2022-2023 Informatics Practical Assignment 6 Programming

Showcase | **Full source** | by **Roham Koohestani** and **Rens Verstappen**

Summary

We have created a program which transforms a video file (mp4 format) into an ASCII-character representation of the edges that gets printed to the command prompt. When the main.py file gets run, the user is prompted to select a video file. After a valid file has been selected, the program will first convert the video to a temporary video file, where only edges are visible. Then, the program converts this video to all its frames, by creating a temporary folder and storing all frames of the video in PNG files. When this has been completed, the program will print each frame individually, as if the video is playing naturally. What gets printed is an ASCII-fied version of each frame. This tool has been created using Python 3.11 and several Python libraries, including ones which are not shipped by default: opencv, PIL, pathlib and numpy.

Mark

We believe the tool we have created exceeds the expectations of this project. We believe we went a step above how most people in our grade will experience Python. We used external libraries and complex logic that are not easy to think of on your own, if you've never used Python. Respectfully, we believe that no one in our class is able to complete a project of this complexity. Therefore, as discussed with our teacher, our preferred mark is a 10.

Technical Summary

- The user runs main.py.
- The user has to select a valid file.
  - If the entered file is not valid, the program asks the user to enter another file.
- The source mp4 gets converted to a temporary mp4 where only edges are visible. (to_edged/to_edged_video)
  - All the video's frames are retrieved.
  - Each has several effects applied. (to_edged/find_edges)
  - Each affected frame is compiled to a new mp4 file, called _output.mp4.
- All the output mp4's frames are compiled to PNGs in a folder, called _temp. (to_frames/to_frames)
  - All the output video's frames are retrieved.
  - Each frame gets saved to a PNG with its name being the frame index.
- Each frame gets individually printed to the console. (to_terminal/play)
  - Video info is gathered. (fps, frame count, video duration)
  - Each image gets converted to an ASCII representation of all pixels and printed (to_ascii/print_frame)
    - First, the image gets put into memory using PIL. (to_ascii/to_image)
      - If image is not found, stop printing of video.
    - Each pixel of the image instance is converted to a luminance value (grayscale value, range: 0-255). (to_ascii/to_luminance)

            - First, to optimize for performance and console printing, the image is resized.
            - The resized images' pixels are converted to luminance values.
            - The pixels are compiled to a 2D-array for easier printing.
        - Each pixel's luminance value is converted to an ASCII character. (to_ascii/to_char)
            - This is done by scaling the luminance value to a list of ASCII characters we have compiled.
                - The higher the luminance value, the whiter it is, so the character should fill more of the space.
    - Between frames, to ensure the same fps as the source video, the main thread is paused by the time it took to get the ASCII representation and print it.
- Information about the entire process gets displayed: how many seconds it took to print the entire video, how much % this differs from the actual video length and which frames were "dropped" (not displayed) if the processing took too long.
- The _temp folder is deleted and opencv has all memory released.

Todo update on final day~

Full Code

main.py

```python
import os
import shutil
from pathlib import Path

import cv2

import to_edged
import to_frames
import to_terminal


def main():
    print("===================================================")
    print("Enter your source video.")

    while True:
        src = input().strip()

        if os.path.exists(src):
            break

        print("That file does not exist. Try again.")

    # get the parent folder to do stuff in
    parent = Path(src).parent.absolute()

    print("Applying edge detection...")
    # convert to edged video
    to_edged.to_edged_video(src, parent)

    print()
    print("Converting frames...")

    # create folder with all frames as pngs
    # isn't done on the fly to ensure the console can play the video at
preferred fps
    folder, video = to_frames.to_frames(src, parent)

    print()
    print("All good now. Enjoy the show!")

    to_terminal.play(folder, video)

    # remove folder with frames
    shutil.rmtree(folder, ignore_errors=True)


if __name__ == '__main__':
    main()
```

to_edged.py

```
import math

import cv2
import numpy as np


def find_edges(image):
    """Converts the provided image to one where only edges are visible."""
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (3, 3), 0)

    # Perform Canny edge detection
    edges = cv2.Canny(blurred, 25, 200)

    # Perform dilation to fill in gaps in the edge map
    kernel = np.ones((5, 5), np.uint8)
    # dilated = cv2.dilate(edges, kernel, iterations=1)
    closed = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel)

    # Find contours in the edge map
    contours, _ = cv2.findContours(closed, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

```
    # Create a mask using the contours
    mask = np.zeros_like(closed)
    cv2.drawContours(mask, contours, -1, 255, -1)

    # Extract the object using the mask
    object = cv2.bitwise_and(image, image, mask=mask)

    # return the object
    return object


def to_edged_video(src, parent):
    # Load the input video
    video = cv2.VideoCapture(src)

    # Get the video frames per second (fps) and frame size
    fps = video.get(cv2.CAP_PROP_FPS)
    frame_size = (int(video.get(cv2.CAP_PROP_FRAME_WIDTH)),
int(video.get(cv2.CAP_PROP_FRAME_HEIGHT)))

    # the current frame
    frame_idx = 0
    frames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
    intervals = list(range(frames))[0::int(frames / 10)]

    # Define the codec and create the output video
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")

    output_video = cv2.VideoWriter(str(parent / "_output.mp4"), fourcc,
fps, frame_size)

    # Process the video frames
    while video.isOpened():
        # Read a frame from the video
        success, frame = video.read()

        # If the frame is not None, process it
        if not success:
            break

        # Convert the frame using the find_edges function
        object = find_edges(frame)

        # Write the processed frame to the output video
        output_video.write(object)

        if frame_idx in intervals:
            print(f"{math.ceil(frame_idx / frames * 100)}% completed...
({frame_idx} / {frames})")

        frame_idx += 1

    # Release the video and output video objects
    video.release()
    output_video.release()

    return output_video

```

to_frames.py

```
import math
import os
from pathlib import Path

import cv2


def to_frames(src, parent):
    """Converts the provided video at file path src to all frames in that
video."""
    # gets the video
    video = cv2.VideoCapture(str(parent / "_output.mp4"))

    frames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
    intervals = list(range(frames))[0::int(frames / 10)]  # for the ...
completed messages
    frame = 0

    folder = Path(src).parent.absolute() / "_temp"
    if not os.path.isdir(folder):
        os.makedirs(folder)  # create folder

    while True:
        success, image = video.read()

        # if frame is not found, stop and return data to be used in the
rest of the code
        if not success:
            return folder, video

        cv2.imwrite(str(folder / f"{frame}.png"), image)

        if frame in intervals:
            print(f"{math.ceil(frame / frames * 100)}% completed...
({frame} / {frames})")

        frame += 1

```

to_terminal.py

```
import time

import cv2

import to_ascii


def play(folder, video):
    """Prints a video to the console."""
    # options
    fps = video.get(cv2.CAP_PROP_FPS)
    frames = video.get(cv2.CAP_PROP_FRAME_COUNT)
    sec_time = frames / fps

    ns_between_frames = 1 / fps * 10 ** 9
    ns_init_sleep = 1_000_000  # time to initiate sleep

    result = 0
    frame = 0
    dropped_frames = []
    begin = time.time_ns()

    while result is not None:
        before = time.time_ns()

        result = to_ascii.print_frame(str(folder / f"{frame}.png"))

        # to ensure video plays at 30 fps we need to calculate the time it
has taken to print and reduce this
        # from wait so the next frame plays on time
        elapsed_time = time.time_ns() - before

        # initiating sleep causes some ns to pass. remove this from
time_to_wait to reduce inaccuracy from
        # ~2.5% (224 secs in execution) to ~-0.3% (218 secs in execution)
        ns_to_wait = ns_between_frames - elapsed_time - ns_init_sleep

        # if the ns to wait is negative, aka the reading took too long, add
this frame to the "dropped'
        if ns_to_wait < 0:
            dropped_frames.append(frame)
            # can't wait negative amount of time
            ns_to_wait = 0

        frame += 1

        time.sleep(ns_to_wait * 10 ** -9)

    end = time.time_ns()

    # measure time it took to actually play video
    deltasecs = (end - begin) / 10 ** 9
    # stats
    print(f"Took: {deltasecs} | "
          f"Accuracy: {(deltasecs - sec_time) / sec_time * 100}% | "
          f"Dropped frames: {dropped_frames}")
```

to_ascii.py

```
from PIL import Image


def to_image(src):
    """Returns a new Image instance to perform operations on."""
    return Image.open(src)


def to_luminance(image):
    """Returns a 2D-list of all rgba values mapped to the specified detail
level."""
    width = int(220)
    height = int(63)

    image = image.resize((width, height)).convert("L")

    pixels = list(image.getdata())
    final = []

    for y in range(height):
        final.append(pixels[(y * width):((y + 1) * width)])

    return final


grays = "#@$%&?!*~^;:'+=-_,.` "


# grays = "#W@$8%M&0GRBENHFKSAX?
PQD45O3YZ62TLC7UVJI1!*~wmikltbdfheagypqsjzxnrvouc^;:'+=-_,.` "
# i just used all the keys on my keyboard lol
# i prefer the smaller version since letters are distracting but if
extended palette is needed you can uncomment it


def to_char(luminance):
    """Transforms a luminance value to an ASCII character."""
    return grays[int((1 - luminance / 255) * (len(grays) - 1))]


def print_grid(pixels):
    """Prints the final pixel grid with the provided pixels at the provided
positions."""
```

```
    to_chars = list(map(lambda row: list(map(to_char, row)), pixels))

    total = ""

    for row in to_chars:
        total += "".join(row) + "\n"

    print(total)


def print_frame(file):
    """Prints the specified file."""
    try:
        img = to_image(file)
    except FileNotFoundError:
        return None

    dots = to_luminance(img)

    print_grid(dots)

    return True
```

Code Explanation

main.py

```
import os
import shutil
from pathlib import Path

import cv2

import to_edged
import to_frames
import to_terminal
```

This code imports all the modules we need to run the main procedure of our program. Importing these modules gives our code access to these modules' features. Importing has two steps: first, the module name is found. Then, that module gets bound to a local variable. The module name is the part after `import`. Syntax: `import (module name)`

The first four import statements are used to import external modules. The last three are used to import local program files. The `from pathlib import Path` import statement imports a specific class from a module. Syntax: `from (module name) import (names of classes)`

```
def main():
    print("====================================================")
    print("Enter your source video.")
```

In the above code, a function called `main` with no arguments is defined. Syntax: `def function_name(arguments seperated by a ,):` After the `:`, every line of code that has been "indented" (preceded by a tab/spaces) will be executed when the function is called. In this case, the `print` function is called. Print prints the provided arguments to the standard output. Syntax: `function_name(arguments seperated by a ,)` The arguments are, in this case, a string literal. A string is text which has been surrounded by quotation marks (a.k.a. air commas). Syntax: `"your text here"`. This snippet features 2 `print` statements.

```
    while True:
        src = input().strip()

        if os.path.exists(src):
            break

        print("That file does not exist. Try again.")
```

The above code starts with a while-loop. Syntax: `while (condition):` A while-loop is a form of loop: it activates the code within the body over and over again, until the condition is false. The condition has to be a boolean value: `True` or `False`. `False` is the opposite of `True`. When the condition is true, the code within the body will be repeated until the condition becomes `False`. If it is `False`, it will not again execute the body. In this case, the value is set to `True`. `True` is a constant value which does not equal `False`. Therefore, the body will be repeated indefinitely.

The second line consists of a variable assignment. Variables are a way to easily associate data with a callable identifier. Syntax: `variable_name = data`. In this case, the variable is named `src`, which means `source`. This variable will contain the source video. The data, in this case, is equal to `input().strip()`. Like print, these are 2 function calls.

- `input` asks the user to input some text. In this case, we want the user to input the source video.
- `strip` removes any trailing and leading spaces from the input string. If these are not removed, it may cause problems later.

Line 4 is an if statement. An if statement performs the code within its body when the condition equals `True`. Syntax: `if (condition):`. In this case, we want to check whether the file, provided by the user, actually exists. To do this, we use the `os` module. Then, we use the `path` property of and then the `exists` method. In this method's arguments, we pass the path the user provided. If this path exists, the method will return `True`. If it doesn't exist, it returns `False`. If the file exists, we want to stop our infinite loop. This is done with the `break` keyword. If the file doesn't exist, we want to user to enter another file. The loop continues. After the if-statement, a print statement is used to tell the user they may input another path. After that print statement, the body of the while loop returns to the start; to the `input().strip()`. This way the user may only continue when they have entered a file that exists.

```
    parent = Path(src).parent.absolute()
```

The above code assigns the variable `parent` to the parent folder of the provided file path. The parent folder of the provided path is gathered using the `Path` class from `pathlib`. `Path(src)` creates a new instance of the `Path` class. Then, we use the `parent` attribute of the class to get the parent folder. The `absolute()` method returns the absolute file path (from the source drive) to the parent of the provided file.

```
    print("Applying edge detection...")

    # convert to edged video
    to_edged.to_edged_video(src, parent)
```

Next, we let the user know what is happening with another print statement. Then, from the `to_edged.py` file, the method `to_edged_video` is called. This takes the source file `src` and parent folder `parent` as arguments. This will be explained in detail in the `to_edged.py` chapter.

```
    print()
    print("Converting frames...")

    # create folder with all frames as pngs
    # isn't done on the fly to ensure the console can play the video at
preferred fps
    folder, video = to_frames.to_frames(src, parent)
```

Afterwards, we use a print statement with no arguments to print a new line. Then, again, we let the user know what is happening with another print statement. Finally, we invoke the `to_frames` method from `to_frames.py`, with the same arguments as `to_edged_video` from `to_edged.py`. This method returns a 2-tuple, which we use to define several new variables on one line. Syntax: `var1, var2 = 2-tuple`. A tuple is a way to send several pieces of information at the same time. A method may, for instance, want to return `True` and `32`. To do this, you can save these values in a tuple and return it. Syntax: `val1, val2`. In this case, `folder` is the folder in which all frames of the output video have been put. `video` is the special opencv version of the output video. We store this and pass it on to the next method, to avoid reading the output video twice.

```
    print()
    print("All good now. Enjoy the show!")

    to_terminal.play(folder, video)
```

Next, we print another new line using an empty print statement. We let the user know that the video finished converting, and that they may enjoy the displaying. Then, the method `play` from `to_terminal.py` is called. This prints the video to the standard output. This takes the `folder` and `video` arguments we received when we invoked the `to_frames` method from `to_frames.py`.

```
    # remove folder with frames
    shutil.rmtree(folder, ignore_errors=True)
```

Finally, we use the `rmtree` method from the `shutil` module to remove the folder that contains all the frames as PNGs. We pass the `folder` variable to it, to indicate which folder to remove. We also set the optional variable `ignore_errors` to `True`. This indicates that the method should ignore any errors which may pop up. By default, this is set to `False`.

```
if __name__ == '__main__':
    main()
```

The above piece of code prevents users from accidentally running the entire file if they ever try to import it. It consists of an if statement, where a special variable called `__name__` is checked to see if it is equal to `"__main__"`. When a user executes this code, `__name__` is equal to `"__main__"`. When this file is imported by another file, `__name__` will equal the file name, which in this case is `"main"`. `"main"` does not equal `"__main__"`, therefore preventing accidental execution.

to_edged.py

This file of the project is intended to turn an input mp4 file into an edge detected version of the same input. Through the most prodominant use of the libraries `opencv` and `numpy` in this file (imported respectively as `cv2` and `np`) an edge detected version of input video is created using a video coded included with the opencv library. What follows is a detailed explanation of the workings of `to_edged.py`, however, some methods are not thoroughly elaborated upon seeing the exceed the scope of this project and would only be the cause of confusion.

```
import math

import cv2
import numpy as np
```

As previously dicussed This file of the project makes extensive use of the `opencv` and `numpy` libraries. Additionally the built-in math library of python is imported for the later use of its ceil function in parts of the document to follow. The syntax used here for import will not be covered seeing it has been covered in previous sections.

```
def find_edges(image):
    """Converts the provided image to one where only edges are visible."""
    .
    .
    .
    .
    # return the object
    return object
```

The function defined here `find_edges` with the previously defined syntax is utilized in the document as a converter function for any given frame. The only function parameter, namely `image`, is a single numpy array containing the data from a singular frame from any given video. This frame is obtained through the `opencv` library (elaboration to follow). After multiple operations and coversions have been preformed on the provided `image` a new `object` i.e. a new instance of a frame is returned as the value of the invocation of the function. This returned value will then be used in further sections of the code to generate an output video.

What follows are elaborations of the code inside the find_edges function

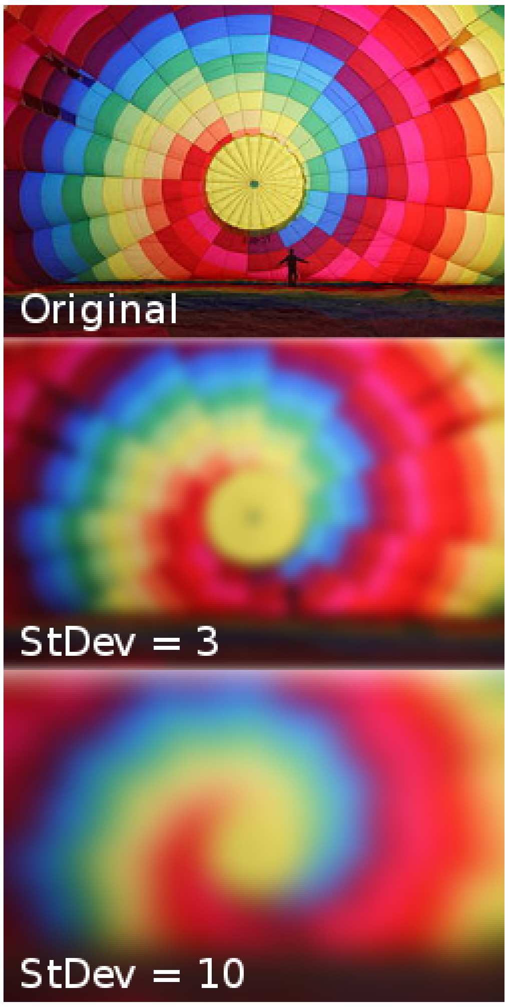
---

```
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

This part of the function takes the provided `image` and the using the image converter function of the `opencv` library in order to get a greyscale representation of the given image according to the opencv `COLOR_BGR2GRAY` colorscale. this value is the stored in the variable gray.

```
    # Apply Gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray, (3, 3), 0)
```

Here, a Gaussian kernel is used in order to smoothen the image in order to optimize the process of edge detection in further parts of code. The provided arguements here are respectively the graysclae representation of the provided image, the x and y standard deviations, and the border type. The function then applies a two dimensional gaussian blur to the image similar to that noticable in the example below. Additionally the basis of the gaussian formula has been also provided.



$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

```
    # Perform Canny edge detection
    edges = cv2.Canny(blurred, 25, 200)
```

After obtaining the blurred representation of the input image, it is passed to the opencv implementation of Canny edge detection with the parameters being respectively the input image and the lower and upper thresholds of the detector. Seeing the implementation of this method exceeds the scope of this assignment the working of this function shall not be covered in this report.

```
    # Perform dilation to fill in gaps in the edge map
    kernel = np.ones((5, 5), np.uint8)
    # dilated = cv2.dilate(edges, kernel, iterations=1)
    closed = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel)

    # Find contours in the edge map
    contours, _ = cv2.findContours(closed, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

After having obtained the edges from the detector it is time to optimize for curves and rough edges that might not have been detected by the default edge detector. In a sequence the morphologyEx and findContours methods from `opencv` are called.

- The variable kernel is is a 5x5 numpy array that is filled with numeric values of type np.uint8 meaning that they are 8 bit numbers (because no more is needed for grayscale iamges that are represented by luminocities from 0 trhough 256 = 2^8)
- The `MorphologyEx` method is a combination of an Erosion (erodes away the boundaries of foreground object) and Dialation (basically reverse Erosion)method that are applied automatically to the input image using the created kernel. This method takes care of removing any unwanted noise is removed from the image whilst keeping the original shape of the image.
- The `findContours` method creates a smooth and consistent line through the edges that have been detected returning a consistent contour of the edge detected `object`. What `CHAIN_APPROX_SIMPLE` does is ensures the removal of redundant poitn and compresses the contour so more memory is saved. The contours i.e. the smoth edges are then stored in a variable `contours` and utilized in further parts of the code.

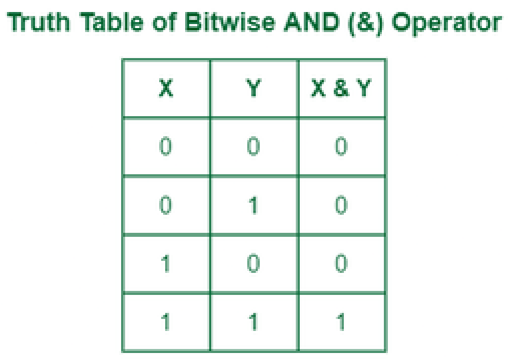
```
    # Create a mask using the contours
    mask = np.zeros_like(closed)
    cv2.drawContours(mask, contours, -1, 255, -1)

    # Extract the object using the mask
    object = cv2.bitwise_and(image, image, mask=mask)
```

After having obtained the contours of the image it is time to transpose these edges onto to original image and get an edge detected representation of the image. To do so, however, first a mask needs to be created which can then be applied to the image. Here, in the first line using the `numpy` library a zero-filled array is created in the shape of the `closed` array. Subsequently the `drawContours` method of opencv is called which does as it is named, given a mask and countours (previously obtained from `findContours` (so logical right!!)) it draws out the contours contours of the `contours` variable onto the mask. The three numeric values then represent respectively the contour to draw (here -1 indicates all), the color of the contour, and finally the thichkness of the contour (with -1 representing the inside of the contour).

By then applying the obtained mask through a `bitwise_and` operation onto the original image an `object` or rather an edge detected representation of the image is obtained which can then be returned as the value of the invocation as discussed previously.

The truth table for the `bitwise_and` operator is as follows



Truth Table of Bitwise AND (&) Operator

X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1

```
def to_edged_video(src, parent):
    .
    .
    .
    .
    return output_video
```

The functiond definition of the `to_edged_video` is shown here. Given the source of the input video (`src`) and the absolute path to the parent directory of the video (`parent`) it creates a new edge detected video from the input video and encodes/saves it to an output video `_output.mp4` in the parent directory of the input video. Furthermore, this function returns the path to this video to the invocator so that it can be used to print the results to the terminal. (explantion will follow).

```
    # Load the input video
    video = cv2.VideoCapture(src)

    # Get the video frames per second (fps) and frame size
    fps = video.get(cv2.CAP_PROP_FPS)
    frame_size = (int(video.get(cv2.CAP_PROP_FRAME_WIDTH)),
int(video.get(cv2.CAP_PROP_FRAME_HEIGHT)))

    # the current frame
    frame_idx = 0
    frames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
    intervals = list(range(frames))[0::int(frames / 10)]
```

In a sequntal fashion the following operations are preformed and their returning values are stored in their respective variables.

- The video from `src` is retrieved through the opencv's `VideoCapture` method and is stored inside of a variable video.
- The video contains an object referring to the input video. This Video object has a `get` method that can be used to retrieve different values and properties about the video. Here, for example the frame count and the frame size of the video are retrieved. Notice that to ensure integer type numbers in relation to the frame size, they are being typecaseted to te integers after their valueas are retrieved.
- The frame index inside of the video is set to 0 and the total frame count of the video is retrieved (with their values being respectively stored in `frame_idx` and `frames`).
- The last line of code presented above serves as type of metric for the user to be able to track the progress of progression. What this line does is take the list of frames and in intervals of (totalframes / 10) take numbers such that updates are given to the user at percentages of interval ~ 10%

```
    # Define the codec and create the output video
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    output_video = cv2.VideoWriter(str(parent / "_output.mp4"), fourcc,
fps, frame_size)
```

These lines serve the purpose of creating a codec/encoder for the output video aand are standrad lines of code coppied from the documentation of the opencv library. Seeing not much has been changes about how this method is utilized not much needs elaborating besides the fact that the output is going to be a mp4 and the the name of the file is going to be `_output.mp4` and that it will be stored inside the parent directory of the input source.

```
    while video.isOpened():
        # Read a frame from the video
        success, frame = video.read()

        # If the frame is not None, process it
        if not success:
            break

        # Convert the frame using the find_edges function
        object = find_edges(frame)

        # Write the processed frame to the output video
        output_video.write(object)

        if frame_idx in intervals:
            print(f"{math.ceil(frame_idx / frames * 100)}% completed...
({frame_idx} / {frames})")

        frame_idx += 1
```

These lines of code are used to convert the video into an edge detected version of the same video using the `find_edges` function as previously defined. Using a while loop as defined previously it is checked whether the video is opened on every iteration of teh loop, if so, the frame is read using the videos `read` method and checked whether the reading of the frame was successful; the value of this check is stored inside of the variable success and checked in the line with the first if statement. If there was an error when loading the frame the whole loop will break and result in the program haulting. If no errors are encountered in retreieving the frame it is passed to the previously defined `find_edges` functiona and the value of it's invocation is stored into a variable called `object`. This object or rather image is then written to the output video using the `write` method of the video encoder. The rest of the code serves as the progression metric previosuly elaborated upon.

```
    video.release()
    output_video.release()
```

As the final step in this function the videos that were being written are now released (in other words saved and closed) and can then be utilized by other parth of the whole program.

to_frames.py

The code inside the file to_frames.py is intended to take a source for a video input and consequently convert that video into it's distinct frames and store these extracted frames into a temporary folder to then be used in furthe rsections of the main application (ex. by the `to_terminal.py` code). What follows is a detailed description of how everything is put together inside this file. Some sections are not fully ellaborated upon seeing they have been thoroughly explained in previous sections of this document.

```
import math
import os
from pathlib import Path

import cv2
```

The lines of code here use both previously covered importing syntaxes of the python programming langauge. With one being used to imort the `math`, `os`, and `open-cv` libraries and the other to specifically import the `Path` module from the `pathlib` library.

```
def to_frames(src, parent):
    """Converts the provided video at file path src to all frames in that
video."""
    .
    .
    .
    .
```

The function definition of the `to_frames` function as defined here takes two arguments `src` and `parent` quite similar to previously seen functions inside of the `to_edged.py` file. This function, however, does not return anything and everything is local to this function.

```
    # gets the video
    video = cv2.VideoCapture(str(parent / "_output.mp4"))
```

Using the parent directory of the input video (which is retrieved in the main python file) the output video from the `to_edged.py` call is retrieved and further utilized in the program. The gathering of the data from the video is similar to that in `to_edged.py`.

```
    frames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
    intervals = list(range(frames))[0::int(frames / 10)]  # for the ...
completed messages
    frame = 0
```

See `to_edged.py`

```
    folder = Path(src).parent.absolute() / "_temp"
    if not os.path.isdir(folder):
        os.makedirs(folder)  # create folder
```

Using the the parent directory of the source file a new path is created for a folder called `_temp`. The a check is ran using the `os` library to see whether this directory already exists, if one already exists nothing is done, however, if such a directory does not exist one is made using the `makedirs` function of the os library.

```
    while True:
        success, image = video.read()

        # if frame is not found, stop and return data to be used in the
rest of the code
        if not success:
            return folder, video

        cv2.imwrite(str(folder / f"{frame}.png"), image)

        if frame in intervals:
            print(f"{math.ceil(frame / frames * 100)}% completed...
({frame} / {frames})")

        frame += 1
```

Similar to the workings of `to_edged.py` a while loop is utilized in order to sequentially read all frames of the given video and using the `imwrite` function of the `open-cv` library store these into the temporary folder with the name of the file being the index of the given frame inside the video. The rest of the guidelines previously discussed apply to this loop as well.

to_ascii.py

The file `to_ascii.py` is the program file that contains the main logic associated with converting a given image into the ascii representation of that given image. Utilizing the `Pillow` library imported as `PIL` inside of the program the process of this conversion is eased and shall be elaborated upon.

Roham's side note: The extensive use of helper functions can be well associated with the programming style of Rens which I highly appreciate. Seperation of concerns is key. My man is and will be a very good functional programmer one day. I just know it 😃

```
from PIL import Image
```

This is the first occurence of the usage of the `Pillow` library inside the program. This library as described on the website of the library is intended to add image processing capabilites to the python interpreter. With it's extensive format support and well-documented codebase it was one of the best options we had. Here we are specifically importing the `Image` module of the `Pillow` library.

```
def to_image(src):
    """Returns a new Image instance to perform operations on."""
    return Image.open(src)
```

Here, the defined `to_image` function takes a soruce/path of an image and returns the image opened using the `open` method of the `Image` module of `PIL`.

```
def to_luminance(image):
    """Returns a 2D-list of all rgba values mapped to the specified detail
level."""
    width = int(220)
    height = int(63)

    image = image.resize((width, height)).convert("L")

    pixels = list(image.getdata())
    final = []

    for y in range(height):
        final.append(pixels[(y * width):((y + 1) * width)])

    return final
```

The function definiton for `to_luminance` takes an image as it's only input and returns a transformed version of the image corelated to the luminance values associated with each given pixel in the image. Initially, however, the width and height of the given terminal window are taken and stored in similarly labeled variables. The image is then resized using the `resize` method of the `Image` object. note that here the `convert` method is the also being called for the image before it is stored. This is to transoft the iamge into a single channel image, AKA a luminance values of the image without color i.e. grayscale. Following this, the values for all pixels are retrieved by typecasting the returned value from calling the `getdata` method of the image to a `list`. An aditional array is the created called final which is also the 2Drepresentations of luminance mapped rgba values that is returned. seeing the returned value of `image.getdata` is a 1-d array we need to convert it to a list of pixel rows. To do this we create a range which include all rows of the array that we need i.e. the height of the iamge in pixels and then split the 1d array of pixels into a final 2d array of n arrays where n represents the pixel height of the image.

```python
grays = "#@$%&?!*~^;:'+=-_,.` "
```

This string is an ordered collection of characters that will be used to convert an image into ascii art. sorted in decending order based on darkness, `#` represents the drakest symbol and the `space` represents the lowest value.

```python
def to_char(luminance):
    """Transforms a luminance value to an ASCII character."""
    return grays[int((1 - luminance / 255) * (len(grays) - 1))]
```

The `to_char` function as defined here takes in one input that is a given luminance value and returns the associated ascii char from the `grays` collection.

```python
def print_grid(pixels):
    """Prints the final pixel grid with the provided pixels at the provided positions."""
    to_chars = list(map(lambda row: list(map(to_char, row)), pixels))

    total = ""

    for row in to_chars:
        total += "".join(row) + "\n"

    print(total)
```

Provided the returned value from the `to_luminance` function as an argument named `pixels`, the `print_grid` function proceeds to convert the given array of pixels into a string representation and prints it. This function does not return anything. The value of the `to_chars` function is obtained by using the map function which applies a function to all values of a given array; This is first done on the first dimension of the array which then allows to manipulate the specific cells inside this 2-d array. The inner map function applies the previously defined `to_char` method to the all the values. After both map's have finished executing the result is typecased to a list and stored to the `to_chars` array. An empty string called `total` is the created and per row inside the `to_chars` array the concatenated values of the rows as a string are concatenated to the `total` string. additionally at the end of each row string representation a `\n` character is added in order to ensure the image is well represented as a string. To finish things off the `total` string is printed.

```
def print_frame(file):
    """Prints the specified file."""
    try:
        img = to_image(file)
    except FileNotFoundError:
        return None

    dots = to_luminance(img)

    print_grid(dots)

    return True
```

The `print_frame` function as defined here takes in a file as it's only input and proceeds to print it using the previuosly defined helper functions. A `try/catch` statement is included to ensure the programs continues it's processing even if the image fails to load. The prevension of haulting causes the smooth printing process of the frames to the terminal. If the process fails this function returns the value `None` and if all goes well the value returned will be `True`. Initially in this function the image is loaded into a variable called `img` inside of the try block. This image is then transformed into luminance values and printed to the terminal using the previosuly defined functions.

to_terminal.py

What follows is the a detailed explanation of the workings of the `to_terminal.py` file. Some lines are not explained seeing constant repetition of certain concepts does not add any additonal value to the report and documentation.

```
import time
import cv2
import to_ascii
```

These import statements, import two libraries and one other previusly elaborated file from the project. This is also the first occurence of the `time` built-in module of python which will be utilized in this file to regulate thread operations and video speed.

```
def play(folder, video):
    """Prints a video to the console."""
    .
    .
    .
    .
```

Given a path folder and video as it's parameters, the `play` function litrally plays a given video inside the terminal where the program is called from.

```
    # options
    fps = video.get(cv2.CAP_PROP_FPS)
    frames = video.get(cv2.CAP_PROP_FRAME_COUNT)
    sec_time = frames / fps

    ns_between_frames = 1 / fps * 10 ** 9
    ns_init_sleep = 1_000_000  # time to initiate sleep

    result = 0
    frame = 0
    dropped_frames = []
    begin = time.time_ns()
```

This part of the code resembels other parts of the proejct that utilize the `open-cv` library and how values of certain attributes are retrieved using the `get` method. A new concept however is that of time and video length. The variable `sec_time` contains the length of the video in second. This value is then used to calculate the time between frames in nanosecond and then this value is stored in `ns_between_frames`. Some initial variables are also defined here namely those of `result`, `frame`, `dropped_frames`, and the inital start time of the program stored in `begin`.

```
while result is not None:
        before = time.time_ns()

        result = to_ascii.print_frame(str(folder / f"{frame}.png"))

        # to ensure video plays at 30 fps we need to calculate the time it
has taken to print and reduce this
        # from wait so the next frame plays on time
        elapsed_time = time.time_ns() - before

        # initiating sleep causes some ns to pass. remove this from
time_to_wait to reduce inaccuracy from
        # ~2.5% (224 secs in execution) to ~-0.3% (218 secs in execution)
        ns_to_wait = ns_between_frames - elapsed_time - ns_init_sleep

        # if the ns to wait is negative, aka the reading took too long, add
this frame to the "dropped'
        if ns_to_wait < 0:
            dropped_frames.append(frame)
            # can't wait negative amount of time
            ns_to_wait = 0

        frame += 1

        time.sleep(ns_to_wait * 10 ** -9)
```

This loop represents the main operation that is running when the terminal is printig the video (never used that one before 😃). As long as the video is being printed the loop goes on. The whole process of printing the ascii representation is timed through the `before` and `elapsed_time` variables and are then used to calculate sleep time. In the meantime the `print_frame` function as previously defined is called with the current frame of the video as the input.

The previously mentioned elapsed time is then used to calculate wait time by taking the previously calculated `ns_between_frames` and deducting the elapsed time and the estimated intial sleep. If the waiting time is less that 0, meaning the frame took longer than expected it is added to the `dropped_frames` array and the wait time is set to 0. After all these have ran the `frame` counter is incremented and the sleep time is implemented using the `sleep` method from the `time` library.

```
    end = time.time_ns()

     # measure time it took to actually play video
     deltasecs = (end - begin) / 10 ** 9

     # stats
     print(f"Took: {deltasecs} | "
           f"Accuracy: {(deltasecs - sec_time) / sec_time * 100}% | "
           f"Dropped frames: {dropped_frames}")
```

After the main loop has finished executing the end time is logged and all the metrics are presented to the user. The time difference stored in `deltasecs` is calculated by taking the difference between the begin and end time and converted to normal seconds from nano seconds by multiplying by 10^9.

Additionally, the accuracy (meaning the percentage of difference between actual time and time take) and the dropped frames are printed to the user.

Conclusions

These explanations conclude our explanation of the code associated with this project. Many Many hours of explaining later we hope to have elaborated enough.

Best of regards

Roham and Rens