CREx: Programming psychological experiments in R

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:

```
library(remotes)

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)

# x and y are position of window's upper-left corner
# w and h give width and height of window

gs <- CRopen(x = 50, y = 50, w = 900, h = 600)

CRwait_ms(1000)

CRclose(gs)
```

CRopen() returns a list containing the window's addresses and other important properties (the "graphics-list"). 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 system 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 100x50

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 for alpha values). 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( )
CRdraw_polygon()    CRfill_polygon( )
```

Choice-RT task

To run a simple choice-RT task:

```
gs <- CRopen(x = 50, y = 50, w = 900, h = 600)

num_trials <- 10 # number of trials

# keycode 'f' and 'j'

left_button <- utf8ToInt("f")

right_button <- utf8ToInt("j")

# 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)
# draw target rectangle
if (conditions[trial] == "L") {
   CRfill_rect(gs, left_rect)
} else if (conditions[trial] == "R") {
   CRfill_rect(gs, right_rect)
}
CRset_render_color(gs, red_color)
# draw non-target rectangle
if (conditions[trial] == "L") {
   CRfill_rect(gs, right_rect)
} else if (conditions[trial] == "R") {
   CRfill_rect(gs, left_rect)
}
stim_time <- CRrender_present(gs) # put stimuli on window
# response loop
respfound <- FALSE
respout <- NULL
while (!respfound) {
   while (!is.null(respout <- CRpoll_event(gs))) {
      if (respout$type == "key_down") {
         if (respout$code == left_button || respout$code == right_button) {
            # left- or right-button pressed
            responses[trial, ] = c(respout$code, respout$timestamp - stim_time)
```

```
                respfound <- TRUE

                CRflush(gs)
            }
        }

        CRwait_ms(1) # avoid hot loop

    }
  }
}

CRclose(gs)
```

There is a lot happening in this example: 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 using CRopen(). Then, a number of variables and vectors are defined that dictate what the experiment is supposed to do. For example, the Utf-8 key codes of the f- and j-keys are stored in variables. The conditions vector consists of 10 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 and right rectangle. We define these rectangles relative to the window: the graphics structure gs holds the width and height of the window. We also define the task's three colors: green, red, and the grey background color.

What follows is the trial loop, which executes once for each of the 10 trials. First the window is cleared to the background color and the task holds for 1500 ms (the inter-trial-interval). We then draw the red and green rectangles. Which color rectangle is shown on the left and right is goverend by the conditions-vector.

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. Note that this kind of time measurement is not 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.

The last major part of our trial loop is the response loop. This loop's task is to keep checking whether a response was given, in our case whether the f- or j-key was pressed. This loop actually consists of two nested loops. The outer loop keeps running until a response was found (that is, until respfound is TRUE). The inner loop keeps running until there are no more events waiting to be processed. When no events are waiting, CRpoll_event() will return NULL, in which case the loop will not execute.

When an event is waiting, the function CRpoll_event() will retrieve it and store it in respout. 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 the Utf-8 code for 'f' or 'j'. 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 respfound to TRUE to end the outer loop and flush all remaining events, so that our next trial does not start by processing some outdated events.

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 <- choose.files(caption = "Choose image file.") # choose an image to load

image_list <- CRload_image(list(), image_file) # load image

# create a texture to draw to

texture_list <- CRcreate_texture(gs, list(), gs$width, gs$height)

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

image_rect <- list(x = gs$width / 2 - 250, y = gs$height / 2 - 250, w = 500, h = 500)

CRdraw_image(gs, image_list, 1, image_rect)

CRset_blend_mode(gs, "blend") # use alpha values when rendering

for (i in 1:300) {

 # draw 300 random filled circles in random colors

 rnd_color <- list(r = runif(1, 1, 255),
                   g = runif(1, 1, 255),
                   b = runif(1, 1, 255),
                   a = runif(1, 1, 255))

 CRset_render_color(gs, rnd_color)

 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(texture_list, image_list, gs)
```

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 you use the more resource-heavy drawing functions (CRfill_circles(), CRdraw_text(), and especially CRdraw_image()), when you re-draw your stimuli every frame – because they are moving, for example – and when you have to compute a lot of other things every frame as well, drawing your stimuli to the window can take too long for fluent presentation. 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 which texture in the texture_list you want to set as rendering target. From now on, draw_ and fill_ functions (e.g. CRfill_circle_n()) will do their work on the texture, not the screen.

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). The function CRdraw_texture() also accepts other arguments, such as a rotation angle.

The example also illustrates how CRset_blend_mode() 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.

```
# define button positions

play <- list(x = 50, y = 100)

pause <- list(x = 50, y = 200)

resume <- list(x = 50, y = 300)

end <- list(x = 50, y = 400)

gs <- CRopen(50, 50, 700, 500)

CRinit_audio() # initialize audio system

# let used choose files

wav_file <- choose.files(caption = "Choose wav-file.")
```

```
wav_list <- CRload_wav(list(), wav_file) # load audio

font_file <- choose.files(caption = "Choose ttf-file.")

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", play)

CRdraw_circle(gs, list(centerx = play$x, centery = play$y, radius = 25))

CRdraw_text(gs, font_list, 1, "PAUSE", pause)

CRdraw_circle(gs, list(centerx = pause$x, centery = pause$y, radius = 25))

CRdraw_text(gs, font_list, 1, "RESUME", resume)

CRdraw_circle(gs, list(centerx = resume$x, centery = resume$y, radius = 25))

CRdraw_text(gs, font_list, 1, "END", end)

CRdraw_circle(gs, list(centerx = end$x, centery = end$y, 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") {

         mouse_pos <- c(respout$x, respout$y)

         if (dist(rbind(mouse_pos, c(play$x, play$y))) < 50) {

            # Play button clicked
            CRplay_wav(wav_list, 1, 0)

         } else if (dist(rbind(mouse_pos, c(pause$x, pause$y))) < 50) {

            # Pause button clicked
            CRaudio_pause(0)

         } else if (dist(rbind(mouse_pos, c(resume$x, resume$y))) < 50) {
```

```
            # Resume button clicked
            CRaudio_resume(0)

         } else if (dist(rbind(mouse_pos, c(end$x, end$y))) < 50) {

            # End button clicked
            end_clicked = TRUE

         }
      }

      CRwait_ms(1) # avoid hot loop

   }
}

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

First we prepare everything by defining some button positions (play, pause, resume, end), opening a window and initializing the audio system (CRinit_audio). CRload_wav() takes as arguments a (possibly empty) wav_list, as well as a string giving the path to a wav file. 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 closed.

On the screen we draw four circles, which the user can click to: 1) play the sound; 2) pause the played sound; 3) resume the sound; and 4) end the program. To draw text next to these circles, we 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.

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. To play, pause and resume sound, we use CRplay_wav(), CRaudio_pause() and CRaudio_resume(). To change the sound's volume, you can use CRaudio_volume(). If the option "end" was clicked, we set end_clicked to TRUE, which stops the loop.

Error handling via status

CREx uses an optional error system via the status-argument that all functions accept. If the status-argument is not provided to a function by the user, the function will at most print out a warning to the console if something goes wrong, but otherwise return no error information. If the user wants to be able to react to errors, for example by saving data, a user-create list with elements "value" and "message" can be passed as the status-argument. In the event of an error, CREx functions will set the value-element to 0 and copy an error-message into the message-element of status. Otherwise, value is set to 1 and message is empty. The value-element can subsequently be used to react to errors.

The following code gives a brief example of how errors can be handled using the status-argument:

```
s <- list(value = 1, message = "")

gc <- CRopen_controller(status = s)
```

```
if (s$value == 1) {

  print("No error.")

} else if (s$value == 0) {

  print(s$message)

}
```

If error-checking is limited to functions that load- or create resources - such as CRopen(), CRcreate_texture(), CRload_image(), CRopen_controller() etc. - an alternative way is via the valid-element of the functions' output.

```
gc <- CRopen_controller()

if (gc$valid == 1) {

  print("No error.")

} else {

  print("Controller not valid!")

}
```

Limitations

The CREx package is provided as-is, without any warranty or implied warranties regarding its functionalities, the results obtained when using it, its general reliability or fitness for a particular purpose. In no event will the author be held liable for any damages arising from the use of this software. 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. 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.