Post02: Image Processing and Manipulation - Play with Images in R Using Magick

Demi Liu

12/03/2017

Introduction and Motivation

Up to now, we've seen many different useful graphic functions and packages in R, however, we rarely mentioned how to process those images. One motivation for me writing this post is knowing accidentally that an awesome collage was actually created by R (I will show you the example, so please keep reading), and one way to realize that is by using the `Magick` package. Plus, after coding interactive elements within ShinyApp, I really want to combine magick images in ShinyApps. And after doing some research, I found out that the easiest way to do so is to write image to a `tempfile()` within the `renderImage()` callback function.

So in this post, I will introduce the `Magick` package, some images **transformation** (cut, edit, filter, kernel convolution), **image vectors** (animation, animated graphics), some beautiful **examples**, and how to use `Magick` with Rmarkdown and **ShinyApps** in R.

(Note: This post assumes that you have basic knowledge in graphic packages and ShinyApp in R.)

Background Information of `Magick`

The `Magick` package is a useful tool to modernize and simplify high-quality image processing in R. The `ImageMagick` library has an overwhelming amount of functionality. The current version of Magick exposes a decent chunk of it, but being a first release, documentation is still sparse. This post briefly introduces the most important concepts to get started with `Magick` pacakge.

Installing Requried Packages

```
install.packages("magick")
```

```
library(magick)
```

Read the image

Images can be read directly from a file path, URL, or raw vector with image data with `image_read`. In the following exmaple, we'll be using an image of Cal bear.

```
#Read the image
library(magick)
```

```
## Linking to ImageMagick 6.9.6.6
## Enabled features: cairo, fontconfig, freetype, pango, rsvg, webp
## Disabled features: fftw, ghostscript, lcms, x11
```

```
campanile <- image_read('https://raw.githubusercontent.com/DemiLZZ/post02/master/campanile.png')
campanile
```

Image Transformations

Now we will see some image transformation applications with `Magick`.

Cut and edit images

```
#Cut images
image_crop(campanile, "100x150+50")
```

```
#Rescale images
image_scale(campanile, "x30")
```

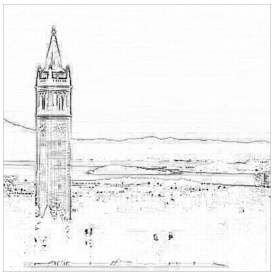
```
image_scale(campanile, "300")
```

```
#Rotate images
image_rotate(campanile, 45)
```

Filters and Effects

```
# Add randomness
image_blur(campanile, 10, 5)
```

```
# Silly filters
image_charcoal(campanile)
```

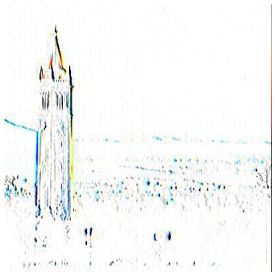


```
image_negate(campanile)
```

Kernel Convolutions

The `image_convolve()` function applies a kernel over the image. Kernel convolution means that each pixel value is recalculated using the weighted neighborhood sum defined in the kernel matrix.

```
#Standard kernels
campanile %>% image_convolve('Sobel') %>% image_negate()
```



```
campanile %>% image_convolve('DoG:0,0,2') %>% image_negate()
```

Image Vectors

The examples above concern single images. However all functions in `magick` have been vectorized to support working with layers, compositions or animation.

The standard base methods `[]`, `c()` and `length()` are used to manipulate vectors of images which can then be treated as layers or frames.

Let us look at an example of a gif first.

```
#Read the gif earth
earth <- image_read('https://raw.githubusercontent.com/DemiLZZ/post02/master/earth.gif')
earth
```

```
#The length of earth gif
length(earth)
```

```
## [1] 44
```

```
#Acsess some of the information of earth gif
head(image_info(earth))
```

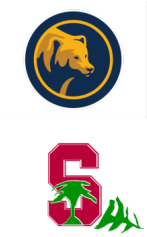
```
##   format width height colorspace filesize
## 1    GIF   400    400       sRGB  1429302
## 2    GIF   400    400       sRGB  1429302
## 3    GIF   400    400       sRGB  1429302
## 4    GIF   400    400       sRGB  1429302
## 5    GIF   400    400       sRGB  1429302
## 6    GIF   400    400       sRGB  1429302
```

```
#Some manipulation of earth gif: reverse the gif and add notes
rev(earth) %>%
  image_flip() %>%
  image_annotate("meanwhile in Australia", size = 20, color = "white")
```

Layers

We can combine layers next to each other we would in Photoshop:

```
#Put images next to each other
stanfurd <- image_read('https://raw.githubusercontent.com/DemiLZZ/post02/master/stanfurd.png')
berkeley <- image_read('https://raw.githubusercontent.com/DemiLZZ/post02/master/calbear.png')
img <- c(berkeley, stanfurd)
img <- image_append(img, stack = TRUE)
img <- image_scale(img, "400x400")
img
```



Animations

Instead of treating vector elements as layers, we can also make them frames in an animation.

Morphing (`image_morph`) creates a sequence of n images that gradually morph one image into another.

```
#Gradually change into another logo through animation
newlogo <- image_scale(image_read("https://www.r-project.org/logo/Rlogo.png"), "x150")
oldlogo <- image_scale(image_read("https://developer.r-project.org/Logo/Rlogo-3.png"), "x150")
frames <- image_morph(c(oldlogo, newlogo), frames = 10)
image_animate(frames)
```

We can also read an existing GIF or video file, in which each frame becomes a layer.

```
image_info(frames)
```

```
##    format width height colorspace filesize
## 1     PNG   197    150       sRGB        0
## 2     PNG   197    150       sRGB        0
## 3     PNG   196    150       sRGB        0
## 4     PNG   196    150       sRGB        0
## 5     PNG   196    150       sRGB        0
## 6     PNG   196    150       sRGB        0
## 7     PNG   195    150       sRGB        0
## 8     PNG   195    150       sRGB        0
## 9     PNG   195    150       sRGB        0
## 10    PNG   195    150       sRGB        0
## 11    PNG   194    150       sRGB        0
## 12    PNG   194    150       sRGB        0
```

Manipulate the individual frames and put them back into an animation:

```
#Choose the background
logo <- image_read("https://www.r-project.org/logo/Rlogo.png")
background <- image_background(image_scale(logo, "200"), "white", flatten = TRUE)

#Choose the foreground, which could be a gif file
banana <- image_read('https://raw.githubusercontent.com/DemiLZZ/post02/master/banana.gif')
banana <- image_scale(banana, "150")

# Combine and flatten frames
newframes <- image_apply(banana, function(frame) {
  image_composite(background, frame)
})

# Turn frames into animation
animation <- image_animate(newframes, fps = 10)
print(animation)
```

```
##   format width height colorspace filesize
## 1    gif   200    155       sRGB        0
## 2    gif   200    155       sRGB        0
## 3    gif   200    155       sRGB        0
## 4    gif   200    155       sRGB        0
## 5    gif   200    155       sRGB        0
## 6    gif   200    155       sRGB        0
## 7    gif   200    155       sRGB        0
## 8    gif   200    155       sRGB        0
```

Animated Graphics

The graphics device supports multiple frames which makes it easy to create animated graphics. The code below shows how you would implement an example similar to the Gapminder World animation, using the package `gapminder` for the data.

We first make sure we have the data we need by calling `install.packages('gapminder')`.

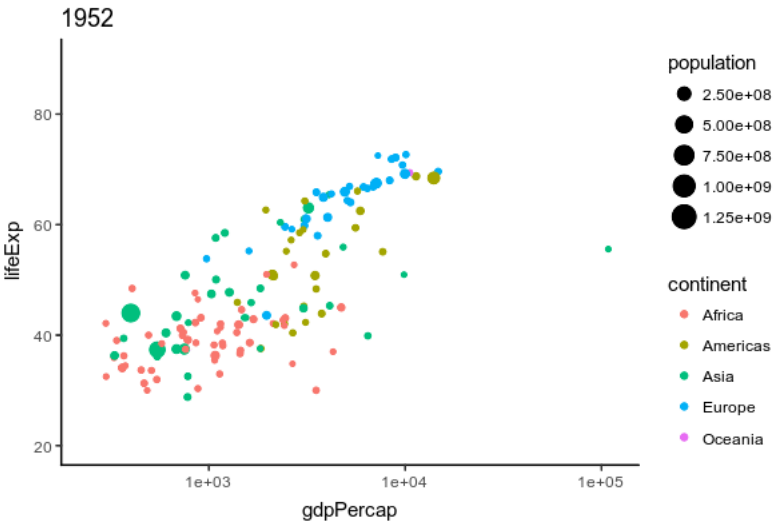
The core of the approach is to treat “frame” (as in, the time point within an animation) as another aesthetic, just like **x**, **y**, **size**, **color**, or so on. Thus, a variable in the data can be mapped to frame just as others are mapped to x or y.

```
#We will need both gapminder and ggplot2 to get the map animation
library(gapminder)
library(ggplot2)

#Produce an image
map <- image_graph(600, 400, res = 96)

#split divides the data in the vector gapminder into the groups defined by the data of years in gapminder
datalist <- split(gapminder, gapminder$year)

#lappy returns a list of the sample length as datalist, each element of which is the result of applying the functi
on data to the corresponding element of x
out <- lapply(datalist, function(data){
  p <- ggplot(data, aes(gdpPercap, lifeExp, size = pop, color = continent)) +
    scale_size("population", limits = range(gapminder$pop)) + geom_point() + ylim(20, 90) +
    scale_x_log10(limits = range(gapminder$gdpPercap)) + ggtitle(data$year) + theme_classic()
  print(p)
})
dev.off()
```

```
## quartz_off_screen
##                 2
```

```
map <- image_background(image_trim(map), 'white')

#And now we animate the map
animation <- image_animate(map, fps = 2)
print(animation)
```

```
##    format width height colorspace filesize
## 1     gif   600    400       sRGB        0
## 2     gif   600    400       sRGB        0
## 3     gif   600    400       sRGB        0
## 4     gif   600    400       sRGB        0
## 5     gif   600    400       sRGB        0
## 6     gif   600    400       sRGB        0
## 7     gif   600    400       sRGB        0
## 8     gif   600    400       sRGB        0
## 9     gif   600    400       sRGB        0
## 10    gif   600    400       sRGB        0
## 11    gif   600    400       sRGB        0
## 12    gif   600    400       sRGB        0
```

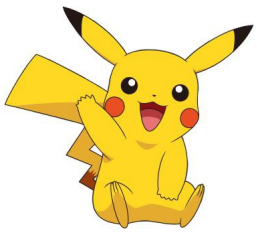


Application of Magick within ShinyApps

Now that we've seen the versatile applications of `magick` package, I am going to show you how to apply it further into ShinyApps. By combining with the interative elements provided by ShinyApps, `magick` can display information in a more direct and easily visualized way.

First, we read the image we are going to insert into the ShinyApp.

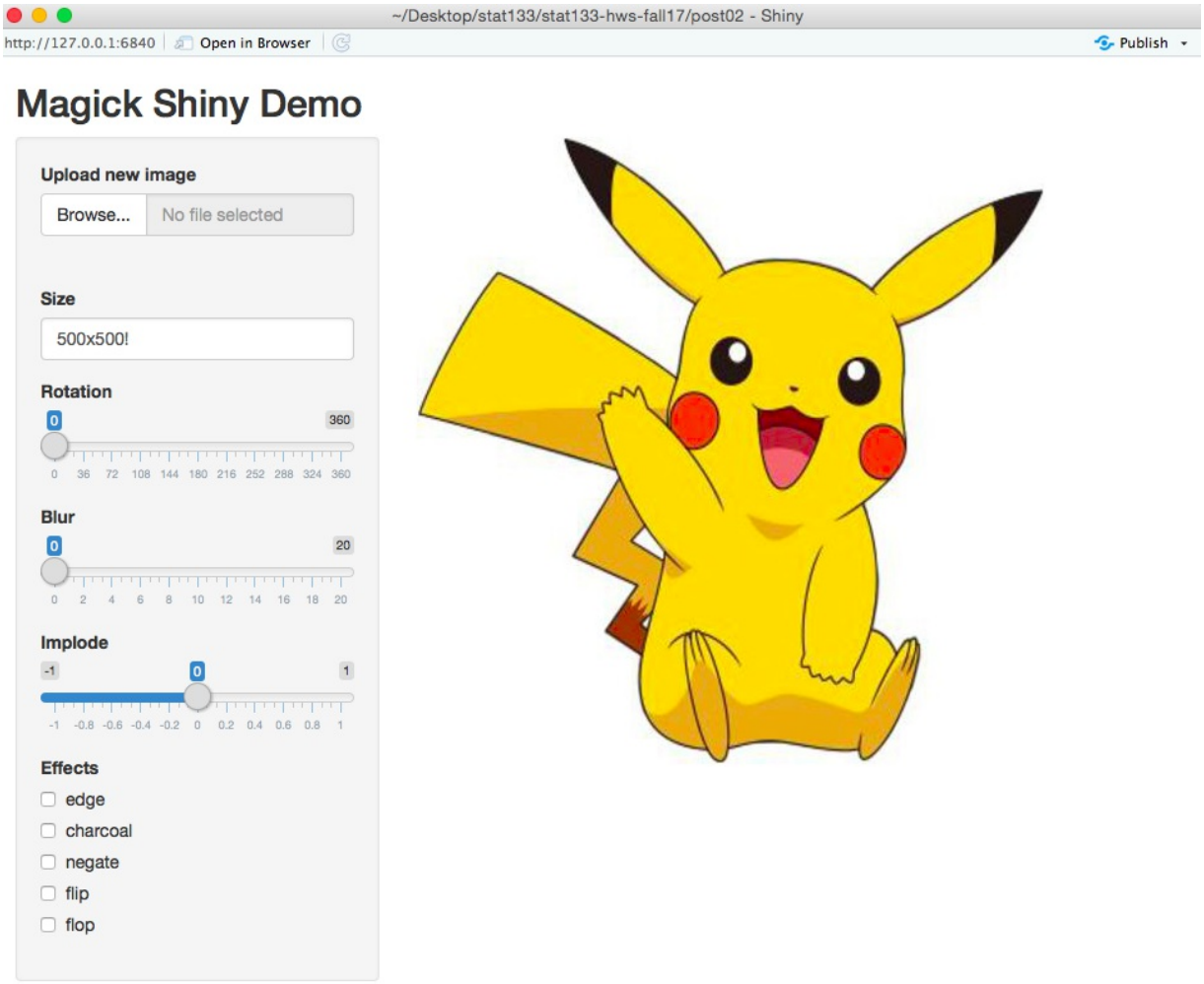
```
#Read the pikachu image
pikachu <- image_read('https://raw.githubusercontent.com/DemiLZZ/post02/master/pikachu.png')
pikachu
```



And the most convinient way doing so is writing `tempfile()` within the `renderImage()` callback function. The code for running the ShinyApp is attached below:

```
  # Minimal example of Shiny widget using 'magick' images
library(magick)
library(shiny)
app <- shinyApp(
ui = fluidPage(
  titlePanel("Magick Shiny Demo"),

  sidebarLayout(

    sidebarPanel(

      fileInput("upload", "Upload new image", accept = c('image/png', 'image/jpeg')),
      textInput("size", "Size", value = "500x500!"),
      sliderInput("rotation", "Rotation", 0, 360, 0),
      sliderInput("blur", "Blur", 0, 20, 0),
      sliderInput("implode", "Implode", -1, 1, 0, step = 0.01),

      checkboxGroupInput("effects", "Effects",
                         choices = list("edge", "charcoal", "negate", "flip", "flop"))
    ),
    mainPanel(
      imageOutput("img")
    )
  )
),

server = function(input, output, session) {

  # Start with placeholder img
  pikachu <- image_read('https://raw.githubusercontent.com/DemiLZZ/post02/master/pikachu.png')

  # When uploading new image
  observeEvent(input$upload, {
    if (length(input$upload$datapath))
      pikachu <<- image_convert(image_read(input$upload$datapath), "jpeg")
    info <- image_info(pikachu)
    updateCheckboxGroupInput(session, "effects", selected = "")
    updateTextInput(session, "size", value = paste0(info$width, "x", info$height, "!"))
  })

  # A plot of fixed size
  output$img <- renderImage({

    # Boolean operators
    if("edge" %in% input$effects)
      pikachu <- image_edge(pikachu)

    if("charcoal" %in% input$effects)
      pikachu <- image_charcoal(pikachu)

    if("negate" %in% input$effects)
      pikachu <- image_negate(pikachu)

    if("flip" %in% input$effects)
      pikachu <- image_flip(pikachu)

    if("flop" %in% input$effects)
      pikachu <- image_flop(pikachu)

    # Numeric operators
    tmpfile <- pikachu %>%
      image_resize(input$size) %>%
      image_implode(input$implode) %>%
      image_blur(input$blur, input$blur) %>%
      image_rotate(input$rotation) %>%
      image_write(tempfile(fileext='jpg'), format = 'jpg')

    # Return a list
    list(src = tmpfile, contentType = "image/jpeg")
  })
}
)

runApp(app)
```

If you run the ShinyApp, you will see an application of different image effects of Pikachu



Unfortunately, R cannot display ShinyApp within the html file, but you can always check and play with the Pikachu Magick ShinyApp by browsing my R script to write the App here.

Discussion

`Magick` is an advanced graphics and image-processing package in R, and it is the most comprehensive open-source image processing library available. It supports many common formats (png, jpeg, tiff, pdf, etc) and manipulations (rotate, scale, crop, trim, flip, blur, etc). All operations are vectorized via the Magick++ STL, meaning that they operate either on a single frame or a series of frames for working with layers, collages, or animation. In RStudio images are automatically previewed when printed to the console, resulting in an interactive editing environment. And the interactive environment `magick` creates can be combined with ShinyApp to create a more visible and easily-controlled user environment for image processing and manipulation.

Conclusion and Take-Home-Messages

As we've seen above, `magick` is a super powerful tool with diverse applications.

And some features of `magick` include:

- Format conversion: convert an image from one format to another (e.g. PNG to JPEG).
- Transform: resize, rotate, crop, flip or trim an image. (Applies these without generation loss on JPEG files, where possible.)
- Transparency: render portions of an image invisible.
- Draw: add shapes or text to an image.
- Decorate: add a border or frame to an image.
- Special effects: blur, sharpen, threshold, or tint an image.
- Animation: assemble a GIF animation file from a sequence of images.
- Text & comments: insert descriptive or artistic text in an image.
- Image identification: describe the format and attributes of an image.
- Composite: overlap one image over another.
- Montage: juxtapose image thumbnails on an image canvas.
- Generalized pixel distortion: correct for, or induce image distortions including perspective.
- Morphology of shapes: extract features, describe shapes and recognize patterns in
- Motion picture support: read and write the common image formats used in digital film work.

- Image calculator: apply a mathematical expression to an image or image channels.
- Discrete Fourier transform: implements forward and inverse DFT.
- Color management: accurate color management with color profiles or in lieu of – built-in gamma compression or expansion as demanded by the colorspace.
- High-dynamic-range images: accurately represent the wide range of intensity levels found in real scenes ranging from the brightest direct sunlight to the deepest darkest shadows.
- Encipher or decipher an image: convert ordinary images into unintelligible gibberish and back again.
- Virtual pixel support: convenient access to pixels outside the image region.
- Large image support: read, process, or write mega-, giga-, or tera-pixel image sizes.
- Threads of execution support: ImageMagick is thread safe and most internal algorithms execute in parallel to take advantage of speed-ups offered by multi-core processor chips.
- Heterogeneous distributed processing: certain algorithms are OpenCL-enabled to take advantage of speed-ups offered by executing in concert across heterogeneous platforms consisting of CPUs, GPUs, and other processors.
- Distributed pixel cache: offload intermediate pixel storage to one or more remote servers.
- ImageMagick on the iPhone: convert, edit, or compose images on your iOS computing device such as the iPhone or iPad.

We covered some of them, but there are still many more waiting for us to explore.

References

Magick Package on Wiki

The Gapminder

Magicak Pacakge on CRAN

Advanced Image-Processing in R with Magick

Using Magick with Shiny

Image Processing and Manipulation with magick in R

Advanced Image-Processing in R