Package ‘plotKML’

January 9, 2019

Version 0.5-9

Date 2019-01-04

Title Visualization of Spatial and Spatio-Temporal Objects in Google Earth

Maintainer Tomislav Hengl <tom.hengl@opengeohub.org>

Depends R (>= 2.13.0)

Imports methods, tools, utils, XML, sp, raster, rgdal, spacetime, colorspace, plotrix, dismo, aqp, pixmap, plyr, stringr, colorRamps, scales, gstat, zoo, RColorBrewer, RSAGA, classInt

Suggests adehabitatLT, maptools, fossil, spcosa, rjson, animation, spatstat, RCurl, rgbif, Hmisc, GSIF, uuid, intervals, reshape, gdalUtils, snowfall, parallel

Description Writes sp-class, spacetime-class, raster-class and similar spatial and spatio-temporal objects to KML following some basic cartographic rules.

URL http://plotkml.r-forge.r-project.org/

LazyLoad yes

RoxygenNote 6.0.1

NeedsCompilation no

Author Tomislav Hengl [cre, aut],
Pierre Roudier [ctb],
Dylan Beaudette [ctb],
Edzer Pebesma [ctb],
Michael Blaschek [ctb]

R topics documented:

`plotKML-package` *Visualization of spatial and spatio-temporal objects in Google Earth*

Description

A suite of functions for converting 2D and 3D spatio-temporal (sp, raster and spacetime package classes) objects into KML or KMZ documents for use in Google Earth.

Details

Package:	plotKML
Type:	Package
URL:	`http://plotkml.r-forge.r-project.org/`
License:	GPL
LazyLoad:	yes

Note

This package has been developed as a part of the Global Soil Information Facilities project, which is run jointly by the ISRIC Institute and collaborators. ISRIC is a non-profit organization with a mandate to serve the international community as custodian of global soil information and to increase awareness and understanding of the role of soils in major global issues.

Author(s)

Tomislav Hengl (<`tom.hengl@opengeohub.org`>), Pierre Roudier (<`pierre.roudier@landcare.nz`>), Dylan Beaudette (<`debeaudette@ucdavis.edu`>), Edzer Pebesma (<`edzer.pebesma@uni-muenster.de`>)

References

- KML documentation (`http://code.google.com/apis/kml/documentation/`)
- **Google Earth Outreach** project (`http://earth.google.com/outreach/tutorials.html`)
- Hengl, T., Roudier, P., Beaudette, D. and Pebesma, E. (2015) plotKML: Scientific Visualization of Spatio-Temporal Data. Journal of Statistical Software, 63(5): 1–25.

aesthetics *Plotting aesthetics parameters*

Description

Parses various object parameters / columns to KML aesthetics: size of the icons, fill color, labels, altitude, width, ...

Usage

```
kml_aes(obj, ...)
```

Arguments

`obj`	space-time object for plotting
`...`	other arguments

Details

Valid aesthetics: `colour = "black"`, `fill = "white"`, `shape`, `whitening`, `alpha`, `width = 1`, `labels`, `altitude = 0`, `size`, `balloon = FALSE`. Specific features (target variables and the connected hot-spots) can be emphasized by using two or three graphical parameters for the same variable. See plotKML package homepage / vignette for more examples.

Author(s)

Pierre Roudier

See Also

`kml-methods`

baranja *Baranja hill case study*

Description

Baranja hill is a 4 by 4 km large study area in the Baranja region, eastern Croatia (corresponds to a size of an aerial photograph). This data set has been extensively used to describe various DEM modelling and analysis steps (see Hengl and Reuter, 2008; Hengl et al., 2010). Object `barxyz` contains 6370 precise observations of elevations (from field survey and digitized from the stereo images); `bargrid` contains *observed* probabilities of streams (digitized from the 1:5000 topo map); `barstr` contains 100 simulated stream networks (`"SpatialLines"`) using `barxyz` point data as input (see examples below).

Usage

```
data(bargrid)
```

Format

The `bargrid` data frame (regular grid at 30 m intervals) contains the following columns:

`p.obs` observed probability of stream (0-1)

`x` a numeric vector; x-coordinate (m) in the MGI / Balkans zone 6

`y` a numeric vector; y-coordinate (m) in the MGI / Balkans zone 6

Note

Consider using the 30 m resolution grid (see `bargrid`) as the target resolution (output maps).

Author(s)

Tomislav Hengl

References

- Hengl, T., Reuter, H.I. (eds), (2008) Geomorphometry: Concepts, Software, Applications. Developments in Soil Science, vol. 33, Elsevier, 772 p.
- Hengl, T., Heuvelink, G. B. M., van Loon, E. E., (2010) On the uncertainty of stream networks derived from elevation data: the error propagation approach. Hydrology and Earth System Sciences, 14:1153-1165.
- `http://geomorphometry.org/content/baranja-hill`

Examples

```
library(sp)
library(gstat)
## sampled elevations:
data(barxyz)
prj = "+proj=tmerc +lat_0=0 +lon_0=18 +k=0.9999 +x_0=6500000 +y_0=0 +ellps=bessel +units=m
+towgs84=550.499,164.116,475.142,5.80967,2.07902,-11.62386,0.99999445824"
coordinates(barxyz) <- ~x+y
proj4string(barxyz) <- CRS(prj)
## grids:
data(bargrid)
data(barstr)
coordinates(bargrid) <- ~x+y
gridded(bargrid) <- TRUE
proj4string(bargrid) <- barxyz@proj4string
bargrid@grid
## Not run: ## Example with simulated streams:
data(R_pal)
library(rgdal)
library(RSAGA)
pnt = list("sp.points", barxyz, col="black", pch="+")
spplot(bargrid[1], sp.layout=pnt,
  col.regions = R_pal[["blue_grey_red"]])
## Deriving stream networks using geostatistical simulations:
Z.ovgm <- vgm(psill=1831, model="Mat", range=1051, nugget=0, kappa=1.2)
sel <- runif(length(barxyz$Z))<.2
N.sim <- 5
## geostatistical simulations:
DEM.sim <- krige(Z~1, barxyz[sel,], bargrid, model=Z.ovgm, nmax=20,
```

```
   nsim=N.sim, debug.level=-1)
## Note: this operation can be time consuming

stream.list <- list(rep(NA, N.sim))
## derive stream networks in SAGA GIS:
for (i in 1:N.sim) {
  writeGDAL(DEM.sim[i], paste("DEM", i, ".sdat", sep=""),
     drivername = "SAGA", mvFlag = -99999)
  ## filter the spurious sinks:
  rsaga.fill.sinks(in.dem=paste("DEM", i, ".sgrd", sep=""),
     out.dem="DEMflt.sgrd", check.module.exists = FALSE)
  ## extract the channel network SAGA GIS:
  rsaga.geoprocessor(lib="ta_channels", module=0,
    param=list(ELEVATION="DEMflt.sgrd",
    CHNLNTWRK=paste("channels", i, ".sgrd", sep=""),
    CHNLROUTE="channel_route.sgrd",
    SHAPES="channels.shp",
    INIT_GRID="DEMflt.sgrd",
    DIV_CELLS=3, MINLEN=40),
    check.module.exists = FALSE,
    show.output.on.console=FALSE)
  stream.list[[i]] <- readOGR("channels.shp", "channels",
    verbose=FALSE)
  proj4string(stream.list[[i]]) <- barxyz@proj4string
}
# plot all derived streams at top of each other:
streams.plot <- as.list(rep(NA, N.sim))
for(i in 1:N.sim){
  streams.plot[[i]] <- list("sp.lines", stream.list[[i]])
}
spplot(DEM.sim[1], col.regions=grey(seq(0.4,1,0.025)), scales=list(draw=T),
sp.layout=streams.plot)

## End(Not run)
```

---

bigfoot *Bigfoot reports (USA)*

Description

2984 observations of bigfoot (with attached dates). The field occurrence records have been obtained from the BigFoot Research Organization (BFRO) website. The BFRO reports generally consist of a description of the event and where it occurred, plus the quality classification. Similar data set has been used by Lozier et al. (2009) to demonstrate possible miss-interpretations of the results of species distribution modeling. The maps in the `USAWgrids` data set represent typical gridded environmental covariates used for species distribution modeling.

Usage

```
data(bigfoot)
```

Format

The `bigfoot` data frame contains the following columns:

`Lon` a numeric vector; x-coordinate / longitude in the WGS84 system

`Lat` a numeric vector; y-coordinate / latitude in the WGS84 system

`NAME` name assigned by the observer (usually referent month / year)

`DATE` 'POSIXct' class vector

`TYPE` confidence levels; according to the BFRO website: *"Class A" reports involve clear sightings in circumstances where misinterpretation or misidentification of other animals can be ruled out with greater confidence; "Class B" and "Class C" reports are less credible*.

The `USAWgrids` data frame (46,018 pixels; Washington, Oregon, Nevada and California state) contains the following columns:

`globedem` a numeric vector; elevations from the ETOPO1 Global Relief Model

`nlights03` an integer vector; lights at night image for 2003 (Version 2 DMSP-OLS Nighttime Lights Time Series)

`sroads` a numeric vector; distance to main roads and railroads (National Atlas of the United States)

`gcarb` a numeric vector; Global Biomass Carbon Map (New IPCC Tier-1 Global Biomass Carbon Map for the Year 2000)

`dTRI` a numeric vector; density of pollutant releases (North American Pollutant Releases and Transfers database)

`twi` a numeric vector; Topographic Wetness Index based on the globedem

`states` an integer vector; USA states

`globcov` land cover classes based on the MERIS FR images (GlobCover Land Cover version V2.2)

`s1` a numeric vector; x-coordinates in the Albers equal-area projection system

`s2` a numeric vector; y-coordinates in the Albers equal-area projection system

Note

According to the Time.com, a team of a dozen-plus experts from as far afield as Canada and Sweden have proclaimed themselves 95 percent certain of the mythical animal's existence on Kemerovo region territory some 3,000 kilometers east of Moscow (announced at the Tashtagol conference in 2011).

Author(s)

Tomislav Hengl

References

- Lozier, J.D., Aniello, P., Hickerson, M.J., (2009) Predicting the distribution of Sasquatch in western North America: anything goes with ecological niche modelling. Journal of Biogeography, 36(9):1623-1627.
- BigFoot Research Organization (`http://www.bfro.net`)

Examples

```
## Not run: # Load the BFRO records:
library(sp)
data(bigfoot)
aea.prj <- "+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=23 +lon_0=-96
+x_0=0 +y_0=0 +ellps=GRS80 +datum=NAD83 +units=m +no_defs"
library(sp)
coordinates(bigfoot) <- ~Lon+Lat
proj4string(bigfoot) <- CRS("+proj=latlon +datum=WGS84")
library(rgdal)
bigfoot.aea <- spTransform(bigfoot, CRS(aea.prj))
# Load the covariates:
data(USAWgrids)
gridded(USAWgrids) <- ~s1+s2
proj4string(USAWgrids) <- CRS(aea.prj)
# Visualize data:
data(SAGA_pal)
pnts <- list("sp.points", bigfoot.aea, pch="+", col="yellow")
spplot(USAWgrids[2], col.regions=rev(SAGA_pal[[3]]), sp.layout=pnts)

## End(Not run)
```

---

check_projection — *Extracts the proj4 parameters and checks if the projection matches the referent CRS*

Description

Function `parse_proj4` gets the proj4 string from a space-time object and `check_projection` checks if the input projection is compatible with the referent projection system. The referent system is by default the `longlat` projection with WGS84 datum (KML-compatible coordinates).

Usage

```
check_projection(obj, control = TRUE,
                 ref_CRS = get("ref_CRS", envir = plotKML.opts))
```

Arguments

`obj`	object of class `Spatial*` or `Raster*`
`control`	logical; if TRUE, a logical value is returned, if FALSE, an error is thrown if the test failed
`ref_CRS`	the referent coordinate system.

Details

A cartographic projection is KML compatible if: (a) geographical coordinates are used, and (b) if they relate to the WGS84 ellipsoid (`"+proj=longlat +datum=WGS84"`). You can also set your own local referent projection system by specifiying `plotKML.env(ref_CRS = ...)`.

Warning

`obj` needs to have a proper proj4 string (CRS), otherwise `check_projection` will not run. If the geodetic datum is defined via the `+towgs`, consider converting the coordinates manually i.e. by using the `spTransform` or `reproject` method.

Author(s)

Pierre Roudier, Tomislav Hengl, and Dylan Beaudette

References

- WGS84 (http://spatialreference.org/ref/epsg/4326/)

See Also

`reproject`, `rgdal::CRS-class`

Examples

```
data(eberg)
library(sp)
library(rgdal)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
check_projection(eberg)
# not yet ready for export to KML;
parse_proj4(proj4string(eberg))
eberg.geo <- reproject(eberg)
check_projection(eberg.geo)
# ... now ready for export
```

`col2kml` *Convert a color strings to the KML format*

Description

Converts some common color formats (internal R colors, hexadecimal format, Munsell color codes) color to KML format.

Usage

```
col2kml(colour)
```

Arguments

`colour`	R color string

Value

KML-formatted color as `#aabbggrr` where aa=alpha (00 to ff), bb=blue (00 to ff), gg=green (00 to ff), `rr`=red (00 to ff).

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

See Also

`aqp::munsell2rgb`

Examples

```
col2kml("white")
col2kml(colors()[2])
hex2kml(rgb(1,1,1))
x <- munsell2kml("10YR", "2", "4")
kml2hex(x)
```

---

count.GridTopology — *Counts the number of occurrences of a list of vector object over a GridTopology*

Description

Counts the number of occurrences of a vector object over a "GridTopology" for a list of vector objects (usually multiple realizations of the same process).

Usage

```
count.GridTopology(x, vectL, ...)
```

Arguments

x	object of type "GridTopology"
vectL	list of vectors of class "SpatialPoint*", "SpatialLines*" or "SpatialPolygons*" (equiprobable realizations of the same process)
...	(optional) arguments passed to the lower level functions

Author(s)

Tomislav Hengl

See Also

SpatialVectorsSimulations-class, vect2rast `display.pal` *Display a color palette*

Description

Plots a color palette in a new window.

Usage

```
display.pal(pal, sel=1:length(pal), names=FALSE)
```

Arguments

`pal`	list; each palette a vector of HEX-formated colors
`sel`	integer; selection of palettes to plot
`names`	logical; specifies whether to print also the class names

Details

The internal palettes available in plotKML typically consists of 20 elements. If class names are requrested (`names=TRUE`) than only one palette will be plotted.

Author(s)

Tomislav Hengl and Pierre Roudier

See Also

`SAGA_pal`, `R_pal`, `worldgrids_pal`

Examples

```
# SAGA GIS palette (http://saga-gis.org/en/about/software.html)
data(SAGA_pal)
names(SAGA_pal)
## Not run: # display palettes:
display.pal(pal=SAGA_pal, sel=c(1,2,7,8,10,11,17,18,19,21,22))
dev.off()
data(worldgrids_pal)
worldgrids_pal[["globcov"]]
display.pal(pal=worldgrids_pal, sel=c(5), names = TRUE)
dev.off()
# make icons (http://www.statmethods.net/advgraphs/parameters.html):
for(i in 0:25){
  png(filename=paste("icon", i, ".png", sep=""), width=45, height=45,
  bg="transparent", pointsize=16)
  par(mar=c(0,0,0,0))
  plot(x=1, y=1, axes=FALSE, xlab='', ylab='', pch=i, cex=4, lwd=2)
  dev.off()
}

## End(Not run)
```

eberg — *Ebergotzen — soil mapping case study*

Description

EbergÃ¶tzen is 10 by 10 km study area in the vicinity of the city of GÃ¶ttingen in Central Germany. This area has been extensively surveyed over the years, mainly for the purposes of developing operational digital soil mapping techniques (Gehrt and BÃ¶hner, 2001), and has been used by the SAGA GIS development team to demonstrate various processing steps.
eberg table contains 3670 observations (augers) of soil textures at five depths (0–10, 10–30, 30–50, 50–70, and 70–90), and field records of soil types according to the German soil classification system. eberg_grid contains gridded maps at 100 m resolution that can be used as covariates for spatial prediction of soil variables. eberg_grid25 contains grids at finer resolution (25 m). eberg_zones is a polygon map showing the distribution of parent material (Silt and sand, Sandy material, Clayey derivats, Clay and loess). eberg_contours shows contour lines derived from the 25 m DEM of the area using 10 m equidistance.

Usage

```
data(eberg)
```

Format

The eberg data frame (irregular points) contains the following columns:

- ID universal identifier
- soiltype a vector containing factors; soil classes according to the German soil classification system: "A" (Auenboden), "B" (Braunerde), "D" (Pelosol), "G" (Gley), "Ha" (Moor), "Hw" (HMoor), "K" (Kolluvisol), "L" (Parabraunerde), "N" (Ranker), "Q" (Regosol), "R" (Rendzina), "S" (Pseudogley), "Z" (Pararendzina)
- TAXGRSC a vector containing factors; full soil class names according to the German soil classification system (see soiltype column)
- X a numeric vector; x-coordinate (m) in DHDN / Gauss-Krueger zone 3 (German coordinate system)
- Y a numeric vector; y-coordinate (m) in DHDN / Gauss-Krueger zone 3 (German coordinate system)
- UHDICM_* a numeric vector; upper horizon depth in cm per horizon
- LHDICM_* a numeric vector; lower horizon depth in cm per horizon
- SNDMHT_* a numeric vector; sand content estimated by hand per horizon (0-100 percent); see Adhoc-AG Boden (2005) for more details
- SLTMHT_* a numeric vector; silt content estimated by hand per horizon (0-100 percent)
- CLYMHT_* a numeric vector; clay content estimated by hand per horizon (0-100 percent)

The eberg_grid data frame (regular grid at 100 m resolution) contains the following columns:

- PRMGEO6 a vector containing factors, parent material classes from the geological map (mapping units)
- DEMSRT6 a numeric vector; elevation values from the SRTM DEM
- TWISRT6 a numeric vector; Topographic Wetness Index derived using the SAGA algorithm

TIRAST6 a numeric vector; Thermal Infrared (TIR) reflection values from the ASTER L1 image band 14 (2010-06-05T10:26:50Z) obtained via the NASA's GloVis browser

LNCCOR6 a vector containing factors; Corine Land Cover 2006 classes

x a numeric vector; x-coordinate (m) in DHDN / Gauss-Krueger zone 3 (German coordinate system)

y a numeric vector; y-coordinate (m) in DHDN / Gauss-Krueger zone 3 (German coordinate system)

The eberg_grid25 data frame (regular grid at 25 m resolution) contains the following columns:

DEMTOPx a numeric vector; elevation values from the topographic map

HBTSOLx a vector containing factors; main soil type according to the German soil classification system (see column "soiltype" above) estimated per crop field

TWITOPx a numeric vector; Topographic Wetness Index derived using the SAGA algorithm

NVILANx a numeric vector; NDVI image derived using the Landsat image from the Image 2000 project

x a numeric vector; x-coordinate (m) in DHDN / Gauss-Krueger zone 3 (German coordinate system)

y a numeric vector; y-coordinate (m) in DHDN / Gauss-Krueger zone 3 (German coordinate system)

Note

Texture by hand method can be used to determine the content of soil earth fractions only to an accuracy of Â±5–10% (Skaggs et al. 2001). A surveyor distinguishes to which of the 32 texture classes a soil samples belongs to, and then estimates the content of fractions; e.g. texture class St2 has 10% clay, 25% silt and 65% sand (Ad-hoc-AG Boden, 2005).

Author(s)

The EbergÃ¶tzen dataset is courtesy of Gehrt Ernst (<Ernst.Gehrt@niedersachsen.de>), the State Authority for Mining, Energy and Geology, Hannover, Germany and Olaf Conrad, University of Hamburg (<conrad@geowiss.uni-hamburg.de>). The original data set has been prepared for this exercise by Tomislav Hengl (<tom.hengl@opengeohub.org>).

References

- Ad-hoc-AG Boden, (2005) Bodenkundliche Kartieranleitung. 5th Ed, Bundesanstalt fÃ¼r Geowissenschaften und Rohstoffe und Niedersaechsisches Landesamt fÃ¼r Bodenforshung, Hannover, p. 423.
- BÃ¶hner, J., McCloy, K. R. and Strobl, J. (Eds), (2006) SAGA — Analysis and Modelling Applications. GÃ¶ttinger Geographische Abhandlungen, Heft 115. Verlag Erich Goltze GmbH, GÃ¶ttingen, 117 pp.
- Gehrt, E., BÃ¶hner, J., (2001) Vom punkt zur flache — probleme des 'upscaling' in der bodenkartierung. In: Diskussionsforum Bodenwissenschaften: Vom Bohrstock zum Bildschirm. FH, OsnabrÃ¼ck, pp. 17-34.
- Skaggs, T. H., Arya, L. M., Shouse, P. J., Mohanty, B. P., (2001) Estimating Particle-Size Distribution from Limited Soil Texture Data. Soil Science Society of America Journal 65 (4): 1038-1044.
- http://geomorphometry.org/content/ebergotzen

Examples

```
data(eberg)
data(eberg_grid)
data(eberg_zones)
data(eberg_contours)
library(sp)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
gridded(eberg_grid) <- ~x+y
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
# visualize the maps:
data(SAGA_pal)
l.sp <- list("sp.lines", eberg_contours, col="black")
## Not run:
spplot(eberg_grid["DEMSRT6"], col.regions = SAGA_pal[[1]], sp.layout=l.sp)
spplot(eberg_zones, sp.layout=list("sp.points", eberg, col="black", pch="+"))

## End(Not run)
```

---

fmd *2001 food-and-mouth epidemic, north Cumbria (UK)*

Description

This data set gives the spatial locations and reported times of food-and-mouth disease in north Cumbria (UK), 2001. It is of no scientific value, as it deliberately excludes confidential information on farms at risk in the study-region. It is included in the package purely as an illustrative example.

Usage

```
data(fmd)
```

Format

A matrix containing (x,y,t) coordinates of the 648 observations.

Author(s)

Edith Gabriel <edith.gabriel@univ-avignon.fr>

References

Diggle, P., Rowlingson, B. and Su, T. (2005). Point process methodology for on-line spatio-temporal disease surveillance. Environmetrics, 16, 423–34.

See Also

`northcumbria` for boundaries of the county of north Cumbria.

geopath *Geopath — shortest trajectory line between two geographic locations*

Description

Derives a SpatialLines class object showing the shortest path between the two geographic locations and based on the Haversine Formula for Great Circle distance.

Usage

```
geopath(lon1, lon2, lat1, lat2, ID, n.points, print.geo = FALSE)
```

Arguments

`lon1`	longitude coordinate of the first point
`lon2`	longitude coordinate of the second point
`lat1`	latitude coordinate of the first point
`lat2`	latitude coordinate of the second point
`ID`	(optional) point ID character
`n.points`	number of intermediate points
`print.geo`	prints the distance and bearing

Details

Number of points between the start and end point is derived using a simple formula:

```
round(sqrt(distc)/sqrt(2), 0)
```

where `distc` is the Great Circle Distance.

Value

Bearing is expressed in degrees from north. Distance is expressed in kilometers (Great Circle Distance).

Author(s)

Tomislav Hengl

References

- fossil package (https://CRAN.R-project.org/package=fossil)
- Haversine formula from Math Forums (http://mathforum.org/dr.math/)

See Also

`kml_layer.SpatialLines`, `kml_layer.STTDF`, `fossil::earth.bear`

Examples

```
library(fossil)
ams.ny <- geopath(lon1=4.892222, lon2=-74.005973, lat1=52.373056, lat2=40.714353,
  print.geo=TRUE)
# write to a file:
kml(ams.ny)
```

---

getCRS-methods *Methods to get the proj4 string*

Description

Gets the proj4 string from a object of type "Spatial" or "Raster".

Usage

```
## S4 method for signature 'Spatial'
getCRS(obj)
## S4 method for signature 'Raster'
getCRS(obj)
```

Arguments

`obj`	object of type `"Spatial"` or `"Raster"`

Details

For more details about the PROJ.4 parameters refer to the `https://proj4.org/usage/projections.html`.

Author(s)

Tomislav Hengl and Pierre Roudier

See Also

`sp::CRS`, `raster::raster`, `check_projection`

Examples

```
data(eberg_grid)
library(sp)
coordinates(eberg_grid) <- ~x+y
gridded(eberg_grid) <- TRUE
library(rgdal)
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
library(raster)
r <- raster(eberg_grid[1])
getCRS(r)
r.ll <- reproject(r)
getCRS(r.ll)
```

`getWikiMedia.ImageInfo` *Gets EXIF information*

Description

`getWikiMedia.ImageInfo` function fetches the EXIF (Exchangeable image file format) data via the Wikimedia API for any donated image. The resulting EXIF data (named list) can then be further used to construct an object of class `"SpatialPhotoOverlay"`, which can be parsed to KML.

Usage

```
getWikiMedia.ImageInfo(imagename,
     APIsource = "https://commons.wikimedia.org/w/api.php",
     module = "imageinfo",
     details = c("url", "metadata", "size", "extlinks"), testURL = TRUE)
```

Arguments

`imagename`	Wikimedia commons unique image title
`APIsource`	location of the API service
`module`	default module
`details`	detailed parameters of interest
`testURL`	logical; species if the program should first test whether the image exist at all (recommended)

Details

Although this is often not visible in picture editing programs, almost any image uploaded to Wikimedia contains usefull EXIF metadata. However, it is highly recommended that you insert the some important tags in the image header yourself, by using e.g. the EXIF tool (courtesy of Phil Harvey), before uploading the files to Wikimedia. The `getWikiMedia.ImageInfo` function assumes that all required metadata has already been entered by the user before the upload, hence no further changes in the metadata will be possible. Examples of how to embed EXIF tags into an image file are available here.
To geocode an uploaded image consider adding:

```
{{location|lat deg|lat min|lat sec|NS|long deg|long min|long sec|EW}}
```

tag to the file description, in which case `getWikiMedia.ImageInfo` will automatically look for the attached coordinates via the external links. For practical purposes and because the image properties information determined by the Wikimedia system can are more reliable, the function will rewrite some important EXIF metadata (image width and height) using the actual values determined by Wikimedia server.
For a list of modules and parameters that can be used via `getWikiMedia.ImageInfo`, please refer to Wikimedia API manual.

Author(s)

Tomislav Hengl

References

- Wikimedia API (http://www.mediawiki.org/wiki/API)
- EXIF tool (http://www.sno.phy.queensu.ca/~phil/exiftool/)
- EXIF Tags (http://www.sno.phy.queensu.ca/~phil/exiftool/TagNames/EXIF.html)

See Also

spPhoto, Rexif::getExifPy

Examples

```
## Not run: # Photo taken using a GPS-enabled camera:
imagename = "Africa_Museum_Nijmegen.jpg"
x <- getWikiMedia.ImageInfo(imagename)
# Get the GPS info:
x$metadata[grep(names(x$metadata), pattern="GPS")]
# prints the complete list of metadata tags;

## End(Not run)
```

---

gpxbtour   *GPS log of a bike tour*

Description

GPS log of a bike tour from Wageningen (the Netherlands) to MÃ¼nster (Germany). The table contains 3228 records of GPS locations, speed and elevation.

Usage

```
data(gpxbtour)
```

Format

The data frame contains the following columns:

`lon` longitude (x-coordinate)

`lat` latitude (y-coordinate)

`ele` GPS-estimated elevation in m

`speed` GPS-estimated speed in km per hour

`time` XML Schema time

Note

The log was produced using the GlobalSat GH-615 GPS watch. The original data log (trackpoints) was first saved to GPX exchange format (http://www.topografix.com/gpx.asp) and then imported to R using the XML package and formatted to a data frame.

Author(s)

Tomislav Hengl

Examples

```
## Not run: ## load the data:
data(gpxbtour)
library(sp)
## format the time column:
gpxbtour$ctime <- as.POSIXct(gpxbtour$time, format="%Y-%m-%dT%H:%M:%SZ")
coordinates(gpxbtour) <- ~lon+lat
proj4string(gpxbtour) <- CRS("+proj=longlat +datum=WGS84")
## convert to a STTDF class:
library(spacetime)
library(adehabitatLT)
gpx.ltraj <- as.ltraj(coordinates(gpxbtour), gpxbtour$ctime, id = "th")
gpx.st <- as(gpx.ltraj, "STTDF")
## Google maps plot:
library(RgoogleMaps)
llc <- c(mean(gpx.st@sp@bbox[2,]), mean(gpx.st@sp@bbox[1,]))
MyMap <- GetMap.bbox(center=llc, zoom=8, destfile="map.png")
PlotOnStaticMap(MyMap, lat=gpx.st@sp@coords[,2], lon=gpx.st@sp@coords[,1],
  FUN=lines, col="black", lwd=4)

## End(Not run)
```

`grid2poly` *Converts a gridded map to a polygon map*

Description

Converts a "SpatialGridDataFrame" object to a polygon map with each available grid node represented with a polygon. To allow further export to KML, `grid2poly` will, by default, convert any projected coordinates to the lat-lon system (geographic coordinates in the WGS84 system).

Usage

```
grid2poly(obj, var.name = names(obj)[1], reproject = TRUE,
    method = c("sp", "raster", "RSAGA")[1], tmp.file = TRUE,
    saga_lib = "shapes_grid", saga_module = 3, silent = FALSE, ...)
```

Arguments

`obj`	"SpatialGridDataFrame" object
`var.name`	target variable column name
`reproject`	logical; reproject coordinates to lat lon system?
`method`	decide to convert grids to polygons either using "sp", "raster" or "RSAGA" packages
`tmp.file`	logical; specify whether to create a temporary file, or to actually write to the workding directory (in the case of SAGA GIS is used to convert grids)
`saga_lib`	string; SAGA GIS library name
`saga_module`	SAGA GIS module number; see `?rsaga_get_modules` for more details
`silent`	logical; specifies whether to print the SAGA GIS output
`...`	additional arguments that can be parsed to the `rasterToPolygons` command

Details

grid2poly is not recommended for large grids (»10e4 pixels). Consider splitting large input grids into tiles before running grid2poly. For converting large grids to polygons consider using SAGA GIS (method = "RSAGA") instead of using the default sp method.

Author(s)

Tomislav Hengl

See Also

vect2rast, raster::rasterToPolygons

Examples

```
data(eberg_grid)
library(sp)
coordinates(eberg_grid) <- ~x+y
gridded(eberg_grid) <- TRUE
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
data(SAGA_pal)
## Not run: # compare various methods:
system.time(dem_poly <- grid2poly(eberg_grid, "DEMSRT6", method = "raster"))
system.time(dem_poly <- grid2poly(eberg_grid, "DEMSRT6", method = "sp"))
system.time(dem_poly <- grid2poly(eberg_grid, "DEMSRT6", method = "RSAGA"))
## plotting large polygons in R -> not a good idea
# spplot(dem_poly, col.regions = SAGA_pal[[1]])
## visualize the data in Google Earth:
kml(dem_poly, colour_scale = SAGA_pal[[1]], colour = DEMSRT6, kmz = TRUE)

## End(Not run)
```

HRprec08 — *Daily precipitation for Croatia for year 2008*

Description

The daily measurements of precipitation (rain gauges) for year 2008 kindly contributed by the Croatian National Meteorological Service. HRprec08 contains 175,059 measurements of precipitation sums (489 stations by 365 days).

Usage

```
data(HRprec08)
```

Format

The HRprec08 data frames contain the following columns:

NAME name of the meteorological station

Lon a numeric vector; x-coordiante / longitude in the WGS84 system

Lat a numeric vector; y-coordinate / latitude in the WGS84 system

DATE 'Date' class vector

PREC daily cummulative precipitation in mm (precipitation from the day before)

Note

The precipitation estimates in mm (`HRprec08`) are collected in a bottle within the rain gauge and readings are usually manual by an observer at 7 a.m. The precipitation collected in the morning refer to the precipitation for previous 24 hours. To project coordinates we suggest using the UTM zone 33N system as this coordinate system was used to prepare the gridded predictors.

Author(s)

Tomislav Hengl and Melita Percec Tadic

References

- Testik, F.Y. and Gebremichael, M. Eds (2011) Rainfall: State of the Science. Geophysical monograph series, Vol. 191, 287 p.
- Zaninovic K., Gajic-Capka, M., Percec Tadic, M. et al., (2010) Klimatski atlas Hrvatske / Climate atlas of Croatia 1961-1990., 1971-2000. Zagreb, Croatian National Meteorological Service, 200 p.
- AGGM book datasets (`http://spatial-analyst.net/book/HRclim2008`)

See Also

`HRtemp08`

Examples

```
data(HRprec08)
library(sp)
## Not run: # subset:
prec.2008.05.01 <- HRprec08[HRprec08$DATE=="2008-05-01",]
coordinates(prec.2008.05.01) <- ~Lon+Lat
proj4string(prec.2008.05.01) <- CRS("+proj=lonlat +datum=WGS84")
# write to KML:
shape = "http://plotkml.r-forge.r-project.org/circle.png"
data(SAGA_pal)
kml(prec.2008.05.01, size = PREC, shape = shape, colour = PREC,
    colour_scale = SAGA_pal[[9]], labels = PREC)

## End(Not run)
```

`HRtemp08` *Daily temperatures for Croatia for year 2008*

Description

The daily measurements of temperature (thermometers) for year 2008 kindly contributed by the Croatian National Meteorological Service. `HRtemp08` contains 56,608 measurements of temperature (159 stations by 365 days).

Usage

```
data(HRtemp08)
```

Format

The `HRtemp08` data frames contain the following columns:

`NAME` name of the meteorological station

`Lon` a numeric vector; x-coordiante / longitude in the WGS84 system

`Lat` a numeric vector; y-coordinate / latitude in the WGS84 system

`DATE` 'Date' class vector

`TEMP` daily temperature measurements in degree C

Note

The precision of the temperature readings in `HRtemp08` is tenth of degree C. On most climatological stations temperature is measured three times a day, at 7 a.m., 1 p.m. and 9 p.m. The daily mean can be calculated as a weighted average.

Author(s)

Tomislav Hengl, Melita Percec Tadic and Benedikt Gräler

References

- Hengl, T., Heuvelink, G.B.M., Percec Tadic, M., Pebesma, E., (2011) Spatio-temporal prediction of daily temperatures using time-series of MODIS LST images. Theoretical and Applied Climatology, 107(1-2): 265-277.
- AGGM book datasets (`http://spatial-analyst.net/book/HRclim2008`)

See Also

`HRprec08`

Examples

```
data(HRtemp08)

## Not run:
## examples from: http://dx.doi.org/10.1007/s00704-011-0464-2
library(spacetime)
library(gstat)
library(sp)
sp <- SpatialPoints(HRtemp08[,c("Lon","Lat")])
proj4string(sp) <- CRS("+proj=longlat +datum=WGS84")
HRtemp08.st <- STIDF(sp, time = HRtemp08$DATE-.5,
     data = HRtemp08[,c("NAME","TEMP")],
     endTime = as.POSIXct(HRtemp08$DATE+.5))
## Country borders:
con0 <- url("http://www.gadm.org/data/rda/HRV_adm1.RData")
load(con0)
stplot(HRtemp08.st[,"2008-07-02::2008-07-03","TEMP"],
   na.rm=TRUE, col.regions=SAGA_pal[[1]],
   sp.layout=list("sp.polygons", gadm))

## Load covariates:
con <- url("http://plotkml.r-forge.r-project.org/HRgrid1km.rda")
load(con)
```

```
str(HRgrid1km)
sel.s <- c("HRdem","HRdsea","HRtwi","Lat","Lon")
## Prepare static covariates:
begin <- as.Date("2008-01-01")
endTime <- as.POSIXct(as.Date("2008-12-31"))
sp.grid <- as(HRgrid1km, "SpatialPixels")
HRgrid1km.st0 <- STFDF(sp.grid, time=begin,
    data=HRgrid1km@data[,sel.s], endTime=endTime)
## Prepare dynamic covariates:
sel.d <- which(!names(HRgrid1km) %in% sel.s)
dates <- sapply(names(HRgrid1km)[sel.d],
     function(x){strsplit(x, "LST")[[1]][2]}
)
dates <- as.Date(dates, format="%Y_%m_%d")
## Sort values of MODIS LST bands:
m <- data.frame(MODIS.LST = as.vector(unlist(HRgrid1km@data[,sel.d])))
## >10M values!
## Create an object of type STFDF:
HRgrid1km.stD <- STFDF(sp.grid, time=dates-4, data=m,
     endTime=as.POSIXct(dates+4))

## Overlay in space and time:
HRtemp08.stxy <- spTransform(HRtemp08.st, CRS(proj4string(HRgrid1km)))
ov.s <- over(HRtemp08.stxy, HRgrid1km.st0)
ov.d <- over(HRtemp08.stxy, HRgrid1km.stD)
## Prepare the regression matrix:
regm <- do.call(cbind, list(HRtemp08.stxy@data, ov.s, ov.d))
## Estimate cumulative days:
regm$cday <- floor(unclass(HRtemp08.stxy@endTime)/86400-.5)
str(regm)
## Plot a single station:
scatter.smooth(regm$cday[regm$NAME=="Zavi<c5><be>an"],
    regm$TEMP[regm$NAME=="Zavi<c5><be>an"],
    xlab="Cumulative days",
    ylab="Mean daily temperature (\260C)",
    ylim=c(-12,28), main="GL039 (Zavi\236an)",
    col="grey")
## Run PCA so we can filter missing pixels in the MODIS images:
pca <- prcomp(~HRdem+HRdsea+Lat+Lon+HRtwi+MODIS.LST,
    data=regm, scale.=TRUE)
selc <- c("TEMP","Lon","Lat","cday")
regm.pca <- cbind( regm[-pca$na.action, selc],
    as.data.frame(pca$x))
## Fit a spatio-temporal regression model:
theta <- min(regm.pca$cday)
lm.HRtemp08 <- lm(TEMP~PC1+PC2+PC3+PC4+PC5+PC6
     +cos((cday-theta)*pi/180), data=regm.pca)
summary(lm.HRtemp08)

## Prediction locations -> focus on Istria:
data(LST)
gridded(LST) <- ~lon+lat
proj4string(LST) <- CRS("+proj=longlat +datum=WGS84")
LST.xy <- reproject(LST[1], proj4string(HRgrid1km))
LST.xy <- as(LST.xy, "SpatialPixels")
## targeted dates:
t.dates <- as.Date(c("2008-02-01","2008-05-01","2008-08-01"),
```

```
     format="%Y-%m-%d")
LST.st <- STF(geometry(LST.xy), time=t.dates)
## get values of covariates:
ov.s.IS <- over(LST.st, HRgrid1km.st0)
ov.d.IS <- over(LST.st, HRgrid1km.stD)
LST.stdf <- STFDF(geometry(LST.xy), time=t.dates,
    data=cbind(ov.s.IS, ov.d.IS))
## predict Principal Components:
LST.pca <- as.data.frame(predict(pca, LST.stdf@data))
LST.stdf@data[,paste0("PC",1:6)] <- LST.pca
cday.l <- as.vector(sapply(
    floor(unclass(LST.stdf@endTime)/86400-.5),
    rep, nrow(LST.xy@coords)))
LST.stdf@data[,"cday"] <- cday.l
stplot(LST.stdf[,,"PC1"], col.regions=SAGA_pal[[1]])
stplot(LST.stdf[,,"PC2"], col.regions=SAGA_pal[[1]])

## Predict spatio-temporal regression:
LST.stdf@data[,"TEMP.reg"] <- predict(lm.HRtemp08,
     newdata=LST.stdf@data)
## Plot predictions:
gadm.ll <- as(spTransform(gadm,
     CRS(proj4string(HRgrid1km))), "SpatialLines")
stplot(LST.stdf[,,"TEMP.reg"], col.regions=SAGA_pal[[1]],
  sp.layout=list( list("sp.lines", gadm.ll),
   list("sp.points", HRtemp08.stxy, col="black", pch=19) )
  )

## End(Not run)
```

kml-methods	*Write to a KML file*

Description

Writes any `Spatial*` object (from the sp package) or `Raster*` object (from the raster package) to a KML file via the `plotKML.fileIO` environment. Various *aesthetics* parameters can be set via colour, alpha, size, shape arguments. Their availability depends on the class of the object to plot.

Usage

```
## S4 method for signature 'Raster'
kml(obj, folder.name, file.name, kmz, ...)
## S4 method for signature 'Spatial'
kml(obj, folder.name, file.name, kmz, ...)
## S4 method for signature 'STIDF'
kml(obj, folder.name, file.name, kmz, ...)
## S4 method for signature 'SoilProfileCollection'
kml(obj, folder.name, file.name, kmz, ...)
## S4 method for signature 'SpatialPhotoOverlay'
kml(obj, folder.name, file.name, kmz, ...)
```

Arguments

`obj`	object inheriting from the `Spatial*` or the `Raster*` classes
`folder.name`	character; folder name in the KML file
`file.name`	character; output KML file name
`kmz`	logical; specief whether to compress the KML file
`...`	additional aesthetics arguments (see details below)

Details

To `kml` you can also pass `folder.name`, `file.name` (output file name *.kml), `overwrite` (logical; overwrites the existing file) and `kmz` (logical; specifies whether to compress the kml file) arguments. Gridded objects (objects of class `"SpatialGridDataFrame"` or `"RasterLayer"` require at least one aesthetics parameter to run, usually the `colour`.)

Value

A KML file. By default parses the object name and adds a ".kml" extension.

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

See Also

`kml_open`, `kml_aes`, `kml_close`, `kml_compress`

Examples

```
# Plotting a SpatialPointsDataFrame object
library(rgdal)
data(eberg)
eberg <- eberg[runif(nrow(eberg))<.1,]
library(sp)
library(rgdal)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
## Not run: # Simple plot
kml(eberg, file = "eberg-0.kml")
# Plot using aesthetics
shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png"
kml(eberg, colour = SNDMHT_A, size = CLYMHT_A,
   alpha = 0.75, file = "eberg-1.kml", shape=shape)

## End(Not run)
```

kml.tiles *Write vector object as tiled KML*

Description

Writes vector object as tiled KML. Suitable for plotting large vectors i.e. large spatial data sets.

Usage

```
kml.tiles(obj, folder.name, file.name,
   block.x, kml.logo, cpus, home.url=".", desc=NULL,
   open.kml=TRUE, return.list=FALSE, ...)
```

Arguments

`obj`	`"SpatialPoints*"` or `"SpatialLines*"` or `"SpatialPolygons*"`; vector layer
`folder.name`	character; KML folder name
`file.name`	character; output KML file name
`block.x`	numeric; size of block in decimal degrees (geographical coordinates)
`kml.logo`	character; optional project logo file (PNG)
`cpus`	integer; specifies number of CPUs to be used by the snowfall package to speed things up
`home.url`	character; optional web-directory where the PNGs will be stored
`desc`	character; optional layer description
`open.kml`	logical; specifies whether to open the KML file after writing
`return.list`	logical; specifies whether to return list of tiled objects
`...`	(optional) aesthetics arguments (see `aesthetics`)

Value

Returns a list of KML files.

Note

This operation can be time-consuming for processing very large vectors. To speed up writing of KMLs, use the snowfall package.

Author(s)

Tomislav Hengl

See Also

`plotKML`, `plotKML.GDALobj`

Examples

```
## Not run:
library(sp)
library(snowfall)
library(GSIF)
library(rgdal)

data(eberg)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
## plot using tiles:
shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png"
tiles.p <- kml.tiles(eberg["SNDMHT_A"], block.x=0.05,
   size=0.8, z.lim=c(20,50), colour=SNDMHT_A, shape=shape,
   labels=SNDMHT_A, return.list=TRUE)
## Returns a list of tiles
data(eberg_contours)
tiles.l <- kml.tiles(eberg_contours, block.x=0.05,
   colour=Z, z.lim=range(eberg_contours$Z),
   colour_scale=SAGA_pal[[1]], return.list=TRUE)

## End(Not run)
```

`kml_compress` *Compress a KML file with auxiliary files*

Description

Compresses the KML file toghether with the auxiliary files (images, models, textures) using the default ZIP program.

Usage

```
kml_compress(file.name, zip = "", files = "", rm = FALSE, ...)
```

Arguments

`file.name`	KML file name
`zip`	(optional) location of an external ZIP program
`files`	a character vector specifying the list of auxiliary files
`rm`	logical; specify whether to remove temporary files
`...`	other `kml` arguments

Details

The KMZ file can carry the model files (`.dae`), textures and ground overlay images. For practical purposes, we recommend that you, instead of compressing the images together with the KML file, consider serving the ground overlay images via a server i.e. as network links.
If no internal ZIP program exists, the function looks for the system ZIP program:

```
Sys.getenv("R_ZIPCMD", "zip")
```

External ZIP program can also be specified via the `zip` argument.

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

References

- KMZ description (http://code.google.com/apis/kml/documentation/)

See Also

kml-methods, kml_open

Examples

```
data(eberg)
eberg <- eberg[runif(nrow(eberg))<.1,]
library(sp)
library(rgdal)
library(raster)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
kml_open("eberg.kml")
kml_layer(eberg, colour = CLYMHT_A)
kml_close("eberg.kml")
# compress:
kml_compress("eberg.kml")
```

---

kml_description *Generate a table description from a data frame*

Description

Converts a two-column data frame to a table in HTML format. This can then parsed to a KML file as the layer description.

Usage

```
kml_description(x, iframe = NULL, caption = "Object summary",
                fix.enc = TRUE, cwidth = 150, twidth = 300,
                delim.sign = "_", asText = FALSE)
```

Arguments

x	object of class "data.frame" with two columns
iframe	(optional) iframe content
caption	character; table caption
fix.enc	logical; specify whether to fix encoding
cwidth	numeric; first column width
twidth	numeric; table width
delim.sign	character; delimiter sign
asText	logical; specifies whether to return the formatted table as text or XML

Author(s)

Tomislav Hengl

See Also

kml-methods

---

kml_layer-methods   *Write objects to a KML connection*

Description

Writes any Spatial* object (from the sp package), spatio-temporal object (from the ST-class package) or Raster* object (from the raster package) to a KML file (connection) as a separate layer. Various *aesthetics*, i.e. ways to represent target variables, can be set via colour, transparency, size, width, shape arguments. Their availability depends on the class of the object to plot.

Usage

```
kml_layer(obj, ...)
```

Arguments

obj	object inheriting from the Spatial* or the Raster* classes
...	additional aesthetics arguments; see details for each kml_layer function and the kml_aes function

Value

An XML object that can be further parsed to a KML file (via an open connection).

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

See Also

kml_layer.SpatialPoints, kml_layer.Raster, kml_layer.SpatialLines, kml_layer.SpatialPolygons, kml_layer.STIDF, kml_layer.STTDF, kml_layer.SoilProfileCollection, kml-methods, kml_open, kml_close

Examples

```
library(rgdal)
data(eberg_grid)
library(sp)
library(raster)
gridded(eberg_grid) <- ~x+y
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
data(SAGA_pal)
data(R_pal)
## Not run: # Plot two layers one after the other:
```

```
kml_open("eberg_grids.kml")
kml_layer(eberg_grid, colour=DEMSRT6, colour_scale=R_pal[["terrain_colors"]])
kml_layer(eberg_grid, colour=TWISRT6, colour_scale=SAGA_pal[[1]])
kml_close("eberg_grids.kml")
# print the result:
library(XML)
xmlRoot(xmlTreeParse("eberg_grids.kml"))[["Document"]]

## End(Not run)
```

`kml_layer.Raster` *Writes raster objects to KML*

Description

Writes rasters to PNG images and makes a KML code (ground overlays). Works with `"RasterLayer"` and `"RasterStack"` class objects. Target attributes can be specified using aesthetics arguments (e.g. `"colour"`).

Usage

```
kml_layer.Raster(obj, subfolder.name = paste(class(obj)), plot.legend = TRUE,
     metadata = NULL, raster_name,
     png.width = ncol(obj), png.height = nrow(obj),
     min.png.width = 800, TimeSpan.begin, TimeSpan.end,
     layer.name, png.type, ...)
```

Arguments

`obj`	object of class `"RasterLayer"`, `"SpatialPixelsDataFrame"` or `"SpatialGridDataFrame"`
`subfolder.name`	character; optional subfolder name
`plot.legend`	logical; specify whether a map legend should be generated automatically
`metadata`	(optional) specify the metadata object
`raster_name`	(optional) specify the output file name (PNG)
`png.width`	(optional) width of the PNG file
`png.height`	(optional) height of the PNG file
`min.png.width`	(optional) minimum width of the PNG file
`TimeSpan.begin`	object of class `"POSIXct"`; (optional) begin of the sampling period
`TimeSpan.end`	object of class `"POSIXct"`; (optional) end of the sampling period
`layer.name`	character; optional layer name
`png.type`	character; PNG type
`...`	additional `aesthetics` arguments

Details

Google Earth does not properly handle a 24-bit PNG file which has a single transparent color (read more at Google Earth Help). To force transparency, plotKML will try to set it using the `-matte -transparent "#FFFFFF"` option in the ImageMagick convert program (ImageMagick needs to be installed separately and located using `plotKML.env()`). On some Unix run machines the `png.type` argument has to be set manually to avoid producing empty PNGs.

Author(s)

Tomislav Hengl, Pierre Roudier and Dylan Beaudette

See Also

kml-methods, kml_open, kml_layer.RasterBrick, plotKML-method

Examples

```
data(eberg_grid)
library(sp)
coordinates(eberg_grid) <- ~x+y
gridded(eberg_grid) <- TRUE
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
data(SAGA_pal)
library(raster)
r <- raster(eberg_grid["TWISRT6"])
## Not run: # KML plot with a single raster:
kml(r, colour_scale = SAGA_pal[[1]], colour = TWISRT6)

## End(Not run)
```

---

kml_layer.RasterBrick *Export a time series of images to KML*

Description

Writes a series of images to PNGs and uses them to create ground overlays. Works only with "RasterBrick" class objects with time dimension specified via the "@zvalue".

Usage

```
kml_layer.RasterBrick(obj, plot.legend = TRUE, dtime = "", tz = "GMT",
      z.lim = c(min(minValue(obj), na.rm=TRUE), max(maxValue(obj), na.rm=TRUE)),
          colour_scale = get("colour_scale_numeric", envir = plotKML.opts),
          home_url = get("home_url", envir = plotKML.opts),
          metadata = NULL, html.table = NULL,
          altitudeMode = "clampToGround", balloon = FALSE,
          png.width, png.height, min.png.width = 800, png.type, ...)
```

Arguments

obj	object of class "RasterBrick" (e.g. a time series of images)
plot.legend	logical; specify whether a map legend should be generated automatically
dtime	temporal support (point or block) expressed in seconds
tz	referent time zone
z.lim	upper and lower limits (unique for all maps in the time series); the function by default uses the absolute minimum and maximum in values
colour_scale	color palette; by default uses the color scale for numeric variables
home_url	(optional) URL directory / location of the images

metadata	(optional) the metadata object
html.table	(optional) the description block (html)
altitudeMode	character; the default altitudeMode
balloon	logical; specifies whether to display balloon for each element
png.width	(optional) width of the PNG files
png.height	(optional) height of the PNG files
min.png.width	(optional) minimum width of the PNG file
png.type	character; PNG type
...	additional arguments (see aesthetics)

Details

This method is recommended for visualization of numeric bands representing the same variable i.e. time series of images. To export a stack of images of different type see kml_layer.Raster. If the "@zvalue" slot is empty, dates will be added by subtracting days from the current day with 1–day increments.

Author(s)

Tomislav Hengl

See Also

kml-methods, kml_open, kml_layer.Raster, plotKML-method

---

kml_layer.SoilProfileCollection

Writes a list of soil profiles to KML

Description

Writes object of type "SoilProfileCollection" (a number of soil profiles with site and horizon data) to KML. Several attributes such as horizontal and vertical exaggeration can be passed via arguments.

Usage

```
kml_layer.SoilProfileCollection(obj,
     var.name, var.min = 0, var.scale,
     site_names = profile_id(obj),
     method = c("soil_block", "depth_function")[1],
     block.size = 100,
     color.name, z.scale = 1, x.min, max.depth = 300,
     plot.points = TRUE,
     LabelScale = get("LabelScale", envir = plotKML.opts) * 0.7,
     IconColor = "#ff0000ff",
     shape = paste(get("home_url", envir = plotKML.opts),
         "circlesquare.png", sep = ""),
     outline = TRUE, visibility = TRUE, extrude = TRUE, tessellate = TRUE,
```

```
altitudeMode = "relativeToGround", camera.distance = 0.01,
tilt = 90, heading = 0, roll = 0,
metadata = NULL, html.table = NULL, plot.scalebar = TRUE,
scalebar = paste(get("home_url", envir = plotKML.opts),
    "soilprofile_scalebar.png", sep = ""),
... )
```

Arguments

`obj`	object of class `"SoilProfileCollection"` (package aqp)
`var.name`	target column name in the `horizons` slot
`var.min`	smallest value
`var.scale`	exaggeration in vertical dimension
`site_names`	site names as listed in the site table
`method`	visualization type (soil block or depth-function)
`block.size`	(optional) size of the block of land
`color.name`	(optional) column name carrying the color information for each horizon
`z.scale`	exaggeration in horizontal direction
`x.min`	offset in longitude direction (in decimal degrees)
`max.depth`	maximum height/depht of a profile in cm
`plot.points`	logical; specifies whether to plot horizon centres with attribute values
`LabelScale`	numeric; specifies size of the labels for each horizon
`IconColor`	colors for the labels for each horizon
`shape`	default icon for Google placemarks
`outline`	logical; specifies whether to draw outline for the soil-depth functions (or simply a line)
`visibility`	logical; specifies whether to make the layer visible
`extrude`	logical; specifies whether to extrude horizon centers
`tessellate`	logical; specifies whether to tessellate polygons
`altitudeMode`	by default `relativeToGround`
`camera.distance`	distance from a profile in arc degrees
`tilt`	angle between the direction of the LookAt position and the normal to the surface of the earth
`heading`	orientation towards north
`roll`	rotation about the y axis
`metadata`	(optional) spatial metadata for the input object
`html.table`	(optional) tabular content (attributes) for each horizon
`plot.scalebar`	logical; specifies whether to plot a scale bar next to the profile plot
`scalebar`	default icon for the scale bar
`...`	additional style arguments

Details

Horizon depths are typically expressed in cm, hence the default exaggeration factor (z.scale) is 10. It is highly recommended to turn off the terrain layer in Google Earth, otherwise Google Earth will deform the plots in areas of high relief.

Note

The spatial exaggeration needs to be used because often the detail in the background imagery in Google Earth is limited to a spatial accuracy of 2–20 m, hence there is no point of zooming into objects of size of few meters. These exaggeration factors were selected empirically and will need to be adjusted as the detail in the background imagery increases.

Author(s)

Tomislav Hengl, Dylan Beaudette and Pierre Roudier

References

- Algorithms for Quantitative Pedology (https://CRAN.r-project.org/package=aqp)

See Also

kml_layer.SpatialPhotoOverlay, plotKML-method

Examples

```
## Not run: ## install.packages("aqp", repos="http://R-Forge.R-project.org")
library(aqp)
library(fossil)
library(plyr)
data(ca630)
## Promote to SoilProfileCollection
ca <- join(ca630$lab, ca630$site, type='inner')
depths(ca) <- pedon_key ~ hzn_top + hzn_bot
## extract site data
site(ca) <- ~ mlra + ssa + lon + lat + cntrl_depth_to_top + cntrl_depth_to_bot + sampled_taxon_name
# generate SpatialPoints
library(sp)
coordinates(ca) <- ~ lon + lat
## assign CRS data
proj4string(ca) <- "+proj=longlat +datum=NAD83"
## plot changes in base saturation by sum of cations method (pH 8.2):
kml(ca, method = "depth_function", file.name = "ca_bs_8_2.kml",
  var.name="bs_8.2", balloon = TRUE)
## plot changes in cation exchange capacity by sum of cations method (pH 8.2):
kml(ca, file.name = "ca_CEC8_2.kml", var.name="CEC8.2", IconColor = "#ff009000")
## plot soil profile as 'block':
kml(ca, file.name = "ca_CEC8_2_block.kml", var.name="CEC8.2", balloon = TRUE)

## End(Not run)
```

`kml_layer.SpatialLines` *Writes spatial lines to KML*

Description

Writes object of class "`SpatialLines*`" to KML with a possibility to parse attribute variables using several aesthetics arguments.

Usage

```
kml_layer.SpatialLines(obj, subfolder.name = paste(class(obj)),
  extrude = FALSE, z.scale = 1, metadata = NULL,
  html.table = NULL, TimeSpan.begin = "", TimeSpan.end = "", ...)
```

Arguments

`obj`	object of class "`SpatialLines*`"
`subfolder.name`	character; optional subfolder name
`extrude`	logical; specifies whether to connect the LinearRing to the ground
`z.scale`	vertical exaggeration
`metadata`	(optional) specify the metadata object
`html.table`	optional description block (html) for each GPS point (vertices)
`TimeSpan.begin`	(optional) beginning of the referent time period
`TimeSpan.end`	(optional) end of the referent time period
`...`	additional style arguments (see `aesthetics`)

Details

Only colour and width (aesthetics) are recommended when visualizing `SpatialLines*` objects.
`TimeSpan.begin` and `TimeSpan.end` are optional TimeStamp vectors in the format:
`yyyy-mm-ddThh:mm:sszzzzzz`
Use the same time values for both `TimeSpan.begin` and `TimeSpan.end` if the measurements refer to a single moment in time. `TimeSpan.begin` and `TimeSpan.end` can be either a single value or a vector of values.

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

See Also

`kml-methods`, `kml_open`, `kml_layer.SpatialPolygons`, `plotKML-method`

Examples

```
library(rgdal)
library(sp)
data(eberg_contours)
data(SAGA_pal)
names(eberg_contours)
# KML plot with elevations used as 'colour' argument:
kml(eberg_contours, colour_scale = SAGA_pal[[1]], colour = Z, kmz = TRUE)
```

`kml_layer.SpatialPhotoOverlay` *Exports objects of type SpatialPhotoOverlay to KML*

Description

Writes object of type SpatialPhotoOverlay to KML together with a COLLADA 3D model file (optional).

Usage

```
kml_layer.SpatialPhotoOverlay(obj, method = c("PhotoOverlay", "monolith")[1],
       PhotoOverlay.shape = obj@PhotoOverlay$shape, href = obj@filename,
       coords, dae.name = "", heading = obj@PhotoOverlay$heading,
       tilt = obj@PhotoOverlay$tilt, roll = obj@PhotoOverlay$roll,
       near = obj@PhotoOverlay$near, range =  obj@PhotoOverlay$range,
      leftFov = obj@PhotoOverlay$leftFov, rightFov = obj@PhotoOverlay$rightFov,
      bottomFov = obj@PhotoOverlay$bottomFov, topFov = obj@PhotoOverlay$topFov,
       altitudeMode = "clampToGround", block.size = 100, max.depth = 300,
       scale.x = 1, scale.y = 1, scale.z = 1, refreshMode = "once",
       html.table = NULL, ... )
```

Arguments

`obj`	object of class `"SpatialPhotoOverlay"` (a photograph with spatial coordinates, metadata and orientation)
`method`	visualization type: either `"PhotoOverlay"` or `"monolith"`
`PhotoOverlay.shape`	PhotoOverlay shape value (KML)
`href`	location of the image file
`coords`	(optional) 3D coordinates of the trapesoid corners
`dae.name`	(optional) COLLADA 3D model file name (without the extension)
`heading`	a PhotoOverlay argument; direction (azimuth) of the camera, in degrees
`tilt`	a PhotoOverlay argument; rotation, in degrees, of the camera around the X axis
`roll`	a PhotoOverlay argument; rotation, in degrees, of the camera around the Z axis
`near`	a PhotoOverlay argument; measurement in meters along the viewing direction from the camera viewpoint to the PhotoOverlay shape
`range`	a PhotoOverlay argument; distance in meters from the point specified by <longitude>, <latitude>, and <altitude> to the LookAt position

leftFov	a PhotoOverlay argument; angle, in degrees, between the camera's viewing direction and the left side of the view volume
rightFov	a PhotoOverlay argument; angle, in degrees, between the camera's viewing direction and the right side of the view volume
bottomFov	a PhotoOverlay argument; angle, in degrees, between the camera's viewing direction and the bottom side of the view volume
topFov	a PhotoOverlay argument; angle, in degrees, between the camera's viewing direction and the top side of the view volume
altitudeMode	altitude mode
block.size	width of the block (100 m by default)
max.depth	300 m by default
scale.x	exaggeration in X dimension (COLLADA rectangle)
scale.y	exaggeration in Y dimension (COLLADA rectangle)
scale.z	exaggeration in Z dimension (COLLADA rectangle)
refreshMode	refresh mode for the COLLADA object
html.table	(optional) specify the description block (html) for each point
...	other additional arguments

Details

The default widht and height (100 m and 300 m) were selected based on empirical testing (level of detail in the background imagery in Google Earth). User specified coordinates can be passed via the cords argument. For more info see makeCOLLADA.rectangle.

Author(s)

Tomislav Hengl

References

- KML Reference (http://code.google.com/apis/kml/documentation/kmlreference.html)
- COLLADA Reference (https://www.khronos.org/collada/)

See Also

spPhoto, getWikiMedia.ImageInfo

Examples

```
## Not run: # display spatially referenced photograph in Google Earth:
imagename = "Soil_monolith.jpg"
x1 <- getWikiMedia.ImageInfo(imagename)
sm <- spPhoto(filename = x1$url$url, exif.info = x1$metadata)
kml_open("sm.kml")
kml_layer(sm, method="monolith")
kml_close("sm.kml")
kml_compress("sm.kml", files="Soil_monolith_jpg.dae")

## End(Not run)
```

kml_layer.SpatialPixels
Writes SpatialPixels or SpatialGrid objects to KML

Description

Writes sp classes "SpatialGrid" or "SpatialPixels" to PNG images and makes a KML document (ground overlays). Target attributes can be specified using aesthetics arguments (e.g. "colour").

Usage

```
kml_layer.SpatialPixels(obj, subfolder.name = paste(class(obj)), raster_name,
     plot.legend = TRUE, metadata = NULL,
     png.width = gridparameters(obj)[1,"cells.dim"],
     png.height = gridparameters(obj)[2,"cells.dim"],
     min.png.width = 800, TimeSpan.begin, TimeSpan.end,
     layer.name, png.type, ...)
```

Arguments

obj	object of class "RasterLayer", "SpatialPixelsDataFrame" or "SpatialGridDataFrame"
subfolder.name	character; optional subfolder name
plot.legend	logical; specify whether a map legend should be generated automatically
metadata	(optional) specify the metadata object
raster_name	(optional) specify the output file name (PNG)
png.width	(optional) width of the PNG file
png.height	(optional) height of the PNG file
min.png.width	(optional) minimum width of the PNG file
TimeSpan.begin	object of class "POSIXct"; (optional) begin of the sampling period
TimeSpan.end	object of class "POSIXct"; (optional) end of the sampling period
layer.name	character; optional layer name
png.type	character; PNG type
...	additional aesthetics arguments

Details

Google Earth does not properly handle a 24-bit PNG file which has a single transparent color (read more at Google Earth Help). To force transparency, plotKML will try to set it using the -matte -transparent "#FFFFFF" option in the ImageMagick convert program (ImageMagick needs to be installed separately and located using plotKML.env()). The PNG export uses the 'cairographics', which will never use a palette and normally creates a larger 32-bit ARGB file, but then always allows transparancy. On some Unix run machines the png.type argument has to be set manually to avoid producing empty PNGs.

Author(s)

Tomislav Hengl, Pierre Roudier and Dylan Beaudette

See Also

kml-methods, kml_open, kml_layer.Raster, plotKML-method

Examples

```
data(eberg_grid)
library(sp)
library(rgdal)
library(raster)
coordinates(eberg_grid) <- ~x+y
gridded(eberg_grid) <- TRUE
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
data(SAGA_pal)
## Not run: ## KML plot with a single raster:
kml(eberg_grid, colour_scale = SAGA_pal[[1]], colour = TWISRT6)
## make a larger image:
kml(eberg_grid, colour_scale = SAGA_pal[[1]], colour = TWISRT6,
  png.width = 600, png.height = 600)

## End(Not run)
```

---

kml_layer.SpatialPoints
Writes spatial points to KML

Description

Writes object of class "SpatialPoints*" to KML with a possibility to parse attribute variables using several aesthetics arguments.

Usage

```
kml_layer.SpatialPoints(obj, subfolder.name = paste(class(obj)),
         extrude = TRUE, z.scale = 1,
         LabelScale = get("LabelScale", envir = plotKML.opts),
         metadata = NULL, html.table = NULL, TimeSpan.begin = "",
         TimeSpan.end = "", points_names, ...)
```

Arguments

obj	object of class "SpatialPoints*"
subfolder.name	character; optional subfolder name
extrude	logical; specifies whether to connect the point to the ground with a line
z.scale	numeric; exaggeration in vertical dimension
LabelScale	numeric; scale factor for size of labels
metadata	(optional) specify the metadata object
html.table	(optional) specify the description block (html) for each point
TimeSpan.begin	(optional) beginning of the referent time period
TimeSpan.end	(optional) end of the referent time period
points_names	character; forces the point labels (size of the character vector must equal the number of the points)
...	additional style arguments (see aesthetics)

Details

TimeSpan.begin and TimeSpan.end are optional TimeStamp vectors:

yyyy-mm-ddThh:mm:sszzzzzz

For observations at point support (a single moment in time), use the same time values for both TimeSpan.begin and TimeSpan.end. TimeSpan.begin and TimeSpan.end can be either a single value or a vector of values.
Optional aesthetics arguments are shapes (icons), colour, sizes, altitude (if not a 3D object; variable to be used to specify altitude above ground), altitudeMode (altitude mode type (clampToGround, relativeToGround or absolute). Although this function can be used to plot over five variables, more than three aesthetics arguments is not recommended (e.g. limit to size and colour).

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

See Also

kml_layer.STTDF, plotKML-method

Examples

```
data(eberg)
data(SAGA_pal)
library(sp)
library(rgdal)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
names(eberg)
# subset to 10 percent:
eberg <- eberg[runif(nrow(eberg))<.1,]
## Not run: # plot the measured CLAY content:
kml(eberg, labels = CLYMHT_A)
shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png"
# color only:
kml(eberg, shape = shape, colour = SLTMHT_A, labels = "", colour_scale = SAGA_pal[[1]])
# two variables at the same time:
kml(eberg, shape = shape, size = CLYMHT_A, colour = SLTMHT_A, labels = "")
# two aesthetics elements are effective in emphasizing hot-spots:
kml(eberg, shape = shape, altitude = CLYMHT_A*10, extrude = TRUE,
  colour = CLYMHT_A, labels = CLYMHT_A, kmz = TRUE)

## End(Not run)

## example of how plotKML is programmed:
data(HRtemp08)
HRtemp08[1,]
library(XML)
p1 = newXMLNode("Placemark")
begin <- format(HRtemp08[1,"DATE"]-.5, "%Y-%m-%dT%H:%M:%SZ")
end <- format(HRtemp08[1,"DATE"]+.5, "%Y-%m-%dT%H:%M:%SZ")
txt <- sprintf('<name>%s</name><TimeStamp><begin>%s</begin><end>%s</end></TimeStamp>
```

```
    <Point><coordinates>%.4f,%.4f,%.0f</coordinates></Point>', HRtemp08[1,"NAME"],
    begin, end, HRtemp08[1,"Lon"], HRtemp08[1,"Lat"], 0)
parseXMLAndAdd(txt, parent=p1)
p1
```

---

kml_layer.SpatialPolygons *Writes spatial polygons to KML*

Description

Writes object of class "SpatialPolygons*" to KML with a possibility to parse attribute variables using several aesthetics arguments.

Usage

```
kml_layer.SpatialPolygons(obj, subfolder.name = paste(class(obj)),
        extrude = TRUE, tessellate = FALSE,
        outline = TRUE, plot.labpt = FALSE, z.scale = 1,
        LabelScale = get("LabelScale", envir = plotKML.opts),
        metadata =  NULL, html.table = NULL, TimeSpan.begin = "",
        TimeSpan.end = "", colorMode = "normal", ...)
```

Arguments

obj	object of class "SpatialPolygons*"
subfolder.name	character; optional subfolder name
extrude	logical; specifies whether to connect the point to the ground with a line
tessellate	logical; specifies whether to connect the LinearRing to the ground
outline	logical; specifies whether to outline the polygon
plot.labpt	logical; specifies whether to add the label point (polygon centre)
z.scale	numeric; exaggeration in vertical dimension
LabelScale	numeric; scale factor for size of labels
metadata	(optional) specify the metadata object
html.table	optional description block (html) for each GPS point (vertices)
TimeSpan.begin	(optional) beginning of the referent time period
TimeSpan.end	(optional) end of the referent time period
colorMode	(optional) KML color mode (normal or random)
...	additional style arguments (see aesthetics)

Details

Label points are be default not plotted. We recommend adding the legend to attribute maps instead. Transparency can be set by using the alpha argument.
`TimeSpan.begin` and `TimeSpan.end` are optional TimeStamp vectors:

`yyyy-mm-ddThh:mm:sszzzzzz`

Use the same time values for both `TimeSpan.begin` and `TimeSpan.end` if the measurements refer to a single moment in time. `TimeSpan.begin` and `TimeSpan.end` can be either a single value or a vector of values.

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

See Also

`kml_layer.SpatialLines`, `kml_layer.STIDF`, , `plotKML-method`

Examples

```
library(rgdal)
library(sp)
data(eberg_zones)
names(eberg_zones)
## visualize zones using random colors:
kml(eberg_zones, colorMode = "random")
## with labels:
kml(eberg_zones, colour = ZONES, plot.labpt = TRUE,
   labels = ZONES, kmz = TRUE, balloon=TRUE)
```

`kml_layer.STIDF` *Write irregular spatio-temporal observations (points, lines and polygons) to KML*

Description

Writes an object of class `"STIDF"` (unstructured/irregular spatio-temporal data) to a KML file with a possibility to parse attribute variables using several aesthetics arguments.

Usage

```
kml_layer.STIDF(obj, dtime, ...)
```

Arguments

`obj`	space-time object of class `"STIDF"` (spatio-temporal irregular data frame) or class `"STFDF"` (spatio-temporal full data frame)
`dtime`	temporal support (point or block) expressed in seconds
`...`	additional arguments that can be passed to the `kml_layer.Spatial` method

Details

An object of class `"STIDF"` contains a slot of type `"Spatial*"`, which is parsed via the `kml_layer` method depending on the type of spatial object (points, lines, polygons). The dateTime is defined as:

`yyyy-mm-ddThh:mm:sszzzzzz`

where `T` is the separator between the date and the time, and the time zone is either `Z` (for UTC) or `zzzzzz`, which represents Â±hh:mm in relation to UTC. For more info on how Time Stamps work see `https://developers.google.com/kml/documentation/kml_tut`. If the time is measured at block support, then:

`<TimeStamp><begin> </begin><end> </end></TimeStamp>`

tags will be inserted. Temporal support for any spacetime class, if not specified by the user, is determined as a difference between the `"time"` (indicating begin time) and `"endTime"` slots.

Author(s)

Tomislav Hengl and Benedikt Graeler

References

- Pebesma, E. (2012) Classes and Methods for Spatio-Temporal Data in R. Journal of Statistical Software. 51(7): 1-30.
- spacetime package (`https://CRAN.R-project.org/package=spacetime`)

See Also

`kml_layer.STTDF`, `plotKML-method`

Examples

```
## Not run:
data(HRtemp08)

# format the time column:
HRtemp08$ctime <- as.POSIXct(HRtemp08$DATE, format="%Y-%m-%dT%H:%M:%SZ")

# create a STIDF object:
library(spacetime)
sp <- SpatialPoints(HRtemp08[,c("Lon","Lat")])
proj4string(sp) <- CRS("+proj=longlat +datum=WGS84")
HRtemp08.st <- STIDF(sp, time = HRtemp08$ctime, data = HRtemp08[,c("NAME","TEMP")])

# write to a KML file:
HRtemp08_jan <- HRtemp08.st[1:500]
shape <- "http://maps.google.com/mapfiles/kml/pal2/icon18.png"
kml(HRtemp08_jan, dtime = 24*3600, colour = TEMP, shape = shape, labels = "", kmz=TRUE)

## North Carolina SIDS data set:
library(maptools)
fname <- system.file("shapes/sids.shp", package="maptools")[1]
nc <- readShapePoly(fname, proj4string=CRS("+proj=longlat +datum=NAD27"))
```

```
time <- as.POSIXct(strptime(c(rep("1974-01-01", length(nc)),
   rep("1979-01-01", length(nc))), format="%Y-%m-%d"), tz = "GMT")
data <- data.frame(BIR = c(nc$BIR74, nc$BIR79), NWBIR = c(nc$NWBIR74, nc$NWBIR79),
   SID = c(nc$SID74, nc$SID79))
# copy polygons:
nc.poly <- rep(slot(nc, "polygons"), 2)
# fix the polygon IDs:
for(i in 1:length(row.names(data))) { nc.poly[[i]]@ID = row.names(data)[i] }
sp <- SpatialPolygons(nc.poly, proj4string=CRS("+proj=longlat +datum=NAD27"))
# create a STIDF object:
nct <- STIDF(sp, time = time, data = data)
# write to a KML file:
kml(nct, colour = SID)

## End(Not run)
```

`kml_layer.STTDF` *Write a space-time trajectory to KML*

Description

Writes an object of class `"STTDF"` to a KML file with a possibility to parse attribute variables using several aesthetics arguments.

Usage

```
kml_layer.STTDF(obj, id.name = names(obj@data)[which(names(obj@data)== "burst")],
    dtime, extrude = FALSE,
    start.icon = paste(get("home_url", envir = plotKML.opts),
         "3Dballyellow.png", sep = ""),
    end.icon = paste(get("home_url", envir = plotKML.opts),
          "golfhole.png", sep = ""),
    LabelScale = 0.8 * get("LabelScale", envir = plotKML.opts), z.scale = 1,
    metadata = NULL, html.table = NULL, ... )
```

Arguments

`obj`	space-time object of class `"STTDF"` (spatio-temporal irregular data.frames trajectory)
`id.name`	trajectory ID column name
`dtime`	temporal support size (in seconds)
`extrude`	logical; extrude GPS vertices?
`start.icon`	start icon name (`3Dballyellow.png`)
`end.icon`	destination icon name (`golfhole.png`)
`LabelScale`	the default size of icons
`z.scale`	vertical exaggeration
`metadata`	(optional) specify the metadata object
`html.table`	optional description block (html) for each GPS point (vertices)
`...`	other optional arguments

Details

The dateTime is defined as `yyyy-mm-ddThh:mm:sszzzzzz`, where `T` is the separator between the date and the time, and the time zone is either `Z` (for UTC) or zzzzzz, which represents Â±hh:mm in relation to UTC. For more info on how Time Stamps work see https://developers.google.com/kml/documentation/kml_tut. If the time is measured at block support, then:

<TimeStamp><begin> </begin><end> </end></TimeStamp>

tags will be inserted. Temporal support for any spacetime class, if not specified by the user, is determined as a difference between the `"time"` (indicating begin time) and `"endTime"` slots.

Author(s)

Tomislav Hengl

References

-
- Pebesma, E. (2012) Classes and Methods for Spatio-Temporal Data in R. Journal of Statistical Software. 51(7): 1-30.
- spacetime package (https://CRAN.R-project.org/package=spacetime)

See Also

`readGPX`, `plotKML-method`

`kml_legend.bar` *Generates a legend bar (PNG file)*

Description

Produces a PNG file that can be used as a screen overlay — legend bar for numeric and factor type variables.

Usage

```
kml_legend.bar(x, width, height, pointsize = 14, legend.file, legend.pal,
   z.lim = range(x, na.rm=TRUE, finite=TRUE), factor.labels, png.type = "cairo-png")
```

Arguments

`x`	numeric or factor-type vector
`width`	numeric; (optional) width of image in pixels
`height`	numeric; (optional) height of image in pixels
`pointsize`	numeric; point size for the plot
`legend.file`	PNG file name
`legend.pal`	character; color palette
`z.lim`	numeric; lower and upper limits
`factor.labels`	character; class names if applicable
`png.type`	character; PNG type

Details

When exporting raster layers to KML the legend bar is generated by default. If the width and height are not provided, the function will try to estimate them automatically.

Author(s)

Tomislav Hengl, Pierre Roudier, and Dylan Beaudette

See Also

`grDevices::png`, `kml-methods`, `kml_layer`

`kml_legend.whitening` *Whitening legend (PNG)*

Description

Produces a PNG file that can be used in KML plots (visualization of uncertainty).

Usage

```
kml_legend.whitening(legend.res = 0.01, width = 120, height = 300, pointsize = 14,
                     x.lim, e.lim, leg.asp = 0.3 *  width/height,
                     legend.file = "whitening_legend.png",
                     matte = FALSE, png.type = "cairo-png")
```

Arguments

`legend.res`	numeric; resolution on a 0-1 scale
`width`	integer; image width
`height`	integer; image height
`pointsize`	integer; point size in units for text
`x.lim`	numeric; upper and lower limits for target variable
`e.lim`	numeric; upper and lower limits for the normalized error
`leg.asp`	numeric; legend aspect
`legend.file`	character; output PNG file name
`matte`	logical; specify whether to fix transparency using ImageMagick
`png.type`	character; PNG type

Details

The output PNG file shows a 2D legend with values on the vertical axis and uncertainty on the horizontal axis. Whitening is only valid with Hue-Saturation-Intensity system where Hue's are used to represent values of the target variable, so that the amount of white color can be linearly used to represent uncertainty (i.e. whitening can not be used with different color palettes; or at least we do not recommend this).

Note

Google Earth does not properly handle a 24-bit PNG file which has a single transparent color. In order to force transparency in the output PNG, the function with try using ImageMagick convert function. ImageMagick needs to be installed separately and located using `plotKML.env()`.

Author(s)

Tomislav Hengl

References

- Hengl, T., Heuvelink, G.M.B., Stein, A., (2004) A generic framework for spatial prediction of soil variables based on regression-kriging. Geoderma 122 (1-2): 75-93.
- Hengl, T., (2003) Visualisation of uncertainty using the HSI colour model: computations with colours. 7th International Conference on GeoComputation (CD-ROM), p. 8.

See Also

`whitening`

Examples

```
## Not run: # create the 2D legend for whitening (PNG file):
kml_legend.whitening(x.lim=c(5,20), e.lim=c(.6,1))

## End(Not run)
```

`kml_metadata-methods` *Add metadata table to the active layer*

Description

Adds a selection of metadata to the description box of an active layer.

Usage

```
## S4 method for signature 'SpatialMetadata'
kml_metadata(obj, cwidth = 150, twidth = 500, asText = FALSE)
```

Arguments

`obj`	object of class `"SpatialMetadata"`
`cwidth`	html column width for the field names
`twidth`	html total table width
`asText`	logical; return the output as XML or characters

Details

The kml_metadata function, by default, prints out only a number of selected metadata fields:

1. "Citation_title",
2. "Abstract",
3. "Object_Count",
4. "Beginning_Date",
5. "Ending_Date",
6. "Data_Order_URL",
7. "Other_Citation_Details",
8. "Citation_URL",
9. "Data_Set_Credit",
10. "Data_Distributing_Organization",
11. "Format_Information_Content",
12. "Native_Data_Set_Environment"

See data(mdnames) for a complete list of metadata fields.

Author(s)

Tomislav Hengl

See Also

spMetadata

---

kml_open *Open / close a KML file connection*

Description

Opens a KML file in write mode and initiates the KML header. The same file connection is further accessible by other kml_*() functions such as kml_layer() and kml_close(). kml_View tries to open the produced file using the default application.

Usage

```
kml_open(file.name, folder.name = file.name, kml_open = TRUE,
          kml_visibility = TRUE, overwrite = TRUE, use.Google_gx = FALSE,
          kml_xsd = get("kml_xsd", envir = plotKML.opts),
          xmlns = get("kml_url", envir = plotKML.opts),
          xmlns_gx = get("kml_gx", envir = plotKML.opts))
```

Arguments

`file.name`	KML file name
`folder.name`	character string; KML folder name
`kml_open`	logical; specify whether to open the folder by default
`kml_visibility`	logical; specify whether to make the whole folder visible
`overwrite`	logical; if TRUE, "name" will be overwritten if it exists
`use.Google_gx`	logical; specify whether to use the Google's extended schema
`kml_xsd`	URL of the KML scheme to be used
`xmlns`	URL of the OGC KML standard
`xmlns_gx`	URL of the extended standard
`...`	other arguments

Details

These lower level functions can be used to create customized multi-layered KML files. See plotKML package homepage / manual for more examples.

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

See Also

`plotKML-method`, `kml_layer`, `kml-methods`

`kml_screen` *Add a screen overlay*

Description

Adds an image file (map legend or logo) as screen overlay. The same file connection is further accessible by other `kml_*()` functions such as `kml_layer()` and `kml_close()`. This allows creation of customized multi-layered KML files.

Usage

```
kml_screen(image.file, sname = "",
     position = c("UL","ML","LL","BC","LR","MR","UR","TC")[1],
     overlayXY, screenXY, xyunits = c("fraction", "pixels", "insetPixels")[1],
     rotation = 0, size = c(0,0) )
```

Arguments

`image.file`	image file to be used for screen overlay
`sname`	screen overlay name
`position`	one of the nine standard positions
`overlayXY`	manually specified tie point on the overlay image e.g. `'x="0" y="1"'`
`screenXY`	manually specified matching tie point on the scren e.g. `'x="0" y="1"'`
`xyunits`	values of the XY units ((`"pixels"`, `"fraction"`, or `"insetPixels"`)
`rotation`	(optional) rotation in degrees clock-wise
`size`	size correction in *x* and *y* direction

Details

If nothing else is specified the function looks for some of the nine typical positions: `"UL"` (upper left), `"ML"` (middle left), `"LL"` (lower left), `"BC"` (bottom centre), `"LR"` (lower right), `"MR"` (middle right), `"UR"` (upper right), and `"TC"` (top centre). The *x* and *y* values can be specified in three different ways: as pixels (`"pixels"`), as fractions of the image (`"fraction"`), or as inset pixels (`"insetPixels"`) — an offset in pixels from the upper right corner of the image.

Note

The function, by default, calculates with fractions. If you change the `xyunits` type, all other elements need to be expressed in the same units.

Author(s)

Tomislav Hengl

References

- KML Reference (http://code.google.com/apis/kml/documentation/)

See Also

`kml-methods`

Examples

```
library(rgdal)
library(sp)
data(eberg_zones)
## Not run: # add logo in the top-center:
kml_open("eberg_screen.kml")
kml_layer(eberg_zones)
logo = "http://meta.isric.org/images/ISRIC_right.png"
kml_screen(image.file = logo, position = "TC", sname = "ISRIC logo")
kml_close("eberg_screen.kml")
kml_compress("eberg_screen.kml")

## End(Not run)
```

LST *Time series of MODIS LST images*

Description

LST contains a spatial sub-sample (Istra region in Croatia) of 46 time series of MODIS LST images (estimated Land Surface Temperature in degrees C) at 1 km resolution. The temporal support size of these images is 8-days.

Usage

```
data(LST)
```

Format

The LST data frame contains the following layers:

LST2008_01_01 8-day MODIS LST mosaick for period 2007-12-29 to 2008-01-04

LST2008_01_09 8-day MODIS LST mosaick for period 2008-01-05 to 2008-01-13

... subsequent bands

lon a numeric vector; x-coordinate (m) in the WGS84 system

lat a numeric vector; y-coordinate (m) in the WGS84 system

Note

Time series of 46 day-time and night-time 8-day composite LST images (MOD11A2 product bands 1 and 5) was obtained from the NASA's FTP server (https://ladsweb.modaps.eosdis.nasa.gov/). The original 8-day composite images were created by patching together images from a period of Â±4 days, so that the proportion of clouds can be reduced to a minimum. The "zvalue" slot in the "RasterBrick" object can be used as the dateTime column expressed as:

```
yyyy-mm-ddThh:mm:sszzzzzz
```

where T is the separator between the date and the time, and the time zone is either Z (for UTC) or zzzzzz, which represents Â±hh:mm in relation to UTC.

Author(s)

Tomislav Hengl and Melita Percec Tadic

References

- Hengl, T., Heuvelink, G.B.M., Percec Tadic, M., Pebesma, E., (2011) Spatio-temporal prediction of daily temperatures using time-series of MODIS LST images. Theoretical and Applied Climatology, 107(1-2): 265-277.
- MODIS products (https://lpdaac.usgs.gov/products/modis_products_table)

makeCOLLADA — *Generate a COLLADA file representing the 3D model of a rectangle*

Description

Produces a COLLADA file representing the 3D model of a rectangle with the image specifies via `href` wrapped over the surface (as texture fill). This allows free rotation of any rectangular image in the 3D space.

Usage

```
makeCOLLADA.rectangle(coords, filename, href, DateTime,
          up_axis = "Z_UP", authoring_tool = "plotKML",
          technique_profile = "GOOGLEEARTH",
          double_sided = TRUE)
```

Arguments

`coords`	a matrix defining the rectangle: 4 points with X, Z and Y coordinates (P1 — upper right, P2 — upper left, P3 — lower right, P4 — lower left)
`filename`	output filename with `*.dae` extension
`href`	location of the image used for wrapping (texture fill)
`DateTime`	creation / update time (system time)
`up_axis`	specify which axis is errected
`authoring_tool`	specify authoring tool
`technique_profile`	specify technique profile
`double_sided`	logical; specify whether to drape image on both sides

Details

COLLADA is managed by the nonprofit technology consortium, the Khronos Group. You can also simply drag and drop a COLLADA (`.dae`) file on top of the virtual Earth.

Author(s)

Tomislav Hengl

References

- COLLADA Schema (https://www.khronos.org/collada/)

See Also

`kml_layer.SpatialPhotoOverlay`

Examples

```
## Not run: # image previously uploaded to Wikimedia commons:
imagename = "Soil_monolith.jpg"
x1 <- getWikiMedia.ImageInfo(imagename)
sm <- spPhoto(filename = x1$url$url, exif.info = x1$metadata)
kml(sm, method="monolith")
xmlTreeParse("Soil_monolith_jpg.dae")

## End(Not run)
```

`metadata2SLD-methods` *Methods to create a Styled Layer Description (**SLD**) file*

Description

Creates a Styled Layer Description (**SLD**) file, that can be attached to a spatial layer contributed to GeoServer. It writes the "`sp.pallete`" object (legend entries, titles and colors) to an external file.

Usage

```
## S4 method for signature 'SpatialMetadata'
metadata2SLD(obj, ...)
```

Arguments

`obj`	object of class "`SpatialMetadata`"
`...`	other arguments

Details

The structure of the SLD file is determined by the object class (Point, Polygon, SpatialPixels).

Author(s)

Tomislav Hengl

See Also

`metadata2SLD.SpatialPixels`, `spMetadata`

metadata2SLD.SpatialPixels
*Writes a Styled Layer Description (**SLD**) file*

Description

Writes a Styled Layer Description (**SLD**) file, that can be attached to a spatial layer contributed to GeoServer.

Usage

```
metadata2SLD.SpatialPixels(obj,
    Format_Information_Content = xmlValue(obj@xml[["//formcont"]]),
    obj.name = normalizeFilename(deparse(substitute(obj))),
    sld.file = set.file.extension(obj.name, ".sld"),
    Citation_title = xmlValue(obj@xml[["//title"]]),
    ColorMap_type = "intervals", opacity = 1,
    brw.trg = 'Greys', target.var, ...)
```

Arguments

`obj`	object of class `"SpatialMetadata"`
`Format_Information_Content`	character; class of the object to be written to SLD file
`obj.name`	character; name of the layer
`sld.file`	character; name of the output file
`Citation_title`	character; title of the layer
`ColorMap_type`	character; type of the colorMap see `http://docs.geoserver.org`
`opacity`	logical; specifies the opacity
`brw.trg`	character; color scheme according to www.colorbrewer2.org; default to 'Greys'
`target.var`	character; target variable used to calculate the class-intervals
`...`	additional arguments

Author(s)

Tomislav Hengl

See Also

`spMetadata`

Examples

```
## Not run: # generate missing metadata
data(eberg_grid)
library(sp)
coordinates(eberg_grid) <- ~x+y
gridded(eberg_grid) <- TRUE
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
```

```
# with localy prepared metadata file:
eberg_TWI <- as(eberg_grid["TWISRT6"], "SpatialPixelsDataFrame")
eberg.md <- spMetadata(eberg_TWI, Target_variable="TWISRT6")
# export to SLD format:
metadata2SLD(eberg.md, "eberg_TWI.sld")

## End(Not run)
```

`normalizeFilename` *Normalize filename string*

Description

Remove all reserved characters from the file name.

Usage

```
normalizeFilename(x, form = c("default", "8.3")[1],
                  fix.encoding = TRUE, sub.sign = "_")
```

Arguments

`x`	input character
`form`	target format (standard or the short 8.3 file name)
`fix.encoding`	logical; specifies whether to fix the encoding
`sub.sign`	substitution symbol

Details

This function removes all reserved characters: (less than), (greater than), (colon), (double quote), (forward slash), (backslash), (vertical bar or pipe), (question mark), (asterisk), and empty spaces, from the file name. This is important when writing a list of objects to an external file (e.g. KML) as it prevents from creating erroneous file names.

Author(s)

Tomislav Hengl

See Also

`utils::shortPathName`, `RSAGA:set.file.extension`

Examples

```
normalizeFilename("name[%].txt")
normalizeFilename("name .txt")
```

northcumbria — *Polygon boundary of north Cumbria*

Description

This data set gives the boundary of the county of north Cumbria (UK).

Usage

```
data(northcumbria)
```

Format

A matrix containing (x,y) coordinates of the boundary.

Author(s)

Edith Gabriel <edith.gabriel@univ-avignon.fr>

See Also

fmd for the space-time pattern of food-and-mouth disease in this county in 2001.

plotKML-method — *Methods for plotting results of spatial analysis in Google Earth*

Description

The method writes inputs and outputs of spatial analysis (a list of point, gridded and/or polygon data usually) to KML and opens the KML file in Google Earth (or any other default package used to view KML/KMZ files).

Usage

```
## S4 method for signature 'SpatialPointsDataFrame'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      size, colour, points_names,
      shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png",
    metadata = NULL, kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'SpatialLinesDataFrame'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
    metadata = NULL, kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'SpatialPolygonsDataFrame'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
```

```
      colour, plot.labpt, labels, metadata = NULL,
      kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'SpatialPixelsDataFrame'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      colour, raster_name, metadata = NULL, kmz = FALSE, open.kml = TRUE, ...)
## S4 method for signature 'SpatialGridDataFrame'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      colour, raster_name, metadata = NULL, kmz = FALSE, open.kml = TRUE, ...)
## S4 method for signature 'RasterLayer'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      colour, raster_name, metadata = NULL, kmz = FALSE, open.kml = TRUE, ...)
## S4 method for signature 'SpatialPhotoOverlay'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      dae.name, kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'SoilProfileCollection'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      var.name, metadata = NULL, kmz = get("kmz", envir = plotKML.opts),
      open.kml = TRUE, ...)
## S4 method for signature 'STIDF'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      colour, shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png",
     points_names, kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'STFDF'
plotKML(obj, ...)
## S4 method for signature 'STSDF'
plotKML(obj, ...)
## S4 method for signature 'STTDF'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
     colour, start.icon = "http://maps.google.com/mapfiles/kml/pal2/icon18.png",
      kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'RasterBrickTimeSeries'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      pngwidth = 680, pngheight = 180, pngpointsize = 14,
      kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'RasterBrickSimulations'
plotKML(obj,
```

```
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      obj.summary = TRUE,
      pngwidth = 680, pngheight = 200, pngpointsize = 14,
      kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'SpatialMaxEntOutput'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      html.file = obj@maxent@html,
      iframe.width = 800, iframe.height = 800, pngwidth = 280,
      pngheight = 280, pngpointsize = 14, colour,
      shape = "http://plotkml.r-forge.r-project.org/icon17.png",
      kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE,
     TimeSpan.begin = obj@TimeSpan.begin, TimeSpan.end = obj@TimeSpan.end, ...)
## S4 method for signature 'SpatialPredictions'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""), colour,
      grid2poly = FALSE, obj.summary = TRUE, plot.svar = FALSE,
      pngwidth = 210, pngheight = 580, pngpointsize = 14,
    metadata = NULL, kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'SpatialSamplingPattern'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      colour, kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'SpatialVectorsSimulations'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env = parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""), colour,
      grid2poly = FALSE, obj.summary = TRUE, plot.svar = FALSE,
      kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
## S4 method for signature 'list'
plotKML(obj,
    folder.name = normalizeFilename(deparse(substitute(obj, env=parent.frame()))),
      file.name = paste(folder.name, ".kml", sep=""),
      size = NULL, colour, points_names = "",
      shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png",
      plot.labpt = TRUE, labels = "", metadata = NULL,
      kmz = get("kmz", envir = plotKML.opts), open.kml = TRUE, ...)
```

Arguments

`obj`	input object of specific class; either some sp, or raster or spacetime package class object, or plotKML composite objects containing both inputs and outputs of analysis
`folder.name`	character; folder name in the KML file
`file.name`	character; output KML file name
`size`	for point objects for plotting (see aesthetics)
`colour`	colour variable for plotting (see aesthetics)

`points_names`	vector of characters that can be used as labels
`shape`	character; icons used for plotting (see aesthetics)
`raster_name`	(optional) specify the output file name (PNG)
`var.name`	target variable name (only valid for visualization of `"SoilProfileCollection"` class data
`metadata`	(optional) the metadata object
`plot.labpt`	logical; specifies whether to plot centroids for polygon data
`labels`	character vector; list of labels that will attached to the centroids
`start.icon`	icon for the start position (for trajectory data)
`dae.name`	output DAE file name
`html.file`	specify the location of the html file containing report data (if the input object is of class `"SpatialMaxEntOutput"`)
`iframe.width`	integer; width of the screen for iframe
`iframe.height`	integer; height of the screen for iframe
`TimeSpan.begin`	object of class `"POSIXct"`; begin of the sampling period
`TimeSpan.end`	object of class `"POSIXct"`; end of the sampling period
`pngwidth`	integer; width of the PNG plot (screen image)
`pngheight`	integer; height of the PNG plot (screen image)
`pngpointsize`	integer; text size in the PNG plot (screen image)
`grid2poly`	logical; specifies whether to convert gridded object to polygons
`obj.summary`	logical; specifies whether to print the object summary
`plot.svar`	logical; specifies whether to plot the model uncertainty
`kmz`	logical; specifies whether to compress the output KML file
`open.kml`	logical; specifies whether to directly open the output KML file (i.e. in Google Earth)
`...`	(optional) arguments passed to the lower level functions

Details

This is a generic function to plot various spatial and spatio-temporal R objects that contain both inputs and outputs of spatial analysis. The resulting plots (referred to as *'views'*) are expected to be cartographically complete as they should contain legends, and data and model descriptions. In principle, `plotKML` works with both simple spatial objects, and complex objects such as `"SpatialPredictions"`, `"SpatialVectorsSimulations"`, `"RasterBrickSimulations"`, `"RasterBrickTimeSeries"`, `"SpatialMaxEntOutpu` and similar. To further customize visualizations consider combining the lower level functions `kml_open`, `kml_close`, `kml_compress`, `kml_screen` into your own `plotKML()` method.

All ST-classes are coerced to the STIDF format and hence use the plotKML method for STIDFs.

Note

To prepare a list of objects of class `"SpatialPointsDataFrame"`, `"SpatialLinesDataFrame"`, `"SpatialPolygonsDataFrame"`, or `"SpatialPixelsDataFrame"` consider using the `GSIF::tile` function. Writting large spatial objects via plotKML can be time consuming. Please refer to the package manual for more information.

See Also

SpatialPredictions-class, SpatialVectorsSimulations-class, RasterBrickSimulations-class, RasterBrickTimeSeries-class, SpatialMaxEntOutput-class, SpatialSamplingPattern-class

Examples

```
plotKML.env(kmz = FALSE)
## -------------- SpatialPointsDataFrame --------- ##
library(sp)
library(rgdal)
data(eberg)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
## subset to 20 percent:
eberg <- eberg[runif(nrow(eberg))<.1,]
## Not run: ## bubble type plot:
plotKML(eberg["CLYMHT_A"])
plotKML(eberg["CLYMHT_A"], colour_scale=rep("#FFFF00", 2), points_names="")

## End(Not run)
## plot points with a legend:
shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png"
kml_open("eberg_CLYMHT_A.kml")
kml_layer(eberg["CLYMHT_A"], colour=CLYMHT_A, z.lim=c(20,60),
   colour_scale=SAGA_pal[[1]], shape=shape, points_names="")
kml_legend.bar(x=eberg$CLYMHT_A, legend.file="kml_legend.png",
   legend.pal=SAGA_pal[[1]], z.lim=c(20,60))
kml_screen(image.file="kml_legend.png")
kml_close("eberg_CLYMHT_A.kml")

## -------------- SpatialLinesDataFrame --------- ##
data(eberg_contours)
## Not run:
plotKML(eberg_contours)
## plot contour lines with actual altitudes:
plotKML(eberg_contours, colour=Z, altitude=Z)

## End(Not run)

## -------------- SpatialPolygonsDataFrame --------- ##
data(eberg_zones)
## Not run:
plotKML(eberg_zones["ZONES"])
## add altitude:
zmin = 230
plotKML(eberg_zones["ZONES"], altitude=zmin+runif(length(eberg_zones))*500)

## End(Not run)

## -------------- SpatialPixelsDataFrame --------- ##
library(rgdal)
library(raster)
data(eberg_grid)
gridded(eberg_grid) <- ~x+y
proj4string(eberg_grid) <- CRS("+init=epsg:31467")
TWI <- reproject(eberg_grid["TWISRT6"])
```

```
data(SAGA_pal)
## Not run: ## set limits manually (increase resolution):
plotKML(TWI, colour_scale = SAGA_pal[[1]])
plotKML(TWI, z.lim=c(12,20), colour_scale = SAGA_pal[[1]])

## End(Not run)
## categorical data:
eberg_grid$LNCCOR6 <- as.factor(paste(eberg_grid$LNCCOR6))
levels(eberg_grid$LNCCOR6)
data(worldgrids_pal)
## attr(worldgrids_pal["corine2k"][[1]], "names")
pal = as.character(worldgrids_pal["corine2k"][[1]][c(1,11,13,14,16,17,18)])
LNCCOR6 <- reproject(eberg_grid["LNCCOR6"])
## Not run:
plotKML(LNCCOR6, colour_scale=pal)

## End(Not run)

## -------------- SpatialPhotoOverlay --------- ##
## Not run:
library(RCurl)
imagename = "Soil_monolith.jpg"
urlExists = url.exists("http://commons.wikimedia.org")
if(urlExists){
  x1 <- getWikiMedia.ImageInfo(imagename)
  sm <- spPhoto(filename = x1$url$url, exif.info = x1$metadata)
  # str(sm)
  plotKML(sm)
}

## End(Not run)

## -------------- SoilProfileCollection --------- ##
library(aqp)
library(plyr)
## sample profile from Nigeria:
lon = 3.90; lat = 7.50; id = "ISRIC:NG0017"; FAO1988 = "LXp"
top = c(0, 18, 36, 65, 87, 127)
bottom = c(18, 36, 65, 87, 127, 181)
ORCDRC = c(18.4, 4.4, 3.6, 3.6, 3.2, 1.2)
hue = c("7.5YR", "7.5YR", "2.5YR", "5YR", "5YR", "10YR")
value = c(3, 4, 5, 5, 5, 7); chroma = c(2, 4, 6, 8, 4, 3)
## prepare a SoilProfileCollection:
prof1 <- join(data.frame(id, top, bottom, ORCDRC, hue, value, chroma),
   data.frame(id, lon, lat, FAO1988), type='inner')
prof1$soil_color <- with(prof1, munsell2rgb(hue, value, chroma))
depths(prof1) <- id ~ top + bottom
site(prof1) <- ~ lon + lat + FAO1988
coordinates(prof1) <- ~ lon + lat
proj4string(prof1) <- CRS("+proj=longlat +datum=WGS84")
prof1
## Not run:
plotKML(prof1, var.name="ORCDRC", color.name="soil_color")

## End(Not run)

## -------------- STIDF --------- ##
```

```
library(sp)
library(spacetime)
## daily temperatures for Croatia:
data(HRtemp08)
## format the time column:
HRtemp08$ctime <- as.POSIXct(HRtemp08$DATE, format="%Y-%m-%dT%H:%M:%SZ")
## create a STIDF object:
sp <- SpatialPoints(HRtemp08[,c("Lon","Lat")])
proj4string(sp) <- CRS("+proj=longlat +datum=WGS84")
HRtemp08.st <- STIDF(sp, time = HRtemp08$ctime, data = HRtemp08[,c("NAME","TEMP")])
## subset to first 500 records:
HRtemp08_jan <- HRtemp08.st[1:500]
str(HRtemp08_jan)
## Not run:
plotKML(HRtemp08_jan[,,"TEMP"], LabelScale = .4)

## End(Not run)

## foot-and-mouth disease data:
data(fmd)
fmd0  <- data.frame(fmd)
coordinates(fmd0) <- c("X", "Y")
proj4string(fmd0) <- CRS("+init=epsg:27700")
fmd_sp <- as(fmd0, "SpatialPoints")
dates <- as.Date("2001-02-18")+fmd0$ReportedDay
library(spacetime)
fmd_ST <- STIDF(fmd_sp, dates, data.frame(ReportedDay=fmd0$ReportedDay))
data(SAGA_pal)
## Not run:
plotKML(fmd_ST, colour_scale=SAGA_pal[[1]])

## End(Not run)

## -------------- STFDF --------- ##

## results of krigeST:
library(gstat)
library(sp)
library(spacetime)
library(raster)
## define space-time variogram
sumMetricVgm <- vgmST("sumMetric",
                      space=vgm( 4.4, "Lin", 196.6,  3),
                      time =vgm( 2.2, "Lin",   1.1,  2),
                      joint=vgm(34.6, "Exp", 136.6, 12),
                      stAni=51.7)
## example from the gstat package:
data(air)
rural = STFDF(stations, dates, data.frame(PM10 = as.vector(air)))
rr <- rural[,"2005-06-01/2005-06-03"]
rr <- as(rr,"STSDF")
x1 <- seq(from=6,to=15,by=1)
x2 <- seq(from=48,to=55,by=1)
DE_gridded <- SpatialPoints(cbind(rep(x1,length(x2)), rep(x2,each=length(x1))),
                            proj4string=CRS(proj4string(rr@sp)))
gridded(DE_gridded) <- TRUE
DE_pred <- STF(sp=as(DE_gridded,"SpatialPoints"), time=rr@time)
```

```
DE_kriged <- krigeST(PM10~1, data=rr, newdata=DE_pred,
                     modelList=sumMetricVgm)
gridded(DE_kriged@sp) <- TRUE
stplot(DE_kriged)
## plot in Google Earth:
z.lim = range(DE_kriged@data, na.rm=TRUE)
## Not run:
plotKML(DE_kriged, z.lim=z.lim)
## add observations points:
plotKML(rr, z.lim=z.lim)

## End(Not run)

## -------------- STTDF --------- ##
## Not run:
library(fossil)
library(spacetime)
library(adehabitatLT)
data(gpxbtour)
## format the time column:
gpxbtour$ctime <- as.POSIXct(gpxbtour$time, format="%Y-%m-%dT%H:%M:%SZ")
coordinates(gpxbtour) <- ~lon+lat
proj4string(gpxbtour) <- CRS("+proj=longlat +datum=WGS84")
xy <- as.list(data.frame(t(coordinates(gpxbtour))))
gpxbtour$dist.km <- sapply(xy, function(x) {
  deg.dist(long1=x[1], lat1=x[2], long2=xy[[1]][1], lat2=xy[[1]][2])
} )
## convert to a STTDF class:
gpx.ltraj <- as.ltraj(coordinates(gpxbtour), gpxbtour$ctime, id = "th")
gpx.st <- as(gpx.ltraj, "STTDF")
gpx.st$speed <- gpxbtour$speed
gpx.st@sp@proj4string <- CRS("+proj=longlat +datum=WGS84")
str(gpx.st)
plotKML(gpx.st, colour="speed")

## End(Not run)

## -------------- Spatial Metadata --------- ##
## Not run:
eberg.md <- spMetadata(eberg, xml.file=system.file("eberg.xml", package="plotKML"),
  Target_variable="SNDMHT_A", Citation_title="Ebergotzen profiles")
plotKML(eberg[1:100,"CLYMHT_A"], metadata=eberg.md)

## End(Not run)

## -------------- RasterBrickTimeSeries --------- ##
library(raster)
library(sp)
data(LST)
gridded(LST) <- ~lon+lat
proj4string(LST) <- CRS("+proj=longlat +datum=WGS84")
dates <- sapply(strsplit(names(LST), "LST"), function(x){x[[2]]})
datesf <- format(as.Date(dates, "%Y_%m_%d"), "%Y-%m-%dT%H:%M:%SZ")
## begin / end dates +/- 4 days:
TimeSpan.begin = as.POSIXct(unclass(as.POSIXct(datesf))-4*24*60*60, origin="1970-01-01")
TimeSpan.end = as.POSIXct(unclass(as.POSIXct(datesf))+4*24*60*60, origin="1970-01-01")
## pick climatic stations in the area:
```

```
pnts <- HRtemp08[which(HRtemp08$NAME=="Pazin")[1],]
pnts <- rbind(pnts, HRtemp08[which(HRtemp08$NAME=="Crni Lug - NP Risnjak")[1],])
pnts <- rbind(pnts, HRtemp08[which(HRtemp08$NAME=="Cres")[1],])
coordinates(pnts) <- ~Lon + Lat
proj4string(pnts) <- CRS("+proj=longlat +datum=WGS84")
## get the dates from the file names:
LST_ll <- brick(LST[1:5])
LST_ll@title = "Time series of MODIS Land Surface Temperature images"
LST.ts <- new("RasterBrickTimeSeries", variable = "LST", sampled = pnts,
    rasters = LST_ll, TimeSpan.begin = TimeSpan.begin[1:5],
    TimeSpan.end = TimeSpan.end[1:5])
data(SAGA_pal)
## Not run: ## plot MODIS images in Google Earth:
plotKML(LST.ts, colour_scale=SAGA_pal[[1]])

## End(Not run)

## -------------- Spatial Predictions --------- ##
library(sp)
library(rgdal)
library(gstat)
data(meuse)
coordinates(meuse) <- ~x+y
proj4string(meuse) <- CRS("+init=epsg:28992")
## load grids:
data(meuse.grid)
gridded(meuse.grid) <- ~x+y
proj4string(meuse.grid) <- CRS("+init=epsg:28992")
## Not run: ## fit a model:
library(GSIF)
omm <- fit.gstatModel(observations = meuse, formulaString = om~dist,
   family = gaussian(log), covariates = meuse.grid)
## produce SpatialPredictions:
om.rk <- predict(omm, predictionLocations = meuse.grid)
## plot the whole geostatical mapping project in Google Earth:
plotKML(om.rk, colour_scale = SAGA_pal[[1]])
## plot each cell as polygon:
plotKML(om.rk, colour_scale = SAGA_pal[[1]], grid2poly = TRUE)

## End(Not run)

## -------------- SpatialSamplingPattern --------- ##
## Not run:
library(spcosa)
library(sp)
## read a polygon map:
shpFarmsum <- readOGR(dsn = system.file("maps", package = "spcosa"),
  layer = "farmsum")
## stratify `Farmsum' into 50 strata
myStratification <- stratify(shpFarmsum, nStrata = 50)
## sample two sampling units per stratum
mySamplingPattern <- spsample(myStratification, n = 2)
## attach the correct proj4 string:
library(RCurl)
urlExists = url.exists("http://spatialreference.org/ref/sr-org/6781/proj4/")
if(urlExists){
  nl.rd <- getURL("http://spatialreference.org/ref/sr-org/6781/proj4/")
```

```
  proj4string(mySamplingPattern@sample) <- CRS(nl.rd)
  # prepare spatial domain (polygons):
  sp.domain <- as(myStratification@cells, "SpatialPolygons")
  sp.domain <- SpatialPolygonsDataFrame(sp.domain,
     data.frame(ID=as.factor(myStratification@stratumId)), match.ID = FALSE)
  proj4string(sp.domain) <- CRS(nl.rd)
  # create new object:
  mySamplingPattern.ssp <- new("SpatialSamplingPattern",
     method = class(mySamplingPattern), pattern = mySamplingPattern@sample,
     sp.domain = sp.domain)
  # the same plot now in Google Earth:
  shape = "http://maps.google.com/mapfiles/kml/pal2/icon18.png"
  plotKML(mySamplingPattern.ssp, shape = shape)
}

## End(Not run)

## -------------- RasterBrickSimulations --------- ##
## Not run:
library(sp)
library(gstat)
data(barxyz)
## define the projection system:
prj = "+proj=tmerc +lat_0=0 +lon_0=18 +k=0.9999 +x_0=6500000 +y_0=0
  +ellps=bessel +units=m
  +towgs84=550.499,164.116,475.142,5.80967,2.07902,-11.62386,0.99999445824"
coordinates(barxyz) <- ~x+y
proj4string(barxyz) <- CRS(prj)
data(bargrid)
coordinates(bargrid) <- ~x+y
gridded(bargrid) <- TRUE
proj4string(bargrid) <- CRS(prj)
## fit a variogram and generate simulations:
Z.ovgm <- vgm(psill=1352, model="Mat", range=650, nugget=0, kappa=1.2)
sel <- runif(length(barxyz$Z))<.2
## Note: this operation can be time consuming
sims <- krige(Z~1, barxyz[sel,], bargrid, model=Z.ovgm, nmax=20,
   nsim=10, debug.level=-1)
## specify the cross-section:
t1 <- Line(matrix(c(bargrid@bbox[1,1], bargrid@bbox[1,2], 5073012, 5073012), ncol=2))
transect <- SpatialLines(list(Lines(list(t1), ID="t")), CRS(prj))
## glue to a RasterBrickSimulations object:
library(raster)
bardem_sims <- new("RasterBrickSimulations", variable = "elevations",
  sampled = transect, realizations = brick(sims))
## plot the whole project and open in Google Earth:
data(R_pal)
plotKML(bardem_sims, colour_scale = R_pal[[4]])

## End(Not run)

## -------------- SpatialVectorsSimulations --------- ##
## Not run:
data(barstr)
data(bargrid)
library(sp)
coordinates(bargrid) <- ~ x+y
```

```
gridded(bargrid) <- TRUE
## output topology:
cell.size = bargrid@grid@cellsize[1]
bbox = bargrid@bbox
nrows = round(abs(diff(bbox[1,])/cell.size), 0)
ncols = round(abs(diff(bbox[2,])/cell.size), 0)
gridT = GridTopology(cellcentre.offset=bbox[,1],
  cellsize=c(cell.size,cell.size),
  cells.dim=c(nrows, ncols))
bar_sum <- count.GridTopology(gridT, vectL=barstr[1:5])
## NOTE: this operation can be time consuming!
## plot the whole project and open in Google Earth:
plotKML(bar_sum)

## End(Not run)

## -------------- SpatialMaxEntOutput --------- ##
## Not run:
library(maptools)
library(rgdal)
data(bigfoot)
aea.prj <- "+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=23 +lon_0=-96
   +x_0=0 +y_0=0 +ellps=GRS80 +datum=NAD83 +units=m +no_defs"
data(USAWgrids)
gridded(USAWgrids) <- ~s1+s2
proj4string(USAWgrids) <- CRS(aea.prj)
bbox <- spTransform(USAWgrids, CRS("+proj=longlat +datum=WGS84"))@bbox
sel = bigfoot$Lon > bbox[1,1] & bigfoot$Lon < bbox[1,2] &
    bigfoot$Lat > bbox[2,1] & bigfoot$Lat < bbox[2,2]
bigfoot <- bigfoot[sel,]
coordinates(bigfoot) <- ~Lon+Lat
proj4string(bigfoot) <- CRS("+proj=longlat +datum=WGS84")
library(spatstat)
bigfoot.aea <- as.ppp(spTransform(bigfoot, CRS(aea.prj)))
## Load the covariates:
sel.grids <- c("globedem","nlights03","sdroads","gcarb","twi","globcov")
library(GSIF)
library(dismo)
## run MaxEnt analysis:
jar <- paste(system.file(package="dismo"), "/java/maxent.jar", sep='')
if(file.exists(jar)){
  bigfoot.smo <- MaxEnt(bigfoot.aea, USAWgrids[sel.grids])
  icon = "http://plotkml.r-forge.r-project.org/bigfoot.png"
  data(R_pal)
  plotKML(bigfoot.smo, colour_scale = R_pal[["bpy_colors"]], shape = icon)
}

## End(Not run)
```

`plotKML.env` *plotKML specific environmental variables / paths*

Description

Sets the environmental, package specific parameters and settings (URLs, names, default color palettes and similar) that can be later on passed to other functions.

Usage

```
plotKML.env(colour_scale_numeric = "", colour_scale_factor = "",
        colour_scale_svar = "", ref_CRS, NAflag, icon, LabelScale, size_range,
        license_url, metadata_sel, kmz, kml_xsd, kml_url, kml_gx, gpx_xsd,
        fgdc_xsd, inspire_xsd, convert, gdalwarp, gdal_translate, python,
        home_url, show.env = TRUE, silent = TRUE)
```

Arguments

`colour_scale_numeric`	default colour scheme for numeric variables
`colour_scale_factor`	default colour scheme for factor variables
`colour_scale_svar`	default colour scheme for model error (e.g. mapping error)
`ref_CRS`	the referent CRS proj4string (`"+proj=longlat +datum=WGS84"`)
`NAflag`	the default missing value flag (usually `"-99999"`)
`icon`	the default icon URL
`LabelScale`	the default scale factor for labels
`size_range`	the default size range
`license_url`	the default license URL
`metadata_sel`	a list of the default metadata fields for summary
`kmz`	logical; the default compression setting
`kml_xsd`	the default KML scheme URL
`kml_url`	the default KML format URL
`kml_gx`	the default extended KML scheme URL
`gpx_xsd`	the default GPX scheme URL
`fgdc_xsd`	the default metadata scheme URL
`inspire_xsd`	the default metadata scheme URL
`convert`	a path to ImageMagick convert program
`gdalwarp`	a path to gdalwarp program
`gdal_translate`	a path to gdalwarp program
`python`	a path to Python program
`home_url`	the default location of all icons and auxiliary files
`show.env`	logical; specify whether to print all environmental parameters
`silent`	logical; specify whether to search for paths for external software

Details

The function will try to locate external software tools under either Windows or Unix platform and then save the results to the `plotKML.opts` environment. plotKML-package does not look automatically for software paths (unless you specify this manually in your `"Rprofile.site"`).
The external software tools are not required by default and most of operations in plotKML-package can be run without using them. GDAL, SAGA GIS and Python are highly recommended, however, for processing large data sets. The function `paths` looks for GDAL, ImageMagick, Python, SAGA GIS, in the Windows Registry Hive, the Program Files directory or the usr/bin installation (Unix).

Warning

Under Linux OS you need to install GDAL binaries by using e.g.:
`sudo apt-get install gdal-bin`

Note

To further customize the plotKML options, consider putting:
`library(plotKML); plotKML.env(..., show.env = FALSE)`
in your "`/etc/Rprofile.site`".

Author(s)

Tomislav Hengl, Dylan Beaudette

References

- ImageMagick (http://imagemagick.org)
- GDAL (http://gdal.org)
- SAGA GIS (http://www.saga-gis.org)
- Python (http://python.org)

Examples

```
## Not run: ## look for paths:
library(gdalUtils)
pts <- paths()
pts
plotKML.env(silent = FALSE)
gdalwarp <- get("gdalwarp", envir = plotKML.opts)
## if missing you need to install it!
system(paste(gdalwarp, "--help-general"))
system(paste(gdalwarp, "--formats"), intern = TRUE)

## End(Not run)
plotKML.env(show.env = FALSE)
get("home_url", envir = plotKML.opts)
```

---

plotKML.GDALobj — *Write tiled objects to KML*

Description

Write tiled objects to KML. Suitable for plotting large rasters i.e. large spatial data sets.

Usage

```
plotKML.GDALobj(obj, file.name, block.x, tiles=NULL,
   tiles.sel=NULL, altitude=0, altitudeMode="relativeToGround", colour_scale,
   z.lim=NULL, breaks.lst=NULL, kml.logo, overwrite=TRUE, cpus,
   home.url=".", desc=NULL, open.kml=TRUE, CRS=attr(obj, "projection"),
   plot.legend=TRUE)
```

Arguments

`obj`	`"GDALobj"` object i.e. a pointer to a spatial layer
`file.name`	character; output KML file name
`block.x`	numeric; size of block in meters or corresponding mapping units
`tiles`	data.frame; tiling definition generated using `GSIF::tile`
`tiles.sel`	integer; selection of tiles to be plotted
`altitude`	numeric; altitude of the ground overlay
`altitudeMode`	character; either `"absolute"`, `"relativeToGround"` or `"clampToGround"`
`colour_scale`	character; color palette
`z.lim`	numeric; upper lower boundaries
`breaks.lst`	numeric; optional break lines (must be of size `length(colour_scale)+1`)
`kml.logo`	character; optional project logo file (PNG)
`overwrite`	logical; specifies whether to overwrite PNGs if available
`cpus`	integer; specifies number of CPUs to be used by the snowfall package to speed things up
`home.url`	character; optional web-directory where the PNGs will be stored
`desc`	character; optional layer description
`open.kml`	logical; specifies whether to open the KML file after writing
`CRS`	character; projection string (if missing)
`plot.legend`	logical; indicate whether to plot summary legend

Value

Returns a list of KML files.

Note

This operation can be time-consuming for processing very large rasters e.g. more than 10,000 by 10,000 pixels. To speed up writing of KMLs, use the snowfall package.

Author(s)

Tomislav Hengl

See Also

`plotKML`, `kml.tiles`

Examples

```
## Not run:
library(sp)
library(snowfall)
library(GSIF)
library(rgdal)
fn = system.file("pictures/SP27GTIF.TIF",
 package = "rgdal")
obj <- GDALinfo(fn)
tiles <- getSpatialTiles(obj, block.x=5000,
```

```
  return.SpatialPolygons = FALSE)
## plot using tiles:
plotKML.GDALobj(obj, tiles=tiles, z.lim=c(0,185))
## Even better ideas is to first reproject
## the large grid using 'gdalUtils::gdalwarp', then tile...

## End(Not run)
```

RasterBrickSimulations-class

A class for spatial simulations containing equiprobable gridded features

---

Description

A class containing input and output maps containing multiple realizations of the same feature. Objects of this class can be directly visualized in Google Earth by using the plotKML-method.

Slots

`variable`: character; variable name

`sampled`: object of class "SpatialLines"; one or more lines (cross sections) that can be used to visualize how the values change in space

`realizations`: object of class "RasterBrick"; multiple realizations of the same feature

Methods

plotKML `signature(obj = "RasterBrickSimulations")`: plots all objects in Google Earth

Author(s)

Tomislav Hengl

See Also

SpatialVectorsSimulations-class, RasterBrickTimeSeries-class, plotKML-method

Examples

```
## Not run: # load input data:
data(barxyz)
# define the projection system:
prj = "+proj=tmerc +lat_0=0 +lon_0=18 +k=0.9999 +x_0=6500000 +y_0=0 +ellps=bessel +units=m
+towgs84=550.499,164.116,475.142,5.80967,2.07902,-11.62386,0.99999445824"
library(sp)
coordinates(barxyz) <- ~x+y
proj4string(barxyz) <- CRS(prj)
data(bargrid)
coordinates(bargrid) <- ~x+y
gridded(bargrid) <- TRUE
proj4string(bargrid) <- CRS(prj)
# fit a variogram and generate simulations:
```

```
library(gstat)
Z.ovgm <- vgm(psill=1352, model="Mat", range=650, nugget=0, kappa=1.2)
sel <- runif(length(barxyz$Z))<.2  # Note: this operation can be time consuming
sims <- krige(Z~1, barxyz[sel,], bargrid, model=Z.ovgm, nmax=20, nsim=10, debug.level=-1)
# specify the cross-section:
t1 <- Line(matrix(c(bargrid@bbox[1,1],bargrid@bbox[1,2],5073012,5073012), ncol=2))
transect <- SpatialLines(list(Lines(list(t1), ID="t")), CRS(prj))
# glue to a RasterBrickSimulations object:
bardem_sims <- new("RasterBrickSimulations", variable = "elevations",
  sampled = transect, realizations = brick(sims))
# plot the whole project and open in Google Earth:
data(R_pal)
plotKML(bardem_sims, colour_scale = R_pal[[4]])

## End(Not run)
```

RasterBrickTimeSeries-class

A class for a time series of regular grids

Description

A class containing list of rasters, begin, end times and sample points to allow exploration of the values. Objects of this class can be directly visualized in Google Earth by using the `plotKML-method`.

Slots

`variable`: object of class `"character"`; variable name

`sampled`: object of class `"SpatialPoints"`; one or more points that can be used to visualize temporal changes in the target variable

`rasters`: object of class `"RasterBrick"`; a time-series of raster objects

`TimeSpan.begin`: object of class `"POSIXct"`; begin of sampling for each raster map

`TimeSpan.end`: object of class `"POSIXct"`; end of sampling for each raster map

Methods

plotKML `signature(obj = "RasterBrickTimeSeries")`: plots time-series of rasters in Google Earth

Author(s)

Tomislav Hengl

See Also

`RasterBrickSimulations-class`, `plotKML-method` `readGPX` *Import GPX (GPS track) files*

Description

Reads various elements from a *.gpx file — metadata, waypoints, tracks and routes — and converts them to dataframes.

Usage

```
readGPX(gpx.file, metadata = TRUE, bounds = TRUE,
              waypoints = TRUE, tracks = TRUE, routes = TRUE)
```

Arguments

`gpx.file`	location of the gpx.file
`metadata`	logical; species whether the metadata should be imported
`bounds`	logical; species whether the bounding box coordinates should be imported
`waypoints`	logical; species whether all waypoints should be imported
`tracks`	logical; species whether all tracks should be imported
`routes`	logical; species whether all routes should be imported

Details

Waypoint is a point of interest, or named feature on a map. **Track** is an ordered list of points describing a path. **Route** is an ordered list of waypoints representing a series of turn points leading to a destination.

Author(s)

Tomislav Hengl

References

- GPX data format (http://www.topografix.com/gpx.asp)
- XML tutorial (https://github.com/omegahat/XML)

See Also

`rgdal::readOGR`, `kml_layer.STTDF`

Examples

```
## Not run: # read GPX file from web:
fells_loop <- readGPX("http://www.topografix.com/fells_loop.gpx")
str(fells_loop)

## End(Not run)
```

`readKML.GBIFdensity` *Imports GBIF cell density records*

Description

Read GBIF cell (1–degree) density record counts and converts them to a `"raster"` object.

Usage

```
readKML.GBIFdensity(kml.file, gbif.url = FALSE, silent = FALSE)
```

Arguments

`kml.file`	GBIF cell density file (local file or URL)
`gbif.url`	logical; species whether the cellid and taxon content information should be also imported (usually not used)
`silent`	logical; species whether the progress bar should be printed

Details

This document contains data shared through the GBIF Network — see http://www.gbif.org/occurrence for more information. GBIF records are constantly updated and every map derived refers to a certain date indicated in the `@zname` *Last update* slot.
All usage of these data must be in accordance with the GBIF Data Use Agreement: https://www.gbif.org/terms.

Author(s)

Tomislav Hengl

References

- GBIF cell density description (http://www.gbif.org/occurrence)

See Also

`readGPX`

Examples

```
## Not run: # reading taxon density maps:
kml.file <- "taxon-celldensity-2294100.kml"
# download.file(paste("http://data.gbif.org/occurrences/taxon/celldensity/", kml.file, sep=""),
# destfile=paste(getwd(), kml.file, sep=""))
# this will not run (you must first accept the data usage agreeent);
# instead, obtain the kml file via a web browser, and save it to the working directory:
r <- readKML.GBIFdensity(kml.file)
class(r)
summary(r)
image(r)
# add world borders:
library(maps)
```

```
country.m = map('world', plot=FALSE, fill=TRUE)
IDs <- sapply(strsplit(country.m$names, ":"), function(x) x[1])
library(maptools)
country <- as(map2SpatialPolygons(country.m, IDs=IDs), "SpatialLines")
lines(country)
# to import a list of files, use e.g.:
kml.list <- list(kml.file)
r.lst <- lapply(kml.list, readKML.GBIFdensity, silent = TRUE)
# mask out missing layers (empty KML files):
mask <- !sapply(r.lst, is.null)
r.lst <- brick(r.lst[mask])

## End(Not run)
```

`reproject` *Methods to reproject maps to a referent coordinate system (WGS84)*

Description

This wrapper function reprojects any vector or raster spatial data to some referent coordinate system (by default: geographic coordinates on the World Geodetic System of 1984 / WGS84 datum).

Usage

```
## S4 method for signature 'SpatialPoints'
reproject(obj, CRS, ...)
## S4 method for signature 'SpatialPolygons'
reproject(obj, CRS, ...)
## S4 method for signature 'SpatialLines'
reproject(obj, CRS, ...)
## S4 method for signature 'RasterLayer'
reproject(obj, CRS, program = "raster", tmp.file = TRUE,
      NAflag, show.output.on.console = FALSE, method, ...)
## S4 method for signature 'SpatialGridDataFrame'
reproject(obj, CRS, tmp.file = TRUE, program = "raster",
      NAflag, show.output.on.console = FALSE, ...)
## S4 method for signature 'SpatialPixelsDataFrame'
reproject(obj, CRS, tmp.file = TRUE, program = "raster",
      NAflag, show.output.on.console = FALSE, ...)
## S4 method for signature 'RasterBrick'
reproject(obj, CRS)
## S4 method for signature 'RasterStack'
reproject(obj, CRS)
```

Arguments

`obj`	Spatial* or Raster* object
`CRS`	object of class `"CRS"`; proj4 string
`program`	reprojection engine; either raster package or GDAL
`tmp.file`	logical; specifies whether to create a temporary file or not
`NAflag`	character; missing value flag

`show.output.on.console`
: logical; specifies whether to print the progress

`method`
: character; resampling method e.g."bilinear"

`...`
: arguments evaluated in the context of function `projectRaster` from the raster package

Details

In the case of raster and/or gridded maps, by selecting `program = "GDAL"` `gdalwarp` functionality will be initiated (otherwise it tries to reproject via the package raster). This requires that GDAL are installed and located from R via `paths()`.

Warning

`obj` needs to have a proper proj4 string (CRS), otherwise `reproject` will not run.

Author(s)

Pierre Roudier, Tomislav Hengl and Dylan Beaudette

References

- Raster package (https://CRAN.R-project.org/package=raster)
- GDAL (http://GDAL.org)

See Also

paths, projectRaster, spTransform, CRS-class

Examples

```
## example with vector data:
data(eberg)
library(sp)
library(rgdal)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
eberg.geo <- reproject(eberg)
## Not run: ## example with raster data:
data(eberg_grid25)
gridded(eberg_grid25) <- ~x+y
proj4string(eberg_grid25) <- CRS("+init=epsg:31467")
## reproject to geographical coords (can take few minutes!):
eberg_grid_ll <- reproject(eberg_grid25[1])
## much faster when using GDAL:
eberg_grid_ll2 <- reproject(eberg_grid25[1], program = "GDAL")
## optional: compare processing times:
system.time(eberg_grid_ll <- reproject(eberg_grid25[1]))
system.time(eberg_grid_ll2 <- reproject(eberg_grid25[1], program="GDAL"))

## End(Not run)
```

SAGA_pal *Colour palettes for numeric variables*

Description

SAGA_pal contains 22 colour palettes imported from SAGA GIS (Conrad, 2007). R_pal 12 standard colour palettes used in R to visualize continuous and binary variables. Each colour palette consists of 20 colours in the hexadecimal system. Use display.pal function to plot different sets of palettes.

Usage

```
data(SAGA_pal)
data(R_pal)
```

Note

rainbow_75, heat colors, terrain_colors, topo_colors, and bpy_colors are the standard color palettes used in R to visualize numeric/continuous variables. soc_pal, pH_pal, tex_pal, BS_pal and CEC_pal palettes are suitable for visualization of soil variables (soil organic carbon, pH, soil texture fractions, Base Saturation and Cation Exchange Capacity). blue_grey_red palette is recommended for visualization of binary variables (values in the range 0-1), and grey_black is a white-to-black type color palette that contains no white color (hence it will not confuse low values with NA values in the PNG/GIF files).
Possibly the most used palettes for visualization of numeric variables are rev(rainbow(65)[1:48]) and SAGA_pal[[1]] (the SAGA GIS default palette). It is however worth mentioning that in the data visualization literature (and the cartography literature in particular), the rainbow (sometimes also called spectral) color ramp is generally recognized as a *bad choice* for visualization of sequential/continuous variables (Rogowitz and Treinish, 1998; Borland and Russell, 2007).

Author(s)

SAGA GIS has been created by the SAGA GIS development team (lead by J. BÃ¶hner and O. Conrad, from the Institute of Geography, University of Hamburg, Germany). The colour palettes have been exported from SAGA (as ".sprm" SAGA parameter files) and ported to R. All palletes described here were prepared for R by Tomislav Hengl (<tom.hengl@opengeohub.org>).

References

- Conrad, O., (2007). SAGA — Entwurf, Funktionsumfang und Anwendung eines Systems fÃ¼r Automatisierte Geowissenschaftliche Analysen. Electronic doctoral dissertation, University of GÃ¶ttingen.
- Rogowitz, B.E., Treinish, L.A., (1998, December). Data visualization: the end of the rainbow. Spectrum, IEEE, 35(12):52-59
- Borland, D. and Russell, M. T. II, (2007). Rainbow Color Map (Still) Considered Harmful. Computer Graphics and Applications, IEEE, 27(2):14-17.
- https://cran.r-project.org/package=RColorBrewer
- https://cran.r-project.org/package=colorspace

See Also

worldgrids_pal, RColorBrewer::display.brewer.all

Examples

```
data(SAGA_pal)
data(R_pal)
## Not run: # visualize SAGA GIS palettes:
display.pal(pal=SAGA_pal, sel=c(1,2,7,8,10,11,17,18,19,21,22))
dev.off()
display.pal(R_pal)
names(R_pal)
dev.off()

## End(Not run)
```

`sp.palette-class` *A class for color palette*

Description

A class for color palette that can be further used to create an object of class "SpatialMetadata".

Slots

bounds: object of class "numeric" or "character"; class boundaries

color: object of class "character"; contains HEX colors

icons: object of class "character"; (optional) contains symbols or URI to icons

names: object of class "character"; class names (optional)

type: object of class "character"; variable type

Note

Size of class boundaries (upper and lower) is 1 element larger then the size of colors and element names.

Author(s)

Tomislav Hengl

See Also

spMetadata, metadata2SLD-methods

`SpatialMaxEntOutput-class` *A class for outputs of analysis produced using the dismo package (MaxEnt)*

Description

A class containing input and output data produced by running the `maxent` (Maximum Entropy) species distribution modeling algorithm. Object of this class can be directly visualized in Google Earth by using the `plotKML-method`.

Slots

`sciname`**:** object of class `"character"`; vector of species name compatible with the `rgbif` package; usually latin "genus" and "species" name

`occurrences`**:** object of class `"SpatialPoints"`; occurrence-only records

`TimeSpan.begin`**:** object of class `"POSIXct"`; begin of the sampling period

`TimeSpan.end`**:** object of class `"POSIXct"`; end of the sampling period

`maxent`**:** object of class `"MaxEnt"` (species distribution model); produced as an output of the `dismo::maxent` function or similar

`sp.domain`**:** object of class `"Spatial"` (ideally `"SpatialPolygonsDataFrame"` or `"SpatialPixelsDataFrame"`); assumed spatial domain that can be set by the user or it will be estimated by MaxEnt (see examples below)

`predicted`**:** object of class `"RasterLayer"`; contains results of prediction produced using the MaxEnt software

Methods

plotKML `signature(obj = "SpatialMaxEntOutput")`: plots all MaxEnt output objects in Google Earth

Note

MaxEnt requires the `maxent.jar` file to be in the 'java' folder of the dismo package (see: `system.file("java", packag`
For more info refer to the dismo package documentation. Alternatively use the `maxlike` package (Royle et al. 2012), which does not require Java.

Author(s)

Tomislav Hengl

References

- Hijmans, R.J, Elith, J., (2012) Species distribution modeling with R. CRAN, Vignette for the dismo package, 72 p.
- Royle, J.A., Chandler, R.B., Yackulic, C. and J. D. Nichols. (2012) Likelihood analysis of species occurrence probability from presence-only data for modelling species distributions. Methods in Ecology and Evolution.
- dismo package (`https://CRAN.R-project.org/package=dismo`)
- maxlike package (`https://CRAN.R-project.org/package=maxlike`)
- rgbif package (`https://CRAN.R-project.org/package=rgbif`)

See Also

plotKML-method, dismo::maxent, maxlike::maxlike, rgbif::taxonsearch

---

SpatialMetadata-class *A class for spatial metadata*

Description

A class containing spatial metadata in the Federal Geographic Data Committee (FGDC) Content Standard for Digital Geospatial Metadata.

Slots

`xml`: object of class "XMLInternalDocument"; a metadata slot

`field.names`: object of class "character"; corresponding metadata column names

`palette`: object of class "sp.palette"; contains legend names and colors

`sp`: object of class "Spatial"; bounding box and projection system of the input object

Methods

summary `signature(obj = "SpatialMetadata")`: summarize object

GetPalette `signature(obj = "SpatialMetadata")`: get only the color slot

GetNames `signature(obj = "SpatialMetadata")`: get metadata field names

Author(s)

Tomislav Hengl and Michael Blaschek

See Also

spMetadata, metadata2SLD-methods

Examples

```
## Not run:
data(eberg)
library(sp)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
names(eberg)
# add metadata:
eberg.md <- spMetadata(eberg, xml.file=system.file("eberg.xml", package="plotKML"),
    Target_variable="SNDMHT_A")
p <- GetPalette(eberg.md)
str(p)
x <- summary(eberg.md)
str(x)

## End(Not run)
```

`SpatialPhotoOverlay-class` *A class for Spatial PhotoOverlay*

Description

A class for spatial photographs (spatially and geometrically defined) that can be plotted in Google Earth.

Slots

`filename` object of class "character"; URI of the filename location (typically a URL)

`pixmap` object of class "pixmapRGB"; RGB bands of a bitmapped images

`exif.info` object of class "list"; EXIF photo metadata

`PhotoOverlay` object of class "list"; list of the camera geometry parameters (KML specifications)

`sp` object of class "SpatialPoints"; location of the camera

Extends

Class `"pixmapRGB"`.

Methods

summary `signature(obj = "SpatialMetadata")`: summarize object

Author(s)

Tomislav Hengl

See Also

`plotKML-method`, `spPhoto`

`SpatialPredictions-class` *A class for spatial predictions produced using* `gstat` *package*

Description

A class containing input and output maps generated through the process of geostatistical mapping. Object of this class can be directly visualized in Google Earth by using the `plotKML-method`.

Slots

`variable`: object of class `"character"`; variable name

`observed`: object of class `"SpatialPointsDataFrame"` (must be 2D); see `sp::SpatialPointsDataFrame`

`regModel.summary`: contains the summary of the regression model

`vgmModel`: object of class `"data.frame"`; contains the variogram parameters passed from gstat

`predicted`: object of class `"SpatialPixelsDataFrame"`; see `sp::SpatialPixelsDataFrame`

`validation`: object of class `"SpatialPointsDataFrame"` containing results of validation

Methods

plot `signature(x = "SpatialPredictions")`: spatial predictions, regression model (observed vs predicted), original variogram and variogram for residuals

plotKML `signature(obj = "SpatialPredictions")`: plots all objects in Google Earth

summary `signature(obj = "SpatialPredictions")`: summarize object by showing the mapping accuracy (cross-validation) and the amount of variation explained by the model

Note

`"SpatialPredictions"` saves results of predictions for a single target variable, which can be of type numeric or factor. Multiple variables can be combined into a list.

Author(s)

Tomislav Hengl

References

- Hengl, T. (2009) A Practical Guide to Geostatistical Mapping, 2nd Edt. University of Amsterdam, www.lulu.com, 291 p.
- Hengl, T., Nikolic, M., MacMillan, R.A., (2012) Mapping efficiency and information content. International Journal of Applied Earth Observation and Geoinformation, special issue Spatial Statistics Conference.

See Also

`plotKML-method`, `GSIF::fit.gstatModel`, `gstat::gstat-class`, `RasterBrickSimulations-class`

`SpatialSamplingPattern-class` *A class for spatial samples produced using various* `spsample` *methods*

Description

A class containing input and output objects generated by some sampling optimisation algorithm. Objects of this type can be directly visualized in Google Earth by using the `plotKML-method`.

Slots

`method`: object of class `"character"`; sampling optimisation method

`pattern`: object of class `"SpatialPoints"`; sampling points

`sp.domain`: object of class `"SpatialPolygonsDataFrame"`; spatial domain / strata

Methods

plotKML `signature(obj = "SpatialSamplingPattern")`: plots generated sampling plan in Google Earth

Author(s)

Tomislav Hengl

See Also

plotKML-method, spcosa::spsample, plotKML-method

SpatialVectorsSimulations-class
A class for spatial simulations containing equiprobable line, point or polygon features

---

Description

A class containing input and output maps generated as equiprobable simulations of the same discrete object (for example multiple realizations of stream networks). Objects of this type can be directly visualized in Google Earth by using the plotKML-method.

Slots

`realizations`: object of class "list"; multiple realizations of the same feature e.g. multiple realizations of stream network

`summaries`: object of class "SpatialGridDataFrame"; summary measures

Methods

plotKML `signature(obj = "SpatialVectorsSimulations")`: plots simulated vector objects and summaries (grids) in Google Earth

Author(s)

Tomislav Hengl

See Also

RasterBrickSimulations-class, plotKML-method

Examples

```
## load a list of equiprobable streams:
data(barstr)
data(bargrid)
library(sp)
coordinates(bargrid) <- ~ x+y
gridded(bargrid) <- TRUE
## output topology:
cell.size = bargrid@grid@cellsize[1]
bbox = bargrid@bbox
nrows = round(abs(diff(bbox[1,])/cell.size), 0)
ncols = round(abs(diff(bbox[2,])/cell.size), 0)
gridT = GridTopology(cellcentre.offset=bbox[,1], cellsize=c(cell.size,cell.size),
  cells.dim=c(nrows, ncols))
## Not run: ## derive summaries (observed frequency and the entropy or error):
bar_sum <- count.GridTopology(gridT, vectL=barstr[1:5])
## NOTE: this operation can be time consuming!
```

```
## plot the whole project and open in Google Earth:
plotKML(bar_sum, grid2poly = TRUE)

## End(Not run)
```

spMetadata-methods *Methods to generate spatial metadata*

Description

The spMetadata function will try to generate missing metadata (bounding box, location info, session info, metadata creator info and similar) for any Spatial* object (from the sp package) or Raster* object (from the raster package). The resulting object of class SpatialMetadata-class can be used e.g. to generate a Layer description documents (<description> tag).
The read.metadata function reads the formatted metadata (.xml), prepared following e.g. the Federal Geographic Data Committee (FGDC) Content Standard for Digital Geospatial Metadata or INSPIRE standard, and converts them to a data frame.

Usage

```
## S4 method for signature 'RasterLayer'
spMetadata(obj, bounds, color, ... )
## S4 method for signature 'Spatial'
spMetadata(obj, xml.file, out.xml.file,
    md.type = c("FGDC", "INSPIRE")[1],
    generate.missing = TRUE, GoogleGeocode = FALSE,
    signif.digit = 3, colour_scale, color = NULL, bounds,
    legend_names, icons, validate.schema = FALSE, ...)
```

Arguments

obj	some "Spatial" or "Raster" class object with "data" slot
xml.file	character; optional input XML metadata file
out.xml.file	character; optional output XML metadata file
md.type	character; metadata standard FGDC or INSPIRE
generate.missing	logical; specifies whether to automatically generate missing fields
GoogleGeocode	logical; specifies whether the function should try to use GoogleGeocoding functionality to determine the location name
signif.digit	integer; the default number of significant digits (in the case of rounding)
colour_scale	the color scheme used to visualize this data
color	character; list of colors (rgb()) that can be passed instead of using the pallete
bounds	numeric vector; upper and lower bounds used for visualization
legend_names	character; legend names in the order of bounds
icons	character; file name or URL used for icons (if applicable)
validate.schema	logical; specifies whether to validate the schema using the xmlSchemaValidate
...	additional arguments to be passed e.g. via the metadata.env()

Details

`spMetadata` tries to locate a metadata file in the working directory (it looks for a metadata file with the same name as the object name). If no `.xml` file exists, it will load the template xml file available in the system folder (e.g. `system.file("FGDC.xml", package="plotKML")` or `system.file("INSPIRE_ISO19139.xml", package="plotKML")`). The `FGDC.xml/INSPIRE_ISO19139.xml` files contain typical metadata entries with description and examples. For practical purposes, one metadata object in plotKML can be associated with only one variable i.e. one column in the `"data"` slot (the first column by default). To prepare a metadata xml file following the FGDC standard, consider using e.g. the Tkme software: Another editor for formal metadata, by Peter N. Schweitzer (U.S. Geological Survey). Before commiting the metadata file, try also running a validation test. Before committing the metadata file following the INSPIRE standard, try running the INSPIRE Geoportal Metadata Validator.

`spMetadata` tries to automatically generate the most usefull information, so that a user can easily find out about the input data and procedures followed to generate the visualization (KML). Typical metadata entries include e.g. (FGDC):

- `metadata[["idinfo"]][["native"]]` — Session info e.g.: Produced using R version 2.12.2 (2011-02-25) running on Windows 7 x64.
- `metadata[["spdoinfo"]][["indspref"]]` — Indirect spatial reference estimated using the Google Maps API Web Services.
- `metadata[["idinfo"]][["spdom"]][["bounding"]]` — Bounding box in the WGS84 geographical coordinates estimated by reprojecting the original bounding box.

and for INSPIRE metadata:

- `metadata[["fileIdentifier"]][["CharacterString"]]` — Metadata file identifier (not mandatory for INSPIRE-compl.) created by UUIDgenerate from package UUID (version 4 UUID).
- `metadata[["dateStamp"]][["Date"]]` — Metadata date stamp created using Sys.Date().
- `metadata[["identificationInfo"]][["MD_DataIdentification"]] [["extent"]][["EX_Extent"]][["ge` — Bounding box in the WGS84 geographical coordinates estimated by reprojecting the original bounding box.

By default, plotKML uses the Creative Commons license, but this can be adjusted by setting the `Use_Constraints` argument.

Author(s)

Tomislav Hengl and Michael Blaschek

References

- The Federal Geographic Data Committee, (2006) FGDC Don't Duck Metadata — A short reference guide for writing quality metadata. Vers. 1, `http://www.fgdc.gov/metadata/documents/MetadataQuickGuide.pdf`
- Content Standard for Digital Geospatial Metadata (`http://www.fgdc.gov/metadata/csdgm/`)
- Tkme metadata editor (`http://geology.usgs.gov/tools/metadata/tools/doc/tkme.html`)
- INSPIRE, INS MD, Commission Regulation (EC) No 1205/2008 of 3 December 2008 implementing Directive 2007/2/EC of the European Parliament and of the Council as regards metadata (Text with EEA relevance). See also Corrigendum to INSPIRE Metadata Regulation.
- INSPIRE, INS MDTG, (2013) INSPIRE Metadata Implementing Rules: Technical Guidelines based on EN ISO 19115 and EN ISO 19119, v1.3

See Also

kml_metadata, SpatialMetadata-class, sp::Spatial, kml_open

Examples

```
## Not run:
library(sp)
library(uuid)
library(rjson)
## read metadata from the system file:
x <- read.metadata(system.file("FGDC.xml", package="plotKML"))
str(x)
## generate missing metadata
data(eberg)
coordinates(eberg) <- ~X+Y
proj4string(eberg) <- CRS("+init=epsg:31467")
## no metadata file specified:
eberg.md <- spMetadata(eberg["SNDMHT_A"])
## this generates some metadata automatically e.g.:
xmlRoot(eberg.md@xml)[["eainfo"]][["detailed"]][["attr"]]
## combine with localy prepared metadata file:
eberg.md <- spMetadata(eberg["SNDMHT_A"],
    xml.file=system.file("eberg.xml", package="plotKML"))
## Additional metadat entries can be added by using e.g.:
eberg.md <- spMetadata(eberg["SNDMHT_A"],
  md.type="INSPIRE",
  CI_Citation_title = 'Ebergotzen data set',
  CI_Online_resource_URL = 'http://geomorphometry.org/content/ebergotzen')
## the same using the FGDC template:
eberg.md <- spMetadata(eberg["SNDMHT_A"],
  Citation_title = 'Ebergotzen data set',
  Citation_URL = 'http://geomorphometry.org/content/ebergotzen')
## Complete list of names:
mdnames <- read.csv(system.file("mdnames.csv", package="plotKML"))
mdnames$field.names
## these can be assigned to the "metadata" environment by using:
metadata.env(CI_Citation_title = 'Ebergotzen data set')
get("CI_Citation_title", metadata)

## write data and metadata to a file:
library(rgdal)
writeOGR(eberg["SNDMHT_A"], "eberg_SAND.shp", ".", "ESRI Shapefile")
saveXML(eberg.md@xml, "eberg_SAND.xml")
## export to SLD format:
metadata2SLD(eberg.md, "eberg.sld")
## plot the layer with the metadata:
kml(eberg, file.name = "eberg_md.kml", colour = SNDMHT_A, metadata = eberg.md, kmz = TRUE)

## End(Not run)
```

---

spPhoto *Generate an object of class* "SpatialPhotoOverlay"

Description

spPhoto function can be used to wrap pixel map (pixmapRGB), EXIF (Exchangeable Image File format) data, spatial location information (standing point), and PhotoOverlay (geometry) parameters to create an object of class "SpatialPhotoOverlay". This object can then be parsed to KML and visualized using Google Earth.

Usage

```
spPhoto(filename, obj, pixmap, exif.info = NULL, ImageWidth = 0,
   ImageHeight = 0, bands = rep(rep(1, ImageHeight*ImageWidth), 3),
   bbox = c(0, 0, 3/36000*ImageWidth, 3/36000*ImageHeight),
   DateTime = "", ExposureTime = "", FocalLength = "50 mm",
   Flash = "No Flash", rotation = 0, leftFov = -30, rightFov = 30,
   bottomFov = -30, topFov = 30, near = 50,
   shape = c("rectangle", "cylinder", "sphere")[1], range = 1000, tilt = 90,
   heading = 0, roll = 0, test.filename = TRUE)
```

Arguments

`filename`	file name with extension (ideally an URL)
`obj`	object of class "SpatialPoints" (requires a single point object)
`pixmap`	object of class "pixmapRGB" (see package pixmap)
`exif.info`	named list containing all available EXIF metadata
`ImageWidth`	(optional) image width in pixels
`ImageHeight`	(optional) image height in pixels
`bands`	(optional) RGB bands as vectors (see `pixmap::pixmapRGB`)
`bbox`	(optional) bounding box coordinates (by default 1 pixel is about 1 m in arc degrees)
`DateTime`	(optional) usually available from the camera EXIF data
`ExposureTime`	(optional) usually available from the camera EXIF data
`FocalLength`	(optional) usually available from the camera EXIF data
`Flash`	(optional) usually available from the camera EXIF data
`rotation`	(optional) rotation angle in 0–90 degrees
`leftFov`	(optional) angle, in degrees, between the camera's viewing direction and the left side of the view volume (-180 – 0)
`rightFov`	(optional) angle, in degrees, between the camera's viewing direction and the right side of the view volume (0 – 180)
`bottomFov`	(optional) angle, in degrees, between the camera's viewing direction and the bottom side of the view volume (-90 – 0)
`topFov`	(optional) angle, in degrees, between the camera's viewing direction and the top side of the view volume (0 – 90)
`near`	(optional) measurement in meters along the viewing direction from the camera viewpoint to the PhotoOverlay shape
`shape`	(optional) shape type — rectangle (standard photograph), cylinder (for panoramas), or sphere (for spherical panoramas)
`range`	(optional) distance from the camera to the placemark

tilt	(optional) rotation, in degrees, of the camera around the X axis
heading	(optional) direction (azimuth) of the camera, in degrees (0 – 360)
roll	(optional) rotation about the y axis, in degrees (0 – 180)
test.filename	logical; species whether a test should be first performed that the file name really exists (recommended)

Details

The most effective way to import a field photograph to SpatialPhotoOverlay for parsing to KML is to: (**a**) use the EXIF tool (courtesy of Phil Harvey) to add any important tags in the image file, (**b**) once you've added all important tags, you can upload your image either to a local installation of Mediawiki or to a public portal such as the Wikimedia Commons, (**c**) enter the missing information if necessary and add an image description. Once the image is on the server, you only need to record its unique name and then read all metadata from the Wikimedia server following the examples below.
You can also consider importing images to R by using the pixmap package, and reading the technical information via e.g. the exif package. If the image is taken using a GPS enabled camera, by getting the EXIF metadata you can generate the complete SpatialPhotoOverlay object with minimum user interaction. Otherwise, you need to at least specify: creation date, file name, and location of the focal point of the camera (e.g. by creating "SpatialPoints" object).

Value

Returns an object of class "SpatialPhotoOverlay":

filename	URL location of the original image
pixmap	optional; local copy of the image ("pixmapRGB" class)
exif.info	list of EXIF metadata
PhotoOverlay	list of the camera geometry parameters (KML specifications)
sp	location of the camera ("SpatialPoints" class)

Note

The spPhoto function will try to automatically fix the aspect ratio of the ViewVolume settings (leftFov, rightFov, bottomFov, topFov), and based on the original aspect ratio as specified in the EXIF data. This might not work for all images, in which case you will have to manually adjust those parameters.
Dimension of 3/36000*ImageWidth in decimal degrees is about 10 m in nature (3-arc seconds is about 100 m, depending on the latitude).

Author(s)

Tomislav Hengl

References

- EXIF tool (http://www.sno.phy.queensu.ca/~phil/exiftool/)
- Wikimedia API (http://www.mediawiki.org/wiki/API)

See Also

getWikiMedia.ImageInfo, pixmap::pixmapRGB, spMetadata

Examples

```
## Not run: # two examples with images on Wikimedia Commons
# (1) soil monolith (manually entered coordinates):
imagename = "Soil_monolith.jpg"
# import EXIF data using the Wikimedia API:
x1 <- getWikiMedia.ImageInfo(imagename)
# create a SpatialPhotoOverlay:
sm <- spPhoto(filename = x1$url$url, exif.info = x1$metadata)
# plot it in Google Earth
kml(sm, method="monolith", kmz=TRUE)
# (2) PhotoOverlay (geotagged photo):
imagename = "Africa_Museum_Nijmegen.jpg"
x2 <- getWikiMedia.ImageInfo(imagename)
af <- spPhoto(filename = x2$url$url, exif.info = x2$metadata)
kml(af)

## End(Not run)
```

`vect2rast` *Convert points, lines and/or polygons to rasters*

Description

Converts any "SpatialPoints*", "SpatialLines*", or "SpatialPolygons*" object to a raster map, and (optional) writes it to an external file (GDAL-supported formats; writes to SAGA GIS format by default).

Usage

```
## S4 method for signature 'SpatialPoints'
vect2rast(obj, fname = names(obj)[1], cell.size, bbox,
      file.name, silent = FALSE, method = c("raster", "SAGA")[1],
      FIELD = 0, MULTIPLE = 1, LINE_TYPE = 0, GRID_TYPE = 2, ... )
## S4 method for signature 'SpatialLines'
vect2rast(obj, fname = names(obj)[1], cell.size, bbox,
      file.name, silent = FALSE, method = c("raster", "SAGA")[1],
      FIELD = 0, MULTIPLE = 1, LINE_TYPE = 1, GRID_TYPE = 2, ... )
## S4 method for signature 'SpatialPolygons'
vect2rast(obj, fname = names(obj)[1], cell.size, bbox,
      file.name, silent = FALSE, method = c("raster", "SAGA")[1],
      FIELD = 0, MULTIPLE = 0, LINE_TYPE = 1, GRID_TYPE = 2, ... )
```

Arguments

`obj`	Spatial-PointsDataFrame,-LinesDataFrame or -PolygonsDataFrame object
`fname`	character; target variable
`cell.size`	numeric; output cell size
`bbox`	matrix; output bounding box
`file.name`	character; (optional) output file name
`silent`	logical; specifies whether to print the output

`method`	character; output rasterization engine (either raster package or SAGA GIS)
`FIELD`	integer; target column in the output shape file (see `rsaga.get.usage("grid_gridding", 0)`)
`MULTIPLE`	integer; method for multiple values (see `rsaga.get.usage("grid_gridding", 0)`)
`LINE_TYPE`	integer; method for lines (see `rsaga.get.usage("grid_gridding", 0)`)
`GRID_TYPE`	integer; preferred target grid type (see `rsaga.get.usage("grid_gridding", 0)`)
`...`	additional arguments that can be passed to the `raster::rasterize` command

Details

This function basically extends the `rasterize` function available in the raster package. The advantage of `vect2rast`, however, is that it requires no input from the user's side i.e. it automatically determines the grid cell size and the bounding box based on the properties of the input data set. The grid cell size is estimated based on the density/size of features in the map (`nndist` function in spatstat package): (a) in the case of `"SpatialPoints"` cell size is determined as half the mean distance between the nearest points; (b) in the case of `"SpatialLines"` half cell size is determined as half the mean distance between the lines; (c) in the case of polygon data cell size is determined as half the median size (area) of polygons of interest. For more details see Hengl (2006). To process larger vector maps consider using `method="SAGA"`.

Value

Returns an object of type `"SpatialGridDataFrame"`.

Author(s)

Tomislav Hengl

References

- Hengl T., (2006) Finding the right pixel size. Computers and Geosciences, 32(9): 1283-1298.
- Raster package (`https://CRAN.R-project.org/package=raster`)
- SpatStat package (`http://www.spatstat.org`)

See Also

`vect2rast.SpatialPoints`, `raster::rasterize`, `spatstat::nndist`

Examples

```
## Not run:
data(eberg)
library(sp)
library(maptools)
library(spatstat)
coordinates(eberg) <- ~X+Y
data(eberg_zones)
# point map:
x <- vect2rast(eberg, fname = "SNDMHT_A")
image(x)
# polygon map:
x <- vect2rast(eberg_zones)
image(x)
# for large data sets use SAGA GIS:
```

```
x <- vect2rast(eberg_zones, method = "SAGA")

## End(Not run)
```

---

vect2rast.SpatialPoints *Converts points to rasters*

Description

Converts object of class "SpatialPoints*" to a raster map, and (optional) writes it to an external file (GDAL-supported formats; it used the SAGA GIS format by default).

Usage

```
vect2rast.SpatialPoints(obj, fname = names(obj)[1], cell.size, bbox,
      file.name, silent = FALSE, method = c("raster", "SAGA")[1], FIELD = 0,
      MULTIPLE = 1, LINE_TYPE = 0, GRID_TYPE = 2, ... )
```

Arguments

obj	"SpatialPoints*" object
fname	target variable name in the "data" slot
cell.size	(optional) grid cell size in the output raster map
bbox	(optional) output bounding box (class "bbox") for cropping the data
file.name	(optional) file name to export the resulting raster map
silent	logical; specifies whether to print any output of processing
method	character; specifies the gridding method
FIELD	character; SAGA GIS argument attribute table field number
MULTIPLE	character; SAGA GIS argument method for multiple values — [0] first, [1] last, [2] minimum, [3] maximum, [4] mean
LINE_TYPE	character; SAGA GIS argument method for rasterization — [0] thin, [1] thick
GRID_TYPE	character; SAGA GIS argument for coding type — [0] integer (1 byte), [1] integer (2 byte), [2] integer (4 byte), [3] floating point (4 byte), [4] floating point (8 byte)
...	additional arguments that can be passed to the raster::rasterize command

Value

Returns an object of type "SpatialGridDataFrame".

Author(s)

Tomislav Hengl

See Also

vect2rast

Examples

```
## Not run:
library(sp)
data(meuse)
coordinates(meuse) <- ~x+y
# point map:
x <- vect2rast(meuse, fname = "om")
data(SAGA_pal)
sp.p <- list("sp.points", meuse, pch="+", cex=1.5, col="black")
spplot(x, col.regions=SAGA_pal[[1]], sp.layout=sp.p)

## End(Not run)
```

`whitening` *whitening*

Description

Derives a *'whitenned'* color based on the Hue-Saturation-Intensity color model. This method can be used to visualize uncertainty: the original color is *leached* proportionally to the uncertainty (white color indicates maximum uncertainty).

Usage

```
whitening(z, zvar, zlim = c(min(z, na.rm=TRUE), max(z, na.rm=TRUE)),
  elim = c(.4,1), global.var = var(z, na.rm=TRUE), col.type = "RGB")
```

Arguments

`z`	numeric; target variable (e.g. predicted values)
`zvar`	numeric; prediction error (variance)
`zlim`	upper and lower limits for target variable
`elim`	upper and lower limits for the normalized error
`global.var`	global variance (either estimated from the data or specified)
`col.type`	characted; `"RGB"` or `"HEX"`

Details

The HSI is a psychologically appealing color model for visualization of uncertainty: hue is used to visualize values and *whitening* (*paleness* or *leaching* percentage) is used to visualize the uncertainty, or in other words the map is incomplete in the areas of high uncertainty. Unlike standard legends for continuous variables, this legend has two axis — one for value range and one for uncertainty range (see also `kml_legend.whitening`).

The standard range for `elim` is 0.4 and 1.0 (maximum). This assumes that a satisfactory prediction is when the model explains more than 85% of the total variation (normalized error = 40%). Otherwise, if the value of the normalized error get above 80%, the model accounts for less than 50% of variability.

Whitening is of special interest for visualization of the prediction errors in geostatistics. Formulas to derive the whitening color are explained in Hengl et al. (2004).

Author(s)

Tomislav Hengl and Pierre Roudier

References

- Hengl, T., Heuvelink, G.M.B., Stein, A., (2004) A generic framework for spatial prediction of soil variables based on regression-kriging. Geoderma 122 (1-2): 75-93.
- Hue-Saturation-Intensity color model (`http://en.wikipedia.org/wiki/HSL_and_HSV`)

See Also

`kml_legend.whitening`

Examples

```
whitening(z=15, zvar=5, zlim=c(10,20), global.var=7)
# significant color;
whitening(z=15, zvar=5, zlim=c(10,20), global.var=4)
# error exceeds global.var -> totally white;
```

`worldgrids_pal` *Standard global color palettes for factor variables*

Description

A number of color palettes used to visualize various environmental categorical / factor variables: land cover classes, water types, anthroms, soil types and similar. Each colour palette consists of a variable number of colours (hexadecimal system). Factor levels names are attached as attributes to the palette.

Usage

```
data(worldgrids_pal)
```

Format

The list contains:

`anthroms` Color palette used for the global map of anthroms (Ellis and Ramankutty, 2008).

`bodemfgr` A simplified color palette for soil types.

`corine2k` Color palette used in the Corine 2000 project for land cover classes (BÃ¼ttner, et al., 2002).

`glc2000` Color palette used for the Global Land Cover 2000 mapping project (Global Land Cover 2000).

`globcov` Color palette used for the ENVISAT-based Global Land Cover map at resolution of 300 m (GlobCover Land Cover version V2.2).

`gtkaart` Color palette used for the Ground water levels map of the Netherlands (Gaast et al. 2005).

`IGBP` Color palette for 17 land cover classes defined by the International Geosphere Biosphere Programme (IGBP).

`lgn3` Color palette used for the Dutch land use map (Hazeu, 2005).

`t250vlak` Color palette used for the most general land use classes at scale 1:250k (TOP250NL).

`watert` Color palette used for the water types (generalized) in the Netherlands.

Note

These colour palettes are only valid for factor-type variables. The names of classes used in the legend can be obtained by loading the palette list.

Author(s)

Tomislav Hengl

References

- Bicheron, P. et al. (2008) GLOBCOVER: Products Description and Validation Report. MEDIAS France, Toulouse, 47 p.
- BÃ¼ttner, G. et al. (2002) Corine Land Cover update 2000, Technical guidelines. EEA (European Environment Agency), Kopenhagen.
- Ellis, E.C., Ramankutty, N. (2008) Putting people in the map: anthropogenic biomes of the world. Frontiers in Ecology and the Environment, Vol. 6, No. 8, pp. 439-447.
- Fritz, S. et al. (2003) Harmonisation, mosaicing and production of the Global Land Cover 2000 database. JRC report EUR 20849 EN, Luxembourg, 41 p.
- Gaast, J.W.J. van der, H.R.J. Vroon en M. Pleijter, (2006) De grondwaterdynamiek in het waterschap Regge en Dinkel. Wageningen, Alterra-rapport 1335.
- Hazeu, G.W., (2005) Landelijk Grondgebruiksbestand Nederland (LGN5). Vervaardiging, nauwkeurigheid en gebruik. Wageningen, Alterra. Alterra-report 1213, 92 pp.
- Puijenbroek, P. van; Clement, J., (2008) Het oppervlaktewater getypeerd: de eerste Nederlandse watertypenkaart. Agro informatica 21(3): 21-25.

See Also

SAGA_pal, R_pal

Examples

```
data(worldgrids_pal)
## Not run: # globcov palette with class names:
display.pal(worldgrids_pal)
dev.off()
display.pal(worldgrids_pal, sel=5, names=TRUE)

## End(Not run)
```

Index