CREx introduction

Lorenz Weise

September 25, 2020

Contents

What is CREx?

The CREx package for R on Windows OS's provides R functions to create windows, draw simple geometric shapes as well as images, play sounds, draw text and gather keyboard and mouse reactions. These functions allow programming cognitive experiments and gather behavioral data using R. Internally, the CREx functions call compiled C-functions, which are distributed along with the CREx package in dynamic-link library files. These C-functions provide limited access to the SDL2 library (https://www.libsdl.org/).

Programming experiments with CREx

The following examples give a brief illustration of how to use CREx.

Installing CREx

The easiest way to install CREx is by using the *remotes* package's *install_github()* function:

```
install_github("lorweiuk/CREx")
```

Alternatively, you can download CREx at https://github.com/lorweiuk/CREx. To install, run:

```
install.packages("path-to-CREx-folder", repos = NULL, type = "source")
```

Opening a window

To open and close a window:

```
library(CREx)
# arguments to CRopen() indicate position of window (x,y) and size (width,height)
gs <- CRopen(x = 50, y = 50, w = 900, h = 600)
CRclose(gs)
```

CRopen() returns a list containing the window's addresses and other important properties (the "*graphics structure*"). These must often be passed to other CREx functions, so make sure to assign them when *CRopen()* returns. Also note that the screen's coordinate systems has its origin in the upper-left corner. That means a rectangle with a y-coordinate of 0 sits at the top of the screen, not at the bottom.

Drawing to the window

To draw a rectangle to the window:

```
gs <- CRopen(x = 50, y = 50, w = 900, h = 600)
# set current color to black
CRset_render_color(gs, color_list = list(r = 0, g = 0, b = 0))
CRrender_clear(gs) # clear window
# set current color to red
CRset_render_color(gs, color_list = list(r = 200, g = 0, b = 0))
# draw filled rectangle at (x=200, y=200) of size (width=100, height=50)
CRfill_rect(gs, rect_list = list(x = 200, y = 200, w = 100, h = 50))
# present everything to the screen
CRrender_present(gs)
# wait 3000 milliseconds
CRwait_ms(3000)
CRclose(gs)
```

This code illustrates several conventions in CREx. Colors are specified via *color_lists*, which are simply lists with elements named r, b and g (and optionally a). Rectangles are specified via *rect_lists*, which are lists with elements x, y, w and h (indicating the position of the rectangle's top-left corner (x, y) and its width and height (w, h). Before drawing something to the window, it should be cleared to some color using *CRrender_clear()*. The color is first set via *CRset_render_color().* After having drawn our stimuli to the screen, we call *CRrender_present()* to show what we have drawn in the window. To make sure the window does not immediately disappear, we call *CRwait_ms()* to wait a certain number of milliseconds.

The following is a table of functions that allow the user to draw. If both a *CRdraw…* and *CRfill…* function exist for a certain shape, the draw function will only draw the shape's outline, while the fill function will also fill it. Some functions have "singular" and "plural" versions (e.g. *CRdraw_line()* and *CRdraw_lines()*). The plural version allows drawing several of the shapes at once.

Table 1. Drawing functions.

CRdraw_circle( )	CRfill_circle( )
CRdraw_circle_n( )	CRfill_circle_n( )
CRdraw_circles( )	CRfill_circles( )
CRdraw_image( )	
CRdraw_line( )	
CRdraw_lines( )	
CRdraw_point( )	
CRdraw_points( )	
CRdraw_rect( )	CRfill_rect( )
CRdraw_rects( )	CRfill_rects( )
CRdraw_text( )	
CRdraw_texture( )	
	CRfill_polygon( )

Choice-RT task

To run a simple choice-RT task:

```
gs <- CRopen(x = 50, y = 50, w = 900, h = 600)
num_trials <- 20 # number of trials

# keycode 'f'; can be looked up at https://wiki.libsdl.org/SDLKeycodeLookup
left_button  <- 102
right_button <- 106

# condition indicates target side
conditions <- sample(c("L", "R"), size = num_trials, replace = T)

# response data structure (column 1: responses; column 2: RTs)
responses <- matrix(NaN, nrow = num_trials, ncol = 2)

# define stimulus rectangles
left_rect  <- list(x = gs$width/2 - 100, y = gs$height/2 - 25, w = 50, h = 50)
right_rect <- list(x = gs$width/2 +  50, y = gs$height/2 - 25, w = 50, h = 50)

# define colors
green_color <- list(r = 0, g = 200, b = 0)
red_color   <- list(r = 200, g = 0, b = 0)
bg_color    <- list(r = 70, g = 70, b = 70)

# trial loop
for (trial in 1:num_trials) {
   # inter-trial-interval
   CRset_render_color(gs, bg_color)
   CRrender_clear(gs) # clear background
   CRrender_present(gs)
   CRwait_ms(1500)

   # draw everything
   CRrender_clear(gs) # clear background
```

```
    CRset_render_color(gs, green_color)
    CRfill_rect(gs, if (conditions[trial] == "L") left_rect else right_rect) # draw target rectangle

    CRset_render_color(gs, red_color)
    CRfill_rect(gs, if (conditions[trial] == "L") right_rect else left_rect) # draw non-target rectangle

    stim_time <- CRrender_present(gs) # put stimuli on window

    # response loop
    r_found <- FALSE

    while (!r_found) {
      respout <- NULL

      while (!is.null(respout <- CRpoll_event(gs))) {
        if (respout$type == "key_down" &&
        (respout$code == left_button || respout$code == right_button)) {
          # left- or right-button response found
          responses[trial, ] = c(respout$code, respout$timestamp - stim_time)
          # save response code and RT
          r_found = TRUE
        }
      }

      CRwait_ms(1) # avoid hot loop
    }
}

CRclose(gs)
```

There is a lot happening in this example, but it illustrates most of what is needed to program a simple experiment. In this task, per-trial two rectangles are shown, one on the left and one on the right. One of them is green, the other is red. The participant is supposed to press the ‘f’ key if the green rectangle is on the left, and the ‘j’ key if it is on the right. Each trial’s response and reaction time (RT) is stored.

First, a window is opened. Then, a number of variables and vectors are defined that dictate what the experiment is supposed to do. For example, the key codes of the f- and j-keys are stored in

variables. Key codes can be looked up at https://wiki.libsdl.org/SDLKeycodeLookup. The *conditions* vector consists of 20 letters, each of which tells you whether the current trial is a left-target or right-target trial. We use the *responses* matrix to store the participant's behavioral data, with column 1 for the responses and column 2 for RTs.

Next, we define the positions for the left rectangle and right rectangle. Until now we always indicated positions in absolute pixel values, but now we start defining them relative to the window: the graphics structure *gs* holds the width and height of the window, which we can retrieve via *gs$width* and *gs$height*. Thus, a position of (x = gs$width / 2, y = gs$height / 2) sits at the center of our window. We want the left rectangle to sit left of the window's center, so we set its x-coordinate to (x = gs$width / 2 – 100). The subtracted amount (100) is dictated by the rectangles width, which we set to 50, and the space we want to see between the two rectangles. We also move the rectangles up a bit (y = gs$height / 2 – 25) so that their center sits at the center of the screen. Before we start the task, we also define the experiment's colors (green, red, and a grey background).

Next comes the *trial loop*. This loop is the central part of most experiments. It loops over our trials, and inside we place the code that executes on each trial. Everything that needs to repeat on every trial – and everything that changes on a trial-by-trial basis – goes in the trial loop. The loops control variable is called *trial*, and we can use it to keep track of which trial we are currently in. A major part of the loop consists of drawing the current trial's stimuli. First we clear the background, then show the resulting empty screen, and wait 1.5 seconds (our inter-trial interval). Next, we clear the window again, draw our two rectangles, and show them in the window. Note that the rectangles' positions passed to *CRfill_rect()* are put inside a conditional statement:

```
if (conditions[trial] == "L") left_rect else right_rect
```

Thus, on left-target trials – in which *conditions[trial] == "L"* is TRUE – the left_rect list is returned. This makes sense, because we are currently drawing the green rectangle, which on left-target trials should be … on the left. When we draw the red rectangle immediately afterwards, the expressions returning the rectangles are reversed. Finally, when we call *CRrender_present()*, we assign the result to *stim_time*. This result is an estimate of when the stimulus appeared on the screen. If you want to compute RTs, they will usually be relative to this time point, so you should keep track of it. Note that this kind of time measurement (when did my stimulus actually appear on the screen?) is not nearly as precise as many people believe. If your project depends on an exact measurement of this time point – the time of physical retinal stimulation from the screen – there is no real way around physically measuring it yourself.

After having presented our two rectangles, we enter the *response loop*. This is the second major part of our trial loop. This loop's task is to keep checking whether a response was given, in our case whether the f- or j-key was hit. This loop actually consists of two nested loops. The outer loop keeps running until a response was found (that is, until *r_found* is TRUE). The inner loop keeps running until

```
respout <- CRpoll_event(gs)
```

returns *NULL* (that is, until there are no more responses waiting to be processed). The function *CRpoll_event(gs)* simply retrieves the next response currently waiting and stores it in *respout*.

If this is not *NULL*, we process it. Should there be no responses left, our outer loop still continues and we keep checking, but we first make a very short break:

```
CRwait_ms(1) # avoid hot loop
```

During processing the current response in *respout*, we check its *type*, which we hope to be *key_down*. This indicates that a keyboard key was pressed. We also check *respout*'s *code* element, which is the *key_code* of the current response. As defined at the beginning of the script, we want this *key_code* to be either *102* (f-key) or *106* (j-key). If any of these conditions is met, we note down the behavioral data by storing the *code* and the RT. The RT is computed as the *timestamp* element of *respout* minus the *stim_time*. Finally, we set *r_found* to TRUE to end the outer loop.

Images and textures

The following example illustrates how to load and present an image, and how to use textures to pre-render your stimuli.

```
gs <- CRopen(50, 50, 900, 600) # open window
image_file <- file.choose() # choose an image to load
image_list <- CRload_image(list(), image_file) # load image
texture_list <- CRcreate_texture(gs, list(), gs$width, gs$height) # create a texture to draw to
CRset_render_target(gs, texture_list, 1) # set this texture as rendering target
CRset_render_color(gs, list(r = 70, g = 70, b = 70)) # clear background to grey
CRrender_clear(gs)

# draw image to texture
CRdraw_image(gs, image_list, 1, list(x = gs$width / 2 - 250, y = gs$height / 2 - 250, w = 500, h = 500))
CRallow_alpha(gs) # allow using alpha values when specifying colors

for (i in 1:300) {
   # draw 300 random filled circles
   CRset_render_color(gs, list(r = runif(1, 1, 255), g = runif(1, 1, 255), b = runif(1, 1, 255), a = runif(1,1,
255)))
   temp <- list(centerx = runif(1, 1, gs$width), centery = runif(1, 1, gs$height), radius = runif(1, 5, 30))
   CRfill_circle_n(gs, temp, 12)
}

CRreset_render_target(gs) # reset render target to window
CRdraw_texture(gs, texture_list, 1) # draw texture to window
CRrender_present(gs)
CRwait_ms(5000)
CRclose(gs, texture_list, image_list)
```

What happens here is not very interesting from an experimental point of view, but it illustrates how *CRload_image()* and *CRdraw_image()* can be used to present images to the window. *CRload_image()* takes as arguments an *image_list*, to which the loaded image is appended, and a string specifying the path to the image file. If the *image_list* provided is simply an empty list, the function's output will be a new *image_list* with the loaded image as the only element.

Furthermore, the use of textures is illustrated. Drawing all your stimuli to the window takes a certain amount of time. This might appear to happen very fast, but when using the more

resource-heavy drawing functions (*CRfill_circles()*, *CRdraw_text()*), when having to re-draw your stimuli every frame – because they are moving, for example – and when having to compute a lot of other things every frame as well, drawing your stimuli to the window might take too long. In these cases, textures come in handy. You can create a texture using *CRcreate_texture()*: you provide a *texture_list* as input (again, this can be an empty list) as well as the intended dimensions of the texture (width and height). The returned *texture_list* has your new texture appended to it. To draw to this texture, you first need to call *CRset_render_target(gs, texture_list, 1)*, where the "1" is an index specifying the position of your intended texture in the *texture_list*. After resetting your window as the rendering target using *CRreset_render_target(gs)*, you can draw your prepared texture to the screen in one go using *CRdraw_texture(gs, texture_list, 1)*, where the "1" is again an index into your texture_list. The advantage this affords is that you can prepare your stimuli during less computationally intensive periods of the task, such as the inter-trial interval. The function *CRdraw_texture()* also accepts other arguments, such as a rotation angle.

The example also illustrates how *CRallow_alpha()* is called in order to allow the use of transparency for your stimuli, by specifying an additional element "a" in your *color_lists*. Lastly, if you have allocated additional resources during your task (such as images or textures), you simply pass the lists to *CRclose()* to de-allocate them.

Sound and text

The following example illustrates how to play sound and draw text.

```
gs <- CRopen(50, 50, 700, 500)
CRinit_audio() # initialize audio system
wave_file <- file.choose()
wave_list <- CRload_wav(list(), wave_file) # load audio
font_file <- file.choose()
font_list <- CRload_font(list(), font_file, 20) # load font
CRset_render_color(gs, list(r = 70, g = 70, b = 70)) # clear background to grey
CRrender_clear(gs)
CRset_render_color(gs, list(r = 200, g = 200, b = 200))
CRdraw_text(gs, font_list, 1, "PLAY", list(x = 50, y = 100))
CRdraw_circle(gs, list(centerx = 50, centery = 100, radius = 25))
CRdraw_text(gs, font_list, 1, "PAUSE", list(x = 50, y = 200))
CRdraw_circle(gs, list(centerx = 50, centery = 200, radius = 25))
CRdraw_text(gs, font_list, 1, "RESUME", list(x = 50, y = 300))
CRdraw_circle(gs, list(centerx = 50, centery = 300, radius = 25))
CRdraw_text(gs, font_list, 1, "END", list(x = 50, y = 400))
CRdraw_circle(gs, list(centerx = 50, centery = 400, radius = 25))
CRrender_present(gs) # show everything

end_clicked <- FALSE

while (!end_clicked) {
   respout <- NULL

   while (!is.null(respout <- CRpoll_event(gs))) {

      if (respout$type == "mouse_button_down" &&
      abs(respout$x - 50) < 50 && abs(respout$y - 100) < 50) {
         # Play button clicked
         CRplay_wav(wave_list, 1)

      } else if (respout$type == "mouse_button_down" &&
```

```
      abs(respout$x - 50) < 50 && abs(respout$y - 200) < 50) {
        # Pause button clicked
        CRaudio_pause()

      } else if (respout$type == "mouse_button_down" &&
      abs(respout$x - 50) < 50 && abs(respout$y - 300) < 50) {
        # Resume button clicked
        CRaudio_resume()

      } else if (respout$type == "mouse_button_down" &&
      abs(respout$x - 50) < 50 && abs(respout$y - 400) < 50) {
        # End button clicked
        end_clicked = TRUE
      }
    }
    CRwait_ms(1) # avoid hot loop
}

CRclose(gs, wave_list, font_list) # close all resources
```

CRload_wav() takes as arguments a (possibly empty) *wav_list*, as well as a string giving the path to a wav file. Note that *CRinit_audio()* must be called before loading and playing sounds. *CRload_font()* takes as arguments a (possibly empty) *font_list*, as well as a string specifying the path to a font file, and the intended font size. Both the *wav_list* and the *font_list* should eventually be passed to *CRclose()* to be de-allocated. To draw text to the window, call *CRdraw_text()*, with the arguments being the graphics structure *gs*, the *font_list*, an index ("1") specifying which font from the *font_list* to use, the text to draw, and a *point_list* giving the point at which to draw the text. *Point_lists* must have elements x and y.

To play, pause and resume sound, use *CRplay_wav()*, *CRaudio_pause()* and *CRaudio_resume()*. To change the sound's volume, you can use *CRaudio_volume()*. In the response loop, the script checks which sound option is clicked by the mouse. This is simply done by checking if a mouse click occurred – that is, checking for response type "*mouse_button_down*". If this happens, the response loop checks whether the position of the mouse (*respout* elements *x* and *y*) was within 50 pixels of any of the audio options. If this was the case, the appropriate action is taken.

Limitations

The CREx package is provided as-is, without any guarantees regarding its functionalities, the results obtained when using it or its general reliability. I have used most of the functions in CREx for several years now to quickly and easily program cognitive experiments, and have found them useful and reliable. However, I make no promises when CREx is run on other systems, and there has been absolutely no testing it on a wide variety of systems. Furthermore, much of CREx is just a (limited) way to make use of the Simple Direct Media Library (SDL2) in R. If you find it useful, much of that is thanks to the people developing SDL2.