Open-CV approach to inject a watermark into video

Mr Ashlin Darius Govindasamy
University of South Africa
Department of Mathematics and Computer Science

January 8, 2023

Abstract

In this paper I attempt to inject a transparent watermark into a video. The watermark is a PNG image with transparent background any other format Open-CV supports is supported. The watermark is injected into the video by overlaying the watermark on the video frame by frame and is written into a video buffer saved into an output video file.

Contents

Chapter 1

Procedure

The watermark is overlayed on the video frame by frame using the following steps:

1. Read the video frame by frame
2. Resize the watermark to the size of the video frame
3. Overlay the watermark on the video frame
4. Write the video frame to the output video buffer
5. Repeat steps 3-4 for all the video frames
6. Save the output video buffer to a file
7. Display the output video with watermark transparently overlayed on the video

The methodology is simple.

You need to open the video file using Open-CV in our case the video file is a MP4 file. We then read the video frame by frame and overlay the watermark on the video frame. We then write the video frame to the output video buffer. We repeat the process for all the video frames. We then save the output video buffer to a file.

Once done using our favourite video software enjoy your content with your own personal branding.

We first open the watermark given by the user. We then resize the watermark to the size of the video frame. We then overlay the watermark on the video frame. We then write the video frame to the output video buffer. We then save the output video buffer to a file.

In terms of Open-CV functions

Using the `cv.imread` function we read the video frame by frame. We then use the `cv.resize` function to resize the watermark to the size of the video frame. We then use the `cv.addWeighted` function to overlay the watermark on the video frame. We then use the `cv.imwrite` function to write the video frame to the output video buffer. We then use the `cv.imwrite` function to save the output video buffer to a file.

The `cv2.IMREAD_UNCHANGED` flag when overloading the `cv.imread` function is used to read the watermark with transparency.

A simple example of the code is given below in Python:

```
import cv2
import numpy as np
watermark = cv2.imread(watermark_path, cv2.
    IMREAD_UNCHANGED)
watermark = cv2.resize(watermark, (100, 100))
```

1.1 Code explained Mathematically

1.1.1 General approach

Mathematically this can be seen taking a video frame V and a watermark W and overlaying the watermark on the video frame as follows:

$$V_{out} = V + W \tag{1.1}$$

When we use the `cv.imread` function we open the watermark as a 3 channel image.

A 3 channel image is an image with 3 colour channels. The 3 colour channels are the Red, Green and Blue channels. The Red, Green and Blue channels are the primary colours of light. The Red, Green and Blue channels are used to create all the colours in the visible spectrum. We use this for the background.

The 4 channel image is an image with 4 colour channels. Which consists of CMYK (Cyan, Magenta, Yellow and Black) channels. The CMYK channels are used to create all the colours in the visible spectrum. We use this for the watermark.

We sum all the frames of the video frame and the watermark and render the output video buffer with the watermark transparently overlayed on the video frame.

$$V_{out} = \sum_{i=1}^{V_f} V_i + W_i \tag{1.2}$$

V_f is the number of frames in the video.
V_i is the i^{th} frame of the video.
W_i is the i^{th} frame of the watermark.
V_{out} is the output video buffer.

Remember in `Computer Image Processing` All of this is done pixel by pixel and each frame is a 2 dimensional array of pixels and each pixel is a 3 dimensional array of colour channels.

Now we need to understand how the `cv.addWeighted` function works.

Understanding the `cv.addWeighted| function`

The `cv.addWeighted` function is used to overlay the watermark on the video frame. The `cv.addWeighted` function takes the following parameters:

- `src1` - The first input array.

- `alpha` - Weight of the first array elements.
- `src2` - The second input array of the same size and channel number as src1.
- `beta` - Weight of the second array elements.
- `gamma` - Scalar added to each sum.
- `dst` - Output array that has the same size and number of channels as the input arrays.
- `dtype` - Optional depth of the output array; when both input arrays have the same depth, dtype can be set to -1, which will be equivalent to src1.depth().

With the `cv.addWeighted` function we can overlay the watermark on the video frame as follows:

$$V_{out} = \alpha V + \beta W + \gamma \tag{1.3}$$

V_{out} is the output video buffer.
V is the video frame.
W is the watermark.
α is the weight of the video frame.
β is the weight of the watermark.
γ is the scalar added to each sum.
Taking the partial derivative of the above equation with respect to α we get:

$$\frac{\partial V_{out}}{\partial \alpha} = V \tag{1.4}$$

This means that the weight of the video frame is the video frame itself.

Taking the partial derivative of the above equation with respect to β we get:

$$\frac{\partial V_{out}}{\partial \beta} = W \tag{1.5}$$

This means that the weight of the watermark is the watermark itself.

Taking the partial derivative of the above equation with respect to γ we get:

$$\frac{\partial V_{out}}{\partial \gamma} = 1 \tag{1.6}$$

This means that the scalar added to each sum is 1.

$$V_{out} = V + W + 1 \tag{1.7}$$

Now using the `cv.addWeighted` function we can overlay the watermark on the video frame as follows:

$$V_{out} = cv.addWeighted(V, 1, W, 1, 1) \tag{1.8}$$

Now i tried this approach the watermark was not transparently overlayed on the video frame. The watermark was overlayed on the video frame but the watermark was not transparent. The watermark was a solid

colour. I was quite upset with the results. Until i met this new approach to overlaying the watermark on the video frame.

1.1.2 New approach to overlaying the watermark on the video frame

We take variables as inputs from the user. The variables are the $V_{background}, V_{foreground}, x_{offset}, y_{offset}$.

$V_{background}$ is the video frame of the background video you want to use.

$V_{foreground}$ is the video frame of the foreground video you want to use.

x_{offset} is the x offset of the watermark on the video frame.

y_{offset} is the y offset of the watermark on the video frame.

We set the $bg_h, bg_w, bg_channels$ variables to the height, width and number of channels of the video frame of the background video.

We set the $fg_h, fg_w, fg_channels$ variables to the height, width and number of channels of the video frame of the foreground video.

Assertion error is raised if the channel of the background video frame is not 3 and the channel of the foreground video frame is not 4.

By default we center the watermark on the video frame. We do this by setting the x_{offset} and y_{offset} variables to the following values:

$$x_{offset} = \frac{bg_w - fg_w}{2} \tag{1.9}$$

$$y_{offset} = \frac{bg_h - fg_h}{2} \tag{1.10}$$

We compute the *width* and *height* of the watermark as follows:

$$width = min(fg_w, bg_w, fg_w + x_{offset}, bg_w - x_{offset}) \tag{1.11}$$

$$height = min(fg_h, bg_h, fg_h + y_{offset}, bg_h - y_{offset}) \tag{1.12}$$

Now if the *width* is less than 1 or the *height* is less than 1 we return the background video frame else we clip foreground and background images to the overlapping regions.

$$\begin{cases} fg_x = 0 & \text{if } x_{offset} \geq 0 \\ fg_x = -x_{offset} & \text{if } x_{offset} < 0 \end{cases} \tag{1.13}$$

$$\begin{cases} fg_y = 0 & \text{if } y_{offset} \geq 0 \\ fg_y = -y_{offset} & \text{if } y_{offset} < 0 \end{cases} \tag{1.14}$$

$$\begin{cases} fg_w = width & \text{if } x_{offset} \geq 0 \\ fg_w = width + x_{offset} & \text{if } x_{offset} < 0 \end{cases} \tag{1.15}$$

$$\begin{cases} fg_h = height & \text{if } y_{offset} \geq 0 \\ fg_h = height + y_{offset} & \text{if } y_{offset} < 0 \end{cases} \tag{1.16}$$

After clipping the foreground and background images to the overlapping regions we set the bounds of the foreground video frame as follows:

$$bg_x = max(0, x_{offset}) \tag{1.17}$$

$$bg_y = max(0, y_{offset}) \tag{1.18}$$

$$fg_x = max(0, -x_{offset}) \tag{1.19}$$

$$fg_y = max(0, -y_{offset}) \tag{1.20}$$

$$bg_w = min(0, x_{offset} * -1) \tag{1.21}$$

$$bg_h = min(0, y_{offset} * -1) \tag{1.22}$$

We set the bounds of the background video frame as follows:

$$V_{foreground}[fg_y : fg_y + height, fg_x : fg_x + width] \quad (1.23)$$

In the above equation we set the bounds of the foreground video frame.

Alternatively we can write the above equation as follows:

$$V_{foreground} = V_{foreground} \begin{bmatrix} fg_y & fg_y + height \\ fg_x & fg_x + width \end{bmatrix} \quad (1.24)$$

We set the bounds of the background video frame as follows:

$$V_{background}[bg_y : bg_y + height, bg_x : bg_x + width] \quad (1.25)$$

In the above equation we set the bounds of the background video frame.

Alternatively we can write the above equation as follows:

$$V_{background} = V_{background} \begin{bmatrix} bg_y & bg_y + height \\ bg_x & bg_x + width \end{bmatrix} \quad (1.26)$$

Now the background subsection variable is the background video frame subsection which is the overlapping region of the background video frame and the foreground video frame.

$$background_{subsection} = background[bg_y : bg_y{+}h, bg_x : bg_x{+}w] \tag{1.27}$$

In the matrix form we can write the above equation as follows:

$$background_{subsection} = background \begin{bmatrix} bg_y & bg_y + height \\ bg_x & bg_x + width \end{bmatrix} \tag{1.28}$$

We now separate the alpha and color channels of the foreground video frame.

$$foreground_{alpha} = foreground[:, :, :3] \tag{1.29}$$

In the matrix form we can write the above equation as follows:

$$foreground_{alpha} = foreground \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{1.30}$$

$$foreground_{color} = foreground[:, :, 3:] \tag{1.31}$$

In the matrix form we can write the above equation as follows:

$$foreground_{color} = foreground \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 3 & 3 \end{bmatrix} \tag{1.32}$$

We now compute the α channel of the background video frame subsection.

$$\alpha_{channel} = foreground[:, :, 3]/255 \tag{1.33}$$

Alternatively

$$\alpha_{channel} = foreground_{\alpha}/255 \tag{1.34}$$

Now we apply an α mask to the α channels to match the image size shape of the background video frame subsection.

$$\alpha_{mask} = np.dstack((alpha_channel, alpha_channel, alpha_channel)) \tag{1.35}$$

We now compute a composite image by combining the foreground and background video frame subsections.

Using this equation we can compute the composite image:

$$composite_{image} = (1 - \alpha_{mask}) * background_{subsection} + \alpha_{mask} * foreground_{color} \tag{1.36}$$

We now set the background video frame subsection to the composite image.

$$background[bg_y : bg_y + h, bg_x : bg_x + w] = composite_{image} \tag{1.37}$$

In the matrix form we can write the above equation as follows:

$$background = background \begin{bmatrix} bg_y & bg_y + height \\ bg_x & bg_x + width \end{bmatrix} = composite_{image} \tag{1.38}$$

We now return the background video frame.

The following is the code wrapper function for the above equations.

```
    def add_transparent_image(background,
        foreground, x_offset=None, y_offset=
        None):
    bg_h, bg_w, bg_channels = background.
        shape
    fg_h, fg_w, fg_channels = foreground.
        shape
    assert bg_channels == 3, f'background␣
        image␣should␣have␣exactly␣3␣channels␣(
        RGB).␣found:{bg_channels}'
    assert fg_channels == 4, f'foreground␣
        image␣should␣have␣exactly␣4␣channels␣(
        RGBA).␣found:{fg_channels}'
    # center by default
```

```
    if x_offset is None: x_offset = (bg_w -
        fg_w) // 2
    if y_offset is None: y_offset = (bg_h -
        fg_h) // 2
    w = min(fg_w, bg_w, fg_w + x_offset, bg_w
         - x_offset)
    h = min(fg_h, bg_h, fg_h + y_offset, bg_h
         - y_offset)
    if w < 1 or h < 1: return
    # clip foreground and background images
        to the overlapping regions
    bg_x = max(0, x_offset)
    bg_y = max(0, y_offset)
    fg_x = max(0, x_offset * -1)
    fg_y = max(0, y_offset * -1)
    foreground = foreground[fg_y:fg_y + h,
        fg_x:fg_x + w]
    background_subsection = background[bg_y:
        bg_y + h, bg_x:bg_x + w]
    # separate alpha and color channels from
        the foreground image
    foreground_colors = foreground[:, :, :3]
    alpha_channel = foreground[:, :, 3] / 255
         # 0-255 => 0.0-1.0
    # construct an alpha_mask that matches
        the image shape
    alpha_mask = np.dstack((alpha_channel,
        alpha_channel, alpha_channel))
    # combine the background with the overlay
         image weighted by alpha
    composite = background_subsection * (1 -
        alpha_mask) + foreground_colors *
        alpha_mask
```

```
    # overwrite the section of the background
        image that has been updated
    background[bg_y:bg_y + h, bg_x:bg_x + w]
       = composite
```

Chapter 2

Injecting a Watermark into a Video

With all the mathematics and understanding now. All we care is if this system works and i can say it works!

Below is attached a Jypyter Notebook of my experiment using Open-CV.

2.0.1 Objective: Inject a watermark into a video

In this notebook i attempt to inject a transparent watermark into a video. The watermark is a png image with transparent background. The watermark is injected into the video by overlaying the watermark on the video frame

by frame. The watermark is overlayed on the video frame by frame using the following steps:

1. Read the video frame by frame
2. Resize the watermark to the size of the video frame
3. Overlay the watermark on the video frame
4. Write the video frame to the output video buffer
5. Repeat steps 1-4 for all the video frames
6. Save the output video buffer to a file
7. Display the output video with watermark transparently overlayed on the video

```
import cv2
import numpy as np

def InjectWatermarkInVideo(position,
    video_path, watermark_path, output_path,
    opacity):
     # make watermark transparent
     watermark = cv2.imread(watermark_path,
         cv2.IMREAD_UNCHANGED)
     watermark = cv2.resize(watermark, (100,
         100))

     # read video using video capture
     video = cv2.VideoCapture(video_path)
     # fourcc is a 4-byte code used to specify
          the video codec
     fourcc = cv2.VideoWriter_fourcc(*'mp4v')
     # we get the fps, width and height of the
          video
     fps = video.get(cv2.CAP_PROP_FPS)
     width = int(video.get(cv2.
```

```
    CAP_PROP_FRAME_WIDTH))
height = int(video.get(cv2.
    CAP_PROP_FRAME_HEIGHT))
# we create a video writer object
video_writer = cv2.VideoWriter(
    output_path, fourcc, fps, (width,
    height))

# allows to set position of watermark
match position:
    case 'top_left':
        position = (0, 0)
    case 'top_right':
        position = (width - watermark.
            shape[1], 0)
    case 'bottom_left':
        position = (0, height - watermark
            .shape[0])
    case 'bottom_right':
        position = (width - watermark.
            shape[1], height - watermark.
            shape[0])
    case 'center':
        position = (width//2 - watermark.
            shape[1]//2, height//2 -
            watermark.shape[0]//2)
    case 'bottom-center':
        position = (width//2 - watermark.
            shape[1]//2, height -
            watermark.shape[0])
    case 'top-center':
        position = (width//2 - watermark.
            shape[1]//2, 0)
```

```
        case 'left-center':
            position = (0, height//2 -
                watermark.shape[0]//2)
        case 'right-center':
            position = (width - watermark.
                shape[1], height//2 -
                watermark.shape[0]//2)
        case _:
            position = position

    # while video is running
    while True:
        # read frame by frame
        ret, frame = video.read()
        if ret == True:
            # add watermark to frame
            add_transparent_image(frame,
                watermark, position[0],
                position[1])
            # write frame to output video
                buffer
            video_writer.write(frame)
        else:
            break
    # free resources
    video.release()
    video_writer.release()
    cv2.destroyAllWindows()

InjectWatermarkInVideo('bottom-center', '
   video.mp4', 'watermarksample.png', 'output
   .mp4', 1)
```

2.0.2 Inputs and Outputs

I have used the following inputs and outputs for my experiment.

- Input Video: video.mp4
- Input Watermark: watermarksample.png

The input video looks like this:

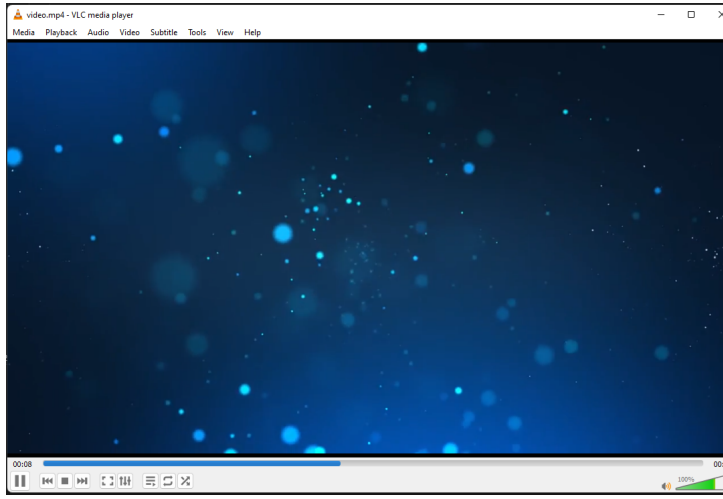

Figure 2.1: Input Video

Our watermark looks like this:
The output video looks like this:

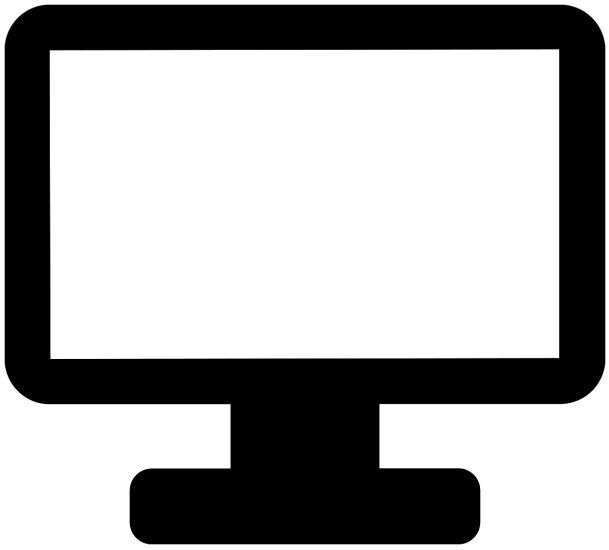

Figure 2.2: Input Watermark

2.0.3 Conclusion

In this notebook/paper, we have learned how to inject a watermark into a video using Open-CV. Now i can successfully inject a watermark into a video. The use case of this technique was to brand a video with a logo or a watermark. This technique can be used to protect the video from being copied or used without the permission of the owner.

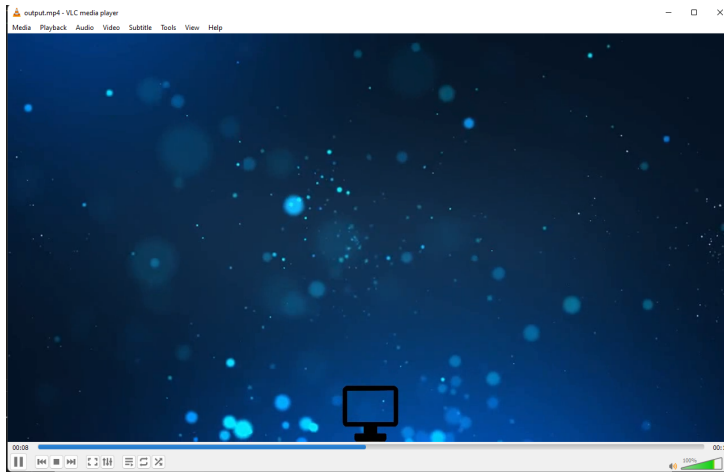

Figure 2.3: Output Video