Light Painting Project Writeup

Team Members

Jane Yang, Melis Gökalp, and William Hackett

Code Instructions

Project repository: https://github.com/william-hackett/lightpainting

Use the project by running the following commands in the project directory:

```
For light painting with green pixel tracking:
$ python main.py

For light painting with You Only Look Once (YOLO) hand detection:
$ python main.py --method yolo

For light painting with the background subtraction model:
$ python main.py --method motion
```

Optional flags:

```
To specify the number of objects to track (default=1) :
$ python main.py --objects <number of objects>

To specify the source video input (default=webcam input):
$ python main.py --source <file path to video>

To save frames as a video in the output_videos/ directory:
$ python main.py --save

To paint with a hard-coded color (default is rainbow):
$ python main.py --color

To simulate motion by shifting all drawn points at each frame:
$ python main.py --shift
```

Overview

Our project implements light painting with several different tracking methods and design choices. Given a webcam or video input, the program follows the path of a chosen subject and generates a light trail capturing its movement across frames.

We implemented three approaches for object tracking:

1. **Green pixel tracking**. By specifying the HSV value range of green pixels, we created a mask for green pixels in each frame, found contours of the green areas, and inferred the center of each green contour as the position of the green region.
2. **YOLO hand detection model**. We use the YOLO model to predict the center coordinates of bounding boxes of hand positions and return predictions over a certain confidence threshold.
3. **Background subtraction model**. We use the MOG2 model to create a background model, consisting of stationary pixels, isolate the foreground object, and threshold on the squared Mahalanobis distance between the pixels and the model. We return the center coordinates for the largest bounding boxes.

Once we obtain positions of points corresponding to individual objects via tracking, we need to identify the trajectory of each object respectively. To this end, we assign the points of interest, returned by the specified tracking method, to point groups, where each group of points represents the path of one object. The number of point groups equals the number of objects, which is specified by the user at runtime. We assume that a given object's position at time t+1 will be closest to the same object's position at time t. Therefore, by comparing the distance between object position in the current frame and the past frame, we assign each point position to its corresponding point group. This gives us a representation of each object's distinct path of movement. We pop a point off a given point group when its length reaches 30 points or when no object is detected in order to create the visual effect of drawing.

After identifying the trajectory of an object, we paint a light trail along the object's path by drawing circles between two points, with the number of circles being half of the distance between two points and the circle radius being proportional to the distance. In the default rainbow mode, the color of the light trail loops over the range of RGB values, and we keep track of each path's current color, so that each path can move continuously through the color spectrum.

(1)

Implementation Details

Painting the Lines

Our baseline implementation was drawing a straight line between two points using the cv2.line() function. However, this produced boring and unsatisfactory images. We decided to form a more visually appealing method, which led to our final approach, which consisted of drawing many circles of a certain radius along the sampled linearspace between two points. This allowed us to modulate the size of the ”brush” with which we were painting.

We also implemented an alpha transition along these circles, so that the circles follow a smooth color gradient. This approach generates smoother trails than the previous implementation.

In order to make the line colors continuous and aesthetically pleasing, we implemented a rainbow_loop() function, which loops through the RGB color spectrum by incrementing or decrementing one color channel's value by 15 depending on the current color state, which is represented as a 3-element tuple of color channel values (B, G, R) in the range [0-255]. This produced seamless color transitions.

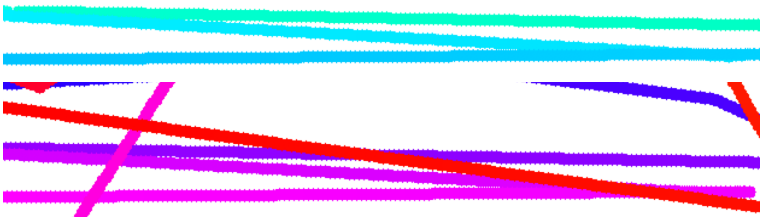

Figure 0: Sample strokes using our custom_line() and rainbow_loop() methods

Listing 1: Drawing circles for light trail

```
points_on_line = np.linspace(p1, p2, int(distance//4))
    for i in range(len(points_on_line)):
        alpha = i/len(points_on_line)
        point = points_on_line[i]
        strip = (np.asarray(c1)*(1.-alpha) + np.asarray(c2)*(alpha))
        output = cv2.circle(output, tuple(point), thickness, strip, -1)
```

Tracking Multiple Objects

We implemented multiple-object tracking by assigning each tracked point to the group which minimizes the distance between the position of the point in the current frame and the stored points from the last frame. We handle edge cases gracefully. For instance, if an object has exited the frame, we gradually erase the corresponding trace by popping points off the list representation of that object's motion path. Also, if we are tracking two objects, our implementation appropriately assigns new points to the stored object

paths when both objects are successfully tracked. However, it also works when only one object is detected, and then resumes drawing the appropriate path if the second object is subsequently detected. This was a much more difficult task than we originally thought, and the final implementation was the best of the many approaches we tried.

Listing 2: Drawing light trail

```
# If more than one newly tracked points, and they are
# at least MIN_DIST = 30 pixels apart, append second newly
# tracked point to the second path
if len(centers) > 1 and np.linalg.norm(centers[0] - centers[1]) > MIN_DIST:
    self.points[1].append(centers[1])
    group_assigned[1] = True
if len(centers) == 1:
    self.points[1].pop(0)
```

Background Subtraction

Generates a foreground mask by performing a subtraction between the current frame and the background model, which contains the static parts of the scene.

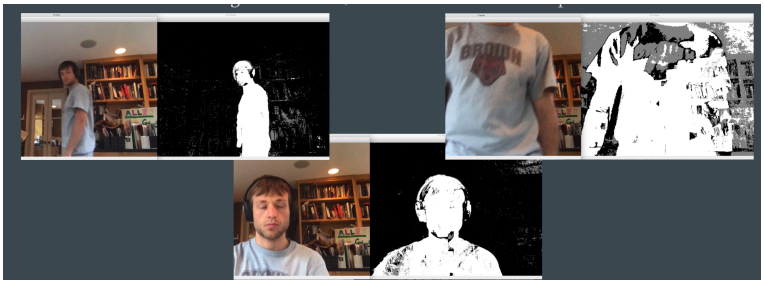

Figure 1: Foreground object mask generated using background subtraction

Limitations

Although we implemented several techniques for generating consistent object masks and drawing smooth lines (such as cv2.erode() and cv2.dilate()), the drawn light trails are not always perfect. Specifically, if the lighting conditions are poor or the video frame rate is very low, especially while using the webcam and Zoom simultaneously, then performance deteriorates. This can be prevented by running the pipeline on a pre-recorded video and controlling for well-lit environments. Generally, we found that the YOLO hand detection model provided the most satisfying results. For green pixel tracking, performance is highly dependent on lighting, and it runs under the assumption that the object constitutes the only green region within the scene. For background subtraction, there is often a high degree of noise, as the background subtraction model generates very large, broad masks based on pixel motion, so it assumes that the object of interest is continuously moving.

We found that this approach was not satisfactory, so we maintain green pixel tracking and YOLO as the best methods for this project.

Results

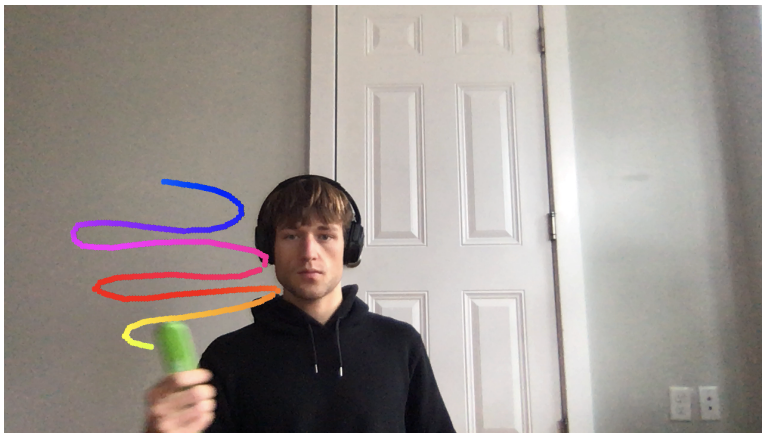

Figure 2: Tracking a green object

Figure 3: Tracking two hands using the YOLO (You Only Look Once) model

Conclusion

Overall, we enjoyed the freedom and creativity inherent in this project. While the underlying idea is quite simple, the implementation proved trickier than anticipated, especially generating smooth lines and tracking multiple objects. We tried several approaches for tracking objects and drawing light trails, and we included several optional flags, so that users can easily change the settings with which they are running the project. Importantly, we refined our code to be as efficient as possible by creating a Painting object class to

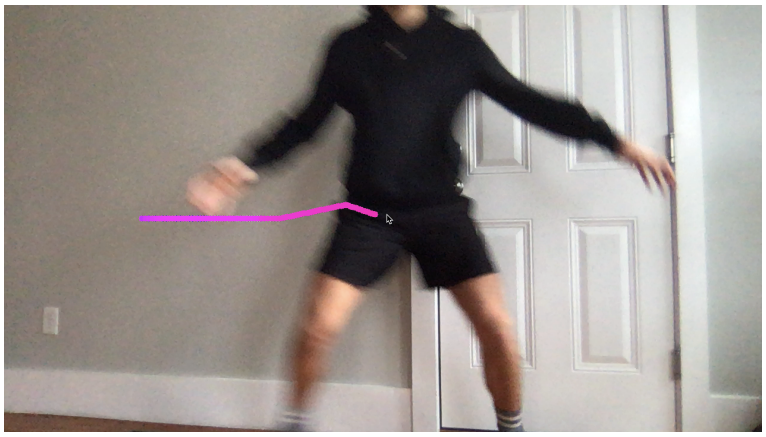

Figure 4: Tracking foreground object using the background subtraction model

Figure 5: Shifting points to simulate rightward motion in the light trails

store points across methods and by minimizing iteration (i.e. for loops). This allowed our project to function well in real-time, drawing light trails quickly even while using the webcam on the user's personal machine. There are still several limitations constraining performance, specifically the amount of light in the user's environment and any lag from their webcam. In the future, it would be interesting to try running this project on high frame rate/high resolution videos and then comparing the results with those obtained by running the project in real-time with a webcam. Ultimately, however, we are happy with our results given the timeline and remote nature of the project! We hope you have fun trying our project for yourself!