Abstract Video Compression

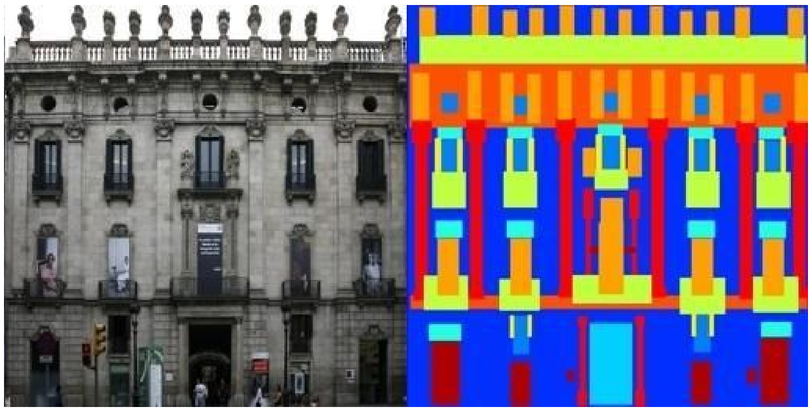
Milan Mauck

Abstract

It is difficult to send video over a very low bandwidth stream. For example, on a drone there is limited power and antenna size, forcing the bandwidth to be minimal. The proposed solution is a type of extreme video compression that would send the basic shapes and colors of each frame and eliminate unnecessary detail. There are a number of both lossless and lossy video compression techniques out there that do a great job at what they were designed to do, but few are able to compress down to an incredibly minimal bandwidth while keeping some sort of recognizable and useful video stream. This project is focused on streams with 3 kbps bandwidth and under. Our solution would provide a clean, recognizable video full of basic shapes meeting this minimal bandwidth requirement by only sending the necessary few bytes per shape. Effectively, we're attempting to find an abstract representation of each frame that looks "close enough" to the original video that its basic features are recognizable to a human observer.

Introduction

The idea here is to preprocess a given video stream by taking each frame, abstracting it into shapes, and streaming only those shapes, their positions, and a single color for that shape. This would theoretically make an incredibly low bandwidth video stream, which would be ideal for low frequency, low power transmissions. Our original idea was for the frames to come in looking like the left image and come out looking something like the right image.



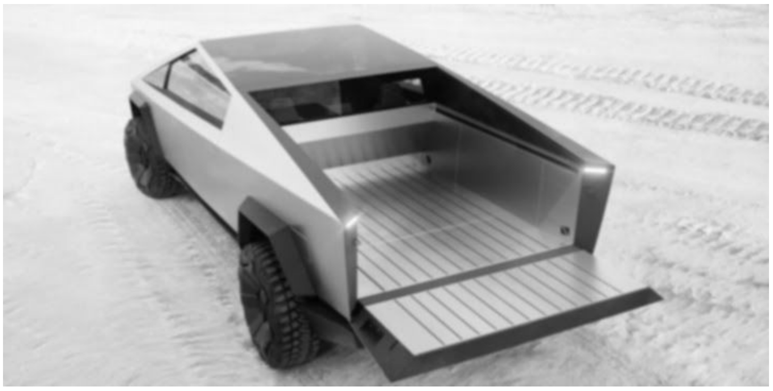
You can see the basic idea of the image is still conveyed, but at far less cost to transmit. Originally, we had experimented with and tried using rectangles to describe images, but this was thrown out in favor of triangles, which are much more flexible for describing a wider variety of shapes.

Method

1. Extract individual frames from input video stream using OpenCV.
    - This would be a step in a practical application - so far we've only used the algorithm on JPEG images.

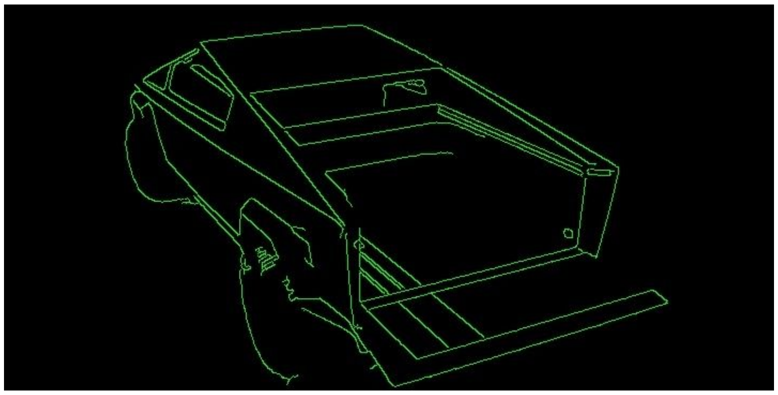
2. Preprocess frame

    - The frame is converted to grayscale, and a 5x5 gaussian filter is applied to smooth the image out.

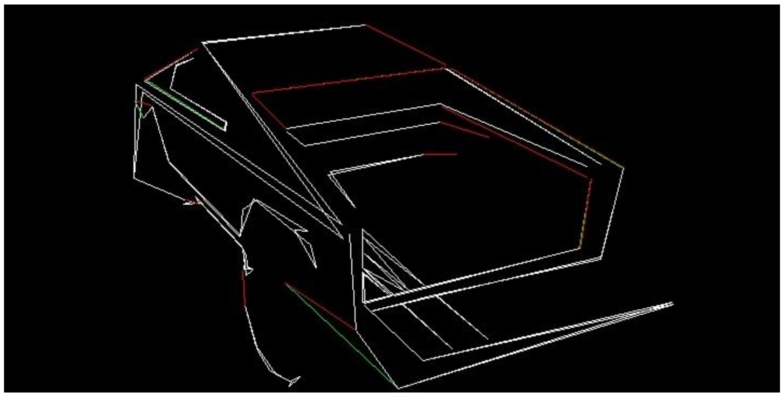


3. On the given frame, detect contours of the frame.
    - This is done with cv2.Canny, which works well for this application.

        - We're using cv2.Canny for this, but there is a high likelihood that we should be using image segmentation first (cv2.Canny). Unfortunately, time constraints have prevented us from taking this next step.



4. Approximate contours to straight lines.
    - This is done with the cv2.approxPolyDP method. Each contour has its minimum side length determined by the contour's arc length.
        - One potential way to improve the results of our algorithm would be to vary the minimum side length based on the size of the object - our current implementation spends a lot of time computing very small triangles for overly detailed representations of small objects.

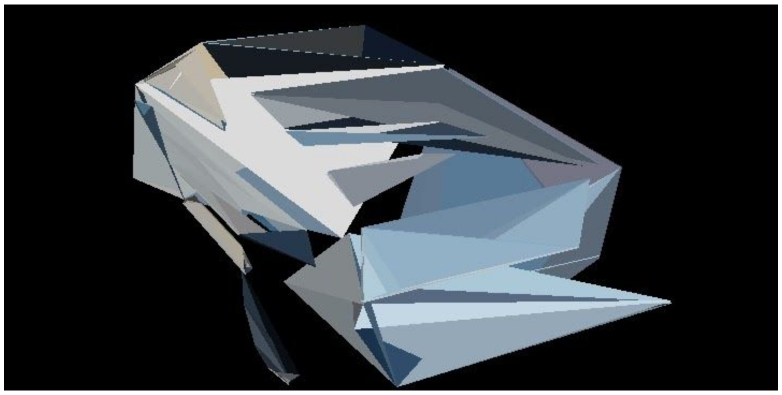


5. With these straight lines, triangulate the entire image based on the points generated.
    - For each contour, connect every vertex to every other vertex with an edge.
        - Except where the edge would intersect an edge already made.
        - This is done in random order, so each triangle doesn't have the first point in the contour as a vertex.
    - For each edge, find the list of edges that share a vertex with that edge.
    - Moving through triples of connected edges, see if they form a triangle.
        - If so, create a triangle based on these edges' vertices, and add it to the list of triangles that composes this polygon.

6. Assign contrasting colors to each triangle generated by flattening the range of colors in the image, similar to some of the algorithms from computer graphics. Then we assign a

representative color to each shape, making sure it doesn't share a color with any shape adjacent to it.

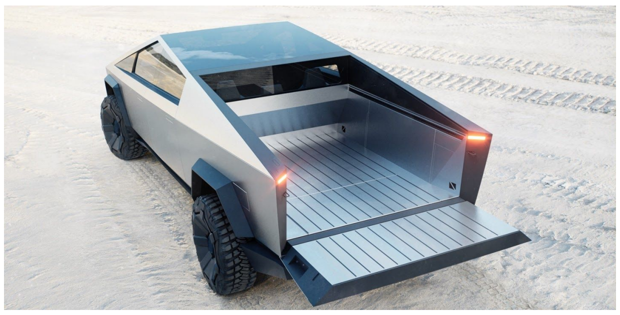
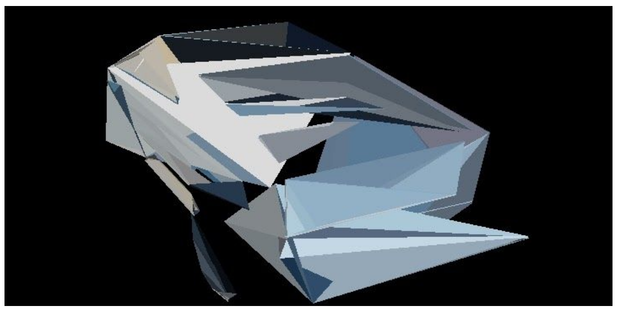
- Current implementation simply finds the coordinates of the midpoint of each triangle, and then finds the color of the pixel at that location in the input image.



7. Send the byte stream representing each triangle and its associated color
    - Similar to the first step, we would implement this in a practical application (finding a compact way to represent triangles in a byte stream), but currently we're just using opencv functions to write the triangulated image to disk as a jpeg.

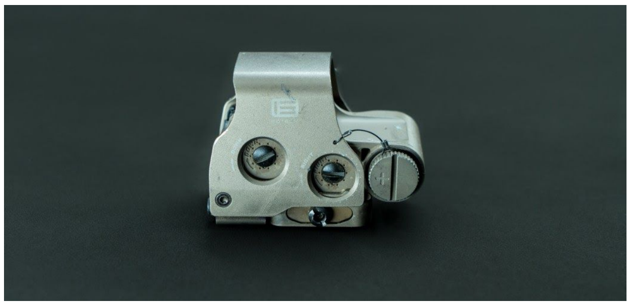
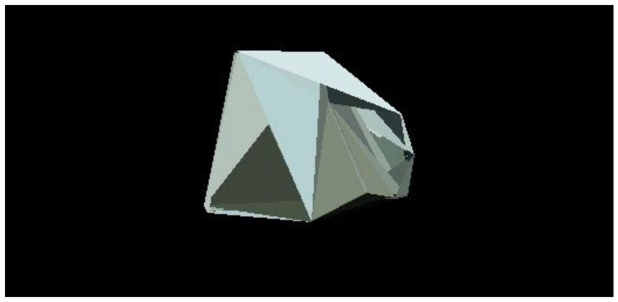
Experiments

Experiment 1:

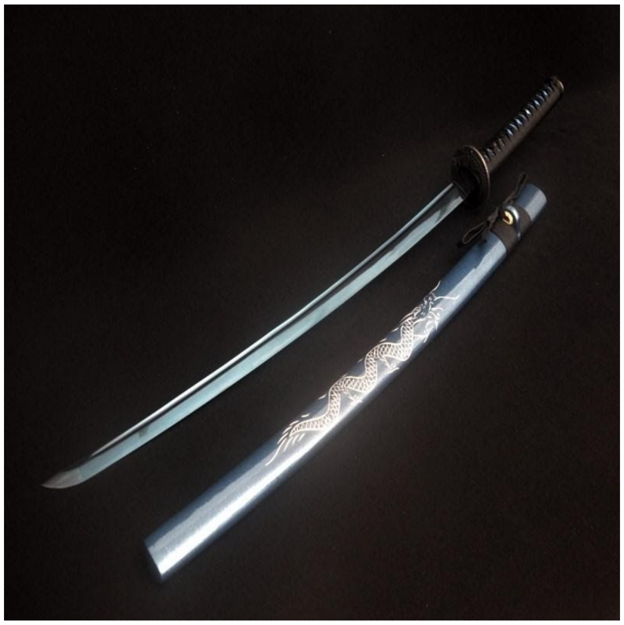
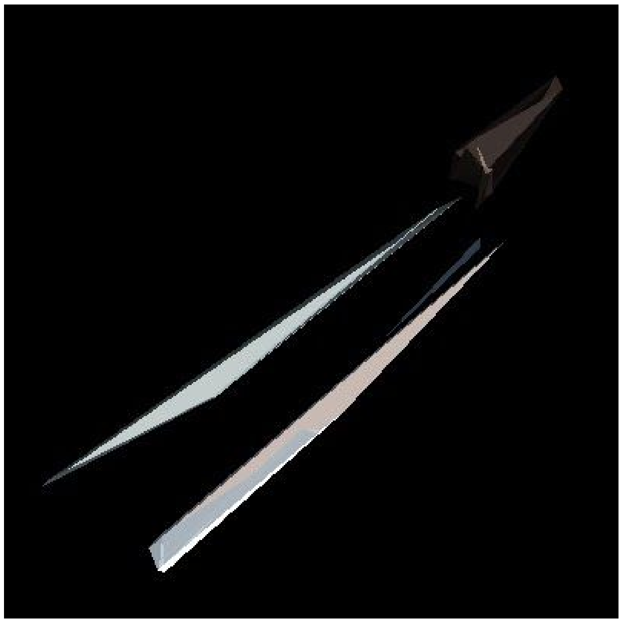
In this first experiment the input frame is on the left and output frame on the right. The results are fairly good, the image has been triangulated and colored well, and much of the overall shape kept. It’s not too hard to recognize that it’s a cybertruck!

Experiment 2:

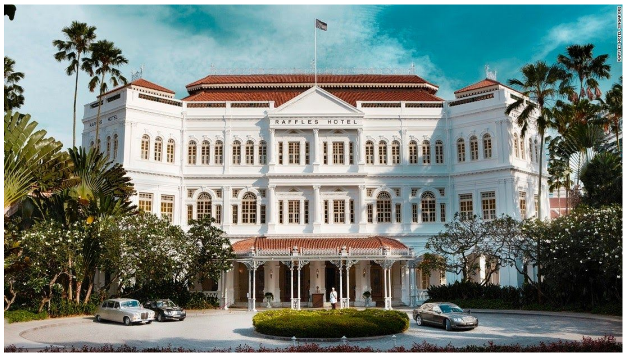
In this experiment the results are fairly good. The overall shape and color of the object is retained with very few triangles used to represent it.

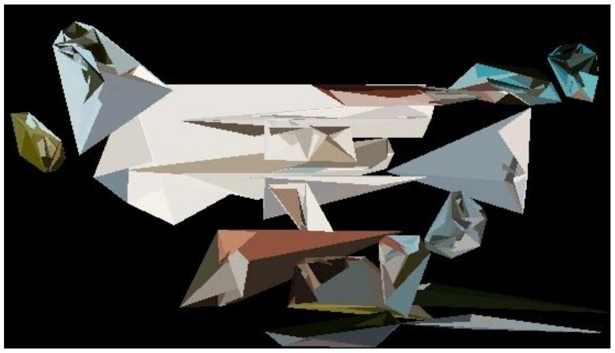
Experiment 3:

In this experiment you can clearly see the basic outline of the sword, handle, and scabbard reflecting in the light. In old experiments the handle would not be retained as it blended too well with the background, but after some major improvements, especially with the preprocessing, you can see the handle shows up quite well in the output image.

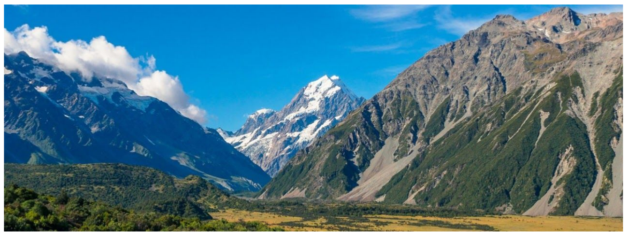
Experiment 4:

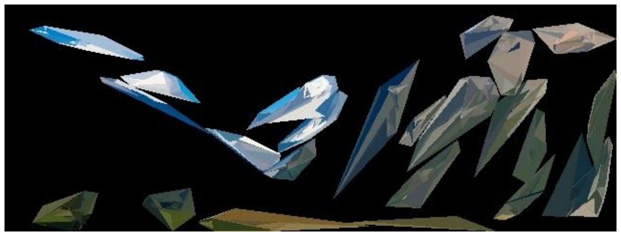




In this experiment, the input image is very busy with lots of shapes and details going on. This image was chosen to see how our algorithms would handle and generalize into triangles so much detail. Overall, the output on the right looks like an absolute mess and not recognizable from the input. However, you can clearly make out some details that carried over well, such as the tree on the left, some of the basic structure of the hotel like the roofs and main walls, and the driveway.

Experiment 5:





In this experiment, the input image is fairly busy, but has some good contrasting colors. The output captures many of the basic outlines and features of the mountains,but much of the smaller detail and overall shapes are lost. Not bad for a very busy image.

Conclusion

We have created a solution for video compression which takes the basic most important features of each frame as triangles and transmits them as a byte stream. Compression was pretty good, our first example with the cybertruck starting at 344KB for the input image and coming down to 4KB worth of triangles, 13 bytes per triangle. Unfortunately not quite hitting my goal, but definitely pretty good results. Many improvements have been made since my last approach to this project. Results are much better thanks to:

- Doing preprocessing on the images with a Gaussian filter to get better contour detections.
- Using cv2.Canny instead of cv2.findContours.
- Optimizations on the triangulation algorithm.
- Numerous tweaks on various constants and thresholds for throwing out contours too small to be worth drawing, saving on computation time. Most of the effort was put here trying to tweak and refine results, especially on busy images.