Today’s [puzzle](https://coord.info/GC606J1) is a single part of a mystery that comprises several smaller puzzles, styled on the cult classic TV show [The Crystal Maze](https://en.wikipedia.org/wiki/The_Crystal_Maze). To get the coordinates there is a “mental” test for the northings, two easy puzzles that I won’t go into here. The westings are the more interesting part, with a “skill” test.

The skill test is a **whack-a-mole** game, with 60 radio buttons. One radio button is highlighted at random and you need to click it. If you hit, you get a point, and if you miss, you lose a point. After 30 seconds, if your score is 30 or more, you are presented with the coordinates. Now, there’s more than one way to skin this particular cat, and one of them doesn’t require any fast fingers *or* making an R programme. But I’ll stay silent on that method and instead use the opportunity to demonstrate how sometimes R can be enhanced by invoking scripts in other languages.

R seems to lack a way to read information about pixels on the screen. It’s easy to do if you’re interested in pixels within an image, but in this case we’re not. I’d like to **read the pixels on the screen** to detect which radio button is lit, then direct the mouse to click there. So I cribbed a small python script from somewhere or other to plug the gap of detecting the colour:

def get\_pixel\_colour(i\_x, i\_y):

import win32gui

i\_desktop\_window\_id = win32gui.GetDesktopWindow()

i\_desktop\_window\_dc = win32gui.GetWindowDC(i\_desktop\_window\_id)

long\_colour = win32gui.GetPixel(i\_desktop\_window\_dc, i\_x, i\_y)

i\_colour = int(long\_colour)

return (i\_colour & 0xff), ((i\_colour >> 8) & 0xff), ((i\_colour >> 16) & 0xff)

The reticulate package allows us to take a python function and “source” it, loading it as though it is an R function. It’s really painless. KeyboardSimulator can send key presses and mouse clicks to the OS. And those two packages do most of the hard work. Here’s the code:

library(KeyboardSimulator)

library(reticulate)

library(magrittr)

## the source\_python takes a few seconds to run, and we're against the clock in this game

## so the first run will stop after it's in memory, and the second run will ignore this block

if (exists("runBefore")==0) {

runBefore <- 1

source\_python("C:/Users/alunh/OneDrive/Documents/Repos/geo2/getPixelColour.py")

stop("python script loaded!")

}

## the position of the radio buttons on my screen

w <- 26

h <- 21

l <- 1324

t <- 290

## the target radio button is dark grey and the non-targets are pale.

## Choose r+g+b < 500 to identify the target

start <- Sys.time()

repeat {

for (x in sample(10)) {

for (y in sample(6)) {

## here's the python function!

p <- get\_pixel\_colour(as.integer(l+w\*(x-1)), as.integer(t+h\*(y-1))) %>%

unlist()

if (sum(p) < 500) {

mouse.move(l + w \* (x-1), t+h \* (y-1))

mouse.click()

}

}

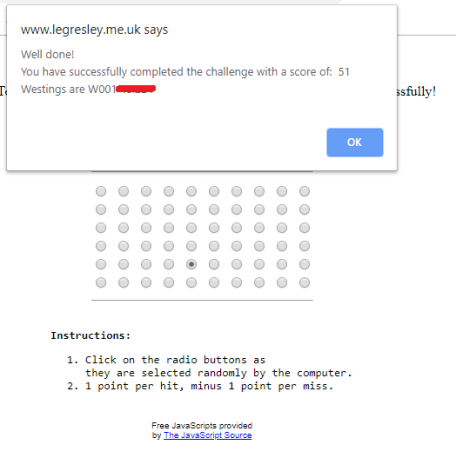
}

## quit after 32 seconds

if ((Sys.time() - start) > 32) break

}

So how did our R/python hybrid do? Remember, the target was 30 correct clicks within 30 seconds, without errors. Using a mouse, I managed **23**, and using a touchscreen device I made it to **34**.

(Final coordinates redacted)

**51!** Not too shabby! Doubtless something bigger and better could be done to improve this score, but I’m very satisfied with my first foray into sourcing python from R.

It’s not often that R has a gap in its functionality, and there might well be a way of doing this without resorting to a python script. Please let me know if you have a better way by using the comment section below.