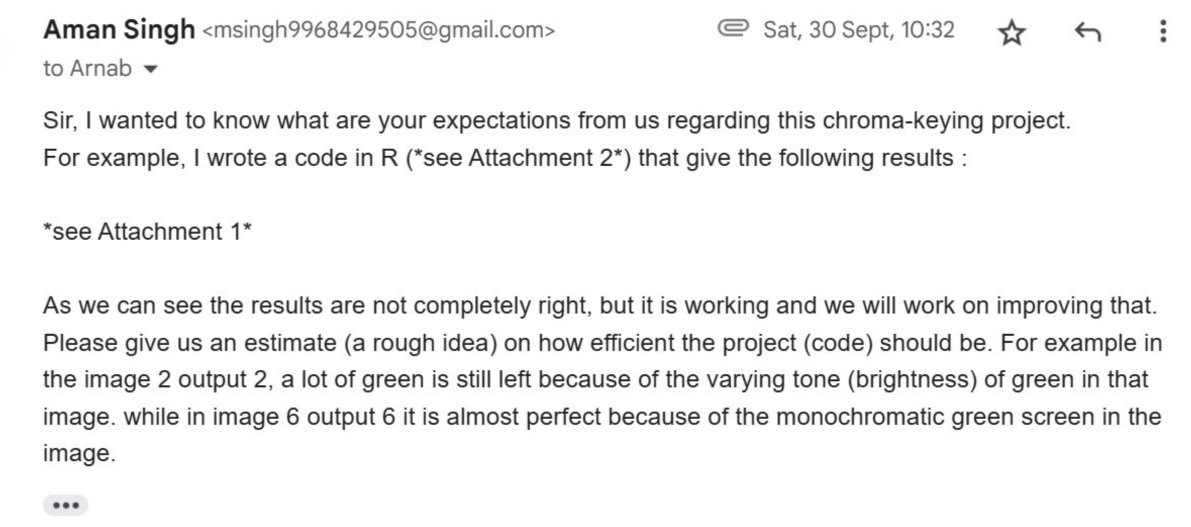
here is the exact topic:

Chroma-keying: See this fun video to understand chroma-keying or green-screening as it is popularly called. Your task will be to work with still photos with a green (or whatever colour you wish) background, and to remove it keeping the foreground as intact as possible. The photos used MUST be taken by the project group (not downloaded).

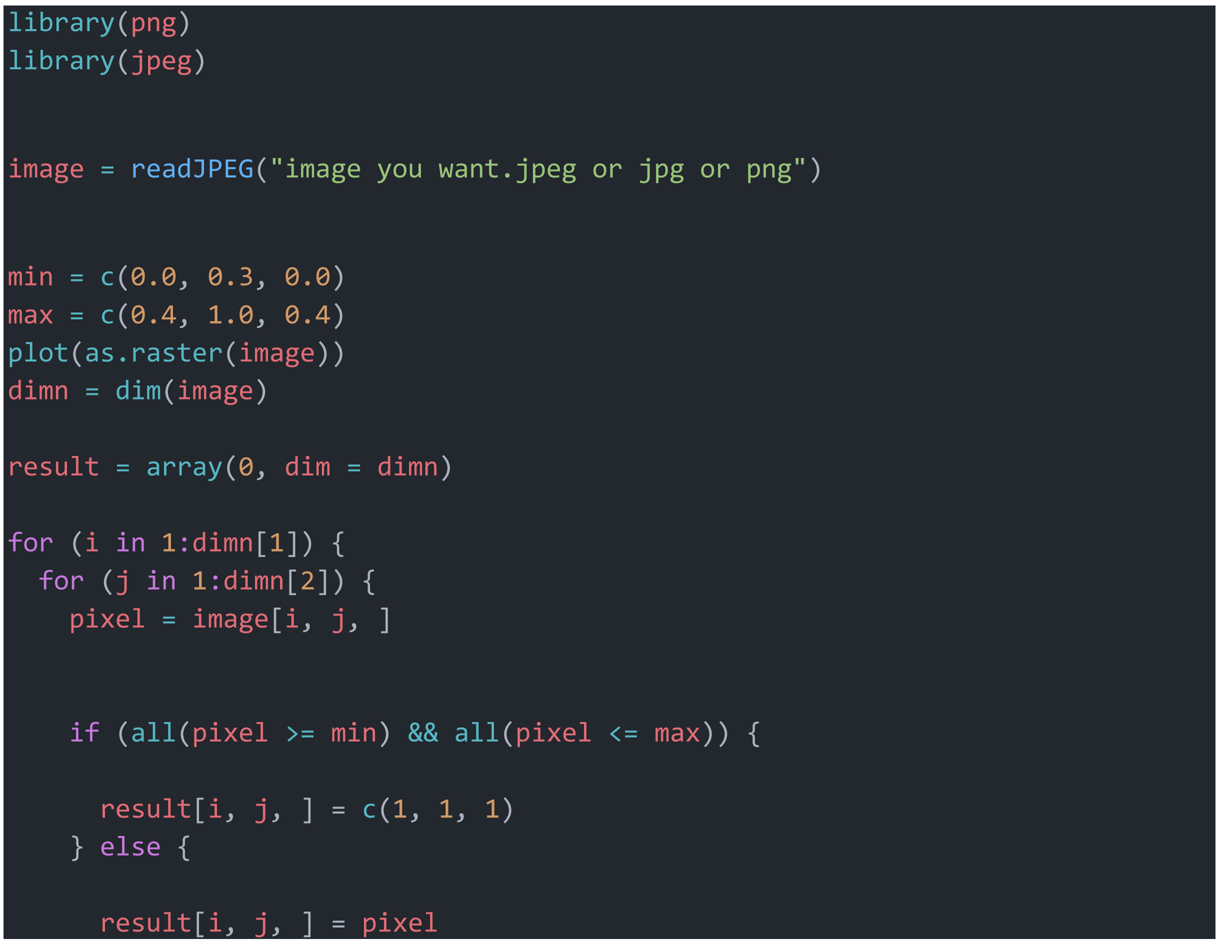


Aman Singh <msingh9968429505@gmail.com> Sat, 30 Sept, 10:32
to Arnab

Sir, I wanted to know what are your expectations from us regarding this chroma-keying project.
For example, I wrote a code in R (*see Attachment 2*) that give the following results :

see Attachment 1

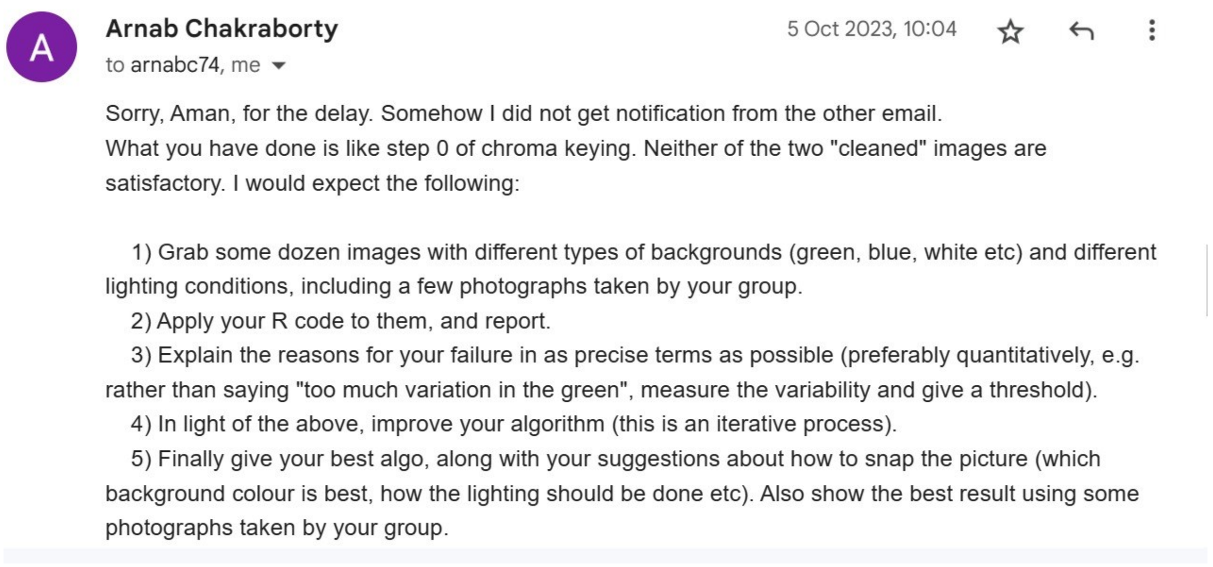
As we can see the results are not completely right, but it is working and we will work on improving that. Please give us an estimate (a rough idea) on how efficient the project (code) should be. For example in the image 2 output 2, a lot of green is still left because of the varying tone (brightness) of green in that image. while in image 6 output 6 it is almost perfect because of the monochromatic green screen in the image.



```
library(png)
library(jpeg)


image = readJPEG("image you want.jpeg or jpg or png")


min = c(0.0, 0.3, 0.0)
max = c(0.4, 1.0, 0.4)
plot(as.raster(image))
dimn = dim(image)

result = array(0, dim = dimn)

for (i in 1:dimn[1]) {
  for (j in 1:dimn[2]) {
    pixel = image[i, j, ]


    if (all(pixel >= min) && all(pixel <= max)) {

      result[i, j, ] = c(1, 1, 1)
    } else {

      result[i, j, ] = pixel
    }
  }
}

plot(as.raster(result))
```

Input-output images for the code above :

https://drive.google.com/file/d/1vZC-61YrwwMSBx4Bh8IKpKTJLlJKAyq_/view?usp=sharing



Arnab Chakraborty 5 Oct 2023, 10:04
to arnabc74, me

Sorry, Aman, for the delay. Somehow I did not get notification from the other email.
What you have done is like step 0 of chroma keying. Neither of the two "cleaned" images are satisfactory. I would expect the following:

1) Grab some dozen images with different types of backgrounds (green, blue, white etc) and different lighting conditions, including a few photographs taken by your group.
2) Apply your R code to them, and report.
3) Explain the reasons for your failure in as precise terms as possible (preferably quantitatively, e.g. rather than saying "too much variation in the green", measure the variability and give a threshold).
4) In light of the above, improve your algorithm (this is an iterative process).
5) Finally give your best algo, along with your suggestions about how to snap the picture (which background colour is best, how the lighting should be done etc). Also show the best result using some photographs taken by your group.

APPROACHES :

Approach 1 : range/threshold

Your approach of using a loop to iterate through each pixel and checking if the RGB components fall within a certain range of green values can work as a basic method for green screen removal, similar to chroma keying. This approach is essentially creating a threshold for green, and if a pixel's color values fall within this threshold, you can replace them with a new background.

PROBLEMS

Precision: Green screen removal often requires high precision, as lighting conditions and variations in the green background can cause the green color to vary slightly. You might need to account for different shades of green to ensure that you don't accidentally remove non-background green elements in the image.

Edge handling: Dealing with the edges of the subject in the foreground can be challenging. You might need to apply a feathering or smoothing technique to blend the edges of the subject with the new background seamlessly.

Alpha matting and feathering are two different techniques used to address the issue of smooth blending between the subject and the new background in the context of green screen removal or chroma keying. They serve slightly different purposes, and you can choose one or both, depending on your specific needs and the look you want to achieve.

*1. **Alpha Matting:***

- Alpha matting is a technique that assigns an alpha (transparency) value to each pixel in the image. It's particularly useful for creating a detailed, pixel-level transparency map that accurately represents the subject's boundaries.

- To implement alpha matting, you need an additional matte image that specifies the transparency of each pixel. This matte can be generated by algorithms such as the closed-form matting method.

- With an alpha matte, you can precisely control the transparency of the subject's edges, enabling a more accurate and realistic blending with the new background.

*2. **Feathering:***

- Feathering is a technique used to create a gradual transition or soft edge between the subject and the background. It involves blending the subject's edges with the background by applying a gradient or smooth transition.

- Feathering is simpler to implement compared to alpha matting and is often used when a more gradual, less precise blending effect is desired.

- You can achieve feathering by applying a Gaussian or radial blur to the edges of the subject or by manually painting a gradient mask to create the desired transition.

In summary, if you need a highly detailed and precise blending of the subject with the background, especially for complex subjects with intricate details and transparency, alpha matting is a suitable choice. However, it can be computationally expensive and may require additional data (the alpha matte).

Feathering is a simpler and computationally efficient technique for creating a soft transition between the subject and the background. It's often used for quick and visually pleasing results when precise detail isn't required.

You can also combine both techniques by using alpha matting for precise transparency and then applying feathering to create a gradual transition, providing the best of both worlds. Your choice depends on your specific use case and the visual result you want to achieve.

Color spill: In some cases, green light can spill onto the subject, especially if the subject is close to the green screen. You might need to account for this color spill and adjust the subject's colors accordingly.

Color decontamination and color correction are two distinct image processing techniques, each serving a different purpose in the context of green screen removal or chroma keying.

*1. **Color Decontamination:***

- Color decontamination is a technique used to remove unwanted color spill or contamination from the subject when it has taken on a color cast from the green screen background. This commonly happens when the subject is positioned very close to the green screen, and green light bounces off the screen onto the subject.

- The goal of color decontamination is to identify the greenish tint on the subject and neutralize it by removing or adjusting the green color without affecting other colors in the image. This helps in achieving a more natural and color-balanced result.

- Techniques for color decontamination often involve isolating the green spill and applying color balance or correction algorithms to counteract the green tint.

*2. **Color Correction:***

- Color correction is a broader process used to adjust and enhance the overall color balance and appearance of an image. It may involve adjusting brightness, contrast, saturation, and color tones to make the image look more visually pleasing.

- In the context of green screen removal, color correction can be used to further refine the subject's appearance after the green screen background has been removed. It helps ensure that the subject's colors match well with the new background and the overall image looks cohesive.

In summary, while both color decontamination and color correction are valuable techniques, they have different purposes:

- Color decontamination is primarily focused on addressing the green color spill on the subject.

- Color correction is more about adjusting the overall color and appearance of the image to make it look its best in the context of the new background.

You might use color decontamination to deal with specific color contamination issues, and then apply color correction to ensure the entire image looks balanced and visually pleasing. The choice between the two depends on the specific needs of your image and the extent of color issues you encounter.

APPROACH 2

involves comparing the green component of each pixel to the red and blue components and removing pixels where the green component is significantly larger than the red and blue components, is a reasonable approach for green screen removal. It is based on the common understanding that green screen backgrounds are predominantly green, and the foreground subject should have a more balanced color distribution.

PROBLEMS :

1. **Green Balance Variation:** While it's generally true that the green background should have a higher green component, the precise green balance can vary due to factors like lighting and color calibration. You should allow for a certain level of flexibility in your algorithm to account for variations in the green background.
2. **Foreground Variations:** The foreground subject's colors can also vary, and in some cases, elements in the subject may have a higher green component than the background. This can happen if the subject contains green-colored objects or if there's green spill from the background onto the subject. You need to be careful not to unintentionally remove parts of the subject.
3. **Tolerance Levels:** Define tolerance levels or thresholds for comparing the green, red, and blue components. These thresholds should be adjustable to accommodate different images and lighting conditions. By allowing users to adjust these thresholds, you make your algorithm more versatile.

APPROACH 3

Your idea of removing pixels with colors that repeat the most, especially if they are predominantly green, is an interesting approach to green screen removal. It's based on the assumption that the green screen background is the most prevalent color in the image.

PROBLEMS :

1. **Colour Dominance:** This method relies on the assumption that the green background colour will dominate the image. However, there may be situations where the subject or other objects in the scene also have prominent colours. This could lead to unintended removal of parts of the subject or other important elements in the image.
2. **Variability in Background:** The effectiveness of this approach may depend on the uniformity of the green screen background. If the background is not consistently colored or contains variations, your algorithm might not accurately identify the background pixels.
3. **Color Preservation:** Be cautious about removing colors that are not part of the background but are still prevalent in the image. For example, if there are green elements in the subject's clothing or props, you wouldn't want to remove those. Or when some other color than green which need not to be removed is also highly frequent.

APPROACH 4

The approach you've described involves using connected components analysis to determine whether the green screen background has been successfully removed. If the number of connected components (distinct regions of pixels) is greater than a specific low threshold (e.g., 10), it indicates that there are still green screen elements present, and you iteratively remove the green until you meet a certain condition.

Here's a simplified step-by-step description of this approach:

1. Green Screen Removal:
   - **Initially, you remove the green screen background from the image using a method like color-based thresholding, as you've described earlier.**
2. Connected Components Analysis:
   - **Perform connected components analysis on the processed image to identify and count the distinct regions of pixels.**
3. Threshold Check:
   - **Check if the number of connected components exceeds a specific low threshold (e.g., 10). If it does, this indicates that some green screen elements may still be present in the image.**
4. Iterative Removal:
   - **If the threshold is exceeded, you can iteratively reapply the green screen removal process. This may involve refining the color-based removal method or applying more advanced techniques to target remaining green elements.**
5. Repeating the Process:
   - **Repeat the connected components analysis and threshold check after each removal iteration until the number of connected components falls below the specified threshold.**
6. Termination:
   - **Terminate the process when the number of connected components is within the threshold, indicating that the green screen background has been effectively removed.**

PROBLEMS :

1. **Computationally Intensive:** The iterative process can be computationally intensive, especially if you have to repeat the removal process multiple times. This can slow down the overall processing of images, making it less suitable for real-time applications.
2. **Dependent on Thresholds:** The effectiveness of the approach relies on choosing the right threshold for the number of connected components. Setting the threshold too high may lead to unnecessary iterations, while setting it too low could result in an incomplete background removal.

3. **Risk of Over-Processing:** Iteratively applying the green screen removal may lead to over-processing the image, potentially removing non-background elements that should be preserved.
4. **User Interaction:** Depending on the image and the green screen removal method used, user interaction may be needed to fine-tune the parameters or manually correct issues that arise during the iterative process.
5. **Sensitivity to Initial Removal:** The quality of the initial green screen removal is critical. If the initial removal is not accurate, the iterative process may not yield satisfactory results.
6. **Complexity:** The approach adds complexity to the removal process, making it more challenging to implement and maintain.
7. **No Guarantee of Success:** There is no guarantee that the iterative approach will always yield a successful result. Some images may not be amenable to this method, and additional techniques may be required.

POSSIBLE SOLUTIONS :

- Instead of a hard-coded RGB range, use a more flexible approach that involves statistical analysis of the green screen color in the image. Calculate the mean and standard deviation of the green color in the background, and then remove pixels that fall within a certain standard deviation range from the mean. This helps account for variations in lighting and different shades of green.
- Provide the user with the ability to fine-tune the removal process. Let users adjust parameters like color range, edge smoothness, and color spill correction. This gives users more control over the output and helps adapt to different image conditions.
- Test your algorithm on a variety of images with different lighting conditions and backgrounds to ensure it performs well in various scenarios.

TRY :

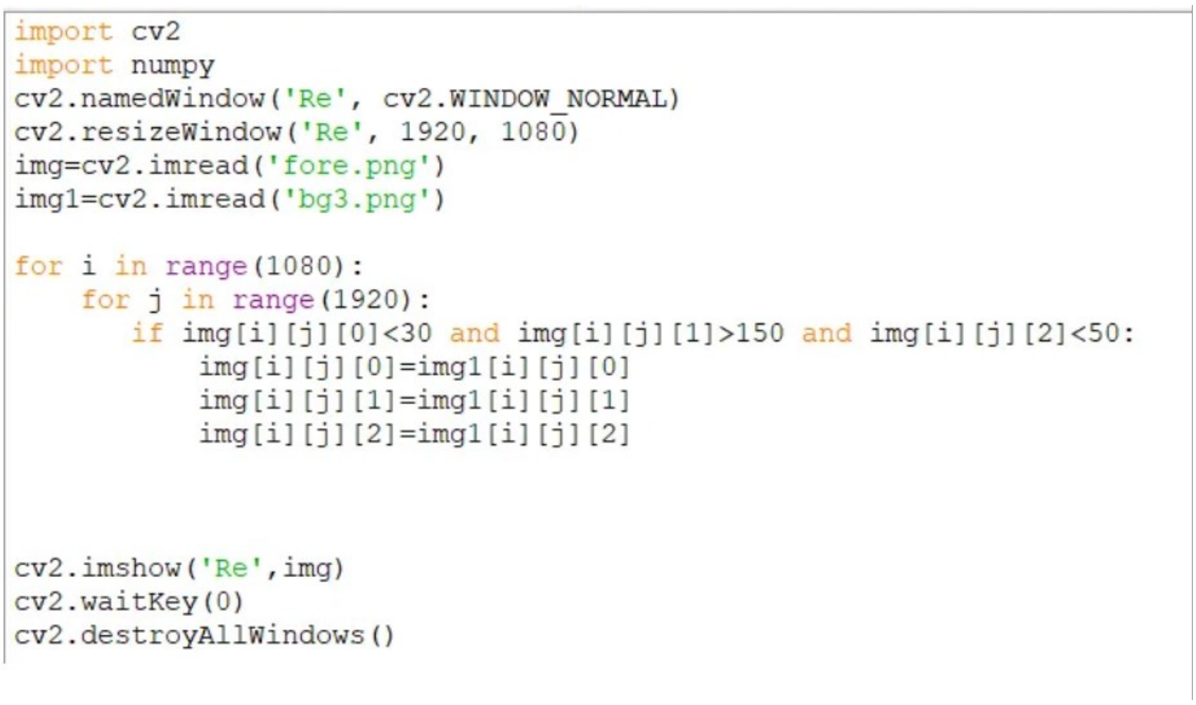
1. **Edge-based background removal-** This technique detects edges in the image and then finds a continuous edge path. All the elements that are outside the path are considered as background.
2. **Foreground detection-** Foreground detection is a technique that detects changes in the image sequence. The background subtraction method is used here to separate the foreground from an image.

3.

```
from PIL import Image
from PIL import ImageFilter
import os
for filename in os.listdir("."): # parse through file list in the current
directory
      if filename[-3:] == "png":
           img = Image.open(filename)
           img = img.convert("RGBA")
           pixdata = img.load()
           for y in xrange(img.size[1]):
                for x in xrange(img.size[0]):
                     r, g, b, a = img.getpixel((x, y))
                     if (r < 130) and (g < 60) and (b < 60):
                          pixdata[x, y] = (255, 255, 255, 0)
                     #Remove anti-aliasing outline of body.
                     if r == 0 and g == 0 and b == 0:
                          pixdata[x, y] = (255, 255, 255, 0)
           img2 = img.filter(ImageFilter.GaussianBlur(radius=1))
           img2.save(filename, "PNG")
```

The basic idea is to load each frame of the animation, loop through each pixel of each frame, and remove the color of the "green screen" background color, including a small range to account for the semi-transparent pixels on the edge of the rendered object that are mixed with the green screen. Finally, a small blurring filter is added to soften the edges of the image, otherwise at the edge where the green screen is removed, it looks more obvious that it was "cut out". As a final touch, after creating the sprite sheet you can use a program like GIMP to add some additional filters to adjust the color and contrast. You can also use GIMP to get the exact RGBA numbers to specify which color to filter out in your program.

4.



```python
import cv2
import numpy
cv2.namedWindow('Re', cv2.WINDOW_NORMAL)
cv2.resizeWindow('Re', 1920, 1080)
img=cv2.imread('fore.png')
img1=cv2.imread('bg3.png')

for i in range(1080):
    for j in range(1920):
       if img[i][j][0]<30 and img[i][j][1]>150 and img[i][j][2]<50:
           img[i][j][0]=img1[i][j][0]
           img[i][j][1]=img1[i][j][1]
           img[i][j][2]=img1[i][j][2]



cv2.imshow('Re',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

5.

```java
   greenToTransparency(image, imageOut);
   MarvinImageIO.saveImage(imageOut, "./res/person_chroma_out1.png");
   // 2. Reduce remaining green pixels
   reduceGreen(imageOut);
   MarvinImageIO.saveImage(imageOut, "./res/person_chroma_out2.png");
   // 3. Apply alpha to the boundary
   alphaBoundary(imageOut, 6);
   MarvinImageIO.saveImage(imageOut, "./res/person_chroma_out3.png");

}

private void greenToTransparency(MarvinImage imageIn, MarvinImage imageOut){
   for(int y=0; y<imageIn.getHeight(); y++){
      for(int x=0; x<imageIn.getWidth(); x++){

         int color = imageIn.getIntColor(x, y);
         int r = imageIn.getIntComponent0(x, y);
         int g = imageIn.getIntComponent1(x, y);
         int b = imageIn.getIntComponent2(x, y);

         double[] hsv = MarvinColorModelConverter.rgbToHsv(new int[]{color});

         if(hsv[0] >= 60 && hsv[0] <= 130 && hsv[1] >= 0.4 && hsv[2] >= 0.3){
            imageOut.setIntColor(x, y, 0, 127, 127, 127);
         }
         else{
            imageOut.setIntColor(x, y, color);
         }

      }
   }
}
```

```
    private void reduceGreen(MarvinImage image){
        for(int y=0; y<image.getHeight(); y++){
            for(int x=0; x<image.getWidth(); x++){
                int r = image.getIntComponent0(x, y);
                int g = image.getIntComponent1(x, y);
                int b = image.getIntComponent2(x, y);
                int color = image.getIntColor(x, y);
                double[] hsv = MarvinColorModelConverter.rgbToHsv(new int[]{color});

                if(hsv[0] >= 60 && hsv[0] <= 130 && hsv[1] >= 0.15 && hsv[2] > 0.15){
                    if((r*b) !=0 && (g*g) / (r*b) >= 1.5){
                        image.setIntColor(x, y, 255, (int)(r*1.4), (int)g, (int)(b*1.4));
                    } else{
                        image.setIntColor(x, y, 255, (int)(r*1.2), g, (int)(b*1.2));
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        new ChromaToTransparency();
    }
}
```

The step **1** simply converts green pixels to transparency. Basically it uses a filtering rule in the HSV color space.
As you mentioned, the hair and some boundary pixels presents colors mixed with green. To reduce this problem, in the step **2**, these pixels are filtered and balanced to reduce its green proportion
Finally, in the step **3**, a gradient transparency is applied to all boundary pixels. The result will be even better with high quality images.

This Java code is used to process an image with a green screen background and convert it to an image with transparency. The code achieves this in three main steps:

1. **Green to Transparency:**
   - The code loads an image, and for each pixel in the image, it checks if the pixel's color falls within a range of green hues and saturation and lightness values. If the pixel color is in the green screen range, it is replaced with a transparent pixel (0, 127, 127, 127), creating transparency for the green screen.

2. **Reduce Green:**
   - In the second step, the code further processes the image to reduce the remaining green pixels. It goes through each pixel in the image, and if the pixel color is in the green range, it adjusts the red and blue components to reduce the green influence. This step helps to minimize the green spill on the subject.

3. **Apply Alpha to the Boundary:**
   - Finally, the code applies alpha (transparency) to the boundary of the image. It checks each pixel and if it's in the green range, it increases the alpha channel (transparency) for the pixel and its nearby pixels, effectively smoothing the transition between the subject and the background.

The code saves the image at each step of the process, resulting in a sequence of images that demonstrate the gradual removal of the green screen background.

This code is useful for removing a green screen background from an image or video, making it transparent, and improving the overall quality of the image. It's a simple example of green screen removal using Java and the Marvin Framework.

6.

Approach:

1. Import all necessary libraries
2. Load the images or videos
3. Resize the images and the videos to the same size
4. Load the upper and lower BGR values of the green color
5. Apply the mask and then use bitwise_and
6. Subtract bitwise_and from the original green screen image

Check for matrix value 0 after subtraction and replace it by the second image. You get the desired results.

#CODE TO PRINT TO SEE A RGB(credit to ChatGPT)

```r
# Load the required libraries
library(shiny)
library(grDevices)
# Define the UI
ui <- fluidPage(
  titlePanel("RGB Color Plot"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("red_value", "Red Value", min = 0, max = 255, value = 0),
      sliderInput("green_value", "Green Value", min = 0, max = 255, value =
140),
      sliderInput("blue_value", "Blue Value", min = 0, max = 255, value = 0)
    ),
    mainPanel(
      plotOutput("color_plot")
    )
  )
)
```

```r
# Define the server logic
server <- function(input, output) {
  output$color_plot <- renderPlot({
    red_value <- input$red_value
    green_value <- input$green_value
    blue_value <- input$blue_value
    color_matrix <- matrix(
      rep(c(red_value, green_value, blue_value), each = 100),
      nrow = 100,
      ncol = 100
    )
    image(
      color_matrix,
      col = rgb(red_value, green_value, blue_value, maxColorValue = 255)
    )
    legend(
      "topright",
      legend = paste("RGB (", red_value, ", ", green_value, ", ", blue_value,
")", sep = ""),
      fill = rgb(red_value, green_value, blue_value, maxColorValue = 255)
    )
    title("RGB Color Plot")
  })
}
# Run the Shiny app
shinyApp(ui, server)
```

# CODE TO SIMULATE THE 'OG' CODE (credit to ChatGPT)

```r
library(shiny)
library(png)
library(jpeg)
library(grid)

# Define the user interface
ui <- fluidPage(
  titlePanel("Image Processing Simulation"),
  sidebarLayout(
    sidebarPanel(
      fileInput("image", "Select an Image (JPEG or PNG)"),
      sliderInput("minR", "Min Red Value", 0, 1, 0, 0.01),
      sliderInput("minG", "Min Green Value", 0, 1, 0.3, 0.01),
      sliderInput("minB", "Min Blue Value", 0, 1, 0, 0.01),
      sliderInput("maxR", "Max Red Value", 0, 1, 0.4, 0.01),
      sliderInput("maxG", "Max Green Value", 0, 1, 1, 0.01),
      sliderInput("maxB", "Max Blue Value", 0, 1, 0.4, 0.01),
      actionButton("processBtn", "Process Image")
    ),
    mainPanel(
      plotOutput("outputImage")
    )
  )
)

# Define the server logic
server <- function(input, output) {
  observeEvent(input$processBtn, {
    req(input$image)
    inFile <- input$image
    if (!is.null(inFile)) {
      ext <- tools::file_ext(inFile$name)
      if (ext %in% c("jpg", "jpeg", "png")) {
        minValues <- c(input$minR, input$minG, input$minB)
        maxValues <- c(input$maxR, input$maxG, input$maxB)
        processImage(inFile$datapath, minValues, maxValues)
      } else {
        showModal(modalDialog(
          title = "Invalid File",
          "Please select a valid JPEG or PNG image."
        ))
      }
    }
  })

  output$outputImage <- renderPlot({
    NULL  # Placeholder for the initial image display
  })

  processImage <- function(imagePath, minValues, maxValues) {
    image <- readJPEG(imagePath)
    dimn <- dim(image)
    result <- array(0, dim = dimn)

    for (i in 1:dimn[1]) {
      for (j in 1:dimn[2]) {
        pixel <- image[i, j, ]

        if (all(pixel >= minValues) && all(pixel <= maxValues)) {
          result[i, j, ] <- c(1, 1, 1)
        } else {
          result[i, j, ] <- pixel
        }
      }
    }

    # Display the processed image using grid graphics
    grid.raster(as.raster(result))
  }
}

# Run the Shiny app
shinyApp(ui, server)
```

# explaination of the code above (credit to ChatGPT):

The code provided is an example of a Shiny web application in R that allows users to interactively adjust the `min` and `max` values for red, green, and blue color channels to process an image.

*1. **Libraries**:*

- We load the necessary libraries, including `shiny` for building the web application and `png` and `jpeg` for image processing.

*2. **User Interface (UI)**:*

- The `ui` object defines the user interface of the Shiny app. It consists of a title panel and a sidebar layout.

- In the sidebar panel, users can:

- Select an image file (JPEG or PNG) using the `fileInput` widget.

- Adjust `min` and `max` values for the red, green, and blue color channels using `sliderInput` widgets.

- Click the "Process Image" button to apply the selected `min` and `max` values to the image.

- The main panel contains a `plotOutput` widget named "outputImage" where the processed image will be displayed.

*3. **Server Logic**:*

- The `server` function contains the server-side logic for the Shiny app.

- The `observeEvent` function is used to trigger an action when the "Process Image" button is clicked. It checks if a valid image file is selected, and if so, it retrieves the `min` and `max` values and calls the `processImage` function with these values.

- The `output$outputImage` function is defined to initially display a placeholder plot for the image.

*4. **Image Processing Function** (`processImage`):*

- The `processImage` function takes the file path of the selected image and the `min` and `max` values for the color channels as arguments.

- It reads the image, processes it according to the specified color thresholds, and then displays the processed image in the main panel.

*5. **Image Display**:*

- To display the processed image, the `grid.raster` function is used from the `grid` package. It converts the processed image array into a raster and displays it in the Shiny app.

*6. **Running the Shiny App**:*

- The Shiny app is launched with `shinyApp(ui, server)`, which combines the defined user interface (`ui`) and server logic (`server`) to create the interactive web application.

The Shiny app allows users to upload an image, set `min` and `max` color thresholds, and see the effect of their adjustments on the displayed image. When users click the "Process Image" button, the image processing function is called to apply the specified color thresholds and display the result in the main panel.

APPROACH 5 :

As noticed while simulating rgb color simulator, that any greenish colors pixel has more green component than that of red and blue, so here using a loop iterating over each pixel of the image, we remove those pixels while have higher green component that red and blue, also we gave user with a threshold that it can change where it applies a further and condition to remove the pixel when the green component in the pixel is higher that the chosen threshold
-added a green upper bound too to ensure the pixels that are mostly green but verry light side do not get removed/can also be handled (may solve color spill problem too)
THINK : it doesn't solve our sharp edges problem
THINK 2 : we should not remove the grey pixels that may have max green component as it is not that green but either grey, white or black, so we can add
THINK 3 : isn't it that if we remove the color spill completely, the edge sharpening problem will also be gone??

```
&&(abs(r - b) <= threshold3) &&(abs(r - g) <= threshold3) &&(abs(b - g) <=
threshold3)
but not working rather working opposite that is for >= , also shows different outputs
for 0 and any other positive value only
```

Oh, sorry it worked, threshold3 should be between 0 and 1 (x/252)

```
# Load required libraries
library(shiny)
library(png)
library(jpeg)
```

```
# Define the UI for the Shiny app
ui <- fluidPage(
    titlePanel("Chroma - Keying"),

    sidebarLayout(
        sidebarPanel(
            fileInput("image_file", "Select an Image"),
            sliderInput("threshold", "Green Lower Bound", min = 0, max = 1,
value = 0.28, step = 0.01),
            sliderInput("threshold2", "Green Upper Bound", min = 0, max = 1,
value = 0.97, step = 0.01),
            sliderInput("threshold3", "Grey Bound", min = 0, max = 1, value =
0, step = 0.005)
        ),

        mainPanel(
            imageOutput("original_image"),
            imageOutput("processed_image")
        )
    )
)

# Define the server for the Shiny app
server <- function(input, output) {
    observe({
        req(input$image_file)

        image <- readJPEG(input$image_file$datapath)

        # Function to process the image
        process_image <- function(image, threshold, threshold2, threshold3) {
            dimn <- dim(image)
            result <- array(0, dim = dimn)

            for (i in 1:dimn[1]) {
                for (j in 1:dimn[2]) {
                    pixel <- image[i, j, ]
                    r <- pixel[1]
                    g <- pixel[2]
                    b <- pixel[3]

                    if (
                        which.max(pixel) == 2 &&
                        g > threshold &&
                        g <= threshold2 &&
                        (abs(r - b) >= threshold3) &&
                        (abs(r - g) >= threshold3) &&
                        (abs(b - g) >= threshold3)
                    ) {
                        result[i, j, ] <- c(1, 1, 1)
                    } else {
                        result[i, j, ] <- pixel
                    }
                }
            }

            return(result)
        }

        output$original_image <- renderImage({
            list(src = input$image_file$datapath,
                 contentType = "image/jpeg",
                 width = "80%")
        })

        output$processed_image <- renderImage({
            processed <- process_image(image, input$threshold,
input$threshold2, input$threshold3)
            outfile <- tempfile(fileext = '.jpeg')
            writeJPEG(processed, target = outfile)

            list(src = outfile, contentType = "image/jpeg", width = "80%")
        })
    })
}

# Run the Shiny app
shinyApp(ui = ui, server = server)
```

decreased image sizes to 80% as outputs were overlapping.

EXPLANATION :

This code is for a simple Shiny app that performs chroma keying on an image. Chroma keying is a technique used in image and video editing to replace a specific color (usually green or blue) with another image or background.

Here's an explanation of the code:

*1. **Library Imports:** The necessary libraries, including Shiny, png, and jpeg, are loaded to perform image processing and create the Shiny app.*

2. **UI Definition:**

   - The app's user interface (UI) is defined within the `ui` variable.

   - The UI includes a title panel with the text "Chroma - Keying."

   - A sidebar layout is set up with input options for the user.

   - `fileInput` allows the user to select an image file.

   - `sliderInput` widgets provide options to adjust the chroma keying parameters:

     - "Green Lower Bound" (`threshold`) - sets the lower limit for green color to be replaced.

     - "Green Upper Bound" (`threshold2`) - sets the upper limit for green color to be replaced.

     - "Grey Bound" (`threshold3`) - sets a threshold for the difference between color channels to consider a pixel for replacement.

3. **Server Function:**

   - The server function is defined, which handles the app's logic.

   - The `observe` function is used to monitor changes in the user's input, specifically when an image file is selected.

4. **Image Processing Function:**

   - The `process_image` function takes the selected image and the user-defined thresholds as input.

   - It iterates through each pixel of the image and checks whether it should be replaced with a specified color. If the pixel falls within the green color range and satisfies the conditions based on the given thresholds, it is replaced with white (c(1, 1, 1)); otherwise, it remains unchanged.

5. **Rendering Original Image:**

   - The `renderImage` function generates the rendered output for the original image.

   - The selected image file is displayed with no changes, and it is wrapped in a `list` with image metadata.

6. **Rendering Processed Image:**

   - Another `renderImage` function generates the rendered output for the processed image.

   - The `process_image` function is called to replace the appropriate pixels, and the processed image is saved as a temporary file with a `.jpeg` extension.

*7. **Running the Shiny App:***

- The Shiny app is initiated with `shinyApp` using the defined UI (`ui`) and server (`server`).

In summary, this Shiny app allows users to upload an image and apply chroma keying by specifying the lower and upper bounds for green color and a threshold for color differences. The original and processed images are displayed in the main panel, with the processed image replacing green pixels that meet the specified conditions.

HERE IS AN APPROACH TO DEAL WITH EDGE SMOOTHNING : (NOT WORKING YET..)

```r
create_disc_se <- function(radius) {
  # Calculate the size of the square matrix to hold the disc
  size <- 2 * radius + 1
  # Create an empty matrix of zeros
  se <- matrix(0, nrow = size, ncol = size)
  # Calculate the coordinates of the center
  center <- radius + 1
  # Loop through the matrix and set values to 1 if they are inside the disc
  for (i in 1:size) {
    for (j in 1:size) {
      if ((i - center)^2 + (j - center)^2 <= radius^2) {
        se[i, j] <- 1
      }
    }
  }
  return(se)
}

# Example: Create a disc structuring element with a radius of 2
radius <- 2
se <- create_disc_se(radius)
print(se)




dilate <- function(binary_image, se) {
  result <- matrix(0, nrow = nrow(binary_image), ncol = ncol(binary_image))
  for (i in 1:nrow(binary_image)) {
    for (j in 1:ncol(binary_image)) {
      if (binary_image[i, j] == 1) {
        # Check the neighborhood defined by the structuring element
```

```r
          for (se_row in 1:nrow(se)) {
            for (se_col in 1:ncol(se)) {
              ni <- i + (se_row - 1)
              nj <- j + (se_col - 1)
              if (ni >= 1 && ni <= nrow(binary_image) && nj >= 1 && nj <=
ncol(binary_image)) {
                result[ni, nj] <- 1
              }
            }
          }
        }
      }
    }
  return(result)
}




erode <- function(binary_image, se) {
  result <- matrix(0, nrow = nrow(binary_image), ncol = ncol(binary_image))
  for (i in 1:nrow(binary_image)) {
    for (j in 1:ncol(binary_image)) {
      all_ones <- TRUE
      for (se_row in 1:nrow(se)) {
        for (se_col in 1:ncol(se)) {
          ni <- i + (se_row - 1)
          nj <- j + (se_col - 1)
          if (ni >= 1 && ni <= nrow(binary_image) && nj >= 1 && nj <=
ncol(binary_image)) {
            if (binary_image[ni, nj] == 0) {
              all_ones <- FALSE
              break
            }
          } else {
            all_ones <- FALSE
          }
        }
        if (!all_ones) {
          break
        }
      }
      if (all_ones) {
        result[i, j] <- 1
      }
    }
  }
  return(result)
}
```

This example will work with binary images (0 for background, 1 for the subject), and it will use a simple square structuring element.

1. **Dilation**:
   Dilation involves expanding the subject regions. You can iterate through the image and, for each pixel, check its neighborhood defined by the structuring element. If any of the pixels in the neighborhood are 1, set the central pixel to 1 as well.

Erosion:
Erosion involves shrinking the subject regions. Similar to dilation, you iterate through the image and, for each pixel, check its neighborhood. If all the pixels in the neighborhood are 1, set the central pixel to 1; otherwise, set it to 0.

The `dilate` and `erode` functions are performing basic morphological operations on binary images using a simple square structuring element. Here's an explanation of what these functions do:

*1. **Dilation**:*

- Dilation is a morphological operation that enlarges bright regions in a binary image. It makes objects in the image thicker and extends them.

- The `dilate` function takes a binary image as input, where 1 represents the subject (foreground) and 0 represents the background.

- For each pixel in the input binary image, the function checks its neighborhood defined by the structuring element (`se`). The structuring element is typically a small binary image that defines the area of influence for dilation. In this example, we use a square structuring element.

- If any of the pixels in the neighborhood of the central pixel is 1 (indicating part of the subject), the central pixel is set to 1 in the result, effectively expanding the subject.

*2. **Erosion**:*

- Erosion is a morphological operation that shrinks bright regions in a binary image. It trims away parts of the subject, making it thinner and more refined.

- The `erode` function also takes a binary image as input, along with a structuring element (`se`).

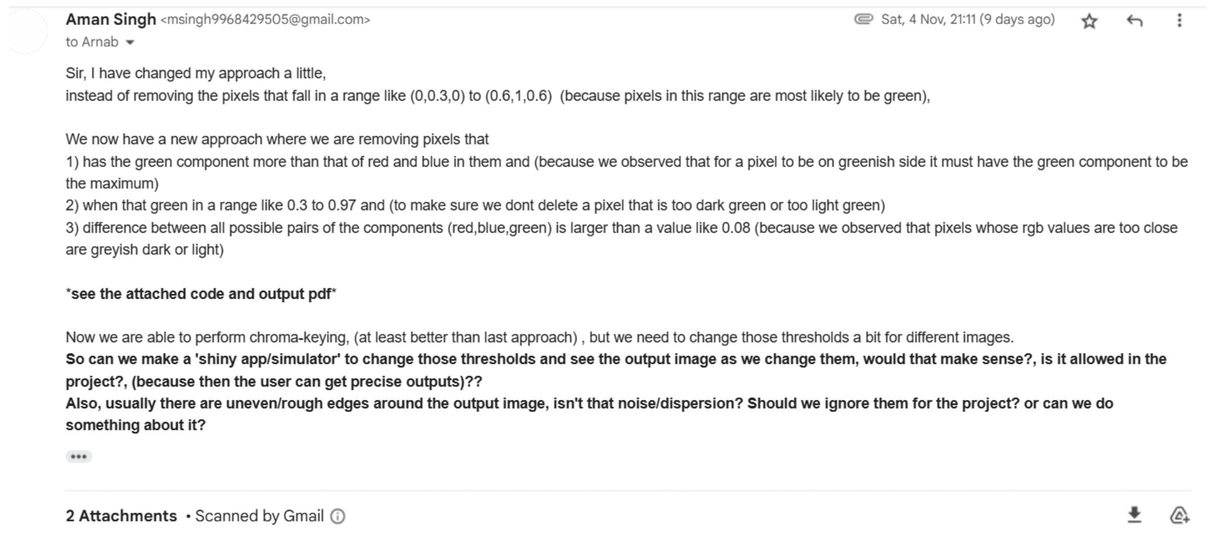
- For each pixel in the input binary image, the function checks its neighborhood defined by the structuring element.

- If all of the pixels in the neighborhood of the central pixel are 1 (indicating that the entire neighborhood is part of the subject), the central pixel is set to 1 in the result. Otherwise, it's set to 0.

- This effectively removes small protrusions and refines the edges of the subject.

These functions are used to process binary images, and they're commonly used in morphological image processing for tasks like noise reduction, edge detection, and feature extraction. In practice, the structuring element and the choice of dilation or erosion depend on the specific image processing goals. The square structuring element used in these examples is simple, but structuring elements can take various shapes and sizes to achieve different effects.

- `create_disc_se` is a function that takes a radius as input and generates a binary matrix (a structuring element) that represents a disc with the specified radius.
- The size of the square matrix is calculated based on the radius, and an empty matrix of zeros is created.
- It then loops through the matrix and sets values to 1 if the corresponding coordinates are inside the disc, determined by the circle equation `(x - center)^2 + (y - center)^2 <= radius^2`.
- The resulting binary matrix represents the disc-shaped structuring element.
- You can change the `radius` variable to create a disc structuring element of the desired size.


Aman Singh <msingh9968429505@gmail.com> Sat, 4 Nov, 21:11 (9 days ago)
to Arnab

Sir, I have changed my approach a little,
instead of removing the pixels that fall in a range like (0,0.3,0) to (0.6,1,0.6) (because pixels in this range are most likely to be green),

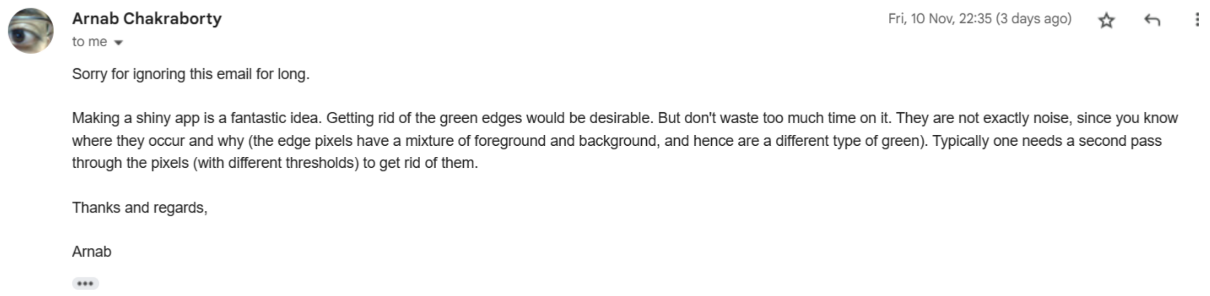
We now have a new approach where we are removing pixels that
1) has the green component more than that of red and blue in them and (because we observed that for a pixel to be on greenish side it must have the green component to be the maximum)
2) when that green in a range like 0.3 to 0.97 and (to make sure we dont delete a pixel that is too dark green or too light green)
3) difference between all possible pairs of the components (red,blue,green) is larger than a value like 0.08 (because we observed that pixels whose rgb values are too close are greyish dark or light)

see the attached code and output pdf

Now we are able to perform chroma-keying, (at least better than last approach) , but we need to change those thresholds a bit for different images.
So can we make a 'shiny app/simulator' to change those thresholds and see the output image as we change them, would that make sense?, is it allowed in the project?, (because then the user can get precise outputs)??
Also, usually there are uneven/rough edges around the output image, isn't that noise/dispersion? Should we ignore them for the project? or can we do something about it?

2 Attachments • Scanned by Gmail

Approach 5 is our final;


Arnab Chakraborty Fri, 10 Nov, 22:35 (3 days ago)
to me

Sorry for ignoring this email for long.

Making a shiny app is a fantastic idea. Getting rid of the green edges would be desirable. But don't waste too much time on it. They are not exactly noise, since you know where they occur and why (the edge pixels have a mixture of foreground and background, and hence are a different type of green). Typically one needs a second pass through the pixels (with different thresholds) to get rid of them.

Thanks and regards,

Arnab

Certainly, let's go through the code line by line, explaining each part:

```r
# Load required libraries
library(shiny)
library(png)
library(jpeg)
```

These lines load the necessary libraries for the Shiny app. `shiny` is the main library for building web applications in R, and `png` and `jpeg` are used for reading images in PNG and JPEG formats, respectively.

```r
# Define the UI for the Shiny app
```

~~~
ui <- fluidPage(
  titlePanel("Chroma - Keying"),

  sidebarLayout(
    sidebarPanel(
      fileInput("image_file", "Select an Image"),
      sliderInput("threshold", "Green Lower Bound", min = 0, max = 1, value = 0.28, step = 0.01),
      sliderInput("threshold2", "Green Upper Bound", min = 0, max = 1, value = 0.97, step = 0.01),
      sliderInput("threshold3", "Grey Bound", min = 0, max = 1, value = 0, step = 0.01)
    ),

    mainPanel(
      imageOutput("original_image"),
      imageOutput("processed_image")
    )
  )
)
```
~~~

This section defines the user interface (UI) of the Shiny app. The UI consists of a title panel, a sidebar layout, and main panels. The sidebar contains file input for selecting an image and slider inputs for setting thresholds. The main panel displays the original and processed images.

~~~
```r
# Define the server for the Shiny app
server <- function(input, output) {
  observe({
    req(input$image_file)

    image <- readJPEG(input$image_file$datapath)
```
~~~

Here, the server function is defined. The `observe` function ensures that the code inside it is re-executed whenever the specified reactive expressions change. In this case, it observes changes in the selected image file.

```r
  # Function to process the image
  process_image <- function(image, threshold, threshold2, threshold3) {
    dimn <- dim(image)
    result <- array(0, dim = dimn)

    for (i in 1:dimn[1]) {
      for (j in 1:dimn[2]) {
        pixel <- image[i, j, ]
        r <- pixel[1]
        g <- pixel[2]
        b <- pixel[3]

        if (
          which.max(pixel) == 2 &&
          g > threshold &&
          g <= threshold2 &&
          (abs(r - b) >= threshold3) &&
          (abs(r - g) >= threshold3) &&
          (abs(b - g) >= threshold3)
        ) {
          result[i, j, ] <- c(1, 1, 1)
        } else {
          result[i, j, ] <- pixel
        }
      }
    }
    return(result)
  }
```

This part defines the `process_image` function, which takes an image and three threshold values as inputs and processes the image based on these thresholds. It iterates over each pixel, checking conditions to determine whether to modify the pixel or leave it unchanged.

```r
  output$original_image <- renderImage({
    list(src = input$image_file$datapath,
         contentType = "image/jpeg",
         width = "100%")
  })
```

This line sets up the rendering of the original image in the Shiny app. The `renderImage` function is used to dynamically display the selected image.

```r
  output$processed_image <- renderImage({
    processed <- process_image(image, input$threshold, input$threshold2, input$threshold3)
    outfile <- tempfile(fileext = '.jpeg')
    writeJPEG(processed, target = outfile)

    list(src = outfile, contentType = "image/jpeg", width = "100%")
  })
 })
}
```

This part sets up the rendering of the processed image in the Shiny app. It uses the `process_image` function to modify the image based on the threshold values selected by the user, creates a temporary file for the processed image, and renders the processed image dynamically.

```r
# Run the Shiny app
shinyApp(ui = ui, server = server)
```

Finally, this line runs the Shiny app by combining the defined UI (`ui`) and server (`server`). It launches the Shiny app in the browser. The user can interact with the app by selecting an image and adjusting the threshold sliders to see the original and processed images.

SOMETHINGS YOU WOULD BE CONFUSED OF ;

Certainly, let's break down the relevant lines of code:

```r
observe({
  req(input$image_file)
```

Here, the `observe` function is used to observe changes in reactive values, and in this case, it's observing changes in the `input$image_file` (the selected image file). The `req` function is used to ensure that the subsequent code within the `observe` block is executed only when `input$image_file` is not null or missing. This helps avoid errors when trying to process an image that hasn't been selected.

```r
renderImage({
  list(src = input$image_file$datapath,
       contentType = "image/jpeg",
       width = "100%")
})
```

This is part of Shiny's reactive programming. The `renderImage` function is used to dynamically render an image in the Shiny app. The content of the image is specified using a list, and in this case, it includes the source (`src`) as the datapath of the selected image file, the content type (`contentType`) as "image/jpeg," and the width of the image as "100%."

```r
outfile <- tempfile(fileext = '.jpeg')
writeJPEG(processed, target = outfile)

list(src = outfile, contentType = "image/jpeg", width = "100%")
```

These lines are part of rendering the processed image. A temporary file (`outfile`) is created with a ".jpeg" extension to store the processed image. The `writeJPEG` function is used to write the processed image data to this temporary file. The final list includes the source (`src`) as the path to the temporary file, the content type (`contentType`) as "image/jpeg," and the width of the image as "100%."

So, to summarize, these lines are responsible for dynamically rendering the original and processed images in the Shiny app. The `observe` block ensures that the processing and rendering are triggered when a new image file is selected by the user. The `renderImage` functions create HTML elements to display the images with the specified properties.

Thanks and Regards.

Signing off

-amansingh