

Core Concepts

gganimate builds on ggplot2's grammar of graphics to provide functions for animation.

You

add them to plots created with ggplot() the same way you add a geom

Main Function Groups

- `transition_*()`: What variable

controls

change and how?

- `view_*()`: Should the axes change with

the

data?

- `enter/exit_*()`: How does new data get

added the plot? How does old data leave?

- `shadow_*()`: Should previous data be

"remembered" and shown with current

data?

- `ease_aes()`: How do you want to

handle the

pace of change between transition values?

Note: you only need a `transition_*()`

or

`view_*()` to make an animation. The other

function groups enable you to add features

or alter

gganimate's default settings.

Starting Plots

```
library(tidyverse)
```

```
library(gganimate)
```

```
a <- ggplot(diamonds, aes(carat, price)) + geom_point()
```

```
b <- ggplot(txhousing,
```

```
  aes(month, sales)) + geom_col()
```

```
c <- ggplot(economics, aes(date, psavert)) + geom_line()
```

enter/exit_*()

```
enter_exit_fade()
anim_a + enter_fade()

enter_grow()/exit_shrink()
anim_a + exit_shrink()

enter_exit_fly()
anim_a + enter_fly(x_loc = 0, y_loc = 0)

enter_exit_drift()
anim_a + exit_drift(x_mod = 3, y_mod = -2)

enter_exit_recolour() (or enter/exit_recolor())
anim_a + enter_recolour(color = "red")
```

Note: `enter/exit_*()` functions can be combined so that you can have old data fade away and shrink to nothing by adding `exit_fade()` and `exit_shrink()` to the plot.

Note: `enter/exit_*()` functions can be

combined so that you can have old data

fade away and

transition_*()

```
transition_states()
a + transition_states(color, transition_length = 3, state_length = 1)

transition_time()
b + transition_time(year, range = c(2002L, 2006L))

transition_reveal()
c + transition_reveal(date)

transition_filters()
a + transition_filter(transition_length = 3, filter_length = 1, cut == "ideal", Deep = depth >= 60)
```

We're cycling between values of color, ... and spending 3 times as long going to the next cut as we do pausing there.

We're cycling through each year of the data... from 2002 to 2006 (range is optional; default is the whole time frame). Unlike `transition_states()`, `transition_time()` treats the data as continuous and so the transition length is based on the actual values. Using 2002L instead of 2002 because the underlying data is an integer.

We're adding each date of the data on top of 'old' data

`transition_length` and `filter_length` work the same as `transition/state_length()` in `transition_states()`...

... but now we're cycling between these two filtering conditions. **Names** are optional, but can be useful (see "Label variables" on next page).

Other transitions

- `transition_manual()`: Similar to `transition_states()`, but without intermediate states.

- `transition_layers()`: Add layers (geoms) one at time.

- `transition_components()`: Transition elements independently from each other.

- `transition_events()`: Each element's duration can be controlled individually.

Baseline Animation

```
anim_a <- a +
  transition_states(color,
    transition_length = 3,
    state_length = 1)
```

Saving animations

view_*()

```
view_follow()
anim_a + view_follow(fixed_x = TRUE, fixed_y = c(2500, NA))

view_step()
anim_a + view_step(pause_length = 2, step_length = 1, nstep = 7)
```

x-axis shows full range, y shows [2500, as much is needed for that frame]. Default is for both axis to vary as needed.

We're spending twice as long moving between views as staying at them...

... and we're cycling between seven views. Seven is the number of steps in the transition, so the view is changing when the points are static, and visa versa. Views are determined by what data is in the current frame.

view_zoom()

`view_zoom()` works similarly to `view_step()`, except it changes the view by zooming and panning.

Note: both `view_step()` and `view_zoom()` have

`view*_manual()` versions for setting views

directly instead of inferring it from frame data

Label variables

```
anim_a + labs(subtitle = "Moving to {next_state}")
```

We're using the `next_state` label variable to tell the viewer where we're going.

Label variable	Description	Transitions
<code>transitioning</code>	TRUE if the current frame is an transition frame, FALSE otherwise	states, layers, filter
<code>previous_state/layer</code>	Last shown state/layer	states, layers
<code>next_state/layer</code>	State/layer that will be shown next	states, layers
<code>closest_state/layer</code>	State/layer that current frame is closest to (if between states/layers, either next or closest).	states, layers
<code>previous/closest/next_filter/expression</code>	Similar to their state/layer analogs. <code>_filter</code> variables return the name of the filter; <code>_expression</code> variables return the condition.	filter
<code>frame_time</code>	Time of current frame	time, components, events
<code>frame_along</code>	Current frame's value for the dimension we're transitioning over	reveal
<code>nlayers</code>	Number of layers (total, not just currently shown)	layer

gganimate's `transition_*()` functions create label variables you can pass to (sub)titles and other labels

with the glue package. For example, `transition_states()` has `next_state`, which is the name of the

state the animation is transitioning towards. Label variables are different between

transitions, and details are included in the documentation of each.

shrink to nothing by adding `exit_fade()` and `exit_shrink()` to the plot

shadow_*

```
shadow_wake()
anim_a + shadow_wake(wake_length = 0.05)
```

Points have a wake of points with the data from the last 5% of frames.

```
shadow_trail()
anim_a + shadow_trail(distance = 0.05)
```

Animation will keep the points from 5% of the frames, spaced as evenly as possible.

```
shadow_mark()
anim_a + shadow_mark(color = "red")
```

Animation will keep past states plotted in red (but not the intermediate frames).

```
animation_to_save <- anim_a +
  exit_shrink()
anim_save("first_saved_animation.gif", animation = animation_to_save)
```

Since the `animation` argument uses your last rendered animation by default, this also works:

```
anim_a + exit_shrink()
anim_save("second_saved_animation.gif")
```

`anim_save()` uses `gifski` to render the animation as a `.gif` file by default. You can use the `renderer` argument for other output types including video files (`av_renderer()` or `ffmpeg_renderer()`) or spritesheets (`sprite_renderer()`):

- # requires you to have the `av` package installed

```
anim_save("third_saved_animation.mp4", renderer = av_renderer())
```

view_*

view_zoom()

`view_zoom()` works similarly to `view_step()`, except it changes the view by zooming and panning.

Note: both `view_step()` and `view_zoom()` have `view_*_manual()` versions for setting views directly instead of inferring it from frame data



By **Pranav V A**
cheatography.com/pranav-v-a/

Published 26th September, 2020.
Last updated 26th September, 2020.
Page 1 of 2.

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
<http://crosswordcheats.com>