Image Manipulation: A Python Tool

Mandy Smoak
Digital Production Art
Clemson University

Project and Code Overview

The objective for this project was to write an image viewing, manipulation, and compositing tool. The application could be written using GLSL with C++ or with Python using an image library of our choice.

At a minimum, the following operations must be included: gamma correction, contrast, edge detection, sharpen, blur, median spatial filter, mix, key mix, over operator, luma keying, chroma keying, color difference method, and a write function to save the results. The distance, add, invert, and two multiply functions were included to complete the required operations more efficiently. I've additionally included the sobel operator, gaussian blur, and the screen operator. Finally, to make the tool user-friendly, I've included a menu prompt and a key handling function.

Color Adjustments

Overview. Color adjustments are generally the simplest type of image manipulation. The basic concept is to iterate through each pixel to perform some operation on each of them. More specifically, a calculation would be performed on each pixel's red, green, and blue channels.

Gamma Correction. Gamma defines the relationship between a pixel's numerical value and its actual luminance. It is a mathematical curve or function that enables to know the correlation between an input signal and the response of a sensor, for instance, our eye. This function is written: output signal = input signal $^{\text{gamma}}$.

```
# Gets gamma value
gamma_value = input("Enter a gamma value between 0.01 and 7.99: ")

# Calculates gamma correction
gamma_correction = 1 / float(gamma_value)

# Creates image with gamma correction
for x in range(output.width):
    for y in range(output.height):
        r, g, b = pixels[x, y]
        # Assigns new value to R, G, B
        r = int(255 * (r / 255) ** gamma_correction)
        g = int(255 * (g / 255) ** gamma_correction)
        b = int(255 * (b / 255) ** gamma_correction)
        draw.point((x, y), (r, g, b))
```

Figure 1. Original image (left), result with γ = 0.5 (middle), and result with γ = 4.0 (right).

Contrast. The contrast is the difference in brightness that makes the objects in a picture more distinguishable. The intensity histogram of an image is the distribution of pixel luminance for an image. In order to improve the contrast, we can use a linear normalization of the intensity histogram: $I_N = (I - I_{min})\ 255\ /\ (I_{max} - I_{min})$, where I_N is the normalized pixel intensity, I_{max} and I_{min} are the minimum and maximum intensity before normalization.

```
# Finds minimum/maximum color values
imin = 255
imax = 0
for x in range(image.width):
    for y in range(image.height):
        r, g, b = input_pixels[x, y]
        i = (r + g + b) / 3
        imin = min(imin, i)
        imax = max(imax, i)

# Creates image with increased contrast
for x in range(output.width):
    for y in range(output.height):
        r, g, b = input_pixels[x, y]
        # Current luminosity
        i = (r + g + b) / 3
        # New luminosity
        ip = 255 * (i - imin) / (imax - imin)
        r = int(r * ip / i)
        g = int(g * ip / i)
        b = int(b * ip / i)
        draw.point((x, y), (r, g, b))
```

Figure 2. Original image (left) and result (right).

Filters

Overview. Image filtering involves the convolution between an image and a kernel. For each pixel, the filter multiplies the current pixel and the surrounding pixels, called a neighborhood, by the kernel's corresponding value. This result becomes the value of the output pixel.

Edge Detection. Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting discontinuous in brightness. The kernel value's used for this operation are all -1's except for the middle value which is 8.

```
# Edge detection kernel
kernel = [[-1, -1, -1],
          [-1, 8, -1],
          [-1, -1, -1]]

# Computes convolution
for x in range(1, image.width - 1):
    for y in range(1, image.height - 1):
        acc = [0, 0, 0]
        for a in range(len(kernel)):
            for b in range(len(kernel)):
                xn = x + a - 1
                yn = y + b - 1
                pixel = pixels[xn, yn]
                acc[0] += pixel[0] * kernel[a][b]
                acc[1] += pixel[1] * kernel[a][b]
                acc[2] += pixel[2] * kernel[a][b]

        draw.point((x, y), (int(acc[0]), int(acc[1]), int(acc[2])))
```

Figure 3. Original image (left) and result (right).

Sobel. The Sobel Operator is a form of edge detection. It uses two kernels which finds gradient magnitudes in both the x and y directions. Once found, they can be combined to find the gradient magnitude.

```
# Calculates pixel intensity
intensity = [[sum(pixels[x, y]) / 3 for y in range(image.height)]
for x in range(image.width)]

# Sobel kernels
kernelx = [[1, 0, -1],
           [2, 0, -2],
           [1, 0, -1]]
kernely = [[1, 2, 1],
           [0, 0, 0],
           [-1, -2, -1]]

# Computes convolution
for x in range(1, image.width - 1):
    for y in range(1, image.height - 1):
        magx, magy = 0, 0
        for a in range(3):
            for b in range(3):
                xn = x + a - 1
                yn = y + b - 1
                magx += intensity[xn][yn] * kernelx[a][b]
                magy += intensity[xn][yn] * kernely[a][b]

       # Draw in black and white the magnitude
        color = int(sqrt(magx ** 2 + magy ** 2))
        draw.point((x, y), (color, color, color))
```

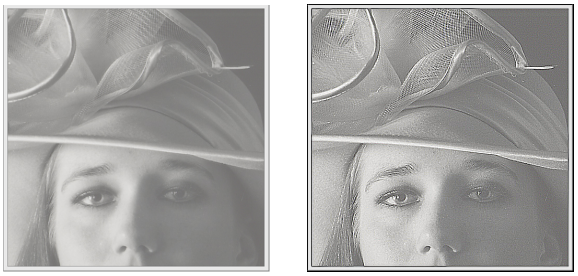

Figure 4. Original image (left) and result (right).

Sharpen. The sharpening filter is used to enhance line structures of other details in an image. The code itself is almost identical to the traditional edge detection. The kernel value's used for this operation are all -1's except for the middle value which is 9.

```
# Sharpening kernel
kernel = [[-1, -1, -1],
          [-1, 9, -1],
          [-1, -1, -1]]

# Computes convolution
for x in range(1, image.width - 1):
    for y in range(1, image.height - 1):
        acc = [0, 0, 0]
        for a in range(len(kernel)):
            for b in range(len(kernel)):
                xn = x + a - 1
                yn = y + b - 1
                pixel = pixels[xn, yn]
                acc[0] += pixel[0] * kernel[a][b]
                acc[1] += pixel[1] * kernel[a][b]
                acc[2] += pixel[2] * kernel[a][b]

        draw.point((x, y), (int(acc[0]), int(acc[1]), int(acc[2])))
```

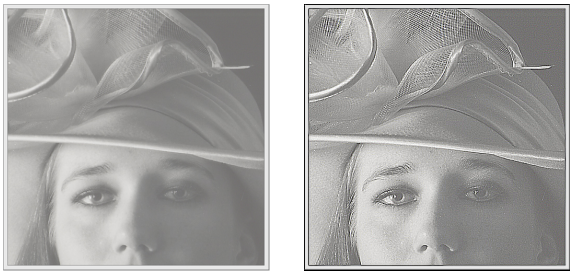

Figure 5. Original image (left) and result (right).

Blur. For this operation, I provided the option for both a box blur and a gaussian blur. The box blur is a 3x3 kernel, while the gaussian kernel is a 5x5 kernel. The box blur is a spatial domain linear filter in which each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image, while the gaussian blur gives more weight to the pixel near the current pixel.

```
# Box Blur kernel
box_kernel = [[1 / 9, 1 / 9, 1 / 9],
              [1 / 9, 1 / 9, 1 / 9],
              [1 / 9, 1 / 9, 1 / 9]]

# Gaussian kernel
gaussian_kernel = [[1 / 256, 4 / 256, 6 / 256, 4 / 256, 1 / 256],
                   [4 / 256, 16 / 256, 24 / 256, 16 / 256, 4 / 256],
                   [6 / 256, 24 / 256, 36 / 256, 24 / 256, 6 / 256],
                   [4 / 256, 16 / 256, 24 / 256, 16 / 256, 4 / 256],
                   [1 / 256, 4 / 256, 6 / 256, 4 / 256, 1 / 256]]

# Choose blur kernel
chosen_kernel = input("Type 'b' to perform a box blur or type 'g' to
perform a gaussian blur: ")

# Select kernel here
if chosen_kernel == "b":
    kernel = box_kernel
elif chosen_kernel == "g":
    kernel = gaussian_kernel
else:
    print("Error: You did not type either 'b' or 'g'.")
    sys.exit()

# Middle of kernel
offset = len(kernel) // 2

# Computes convolution
for x in range(offset, image.width - offset):
    for y in range(offset, image.height - offset):
        acc = [0, 0, 0]
        for a in range(len(kernel)):
            for b in range(len(kernel)):
                xn = x + a - offset
                yn = y + b - offset
                pixel = pixels[xn, yn]
                acc[0] += pixel[0] * kernel[a][b]
                acc[1] += pixel[1] * kernel[a][b]
                acc[2] += pixel[2] * kernel[a][b]
```

Figure 6. Original image (top), box blur (middle), and gaussian blur (bottom).

Median Spatial. The median filter is a sliding-window spatial filter that replaces the center value in the window with the median of all the pixel values in the window. The filter is very useful for reaching noise, or salt-and-pepper, in images. Unfortunately, I could not get this function working; however, I've included my attempt below.

```
lum_list = []
for x in range(1, image.width - 1):
    for y in range(1, image.height - 1):
        kernel = [[pixels[x - 1, y - 1], pixels[x, y - 1], pixels[x + 1, y - 1]],
                  [pixels[x - 1, y], pixels[x, y], pixels[x + 1, y]],
                  [pixels[x - 1, y + 1], pixels[x, y + 1], pixels[x + 1, y + 1]]]
        for a in range(len(kernel)):
            for b in range(len(kernel)):
                xn = x + a - 1
                yn = y + b - 1
                r, g, b = pixels[xn, yn]
                # Finds luminance value
                luminance = (r * 0.2126 + g * 0.7152 + b * 0.0722)
                tup = (a, b, luminance)
                lum_list.append(tup)
            sorted(lum_list, key=itemgetter(2))
            sorted_position_a = [tup[0] for tup in lum_list]
            sorted_position_b = [tup[1] for tup in lum_list]
            r, g, b = kernel[sorted_position_a[4]][sorted_position_b[4]]
            draw.point((x, y), (r, g, b))
```

Compositing

Overview. Compositing is the combination of images from separate sources into single images to create the illusion that all those elements are parts of the same scene. Often, the combination involves a matte, which helps to separate the foreground and background layers. For operations in this section, I used two small images of the same size to decrease computation time.

Mix. The Mix operator is a specialized version of the Add operation, which adds together corresponding pixel values from the two images. Mix is unique, however, in that is is the weighted, normalized addition of the two images. The user is responsible for determining the ratio at which the two images are combined. For example, in my code, if the user types in “70” after the prompt, then the output image will show 70% of the foreground image and 30% of the background image. The formula is O = (MV x A) + [(1 - MV) x B)], where O is the output image, MV is the user-defined mix value, A is the foreground image, and B is the background image.

```
# Calculates MV and (1-MV)
percent = input("What percentage would you like the
foreground to show through?  ")
fg_mv = int(percent) / 100
bg_mv = 1 - fg_mv

# Computes O = (MV * A) + [(1 - MV) * B]
fg_multiplied = multiply1(foreground, fg_mv)
bg_multiplied = multiply1(background, bg_mv)
```

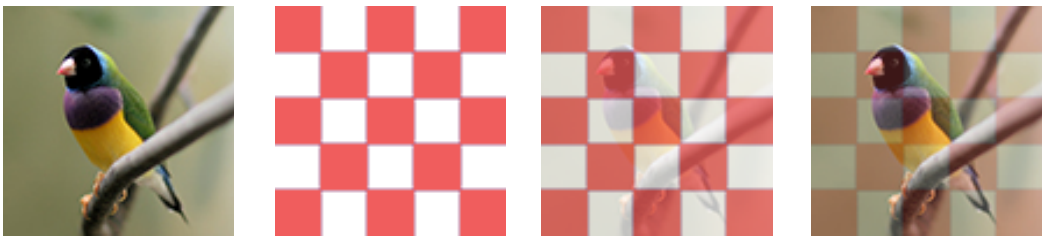

Figure 7. Original foreground (first), original background (second), mix result where MV = 25 (third), and mix result where MV = 75 (fourth).

Keymix. The Keymix operation is much like the Mix operation, only rather than a mix value, there is a specific mix value for each pixel as determined by the matte image. The formula is O = (A x M) + [(1 - M) x B)], where O is the output image, M is the matte, A is the foreground image, and B is the background image.

```
# Computes O = (A * M) + [(1 - M) * B]
new_foreground = multiply2(foreground, foreground_matte)
inverted_matte = invert(foreground_matte)
new_background = multiply2(background, inverted_matte)
output = add(new_foreground, new_background)
```

Figure 8. Original foreground (first), original background (second), foreground matte (third), and result (fourth).

Over. The Over operator is specifically designed to layer a four channel image over another image. It is effectively the same as then Keymix operation, only simplified to take advantage if the integrated matte. The formula is O = (A x M) + [(1 - M) x B)], where O is the output image, M is the matte, A is the foreground image, and B is the background image.

```
# Computes O = A + [(1 - A) * B]
premult_foreground = multiply3(foreground_matte, foreground)
inverted_matte = invert(foreground_matte)
new_bg = multiply3(inverted_matte, background)
output = add(premult_foreground, new_bg)
```

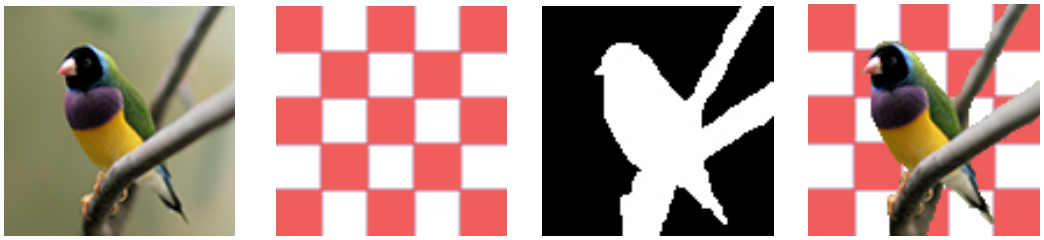

Figure 9. Original foreground (first), original background (second), foreground matte (third), and result (fourth).

Screen. The Screen operator doesn't require any integrated matte channels. It is popular because it works newly for simulating the effect of adding light to a portion of an image. It is a multiplicative operator, only we perform the additional step of inverting both images before multiplying them together, and then invert the result. The formula is O = 1 - [(1 - A) x (1 - B)], where O is the output image, A is the foreground image, and B is the background image.

```
# Computes O = 1 - [(1 - A) * (1 - B)]
inverted_foreground = invert(foreground)
inverted_background = invert(background)
mult_fg_bg = multiply3(inverted_foreground, inverted_background)
output = invert(mult_fg_bg)
```

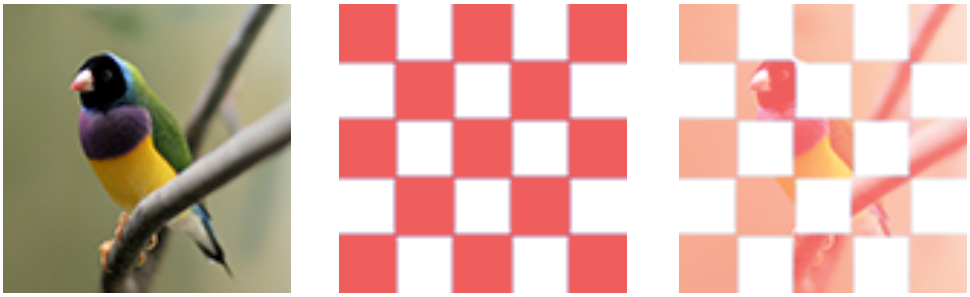

Figure 10. Original foreground (left), original background (middle), result (right).

Keying

Overview. The process of generating a matte for an object is referred to as keying. Keying is not only used to describe the process of creating a matte, but also to include the process of combining one element with another. Note that all of the following functions perform compositing as well.

Luma Keying. Luma Keying is a method used to extract a matte for a given item is based on manipulating the luminance values in a scene. It involves iterating through each pixel using the r, g, and b values to calculate luminosity. You would then apply a threshold to distinguish the light items against the dark background. Note that there must be a significant difference between the brightness of the foreground and the background.

```
for x in range(foreground.width):
    for y in range(foreground.height):
        r, g, b, a = pixels[x, y]
        # Finds luminance value
        luminance = (r * 0.2126 + g * 0.7152 + b * 0.0722)
        # Finds distance to white
        d = distance(luminance, 255)
        # Masking using luminance threshold
        if 225 < d < 255:
            r = 0
            g = 0
            b = 0
            a = 0
        else:
            r = 255
            g = 255
            b = 255
            a = 255
        draw.point((x, y), (r, g, b, a))

new_foreground = multiply3(matte, foreground)
inverted_matte = invert(matte)
new_background = multiply3(inverted_matte, background)
output = add(new_foreground, new_background)
```

Figure 11. Original foreground (top left), original background (top right), result (bottom).

Chroma Keying. Chroma key is a bit more sophisticated than luma keying. To do this, it is best to first convert each pixel's r, g, and b values its corresponding hue (h), saturation (s), and value (v) values. Then, we would continue by picking a certain range of colors or hues and to define only the pixels that fall within that range as being part of the background. The advantage of this method relative to the lumakey is that we can now make the distinction between areas that are dark but still part of the foreground and the true background that we want to remove.

```
for x in range(foreground.width):
    for y in range(foreground.height):
        r, g, b, a = pixels[x, y]
        h, s, v = colorsys.rgb_to_hsv(r / 255, g / 255, b / 255)
        h = h * 360
        s = s * 100
        v = v * 100
        # Alpha-masking using hue threshold (green)
        if 90 < h < 150:
            h = s = v = a = 0
        else:
            h = s = 0
            v = 100
            a = 255
        r, g, b = colorsys.hsv_to_rgb((h / 360), (s / 100), (v / 100))
        draw.point((x, y), (int(r * 255), int(g * 255), int(b * 255), a))

new_foreground = multiply3(matte, foreground)
inverted_matte = invert(matte)
new_background = multiply3(inverted_matte, background)
output = add(new_foreground, new_background)
```

Figure 12. Original foreground (top left), original background (top right), result (bottom).

Color Difference Method. The color difference method includes a combination of steps including matte extraction, color correction, and image combination. This is done by selectively substituting the green channel for the blue channel in every pixel in which the existing blue component has a greater intensity than the green component. The second step involves the creation of the matte itself. We do this by subtracting the maximum of the red or green component from the blue component. The final step is using this matte to combine our two images much like we do in the Over operator.

```
# Performs alpha masking
for x in range(foreground.width):
    for y in range(foreground.height):
        r, g, b, a = pixels[x, y]
        # matte creation
        a = g - max(b, r)
        if a > 0:
            r = g = b = a = 255
        else:
            r = g = b = a = 0
        draw.point((x, y), (r, g, b, a))

# Creates spill suppressed image
ss_foreground = Image.new("RGBA", foreground.size)
draw = ImageDraw.Draw(ss_foreground)

# Performs spill suppression
for x in range(foreground.width):
    for y in range(foreground.height):
        r, g, b, a = pixels[x, y]
        # spill suppression
        if g > b:
            g = b
        else:
            g = g
        draw.point((x, y), (r, g, b, a))

inverted_matte = invert(matte)
new_foreground = multiply3(inverted_matte, ss_foreground)
new_background = multiply3(matte, background)
output = add(new_foreground, new_background)
```

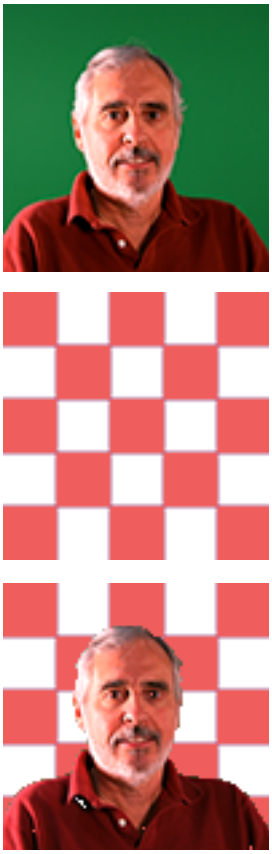

Figure 13. Original foreground (top), original background (middle), result (bottom).

Helper Functions

Overview. These functions were added to make the tool as a whole more efficient. Most of the following operations were used in all of the compositing and keying functions.

Distance. This is a simple function that determines the distance between two values. It is used in the lumakey function to find the distance between the calculated luminance value and true white.

```
# Find distance between two values
def distance(x, y):
    return abs(x-y)
```

Add. This function is used in almost all of the composting and keying functions to combine the foreground and the background. It simply iterates through each pixel of the foreground and the corresponding background pixel and it adds together the matching r, g, and b values from those pixels.

```
# Adds each channel value in pixel by corresponding value in second image
for x in range(image1.width):
    for y in range(image1.height):
        r_1, g_1, b_1, a_1 = image1_pixels[x, y]
        for m in range(image2.width):
            for n in range(image2.height):
                r_2, g_2, b_2, a_2 = image2_pixels[m, n]
                if x == m and y == n:
                    r = int(r_1 + r_2)
                    g = int(g_1 + g_2)
                    b = int(b_1 + b_2)
                    a = int(255)
                    draw.point((m, n), (r, g, b, a))
```

Invert. This is used quite a bit in compositing and keying functions to reverse the matte. Once we have a matte female matte, we can use the invert function to create the male matte, and visa versa. It works by iterating through each pixel and subtracting the r, g, and b values from 225.

```
# Creates image with gamma correction
for x in range(output.width):
    for y in range(output.height):
        r, g, b, a = input_pixels[x, y]
        r = int(255 - r)
        g = int(255 - g)
        b = int(255 - b)
        a = int(255 - a)
        draw.point((x, y), (r, g, b, a))
```

Multiply 1. This version of multiply works by multiplying the image by a constant. It is used in the mix function to multiply the foreground or background by its corresponding mix value. It works the similarly to the add function, but rather than adding to pixels from two images, it multiplies the pixel's r, g, and b values by the mix value.

```
# Multiplies each channel value in pixel by a constant
for x in range(image.width):
    for y in range(image.height):
        r, g, b, a = pixels[x, y]
        r = int(r * value)
        g = int(g * value)
        b = int(b * value)
        a = int(255)
        draw.point((x, y), (r, g, b, a))
```

Multiply 2. This version of multiply performs multiplication on two images. It works identically to the add function, but uses the multiplication operator rather then the addition operator. Note, it takes the alpha channel from the first image, so the matte must be the first parameter.

```
# Multiplies each channel value in pixel by corresponding value in second image
for x in range(image1.width):
    for y in range(image1.height):
        r_1, g_1, b_1, a_1 = image1_pixels[x, y]
        for m in range(image2.width):
            for n in range(image2.height):
                r_2, g_2, b_2, a_2 = image2_pixels[m, n]
                if x == m and y == n:
                    r = int(((r_1 / 255) * (r_2 / 255)) * 255)
                    g = int(((g_1 / 255) * (g_2 / 255)) * 255)
                    b = int(((b_1 / 255) * (b_2 / 255)) * 255)
                    a = int(a_1)
                    draw.point((m, n), (r, g, b, a))
```

Other

Main Function. The tool begins at the main function. This is where the menu is printed for the user. This menu lets the user see what manipulation options are possible along with the keyboard key that corresponds to the function.

```
# Main
def main():

    image = Image.open("girl.jpg")

    # print instructions
    print("\nWelcome to Mandy's Image Manipulator.\n")
    print("Press...")
    print(" 'g' to Gamma Correct            'c' to Contrast")
    print(" 'g' for Sobel Operator          'e' for Edge Detection")
    print(" 'b' to Blur                     's' to Sharpen")
    print(" 'f' for Median Spatial Filter   'm' to Mix")
    print(" 'k' to Key Mix                  'o' for the Over Operator")
    print(" 'n' for the Screen Operator     'l' for Luma Key")
    print(" 'h' for Chroma Key              'd' for Color Difference\n")
    print("  Click 'w' to write the image and 'q' to quit.\n\n")

    key = input("Input Key: ")
    handleKey(key, image)
```

Key Handler. Once the user presses a keyboard key, the key is sent to the key handler function to decide what to do next. If an unrecognized key is pressed, then an error message is printed.

```
# Decides what to do after keyboard key is clicked
def handleKey(key, image):

    # input detection
    if key == "g":
        gamma()
        return

    .  .  .

    if key == "w":
        write(image)
        return
    if key == "q":
        print("\nProgram terminated.\n")
        sys.exit()
    else:
        print("\nError: You did not choose an recognized key\n.")
        main()
    return
```

Write. Finally, the write and quit options are offered after each operation is performed so that the user can save the output image, if desired. The function saves the result in the same directory as the python tool. The output file name includes a time stamp so that each image has a unique name.

```
# Writes image
def write(image):

    time_string = time.strftime("%Y.%m.%d_at_%H.%M.%S")

    image.save("output_" + time_string + ".png")
    print("Image saved as...  output_" + time_string + ".png.")
```