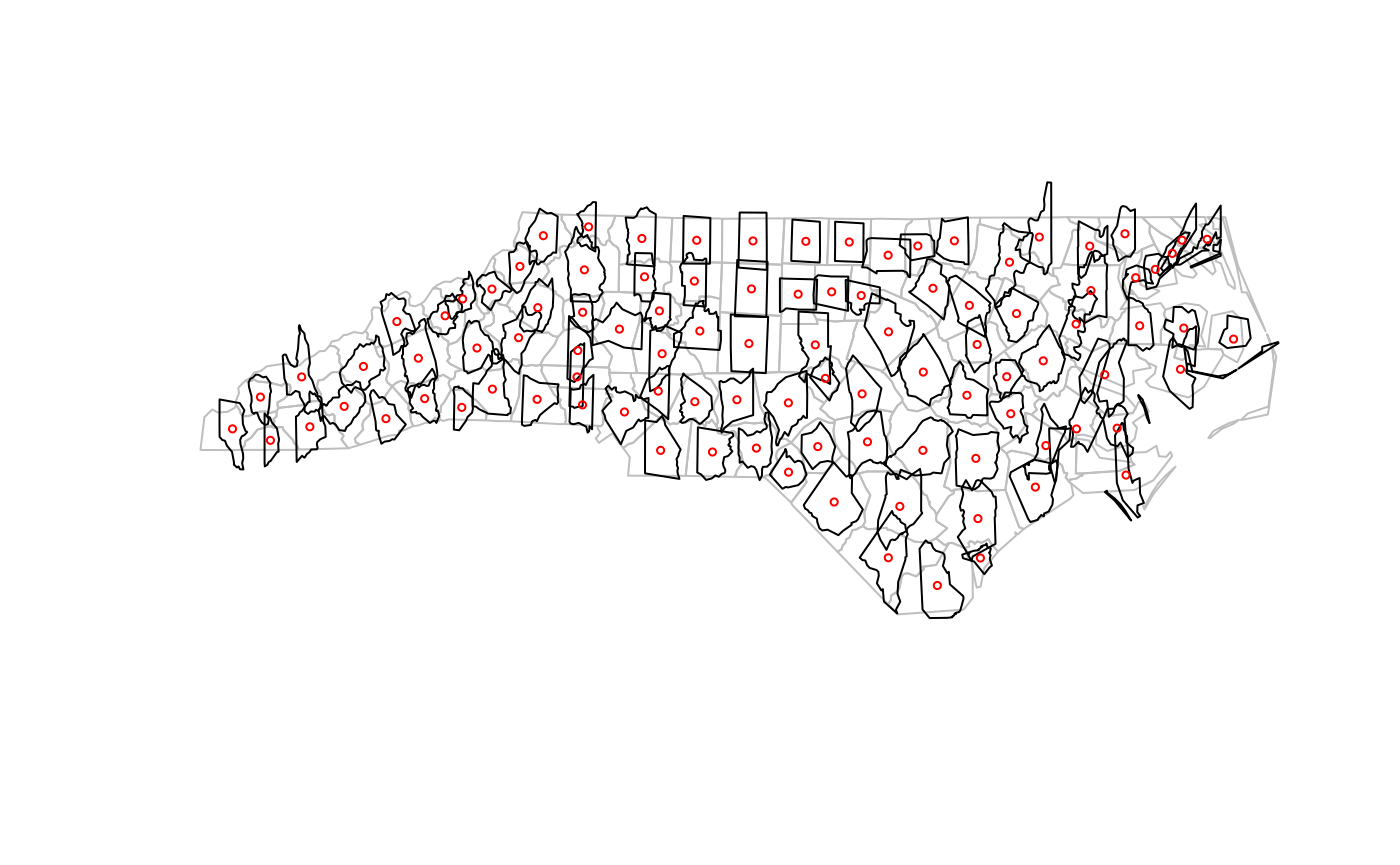
plot(ncg, border = 'grey')

cntrd = st\_centroid(ncg)

ncg2 = (ncg - cntrd) \* rot(pi/2) \* .75 + cntrd

plot(ncg2, add = TRUE)

plot(cntrd, col = 'red', add = TRUE, cex = .5)



Coordinate reference systems conversion and transformation

Getting and setting coordinate reference systems of sf objects

The coordinate reference system of objects of class sf or sfc is obtained by st\_crs, and replaced by st\_crs<-:

library(sf)

geom = st\_sfc(st\_point(c(0,1)), st\_point(c(11,12)))

s = st\_sf(a = 15:16, geometry = geom)

st\_crs(s)

## Coordinate Reference System: NA

s1 = s

st\_crs(s1) <- 4326

st\_crs(s1)

## Coordinate Reference System:

## User input: EPSG:4326

## wkt:

## GEOGCRS["WGS 84",

## DATUM["World Geodetic System 1984",

## ELLIPSOID["WGS 84",6378137,298.257223563,

## LENGTHUNIT["metre",1]]],

## PRIMEM["Greenwich",0,

## ANGLEUNIT["degree",0.0174532925199433]],

## CS[ellipsoidal,2],

## AXIS["geodetic latitude (Lat)",north,

## ORDER[1],

## ANGLEUNIT["degree",0.0174532925199433]],

## AXIS["geodetic longitude (Lon)",east,

## ORDER[2],

## ANGLEUNIT["degree",0.0174532925199433]],

## USAGE[

## SCOPE["Horizontal component of 3D system."],

## AREA["World."],

## BBOX[-90,-180,90,180]],

## ID["EPSG",4326]]

s2 = s

st\_crs(s2) <- "+proj=longlat +datum=WGS84"

all.equal(s1, s2)

## [1] "Component \"geometry\": Attributes: < Component \"crs\": Component \"input\": 1 string mismatch >"

## [2] "Component \"geometry\": Attributes: < Component \"crs\": Component \"wkt\": 1 string mismatch >"

An alternative, more pipe-friendly version of st\_crs<- is

s1 %>% st\_set\_crs(4326)

## Simple feature collection with 2 features and 1 field

## Geometry type: POINT

## Dimension: XY

## Bounding box: xmin: 0 ymin: 1 xmax: 11 ymax: 12

## Geodetic CRS: WGS 84

## a geometry

## 1 15 POINT (0 1)

## 2 16 POINT (11 12)

Coordinate reference system transformations

If we change the coordinate reference system from one non-missing value into another non-missing value, the CRS is is changed without modifying any coordinates, but a warning is issued that this did not reproject values:

s3 <- s1 %>% st\_set\_crs(4326) %>% st\_set\_crs(3857)

## Warning: st\_crs<- : replacing crs does not reproject data; use st\_transform for

## that

A cleaner way to do this that better expresses intention and does not generate this warning is to first wipe the CRS by assigning it a missing value, and then set it to the intended value.

s3 <- s1 %>% st\_set\_crs(NA) %>% st\_set\_crs(3857)

To carry out a coordinate conversion or transformation, we use st\_transform

s3 <- s1 %>% st\_transform(3857)

s3

## Simple feature collection with 2 features and 1 field

## Geometry type: POINT

## Dimension: XY

## Bounding box: xmin: 0 ymin: 111325.1 xmax: 1224514 ymax: 1345708

## Projected CRS: WGS 84 / Pseudo-Mercator

## a geometry

## 1 15 POINT (0 111325.1)

## 2 16 POINT (1224514 1345708)

for which we see that coordinates are actually modified (projected).

Geometrical operations

All geometrical operations st\_op(x) or st\_op2(x,y) work both for sf objects and for sfc objects x and y; since the operations work on the geometries, the non-geometry parts of an sf object are simply discarded. Also, all binary operations st\_op2(x,y) called with a single argument, as st\_op2(x), are handled as st\_op2(x,x).

We will illustrate the geometrical operations on a very simple dataset:

b0 = st\_polygon(list(rbind(c(-1,-1), c(1,-1), c(1,1), c(-1,1), c(-1,-1))))

b1 = b0 + 2

b2 = b0 + c(-0.2, 2)

x = st\_sfc(b0, b1, b2)

a0 = b0 \* 0.8

a1 = a0 \* 0.5 + c(2, 0.7)

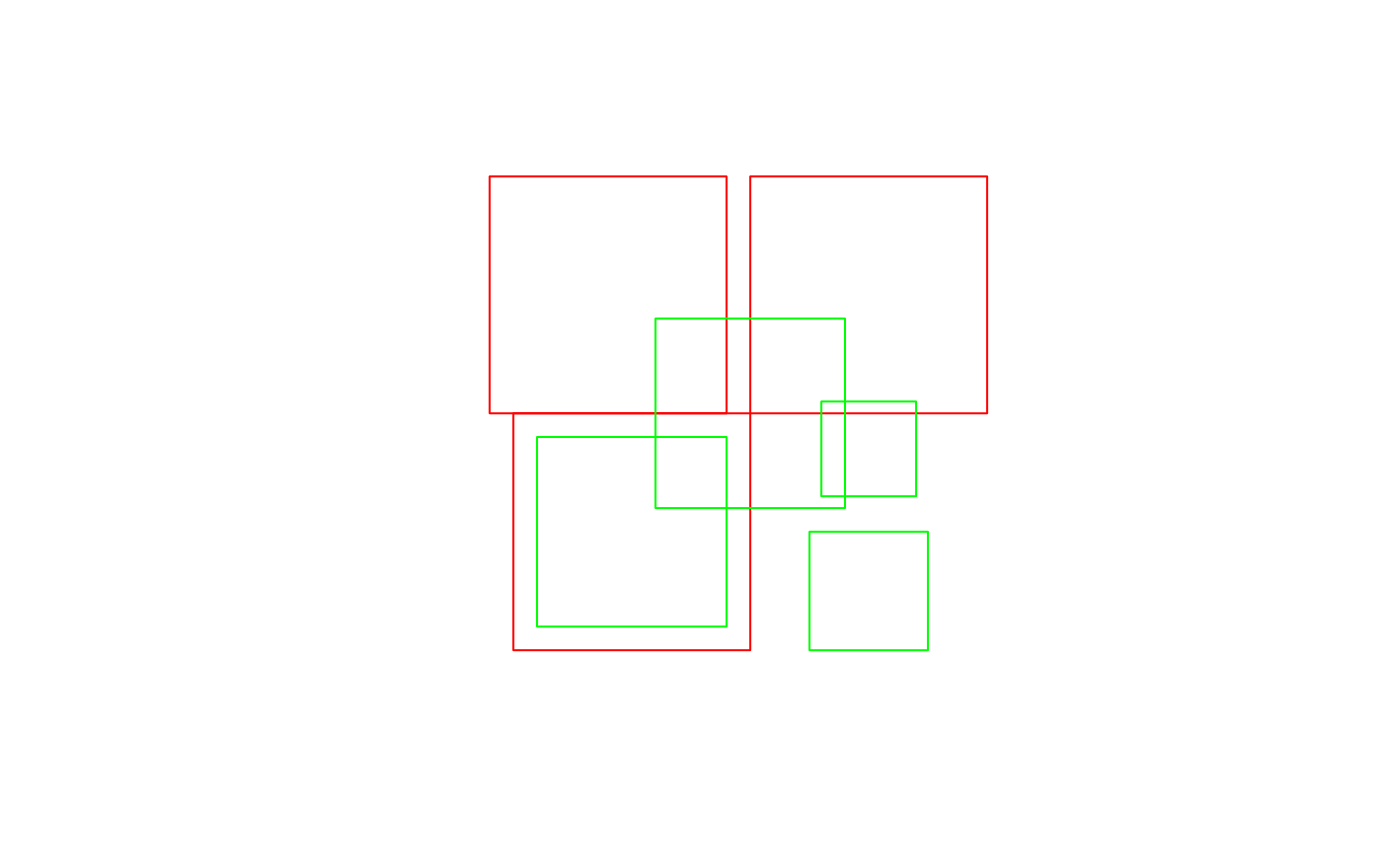
a2 = a0 + 1

a3 = b0 \* 0.5 + c(2, -0.5)

y = st\_sfc(a0,a1,a2,a3)

plot(x, border = 'red')

plot(y, border = 'green', add = TRUE)



Unary operations

st\_is\_valid returns whether polygon geometries are topologically valid:

b0 = st\_polygon(list(rbind(c(-1,-1), c(1,-1), c(1,1), c(-1,1), c(-1,-1))))

b1 = st\_polygon(list(rbind(c(-1,-1), c(1,-1), c(1,1), c(0,-1), c(-1,-1))))

st\_is\_valid(st\_sfc(b0,b1))

## [1] TRUE FALSE

and st\_is\_simple whether line geometries are simple:

s = st\_sfc(st\_linestring(rbind(c(0,0), c(1,1))),

st\_linestring(rbind(c(0,0), c(1,1),c(0,1),c(1,0))))

st\_is\_simple(s)

## [1] TRUE FALSE

st\_area returns the area of polygon geometries, st\_length the length of line geometries:

st\_area(x)

## [1] 4 4 4

st\_area(st\_sfc(st\_point(c(0,0))))

## [1] 0

st\_length(st\_sfc(st\_linestring(rbind(c(0,0),c(1,1),c(1,2))), st\_linestring(rbind(c(0,0),c(1,0)))))

## [1] 2.414214 1.000000

st\_length(st\_sfc(st\_multilinestring(list(rbind(c(0,0),c(1,1),c(1,2))),rbind(c(0,0),c(1,0))))) # ignores 2nd part!

## [1] 2.414214

Binary operations: distance and relate

st\_distance computes the shortest distance matrix between geometries; this is a dense matrix:

st\_distance(x,y)

## [,1] [,2] [,3] [,4]

## [1,] 0.0000000 0.6 0 0.500000

## [2,] 0.2828427 0.0 0 1.000000

## [3,] 0.2000000 0.8 0 1.220656

st\_relate returns a dense character matrix with the DE9-IM relationships between each pair of geometries:

st\_relate(x,y)

## [,1] [,2] [,3] [,4]

## [1,] "212FF1FF2" "FF2FF1212" "212101212" "FF2FF1212"

## [2,] "FF2FF1212" "212101212" "212101212" "FF2FF1212"

## [3,] "FF2FF1212" "FF2FF1212" "212101212" "FF2FF1212"

element [i,j] of this matrix has nine characters, referring to relationship between x[i] and y[j], encoded as IxIy,IxBy,IxEy,BxIy,BxBy,BxEy,ExIy,ExBy,ExEyIxIy,IxBy,IxEy,BxIy,BxBy,BxEy,ExIy,ExBy,ExEy where II refers to interior, BB to boundary, and EE to exterior, and e.g. BxIyBxIy the dimensionality of the intersection of the the boundary BxBx of x[i] and the interior IyIy of y[j], which is one of {0,1,2,F}, indicating zero-, one-, two-dimension intersection, and (F) no intersection, respectively.

Binary logical operations:

Binary logical operations return either a sparse matrix

st\_intersects(x,y)

## Sparse geometry binary predicate list of length 3, where the predicate

## was `intersects'

## 1: 1, 3

## 2: 2, 3

## 3: 3

or a dense matrix

st\_intersects(x, x, sparse = FALSE)

## [,1] [,2] [,3]

## [1,] TRUE TRUE TRUE

## [2,] TRUE TRUE FALSE

## [3,] TRUE FALSE TRUE

st\_intersects(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] TRUE FALSE TRUE FALSE

## [2,] FALSE TRUE TRUE FALSE

## [3,] FALSE FALSE TRUE FALSE

where list element i of a sparse matrix contains the indices of the TRUE elements in row i of the the dense matrix. For large geometry sets, dense matrices take up a lot of memory and are mostly filled with FALSE values, hence the default is to return a sparse matrix.

st\_intersects returns for every geometry pair whether they intersect (dense matrix), or which elements intersect (sparse). Note that the function st\_intersection in this package returns a geometry for the intersection instead of logicals as in st\_intersects (see the next section of this vignette).

Other binary predicates include (using sparse for readability):

st\_disjoint(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] FALSE TRUE FALSE TRUE

## [2,] TRUE FALSE FALSE TRUE

## [3,] TRUE TRUE FALSE TRUE

st\_touches(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] FALSE FALSE FALSE FALSE

## [2,] FALSE FALSE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

st\_crosses(s, s, sparse = FALSE)

## [,1] [,2]

## [1,] FALSE FALSE

## [2,] FALSE FALSE

st\_within(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] FALSE FALSE FALSE FALSE

## [2,] FALSE FALSE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

st\_contains(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] TRUE FALSE FALSE FALSE

## [2,] FALSE FALSE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

st\_overlaps(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] FALSE FALSE TRUE FALSE

## [2,] FALSE TRUE TRUE FALSE

## [3,] FALSE FALSE TRUE FALSE

st\_equals(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] FALSE FALSE FALSE FALSE

## [2,] FALSE FALSE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

st\_covers(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] TRUE FALSE FALSE FALSE

## [2,] FALSE FALSE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

st\_covered\_by(x, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] FALSE FALSE FALSE FALSE

## [2,] FALSE FALSE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

st\_covered\_by(y, y, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] TRUE FALSE FALSE FALSE

## [2,] FALSE TRUE FALSE FALSE

## [3,] FALSE FALSE TRUE FALSE

## [4,] FALSE FALSE FALSE TRUE

st\_equals\_exact(x, y,0.001, sparse = FALSE)

## [,1] [,2] [,3] [,4]

## [1,] FALSE FALSE FALSE FALSE

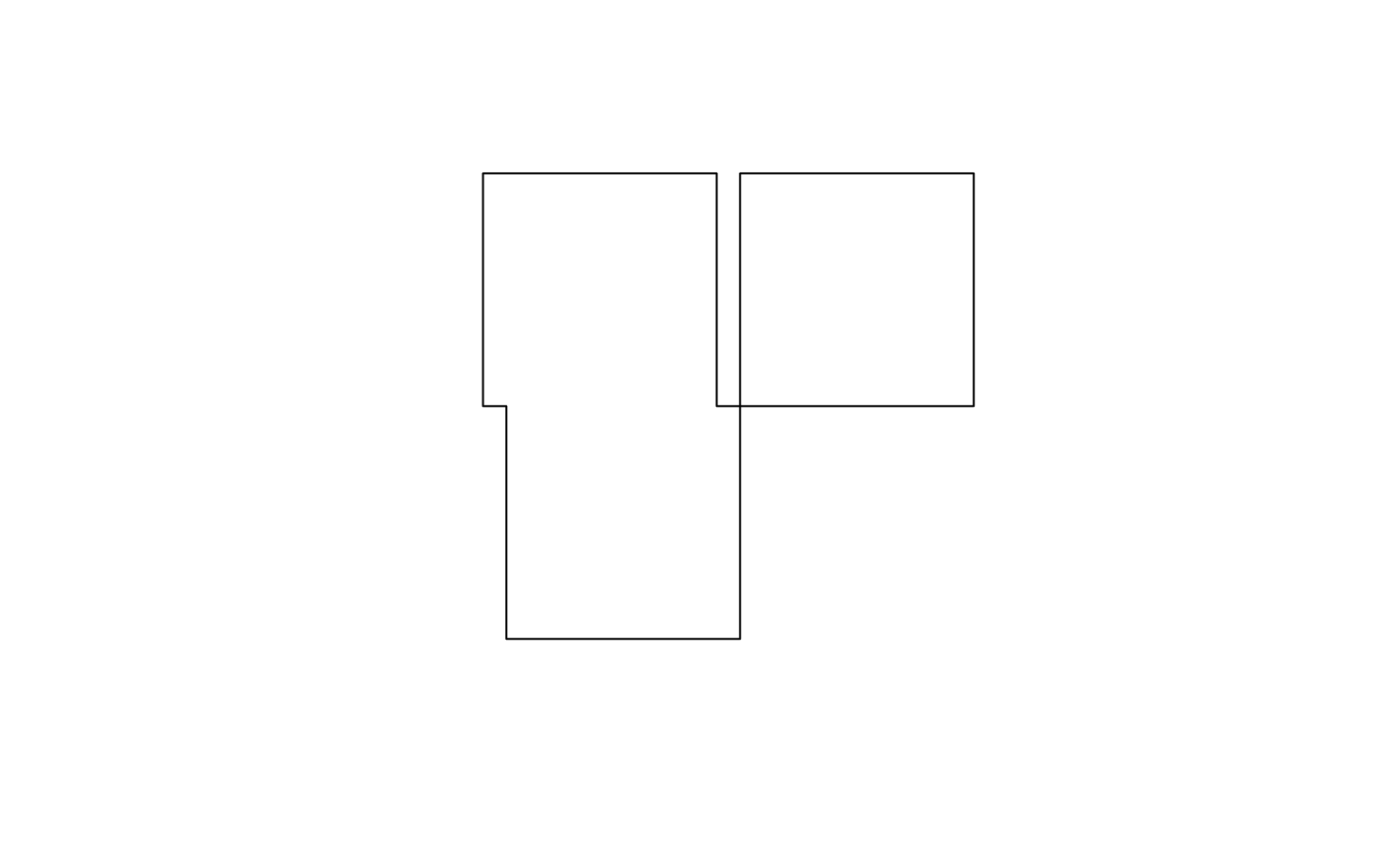
## [2,] FALSE FALSE FALSE FALSE

## [3,] FALSE FALSE FALSE FALSE

Operations returning a geometry

u = st\_union(x)

plot(u)



par(mfrow=c(1,2), mar = rep(0,4))

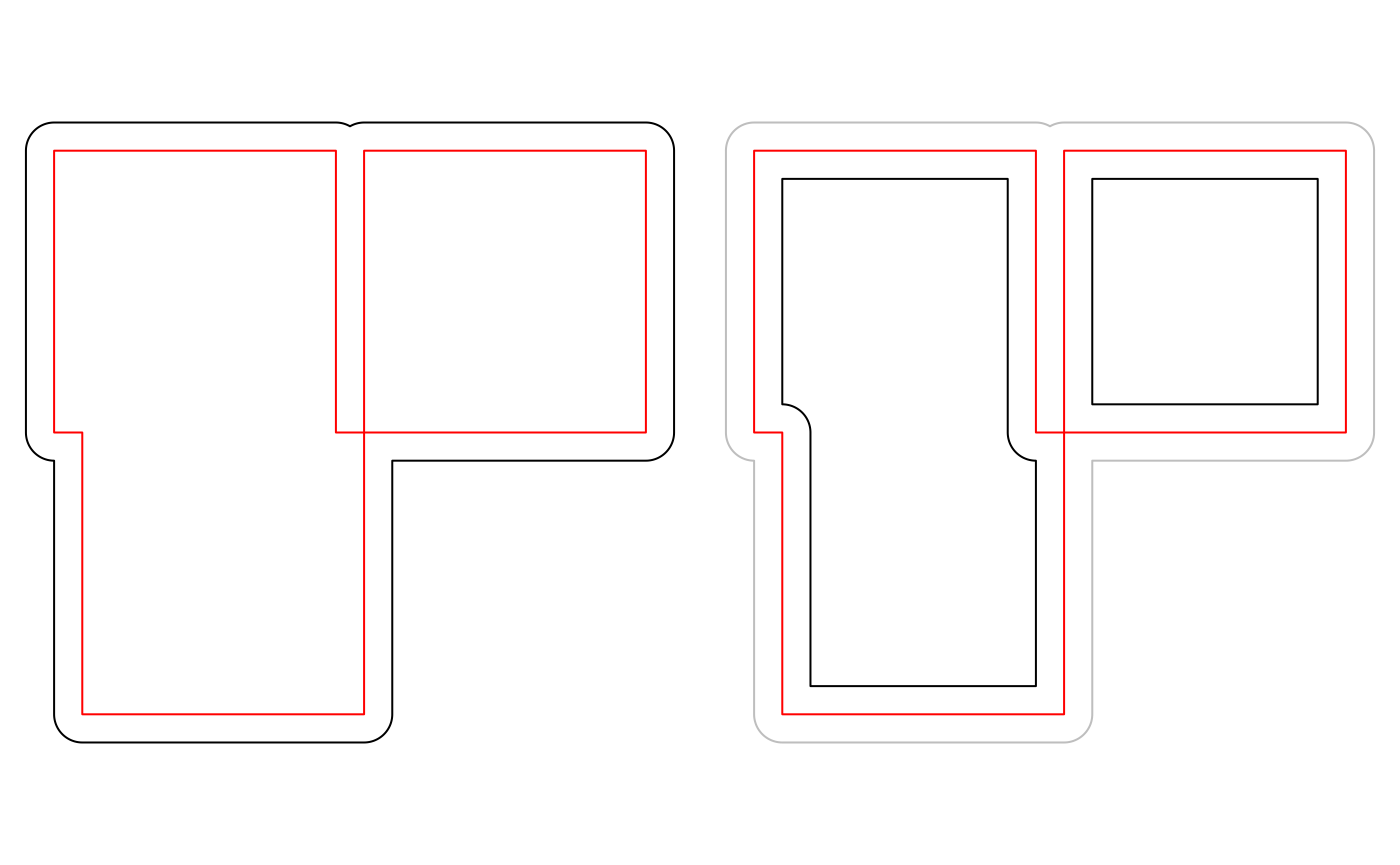
plot(st\_buffer(u, 0.2))

plot(u, border = 'red', add = TRUE)

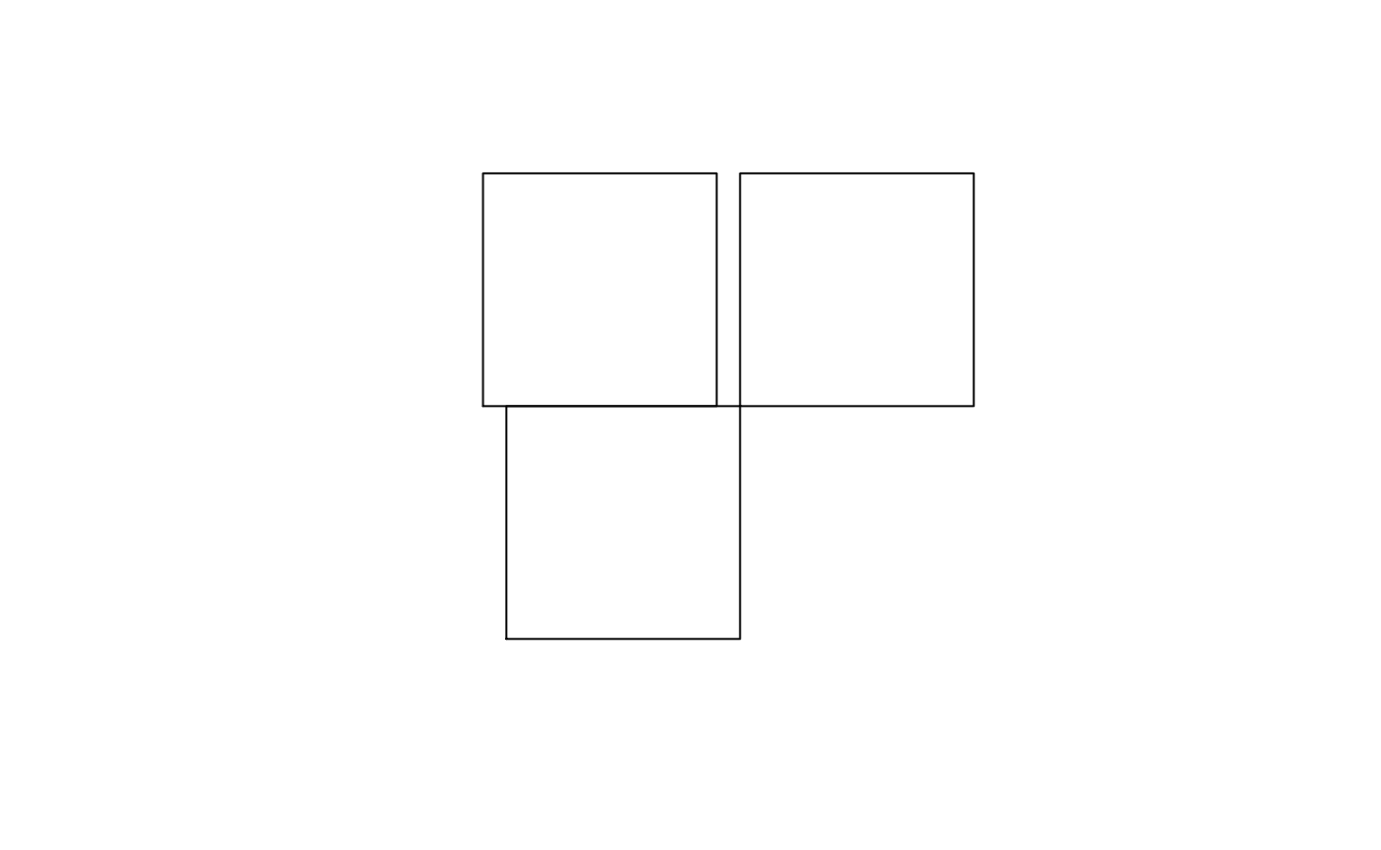
plot(st\_buffer(u, 0.2), border = 'grey')

plot(u, border = 'red', add = TRUE)

plot(st\_buffer(u, -0.2), add = TRUE)



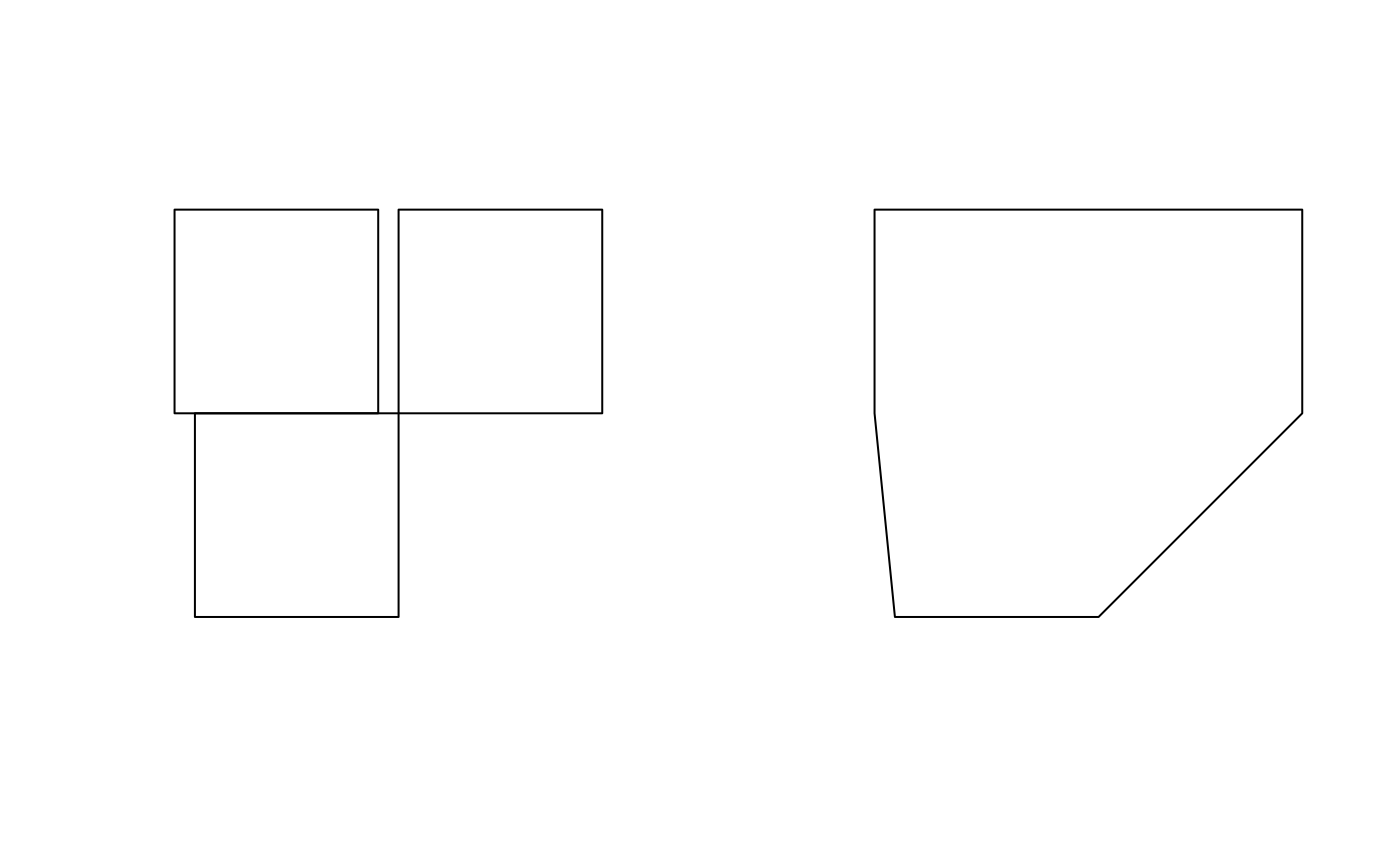
plot(st\_boundary(x))



par(mfrow = c(1:2))

plot(st\_convex\_hull(x))

plot(st\_convex\_hull(u))



par(mfrow = c(1,1))

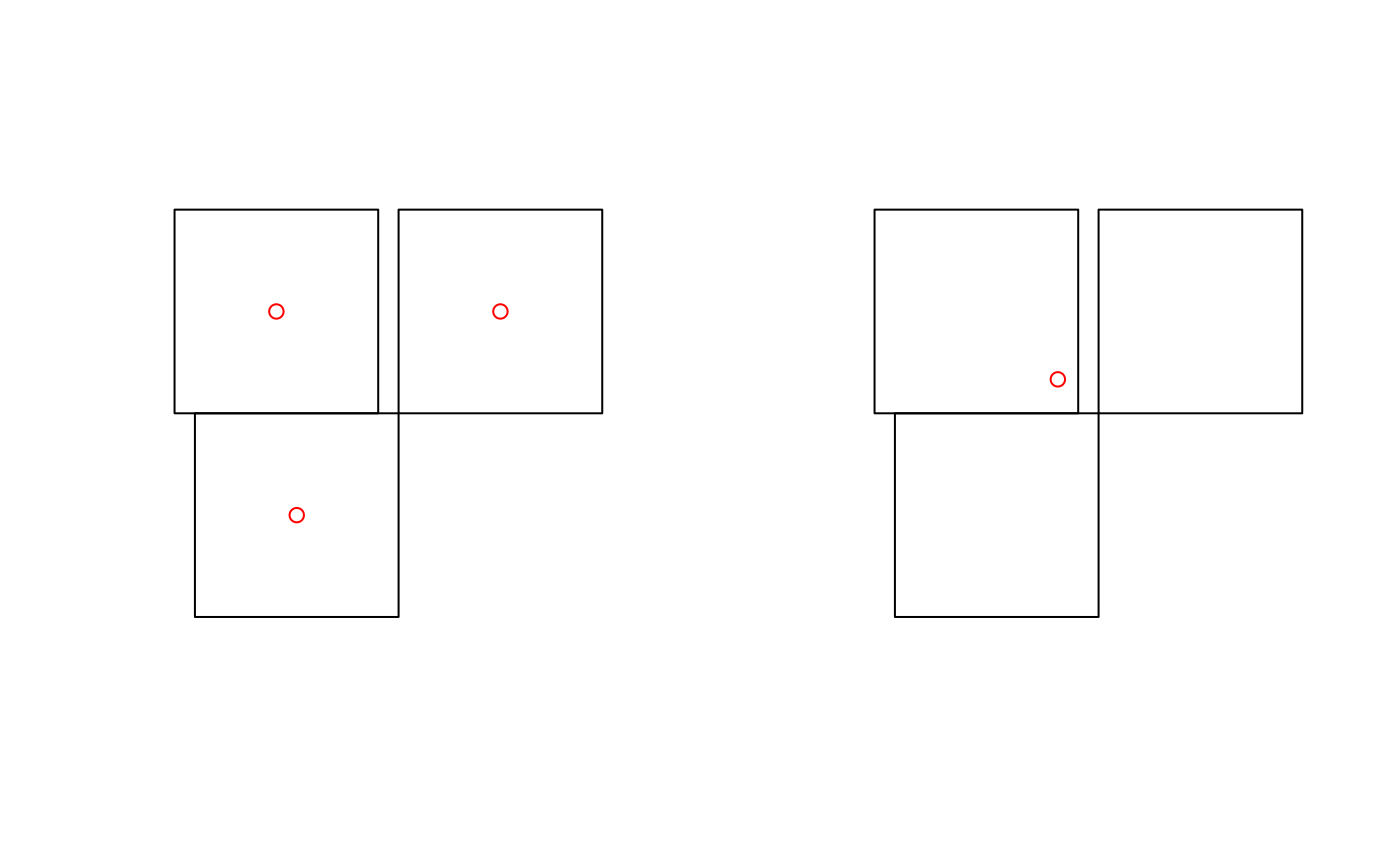
par(mfrow=c(1,2))

plot(x)

plot(st\_centroid(x), add = TRUE, col = 'red')

plot(x)

plot(st\_centroid(u), add = TRUE, col = 'red')

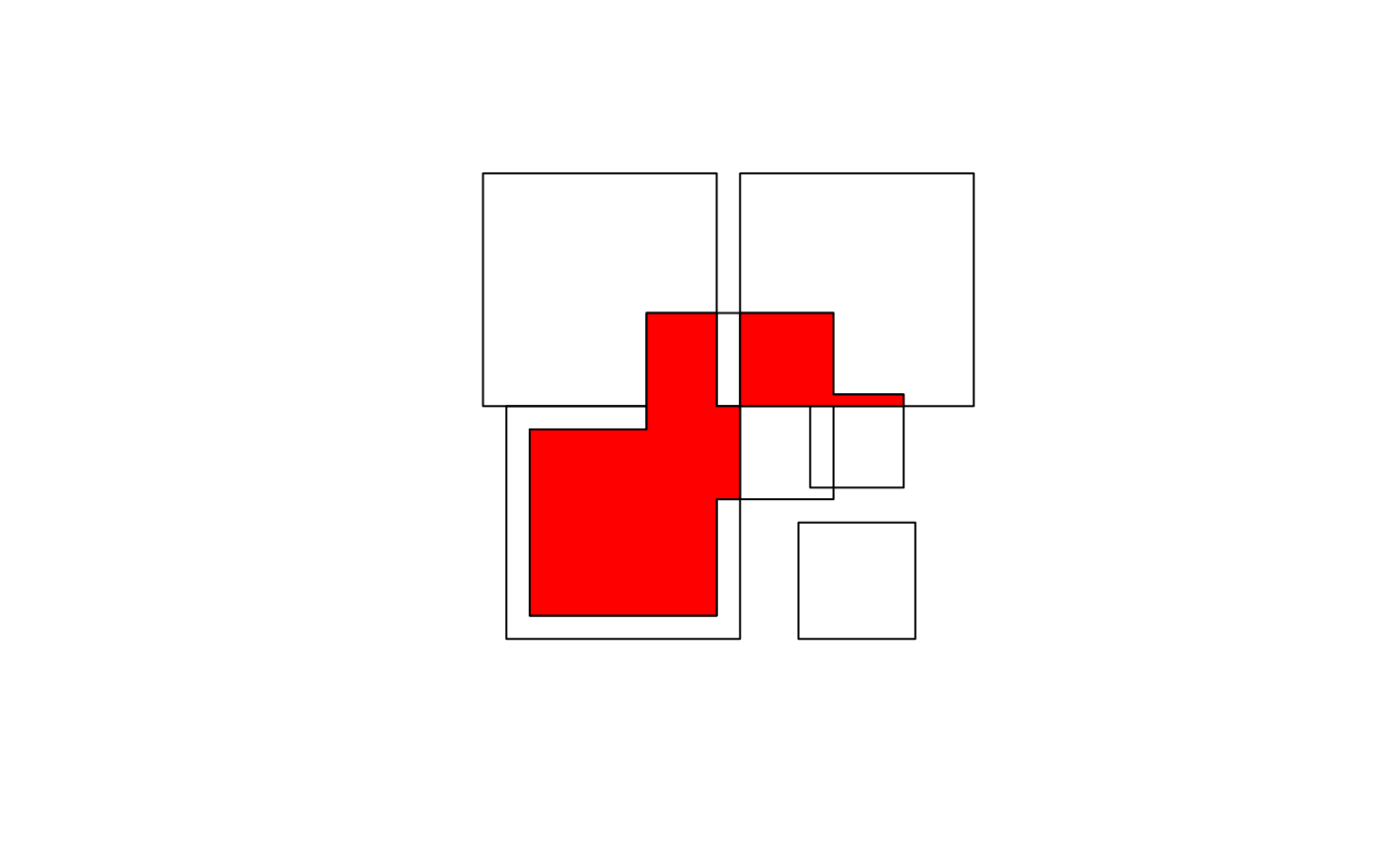


The intersection of two geometries is the geometry covered by both; it is obtained by st\_intersection:

plot(x)

plot(y, add = TRUE)

plot(st\_intersection(st\_union(x),st\_union(y)), add = TRUE, col = 'red')



Note that the function st\_intersects returns a logical matrix indicating whether each geometry pair intersects (see the previous section in this vignette).

To get *everything but* the intersection, use st\_difference or st\_sym\_difference:

par(mfrow=c(2,2), mar = c(0,0,1,0))

plot(x, col = '#ff333388');

plot(y, add=TRUE, col='#33ff3388')

title("x: red, y: green")

plot(x, border = 'grey')

plot(st\_difference(st\_union(x),st\_union(y)), col = 'lightblue', add = TRUE)

title("difference(x,y)")

plot(x, border = 'grey')

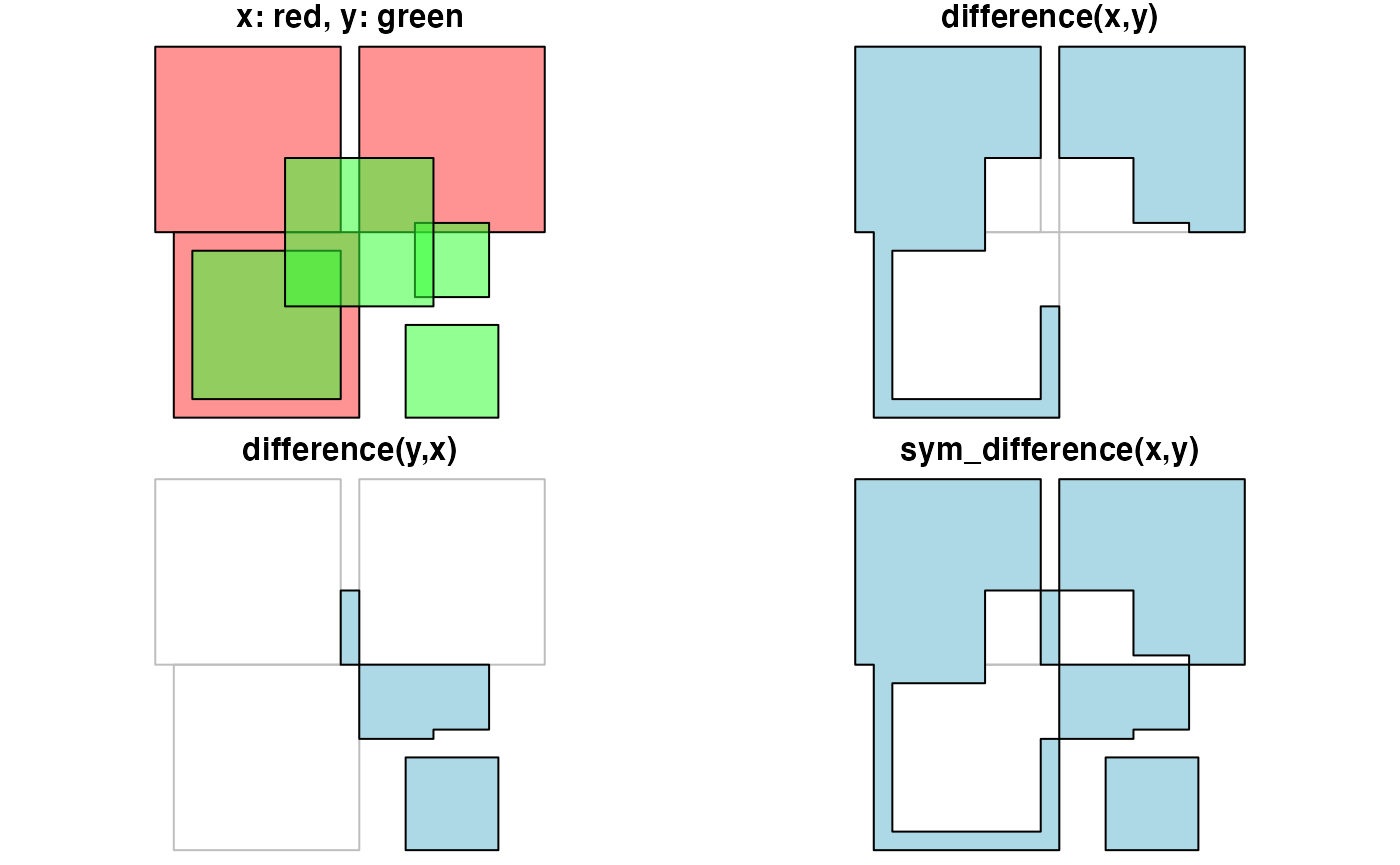
plot(st\_difference(st\_union(y),st\_union(x)), col = 'lightblue', add = TRUE)

title("difference(y,x)")

plot(x, border = 'grey')

plot(st\_sym\_difference(st\_union(y),st\_union(x)), col = 'lightblue', add = TRUE)

title("sym\_difference(x,y)")



Function st\_segmentize adds points to straight line sections of a lines or polygon object:

par(mfrow=c(1,3),mar=c(1,1,0,0))

pts = rbind(c(0,0),c(1,0),c(2,1),c(3,1))

ls = st\_linestring(pts)

plot(ls)

points(pts)

ls.seg = st\_segmentize(ls, 0.3)

plot(ls.seg)

pts = ls.seg

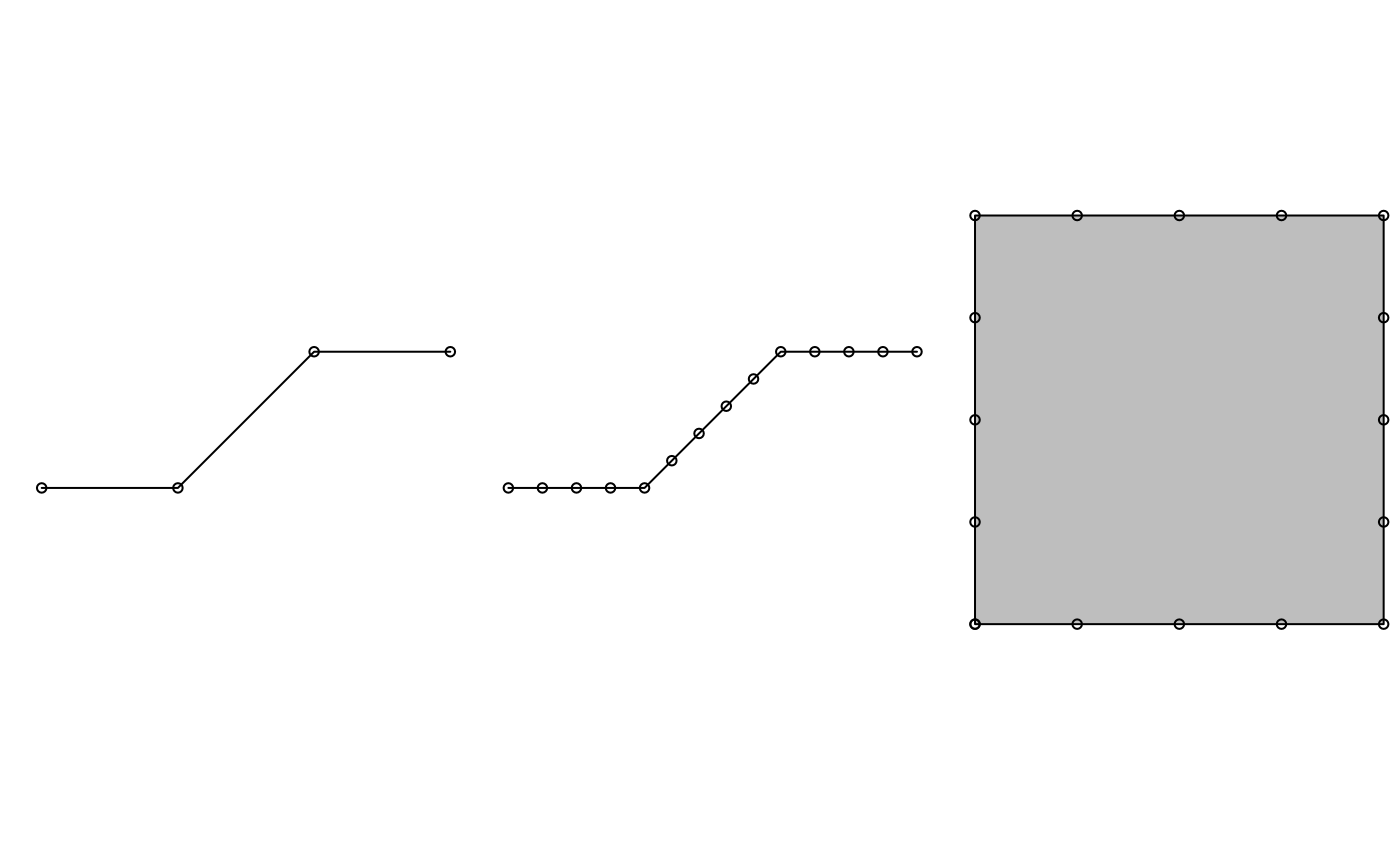
points(pts)

pol = st\_polygon(list(rbind(c(0,0),c(1,0),c(1,1),c(0,1),c(0,0))))

pol.seg = st\_segmentize(pol, 0.3)

plot(pol.seg, col = 'grey')

points(pol.seg[[1]])



Function st\_polygonize polygonizes a multilinestring, as long as the points form a closed polygon:

par(mfrow=c(1,2),mar=c(0,0,1,0))

mls = st\_multilinestring(list(matrix(c(0,0,0,1,1,1,0,0),,2,byrow=TRUE)))

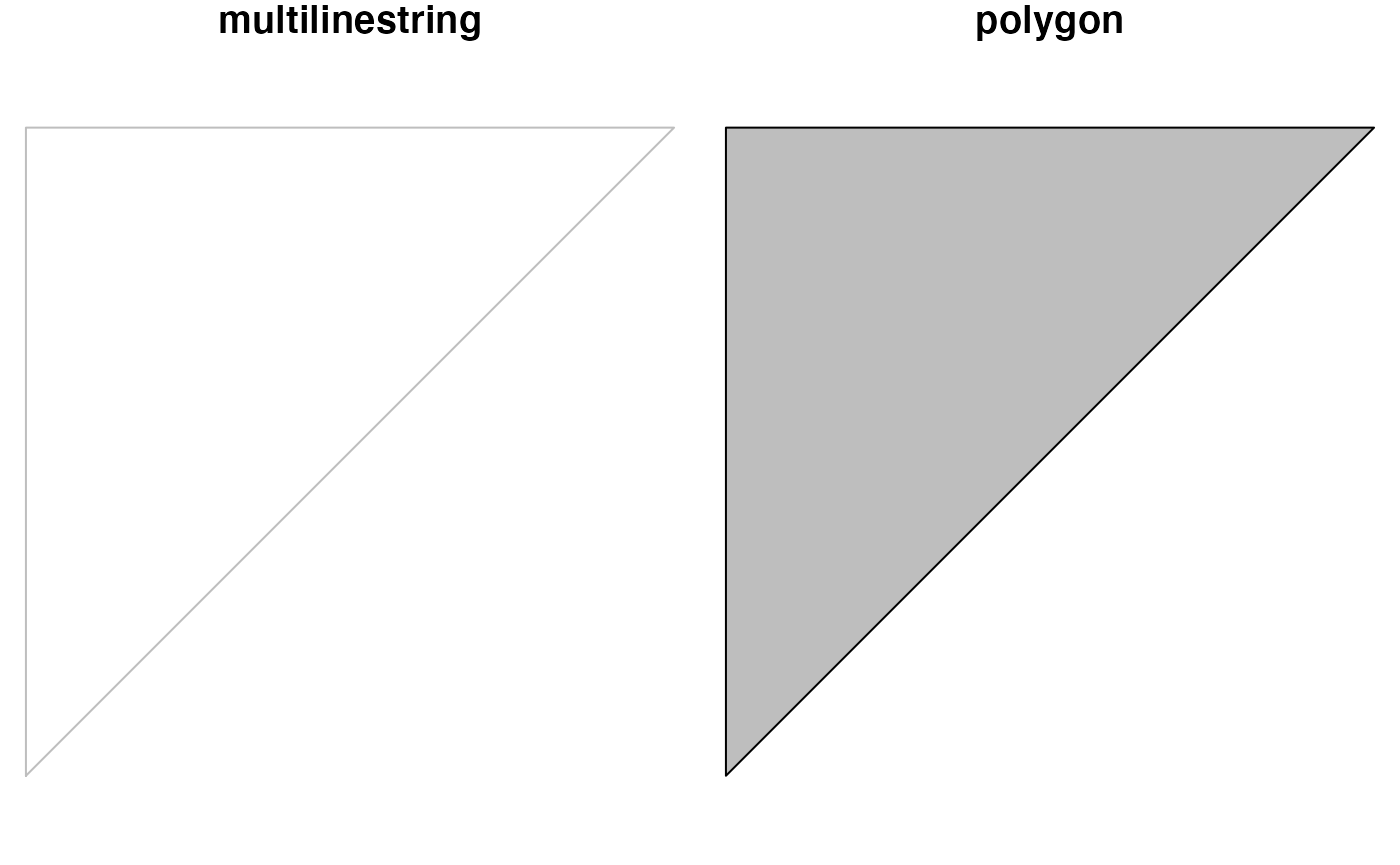
x = st\_polygonize(mls)

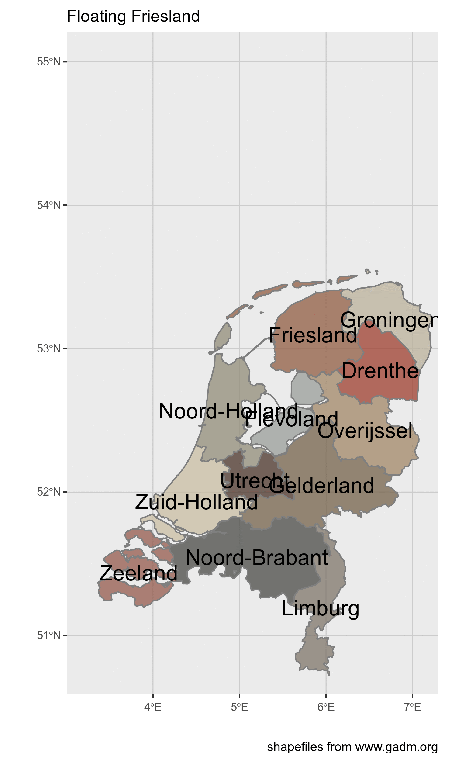
plot(mls, col = 'grey')

title("multilinestring")

plot(x, col = 'grey')

title("polygon")



End result :

**General principles**

* make small functions that do one thing well (not that small in this case)
* combine those
* make imagemagick canvas
* write to the canvas
* animate the canvas

**loading libraries and data**

suppressMessages(library(tidyverse))

library(magick)

library(sf)

library(paletti)

# colorscheme

dutchmasters\_fill <- get\_scale\_fill(get\_pal(dutchmasters))

# the data

NLD <- read\_sf("data/NLD\_adm1.shp") # I cannot redistribute the data from GADM, but you can download and use it for your projects

**basic functions**

I created a function that takes a name, uses that to filter the data and apply a transformation on that part only. (a mutate\_if() could also work, but I didn’t know how). And also one that uses that function to plot. The final function takes a matrix of xy values and sequentially applies every row to the plotting function.

* modify data
* plot a single ggplot version
* loop or apply over range

# basic function that moves an a province

move\_province <- function(provincename, movement){

mov <- quo(movement)

rest <- NLD %>%

filter(NAME\_1 != !!provincename) %>%

filter(TYPE\_1 != "Water body")

#rest %>% st\_centroid() %>% st\_as\_text()

province <- NLD %>%

filter(NAME\_1 == !!provincename) %>%

mutate(geometry = geometry + !!mov) %>%

st\_set\_crs("+proj=longlat +datum=WGS84 +no\_defs")

data1 <-

rbind(province, rest)

centroids <-

data1 %>% st\_centroid() %>% st\_coordinates()

cbind(data1, centroids)

}

# make function to create plot

# using the previous function to move the province

plot\_netherlands <- function(province, movement){

plotunit <- move\_province(provincename = province, movement = movement) %>%

ggplot()+

geom\_sf(aes(fill = NAME\_1),color = "grey50", alpha = 3/4)+

geom\_text(aes(X,Y, label = NAME\_1), size = 6)+

lims(x = c(3.2,7.1), y = c(50.8,55))+

labs(x="", y = "", caption = "shapefiles from www.gadm.org", title = "Floating Friesland")+

dutchmasters\_fill("little\_street")+

theme( legend.position = "empty", # we already labeled the provinces

panel.grid.major = element\_line(colour = "grey80"))

print(plotunit) # you have to explicitly tell it to print so the image is captured

}

# go over every frame and print

plot\_province\_over\_range <-

function(offset\_matrix, province = "Friesland", debug = FALSE){

if(any(is.na(offset\_matrix))){stop("I cannot handle empty movements, there are NA's in movement\_matrix")}

if(NCOL(offset\_matrix) != 2) stop("movement\_matrix needs to have exactly 2 columns")

actionsframe <- data\_frame(x = offset\_matrix[,1], y = offset\_matrix[,2]) %>%

mutate(rownumber = row\_number())

actionsframe$name <- paste0(formatC(actionsframe$rownumber, flag = 0,width = 4))

pb <- progress\_estimated(NROW(actionsframe))

walk(actionsframe$name, ~{

pb$tick()$print()

vars <- filter(actionsframe, name == .x)

if(debug){

message("using values from: ",vars)

}

plot\_netherlands(province = province,movement = c(vars$x[[1]], vars$y[[1]]))

}) # ends the walk action

}

**The plotting and saving**

Nothing happened before the next step (except loading data). All the action and calculation happens here.

## then the creation starts with the movement

Friesland\_moves <- rbind(

matrix(c(c(0,-.1,-.2,-.2,-.3), c(0,.03,.05,.1,.15)) ,ncol = 2),

matrix(c(seq(from = -.3, by = -.1, length.out = 14),seq(from = .2, by = .1, length.out = 14)), ncol = 2)

)

# set up print location

frames <- image\_graph(width = 1500, height = 2500, res = 300, pointsize = 5)

plot\_province\_over\_range(offset\_matrix = Friesland\_moves, province = "Friesland")

# animate

image\_animate(frames, 1) %>%

image\_write(path = "friesland.gif")