Package: animate (via r-universe)

August 22, 2024

Title A Web-Based Graphics Device for Animated Visualisations

Version 0.3.9.8

Description Implements a web-based graphics device for animated visualisations. Modelled on the 'base' syntax, it extends the 'base' graphics functions to support frame-by-frame animation and keyframes animation. The target use cases are real-time animated visualisations, including agent-based models, dynamical systems, and animated diagrams. The generated visualisations can be deployed as GIF images / MP4 videos, as 'Shiny' apps (with interactivity) or as HTML documents through embedding into R Markdown documents.

License MIT + file LICENSE

URL `https://kcf-jackson.github.io/animate/`

BugReports `https://github.com/kcf-jackson/animate/issues`

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Imports R6, httpuv, base64enc, jsonlite, glue, R.utils

Suggests rmarkdown, knitr, shiny, htmltools, pryr, V8, servr

VignetteBuilder knitr

Repository https://kcf-jackson.r-universe.dev

RemoteUrl https://github.com/kcf-jackson/animate

RemoteRef HEAD

RemoteSha 3493092c28717eb3f52a49348aec0c91d213fb42

Contents

`animate` *A web-based graphics device for animated visualisations*

Description

Extends the 'base' graphics functions to support frame-by-frame animation and keyframes animation.

Public fields

`connection` A handle for the WebSocket connection.

`ready_state` The ready state of the connection.

`shiny` TRUE or FALSE; whether the device is used with in a 'Shiny' app.

`session` A 'Shiny' session.

`virtual_meta` A list of device metadata.

`virtual_session` A virtual session simulated with 'V8'.

`event_handlers` A named list of user-defined functions for handling events.

Methods

Public methods:

- `animate$new()`
- `animate$off()`
- `animate$send()`
- `animate$set_max_stacksize()`
- `animate$svg()`
- `animate$bars()`
- `animate$objects()`
- `animate$katex()`
- `animate$plot()`
- `animate$points()`
- `animate$lines()`

- `animate$arrows()`
- `animate$abline()`
- `animate$axis()`
- `animate$text()`
- `animate$image()`
- `animate$event()`
- `animate$chain()`
- `animate$simple_event()`
- `animate$set()`
- `animate$par()`
- `animate$remove()`
- `animate$clear()`
- `animate$delete()`
- `animate$import()`
- `animate$export()`
- `animate$record()`
- `animate$screenshot()`
- `animate$observeAnimateEvent()`
- `animate$clone()`

Method `new()`: Constructor of the device

Usage:

```
animate$new(width, height, id = "SVG_1", launch.browser, ...)
```

Arguments:

`width` An integer; the width in pixels.

`height` An integer; the height in pixels.

`id` A character string; the id assigned to the device.

`launch.browser` A function to launch a viewer; two options are `rstudioapi::viewer` and `utils::browseURL`. It defaults to the first option if the user is using RStudio and to the second option otherwise. The default applies to interactive session only.

`...` Additional arguments. Use `virtual = TRUE` to use the virtual device, `shiny = TRUE` for shiny application; everything else will be passed to the SVG element that hosts the visualisation.

Examples:

```
\donttest{
library(animate)
device <- animate$new(400, 400)  # Launch a WebSocket server
attach(device)
x <- 1:10
y <- 1:10
id <- new_id(x)   # Give each point an ID: c("ID-1", "ID-2", ..., "ID-10")
plot(x, y, id = id)

new_y <- 10:1
plot(x, new_y, id = id, transition = TRUE)  # Use transition
off()
detach(device)
}
```

Method `off()`**:** Switch off the device; this function closes the WebSocket connection.

Usage:

```
animate$off()
```

Method `send()`**:** Send commands to device

Usage:

```
animate$send(message)
```

Arguments:

`message` The message to send to the device.

Method `set_max_stacksize()`**:** Set the maximum size of the stack

Usage:

```
animate$set_max_stacksize(n)
```

Arguments:

`n` The number of commands the plot stack can hold. Use -1 for unlimited number of commands.

Method `svg()`**:** Initialise a SVG element

Usage:

```
animate$svg(width = 800, height = 600, ...)
```

Arguments:

`width` Width of the canvas in pixels.

`height` Height of the canvas in pixels.

`...` Additional parameters. Some commonly used parameters are `id` and `root`. `id` assigns an id to the SVG element for future reference; `root` specifies the DOM element to insert the SVG element into.

Method `bars()`**:** Add bars to a plot

Usage:

```
animate$bars(x, y, w, h, ...)
```

Arguments:

`x` The x coordinates of the bars.

`y` The y coordinates of the bars.

`w` The width of the bars.

`h` The height of the bars.

`...` Additional graphical parameters.

Method `objects()`**:** Add HTML objects to a plot

Usage:

```
animate$objects(x, y, w, h, content, ...)
```

Arguments:

`x` The x coordinates of the objects.

`y` The y coordinates of the objects.

`w` The width of the objects.

`h` The height of the objects.

`content` The content of the objects; the HTML string.

`...` Additional graphical parameters.

Examples:

```
# Add a HTML DIV element
device <- animate::animate$new(500, 500)
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))
# Add a grid to aid positioning
abline(h = 0, v = 0)
abline(h = 0:10, v = 0:10, col = "#22222222")
# Add the DIV element
objects(x=1, y=1, w=2, h=1, id = "obj-1", content = "<div>PlaceHolder</div>")
off()
detach(device)
```

Method `katex()`**:** Add TeX symbols to a plot

Usage:

```
animate$katex(x, y, w, h, tex, ...)
```

Arguments:

`x` The x coordinates of the objects.

`y` The y coordinates of the objects.

`w` The width of the objects.

`h` The height of the objects.

`tex` The TeX string.

`...` Additional graphical parameters.

Details: See https://katex.org/docs/supported for all the supported TeX functions. Currently, the function loads the JavaScript library and the CSS stylesheet from CDN links, so you must be online to use this function. The reason for this is that KaTeX comes with many fonts. If they are all included in the bundle, then the bundle size becomes too large.

Examples:

```
# Add a KaTeX equation (see KaTeX.js )
device <- animate::animate$new(500, 500)
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))
# Add a grid to aid positioning
abline(h = 0, v = 0)
abline(h = 0:10, v = 0:10, col = "#22222222")
# Add the KaTeX equation
katex(x=1, y=3, w=3, h=1, id = 'tex-1', tex = 'a^2 + b^2 = c^2')
off()
detach(device)
```

Method `plot()`**:** Generic X-Y plotting

Usage:

```
animate$plot(x, y, type = "p", ...)
```

Arguments:

`x` The x coordinates of the data.

`y` The y coordinates of the data.

`type` Type of the plot; one of 'p' and 'l'.

`...` Additional graphical parameters.

Method `points()`**:** Add points to a plot

Usage:

```
animate$points(x, y, ...)
```

Arguments:

`x` The x coordinates of the points.

`y` The y coordinates of the points.

`...` Additional graphical parameters.

Details: Options for the "pch" parameter: "circle", "plus", "diamond", "square", "star", "triangle", "wye", "triangle_down", "triangle_left", "triangle_right", "diamond_alt", "diamond_square", "pentagon", "hexagon", "hexagon_alt", "octagon", "octagon_alt", "cross".
The unit of the "cex" parameter is squared pixels, corresponding to how much pixel space the symbol would cover. The convention comes from the 'D3' library, and the choice is (believed) to make plots visually consistent across the different symbols.

Method `lines()`**:** Add line segments / paths to a plot

Usage:

```
animate$lines(x, y, ...)
```

Arguments:

`x` The x coordinates of the line.

`y` The y coordinates of the line.

`...` Additional graphical parameters.

Method `arrows()`**:** Add arrows to a plot

Usage:

```
animate$arrows(x0, y0, x1, y1, length, angle = pi/6, code = 2, ...)
```

Arguments:

`x0` The x coordinates of the start of the arrow.

`y0` The y coordinates of the start of the arrow.

`x1` The x coordinates of the end of the arrow.

`y1` The y coordinates of the end of the arrow.

`length` The length of the arrow head.

`angle` The angle of the arrow head.

`code` The code of the arrow head. Use 1,2,3 or "start"/"end"/"both".

`...` Additional graphical parameters

Details: Currently, the 'arrows' function is not supported as a primitive; instead, it calls the 'lines' function.

Examples:

```
\donttest{
library(animate)
device <- animate$new(500, 500)
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))  # Use static axes
plot(1:10, 1:10, id = paste0("point-", 1:10))
lines(c(3, 6), c(6, 3), id = "line-1")
arrows(4, 8, 8, 4, id='arrow-1')
arrows(1, 2, 3, 6, id='arrow-1', transition = TRUE)
off()
detach(device)
}
```

Method `abline()`**:** Add straight lines to a plot

Usage:

```
animate$abline(a, b, h, v, ...)
```

Arguments:

`a` The intercept.

`b` The slope.

`h` The y-value(s) for horizontal line(s).

`v` The x-value(s) for vertical line(s).

`...` Additional graphical parameters.

Method `axis()`**:** Add an axis to a plot

Usage:

```
animate$axis(x, ...)
```

Arguments:

`x` The x coordinates of the text.

`...` Additional graphical parameters.

`y` The y coordinates of the text.

`labels` The text.

Method `text()`**:** Add text to a plot

Usage:

```
animate$text(x, y, labels, ...)
```

Arguments:

`x` The x coordinates of the text.

`y` The y coordinates of the text.

`labels` The text.

`...` Additional graphical parameters.

Details: Useful tips: use `style = list("text-anchor" = "middle", "dominant-baseline" = "middle")` or `style = list("text-anchor" = "middle", "dominant-baseline" = "central")` to center the text.

Method `image()`**:** Add background image to a plot

Usage:

```
animate$image(href, width, height, ...)
```

Arguments:

`href` The link to the image.

`width` The width of the image.

`height` Th height of the image.

`...` Additional graphical parameters.

Method `event()`**:** Attach an interactive event to an element

Usage:

```
animate$event(selector, event_type, callback)
```

Arguments:

`selector` A character string; a CSS selector.

`event_type` A character string; the event type. For example, "click", "mouseover", "mouse-out". See more options at https://www.w3schools.com/jsref/dom_obj_event.asp.

`callback` A function, to be called when the event is triggered. The function should take an argument to receive the data from the browser end.

Method `chain()`**:** Chain a transition after another.

Usage:

```
animate$chain(callback)
```

Arguments:

`callback` A function, to be called when the event is triggered. The function should take an argument to receive the data from the browser end.

Examples:

```
\donttest{
library(animate)
device <- animate$new(600, 600)  # Launch a WebSocket server
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))
plot(1:10, 1:10, id = 1:10)
points(1:10, sample(10, 10), id = 1:10,
  transition = list(
    duration = 1000,
    on = chain(function(message) {
      print(message)
      points(1:10, sample(10, 10), id = 1:10, bg = "green",
             transition = list(duration = 2000))
      })
  ))
par(xlim = NULL, ylim = NULL)  # Reset `xlim` and `ylim` in `par`
off()
detach(device)
}
```

Method `simple_event()`**:** Attach a captured event to an element

Usage:

`animate$simple_event(selector, event_type, method, param)`

Arguments:

`selector` A character string; a CSS selector.

`event_type` A character string; the event type. For example, "click", "mouseover", "mouseout". See more options at `https://www.w3schools.com/jsref/dom_obj_event.asp`.

`method` A character string; the name of a device function (e.g. "points").

`param` A named list of arguments to be called with.

Details: This function differs from the `event` function in that events registered through `simple_event` do not require R at deployment to work.

Method `set()`**:** Set the active device to a SVG element

Usage:

`animate$set(device_id)`

Arguments:

`device_id` A character vector; ID of the device.

Method `par()`**:** Set the graphical parameters

Usage:

`animate$par(...)`

Arguments:

`...` The graphical parameters

Method `remove()`**:** Remove elements from the active SVG element

Usage:

`animate$remove(id = NULL, selector = "*")`

Arguments:

`id` A character vector; the ID of the elements.

`selector` A character vector; a CSS selector.

Method `clear()`**:** Remove all elements from the active SVG element

Usage:

`animate$clear()`

Method `delete()`**:** Remove a SVG element

Usage:

```
animate$delete(id = NULL)
```

Arguments:

`id` A character string; the ID of the SVG. If not provided, remove the active SVG element. #' @description #' Perform a group of graphical operations to a plot #' @param ... Any number of graphical operations. group = function(...) self$send(Message("fn_group", c(...))) ,

Method `import()`: Import an animated plot

Usage:

```
animate$import(setting)
```

Arguments:

`setting` A JSON file exported from previous runs.

Method `export()`: Export an animated plot

Usage:

```
animate$export(path = "./animate.json", handler = "browser")
```

Arguments:

`path` A character string; the file path to export to.

`handler` 'r' or 'browser'; the program to handle the export operation.

Method `record()`: Record an animated plot as a MP4 video

Usage:

```
animate$record()
```

Details: This function will prompt you to select a screen / window / tab to record. Once started, the recording can be stopped by using the stop button at the notification box, or clicking anywhere on the page near the device. Always confirm that the screen recording notification box is gone. The captured video will be downloaded right after the recording stops.
This uses web browsers' Media Streams API to record the screen and return the captured frames as a video. The entire process runs locally. The source file that provides this functionality can be found at `system.file("addons/screen_record.js", package = "animate")`.
This function is disabled for 'Shiny' app and R Markdown document.
This function does not work in the RStudio viewer. Please use the "show in new window" button to launch the page with a web browser.
See browser compatibility at: `https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_Recording_API#browser_compatibility`
See Media Streams API reference at: `https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API`

Method `screenshot()`: Take screenshot for GIF generation

Usage:

```
animate$screenshot(action, selector = NULL, options = list())
```

Arguments:

`action` One of "new", "capture", "save", "start" and "end". They correspond to two modes of generating the GIF. "new" is used to set up the generator. "capture" and "save" are for frame-by-frame animation, and "start" and "end" are for key-frame animation.

`selector` A string; the element on the page to capture. It captures the "body" element by default, as direct capture of the "svg" element does not work.

`options` Options for the actions. See details for more information.

Details: The function uses html2canvas.js to capture the screenshot and gif.js to string them into a GIF file.
Some known limitations: Simultaneously playing (SVG) animation and rendering them to raster image on the fly can be resource heavy; there is no guarantee that the specified frame rate will be met. Also, since html2canvas is used to take the screenshot, its restrictions also apply here. For cases where `screenshot` is insufficient, use the `record` function instead. See more detail here.
For "new" action, the options are as given by the GIF.js library:

Name	Default	Description
repeat	`0`	repeat count, `-1` = no repeat, `0` = forever
quality	`10`	pixel sample interval, lower is better
workers	`2`	number of web workers to spawn
workerScript	`gif.worker.js`	url to load worker script from
background	`#fff`	background color where source image is transpar
width	`null`	output image width
height	`null`	output image height
transparent	`null`	transparent hex color, `0x00FF00` = green
dither	`false`	dithering method, e.g. `FloydSteinberg-serpen`
debug	`false`	whether to print debug information to console

The option ’workerScript’ is taken care of by `animate`.

Available dithering methods are: `FloydSteinberg`, `FalseFloydSteinberg`, `Stucki`, `Atkinson`.
You can add `-serpentine` to use serpentine scanning, e.g. `Stucki-serpentine`.
For the "capture" action, the options are:

Name	Default	Description
delay	`500`	frame delay
copy	`false`	copy the pixel data
dispose	`-1`	frame disposal code. See GIF89a Spec

Examples:

```
\donttest{
# Frame-by-frame animation example
device <- animate::animate$new(500, 500)
attach(device)

# Initialise the generator
screenshot("new", "body", list(width = 1000, height = 1000, quality = 10))

# Plotting the sine curve
x <- seq(1, 40, 0.1)
y <- sin(x * pi / 6)
```

```
plot(x, y, type="l", id="line-1")
screenshot("capture")  # capture the frame

# Update the plot with the same id
for (n in 41:200) {
  new_x <- seq(1, n, 0.1)
  new_y <- sin(new_x * pi / 6)
  plot(new_x, new_y, type="l", id="line-1")
 screenshot("capture", options = list(delay = 100))  # capture the frame with options
  Sys.sleep(0.2)  # allow ample time for the screen capture to process
}

# Render and save the GIF file (this can take some time to complete)
screenshot("save")
off()
det


# Key-frame animation example
device <- animate::animate$new(500, 500)
attach(device)

# Initialise the generator
screenshot("new", "body", list(width = 1000, height = 900, quality = 10))

# Plot 10 points at random locations
x <- 1:10
y <- 10 * runif(10)
id <- new_id(y, prefix="points")
plot(x, y, bg="orange", id=id)

# Capture screenshot every 100ms and output to GIF frame with 200ms gap in-between
screenshot("start", options = list(delayCapture = 100, delay = 200))

# Update the plot with default transition animation
new_y <- 10 * runif(10)
points(x, new_y, bg="blue", id=id, transition=TRUE)

# Update the plot with specific transition animation
new_y <- 10 * runif(10)
points(x, new_y, bg="green", cex=(1:10)*30, id=id, transition=list(duration = 2000))

# End the screen capturing and generate the GIF file
screenshot("end")

# Clean up
off()
detach(device)
```

```
}
```

Method `observeAnimateEvent()`**:** Event handler

Usage:

```
animate$observeAnimateEvent(input)
```

Arguments:

`input` The input object in the `server` function of a 'Shiny' app.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
animate$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `animate$new`
## ------------------------------------------------


library(animate)
device <- animate$new(400, 400)  # Launch a WebSocket server
attach(device)
x <- 1:10
y <- 1:10
id <- new_id(x)   # Give each point an ID: c("ID-1", "ID-2", ..., "ID-10")
plot(x, y, id = id)

new_y <- 10:1
plot(x, new_y, id = id, transition = TRUE)  # Use transition
off()
detach(device)


## ------------------------------------------------
## Method `animate$objects`
## ------------------------------------------------

# Add a HTML DIV element
device <- animate::animate$new(500, 500)
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))
# Add a grid to aid positioning
abline(h = 0, v = 0)
abline(h = 0:10, v = 0:10, col = "#22222222")
# Add the DIV element
objects(x=1, y=1, w=2, h=1, id = "obj-1", content = "<div>PlaceHolder</div>")
```

```
off()
detach(device)

## ------------------------------------------------
## Method `animate$katex`
## ------------------------------------------------

# Add a KaTeX equation (see KaTeX.js )
device <- animate::animate$new(500, 500)
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))
# Add a grid to aid positioning
abline(h = 0, v = 0)
abline(h = 0:10, v = 0:10, col = "#22222222")
# Add the KaTeX equation
katex(x=1, y=3, w=3, h=1, id = 'tex-1', tex = 'a^2 + b^2 = c^2')
off()
detach(device)

## ------------------------------------------------
## Method `animate$arrows`
## ------------------------------------------------


library(animate)
device <- animate$new(500, 500)
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))  # Use static axes
plot(1:10, 1:10, id = paste0("point-", 1:10))
lines(c(3, 6), c(6, 3), id = "line-1")
arrows(4, 8, 8, 4, id='arrow-1')
arrows(1, 2, 3, 6, id='arrow-1', transition = TRUE)
off()
detach(device)


## ------------------------------------------------
## Method `animate$chain`
## ------------------------------------------------


library(animate)
device <- animate$new(600, 600)  # Launch a WebSocket server
attach(device)
par(xlim = c(0, 10), ylim = c(0, 10))
plot(1:10, 1:10, id = 1:10)
points(1:10, sample(10, 10), id = 1:10,
  transition = list(
    duration = 1000,
    on = chain(function(message) {
      print(message)
      points(1:10, sample(10, 10), id = 1:10, bg = "green",
             transition = list(duration = 2000))
      })
    ))
par(xlim = NULL, ylim = NULL)  # Reset `xlim` and `ylim` in `par`
off()
detach(device)


## ------------------------------------------------
## Method `animate$screenshot`
## ------------------------------------------------


# Frame-by-frame animation example
device <- animate::animate$new(500, 500)
attach(device)

# Initialise the generator
screenshot("new", "body", list(width = 1000, height = 1000, quality = 10))

# Plotting the sine curve
x <- seq(1, 40, 0.1)
y <- sin(x * pi / 6)
plot(x, y, type="l", id="line-1")
screenshot("capture")  # capture the frame

# Update the plot with the same id
for (n in 41:200) {
  new_x <- seq(1, n, 0.1)
  new_y <- sin(new_x * pi / 6)
  plot(new_x, new_y, type="l", id="line-1")
  screenshot("capture", options = list(delay = 100))  # capture the frame with options
  Sys.sleep(0.2)  # allow ample time for the screen capture to process
}

# Render and save the GIF file (this can take some time to complete)
screenshot("save")
off()
det


# Key-frame animation example
device <- animate::animate$new(500, 500)
attach(device)

# Initialise the generator
screenshot("new", "body", list(width = 1000, height = 900, quality = 10))

# Plot 10 points at random locations
x <- 1:10
y <- 10 * runif(10)
id <- new_id(y, prefix="points")
plot(x, y, bg="orange", id=id)
```

```
# Capture screenshot every 100ms and output to GIF frame with 200ms gap in-between
screenshot("start", options = list(delayCapture = 100, delay = 200))

# Update the plot with default transition animation
new_y <- 10 * runif(10)
points(x, new_y, bg="blue", id=id, transition=TRUE)

# Update the plot with specific transition animation
new_y <- 10 * runif(10)
points(x, new_y, bg="green", cex=(1:10)*30, id=id, transition=list(duration = 2000))

# End the screen capturing and generate the GIF file
screenshot("end")

# Clean up
off()
detach(device)
```

`animateDependencies` *The HTML dependency of an 'animate' plot*

Description

The HTML dependency of an 'animate' plot

Usage

```
animateDependencies()
```

`animateOutput` *Create an animate output (container) element*

Description

Create an animate output (container) element

Usage

```
animateOutput(
  outputId = "animateOutput",
  width = "100%",
  height = "400px",
  ...
)
```

Arguments

`outputId`	output variable to read the plot/image from.
`width`	Width of the plot area. Must be a valid CSS unit (like "100%", "400px", "auto").
`height`	Height of the plot area. Must be a valid CSS unit (like "100%", "400px", "auto").
`...`	Optional CSS styling for the container of the plot.

Note

(Advanced usage) A "stack_limit" parameter can be included in the optional parameters to control how many directives the device should keep track of.

Examples

```
# Using 'animate' in a 'Shiny' app
library(shiny)

ui <- fluidPage(
  actionButton("buttonPlot", "Plot"),
  actionButton("buttonPoints", "Points"),
  actionButton("buttonLines", "Lines"),
  animateOutput()
)

server <- function(input, output, session) {
  device <- animate$new(600, 400, session = session)
  id <- new_id(1:10)

  observeEvent(input$buttonPlot, {     # Example 1
    device$plot(1:10, 1:10, id = id)
  })

  observeEvent(input$buttonPoints, {   # Example 2
    device$points(1:10, runif(10, 1, 10), id = id, transition = TRUE)
  })

  observeEvent(input$buttonLines, {    # Example 3
    x <- seq(1, 10, 0.1)
    y <- sin(x)
    id <- "line_1"
    device$lines(x, y, id = id)
    for (n in 11:100) {
      x <- seq(1, n, 0.1)
      y <- sin(x)
      device$lines(x, y, id = id)
      Sys.sleep(0.05)
    }
  })
}

# shinyApp(ui = ui, server = server)  # Launch the 'Shiny' app
```

click_to_loop *Click an element to play all frames*

Description

Playback option for the functions rmd_animate and insert_animate.

Usage

```
click_to_loop(selector = "#SVG_1", start = 2, wait = 20)
```

Arguments

selector	The ID of the DOM element.
start	An integer; the number of frames to execute upon the beginning of the visualisation. This is useful when one wants to start with some set-up instead of an empty canvas.
wait	A number; the number of milliseconds to wait for before the next frame is drawn.

Examples

~~~
input <- tempfile(fileext = ".Rmd")
output <- tempfile(fileext = ".html")
writeLines('
```{r, echo = FALSE, message = FALSE}
# Run / include the following in a code chunk of an R Markdown document
library(animate)
insert_animate(system.file("tests/Lorenz_system.json.gz", package = "animate"),
               options = click_to_loop())
```
', input)
knitr::knit(input, output)
# browseURL(output)
~~~

click_to_play *Click an element to play a frame*

Description

Playback option for the functions rmd_animate and insert_animate.

Usage

```
click_to_play(selector = "#SVG_1", start = 2)
```

Arguments

`selector`	The ID of the DOM element.
`start`	An integer; the number of frames to execute upon the beginning of the visualisation. This is useful when one wants to start with some set-up instead of an empty canvas.

Examples

~~~
input <- tempfile(fileext = ".Rmd")
output <- tempfile(fileext = ".html")
writeLines('
```{r, echo = FALSE, message = FALSE}
# Run / include the following in a code chunk of an R Markdown document
library(animate)
insert_animate(system.file("tests/basic_points.json", package = "animate"),
               options = click_to_play())
```
', input)
knitr::knit(input, output)
# browseURL(output)
~~~

`ffmpeg` *Launch the 'FFmpeg'-based video editor ('Shiny' app)*

Description

Launch the 'FFmpeg'-based video editor ('Shiny' app)

Usage

```
ffmpeg()
```

Note

This requires `ffmpeg` to work. The 'ffmpeg' binary can be downloaded from `https://ffmpeg.org/download.html`.

insert_animate *Insert an animated plot into an R Markdown document*

Description

Insert an animated plot into an R Markdown document

Usage

```
insert_animate(file, options = click_to_play(), style, use_cdn = TRUE)
```

Arguments

`file`	The exported plot.
`options`	A character string; the JavaScript to customise the playback options. Two basic options `click_to_play()` and `click_to_loop()` have been implemented for general usage.
`style`	Optional style for the iframe that hosts the visualisation.
`use_cdn`	TRUE / FALSE; if TRUE, serve the assets from a CDN, otherwise embed the assets into the HTML.

Examples

~~~
input <- tempfile(fileext = ".Rmd")
output <- tempfile(fileext = ".html")
writeLines('
```{r, echo = FALSE, message = FALSE}
# Run / include the following in a code chunk of an R Markdown document
library(animate)
insert_animate(system.file("tests/Lorenz_system.json.gz", package = "animate"),
               options = click_to_loop())
```
', input)
knitr::knit(input, output)
# browseURL(output)
~~~

loop *Loop through the available frames n times*

Description

Playback option for the functions rmd_animate and insert_animate.

Usage

```
loop(times = 1, wait = 20)
```

Arguments

`times`	An integer; the number of times to loop.
`wait`	A number; the number of milliseconds to wait for before the next frame is drawn.

Examples

~~~
input <- tempfile(fileext = ".Rmd")
output <- tempfile(fileext = ".html")
writeLines('
```{r, echo = FALSE, message = FALSE}
# Run / include the following in a code chunk of an R Markdown document
library(animate)
insert_animate(system.file("tests/Lorenz_system.json.gz", package = "animate"),
               options = loop(times = 2, wait = 15))
```
', input)
knitr::knit(input, output)
# browseURL(output)
~~~

`new_id` *A utility function for generating IDs*

Description

A utility function for generating IDs

Usage

```
new_id(x, prefix = "ID", sep = "-")
```

Arguments

`x`	The data that require IDs.
`prefix`	A character string; the prefix to be added to each ID.
`sep`	A character string; the separator to be added between the prefix and an ID.

Examples

```
new_id(x = runif(10), prefix = "points")
```

rmd_animate *In-line rendering of an animated plot in an R Markdown document*

Description

In-line rendering of an animated plot in an R Markdown document

Usage

```
rmd_animate(device, ...)
```

Arguments

`device`	The animate object.
`...`	Optional parameters to pass to insert_animate.

Note

This function should only be used in a code chunk of an R Markdown document.

Examples

~~~
input <- tempfile(fileext = ".Rmd")
output <- tempfile(fileext = ".html")
writeLines('
```{r, echo = FALSE, message = FALSE}
# Run / include the following in a code chunk of an R Markdown document
library(animate)
device <- animate$new(500, 500, virtual = TRUE)  # set `virtual = TRUE` for R Markdown document
attach(device)

# Data
id <- new_id(1:10)
s <- 1:10 * 2 * pi / 10
s2 <- sample(s)

# Plot
par(xlim = c(-2.5, 2.5), ylim = c(-2.5, 2.5))
plot(2*sin(s), 2*cos(s), id = id)
points(sin(s2), cos(s2), id = id, transition = list(duration = 2000))

# Render in-line in an R Markdown document
rmd_animate(device, click_to_play(start = 3))  # begin the plot at the third frame
```
```{r, echo = FALSE, message = FALSE}
par(xlim = NULL, ylim = NULL)  # Reset `xlim` and `ylim` in `par`
# Do some other plots
off()
detach(device)
~~~

```
```
', input)
knitr::knit(input, output)
# browseURL(output)
```

`websocket` *Start a Websocket server*

Description

A thin wrapper of the `httpuv` package, modified to serve animated plots.

Public fields

`app` A list of functions that define the application.

`server` A server handle to be used by 'stopServer'.

`ws` A WebSocket channel to handle the communication between the R session and the browser session.

`in_handler` A function to handle instructions sent by the browser session.

`port` An integer; the TCP port number.

`connected` TRUE or FALSE; whether a connection has been established. One should start the WebSocket server before launching the web page that connects to the server.

`started` TRUE or FALSE; whether a server has been started. Use the `startServer` method to start a server.

Methods

Public methods:

- `websocket$startServer()`
- `websocket$stopServer()`
- `websocket$listServers()`
- `websocket$stopAllServers()`
- `websocket$new()`
- `websocket$clone()`

Method `startServer()`**:** Start a WebSocket server

Usage:

```
websocket$startServer()
```

Method `stopServer()`**:** Stop a WebSocket server

Usage:

```
websocket$stopServer()
```
```

**Method** `listServers()`**:** List all running WebSocket servers

*Usage:*

```
websocket$listServers()
```

Method `stopAllServers()`**:** Stop all running WebSocket servers

Usage:

```
websocket$stopAllServers()
```

Method `new()`**:** Initialise a WebSocket connection

Usage:

```
websocket$new(in_handler, port = 9454)
```

Arguments:

`in_handler` A function to handle incoming message, default to be print which only displays the message without any processing.

`port` An integer; the TCP port number.

Returns: A 'websocket' object.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
websocket$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Index