InstaAuction - 360° Imagery.
ACV Labs
Technical Document

Abstract

Walking around a car and taking a video is trivial. Walking around a car and taking a video with the distance and speed normalized is not. In this iOS application, we try to generate a stabilized 360° low frame rate video that the user can scroll and use while putting their car for an auction.

1. Introduction

InstaAuction, an application developed in iOS is essentially a guide to the ACV Vehicle Condition Inspector(VCI) helping him/her to get a gist of the 360° imagery of the car. The application uses latest technologies like Depth Sensing, Machine Learning, Image Detection, Stabilization to achieve normalized distance walkaround. It involves the VCI moving around the car capturing the best possible frames by following the instructions provided by the application. The application is smart enough to sense the car, detect the side of the car, how fast the user is moving around the car and how far he/she is away from the car. It combines all the technologies and generates a 360° imagery low frame rate video.

As mentioned before the main objective of this application is to help the VCI generate a 360° low frame rate video of the entire automobile. The application initially detects the car using an object detection model. Once the car is detected, the user can start recording the car by moving around the vehicle by following the onscreen instructions. Once the recording is done, the video is run through a stabilization algorithm and later a 360 degree imagery is generated. An in-depth work flow of the app can be found in Figure 1.

Flow Chart:

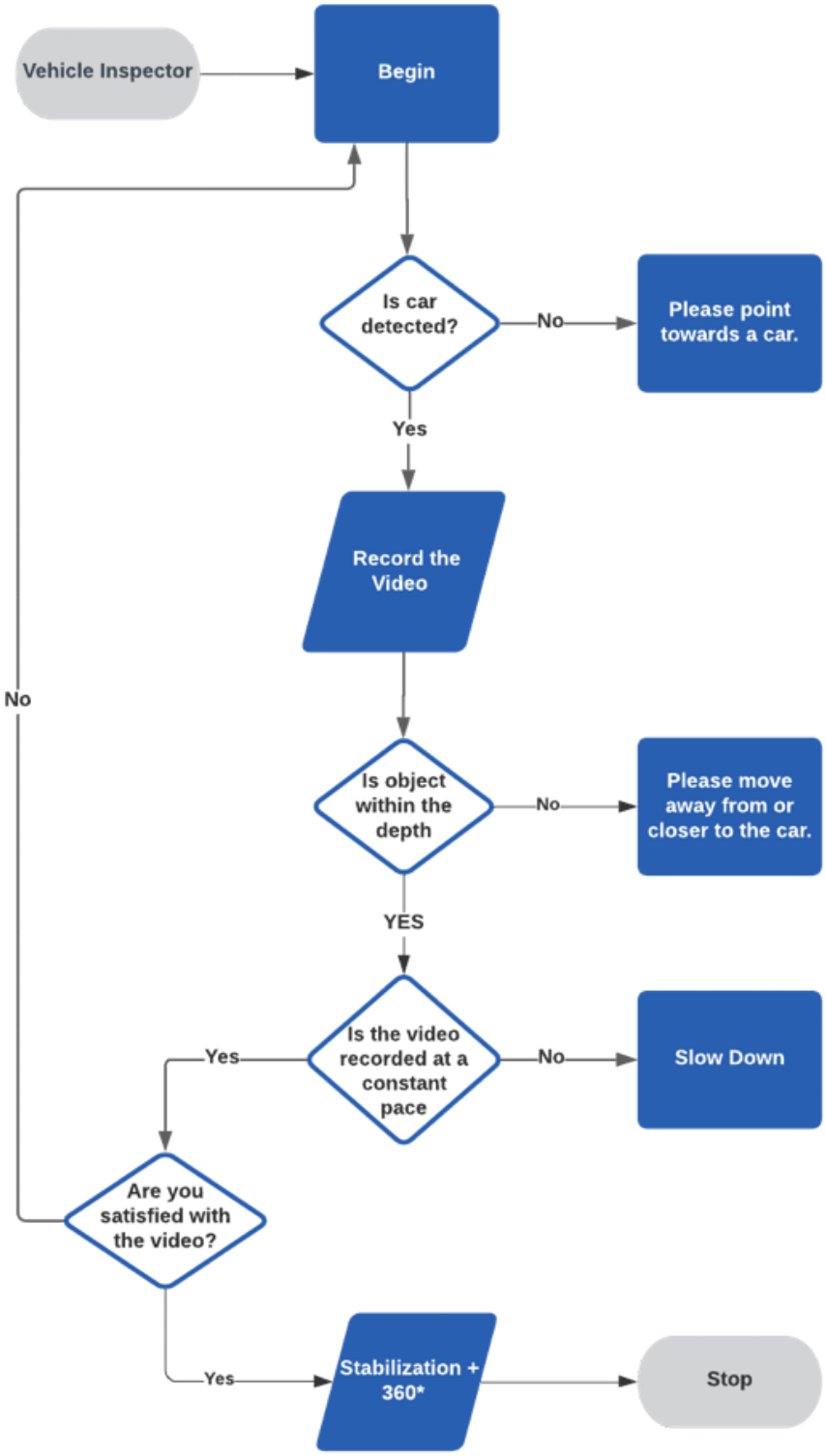

Figure 1: System flow diagram

2. App Functionalities (High level overview)

- Supports video capture on iOS.
- Detects if a vehicle is present in the captured frame using a Machine Learning model (YOLOTinyv3).
- Assists the vehicle inspector in capturing the best possible frames by providing bounding boxes around the car upon detection.
- Detects the correct side of the vehicle using a separate Deep Learning model (CustomResNet34).
- Captures distance between the user and the vehicle using the depth camera module.
- Monitors the pace of the user's movement while recording using CoreMotion.
- Provides the inspector a preview of the captured video, at which point the inspector can discard and retake it.
- Saves the video to the internal memory post the review.
- Stabilizes the captured video using standardized algorithms.
- Converts the stabilized video into 360° imagery.

3. Modules

Major portion of the code is concentrated in two files: VideoViewController.swift and CKFVideoSession.swift. The main components in these file are discussed in detail below:

2.1. Deep Learning

The role of deep learning models in this app is to
i) detect if the car is present in frame or not and
ii) detect the direction of the car at which the camera is pointed.
The first two decision boxes in Figure 1 corresponds to the deep learning module of our app. The model to detect the side of the car is designed by referring to ResNet34 custom architecture and is trained on the 100,000 images and validated on the 10,000 images provided by the ACV team. The training took several days and is done entirely on Google Colab GPU servers. The second one is a pre-trained `YOLOTiny` model from the apples' website. Both the models are initiated in the VideoViewController file and `CMSampleBuffer` from the cameras' output is passed to these initialized models for predictions and the function corresponding these predictions are also present in the same swift file. Then these predictions are used to decide the workflow as shown in the above flowchart.

2.1.1. Architecture (Custom ResNet34)

We trained this model to predict among the 8 different sides of a car: front, back, right side, left side, right-rear corner, left-rear corner, right-front corner and left-front corner. We have created a ResNet34 like network using Keras basic layers. Resnet architecture is built using `resnetBuilder()` which implements a single skip connection. We have used l2 norm penalty regularizer along with `relu` activation in the Conv2D layers. The model is compiled using the Adamax optimizer. Best weights obtained after training are saved and the model is then converted to .coreml using coremltools python package. The model achieves more than 96% accuracy on the provided data. Jupyter Notebook can be found at this link. Once the coreML model is integrated into the iOS app, it expects the input to be `VNClassificationObservation` which only returns the prediction values.

t2.1.2. YOLOTinyv3

Locate and classify 80 different types of objects present in a camera frame or image. This is the model that provides bounding boxes and predictions together. However, we have restricted this model on drawing the boxes around the car and no other object. The model can be found at the link. The model expects the input type as `VNRecognizedObjectObservation` unlike the above model. Also, both these models will be running two separate threads for optimization purposes.

2.1.3 CoreML

The `CoreML` package is used to integrate machine learning models into our app. Core ML provides a unified representation for all models. Our app uses Core ML APIs and camera buffer data to make predictions on the device. `Vision` package is used to apply computer vision algorithms to perform prediction tasks on input frames or video. One of the problems we faced with CoreML is that it has two different kinds of delegate outputs, one which can return the bounding box and the other which only returns the predictions. So, that is the reason we have to use two two separate machine learning models.

2.2 Depth Camera Module

This module is mainly used to detect the depth of the object (an automobile in our case), `AVCaptureDepthDataOutput` is used to get the depth data and is initiated in the CKFVideoSession file, and the delegate `depthvideoCapture` is called from VideoViewController file and the depth generated is passed to the `displayInstruction` function to process the depth data and display the instruction in the labels. As of now the depth data is calculated by taking the average of depth data from the pixels situated in

the centre of the frame. One more backdraw of using a depth camera is that the video that we get is zoomed by *2x*.

2.3 AVFoundation

An object used to write media data to a new file of a specified audiovisual container type. We used `AVAssetWriter` instead of the traditional `AVCaptureMovieFileOutput` cause AVAssetWritter gives us additional advantage of modifying/utilizing the raw data before we write the captured frame into a file. The `CMSampleBuffer` coming is passed to both the machine learning models for various predictions. In future, one addition is to try and check if any additional metadata can be added to the frame, in which we can encode the predictions and other sensor details to each frame metadata, so that they can be used during post processing. This entire portion is done in the CKFVideoSession where the `AVAssetWriter` is initiated and every frame is added to the recording until the recording is stopped. One of the problem we faced with this module is that the `CMSampleBuffer` from `AVCaptureVideoDataOutputSampleBufferDelegate` delegate callback in `captureOutput` is also passed to both the machine learning models for predictions and this sometimes used to cause lag while writing the buffer to `AVAssetWriter`, causing the recorded video to drop some frames. We solved this issue by using multiple threads for each of the machine learning models in an asynchronous way.

2.4 CoreMotion & CoreLocation

These modules are mainly used to calculate speed of the vehicle inspector. The speed is calculated using the user location movements. The thresholds for this module outputs are not optimized and need further study. One of the problems we found while directly computing the velocity is that there is a minor delay in updation of the value which makes the app the updates bit slow, one alternative is to use the `CoreMotionActivity` if the thresholds can be modified for events.

2.5 OpenCV - MeshFlow and 360° imagery.

We have tried various stabilization algorithms with OpenCV both in Objective-C and Python to software stabilize the video captured. Some of noticeable ones are MeshFlow, which is a spatial smooth sparse motion field with motion vectors only at the mesh vertices. The videos are smoothed adaptively by a novel smoothing technique, namely the Predicted Adaptive Path Smoothing (PAPS), which only uses motions from the past. Next one is the implementation of the Nghia Ho work Simple Video Stabilization Using OpenCV, in this we used the `replicate` setting as there is a chance of huge black portions coming up in the stabilized video if there are any sudden moments. After the stabilization, the video is passed to a module (Reel) which converts it into the 360° imagery that can viewed in the browser.

2.6 User Interface

The user interface for our app is very minimal and is divided into two different sections *i.)* user intractable and *ii.)* display labels. In the first we have the basic functionality to control the record button and preview control which comes up immediately after the video recording is stopped, the code corresponding to the preview can be found in the VideoViewController file in `VideoSettingsViewController` class. The instructions which are displaying on the screen are further subdivided into two *a.)* `displayInfo` *and b.)* `displayInstruction,` these two functions are used to display information and instructions respectively. The judgement on what instruction/info to be displayed is based on five signals we discussed above 1. isRecording, 2. isCarPresent from the YOLO model 3. predicted side of the car by the Renset34 model, 4. Depth value and 5. speed of the vehicle inspector recording the video. The signals 3 to 5 are not used until the recording is started.

4. Optimizations

This is one of the areas where we have to invest a lot of time in. The app after the integration was very laggy and prediction updates from the machine learning model used to take around 3 to 4 secs of delay, this was because of the `CMSampleBuffer` that was getting shared across models and is used to write in the `AVAssetWriter`. We were able to optimize this by using multiple threads to handle each of the callbacks asynchronously. Mostly we are using the default threads provided, but this can be achieved using custom threads as well. We are using four different threads in total, the main thread which is used for UI updates and `AVAssetWriter.` Then two separate threads are used for each of the machine learning models. The last one is used for the depth calculation. By using this threading approach we were able to reduce the CPU usage from 110% to 70% which is still high. But this high usage is expected as we are constantly predicting for each frame, we have also added a parameter which allows us control the number of times the models are called per second, we set this value to 10 for the moment. Also, by following this approach we were able to resolve the frame skip issues that we initially faced during writing them to the AVAssetWriter.

5. Challenges

Most of the challenges faced are already discussed above while detailing the modules and in optimization section, here are some of the other challenges that we faced:

- This application uses depth sensing techniques to understand how far the user is far from the vehicle. This would require a depth camera and only iPhone X or above to support depth module.

- Custom ResNet34 model gets high accuracy of over 96% but it still sometimes gets the predictions of symmetric parts like left-rear and right-rear etc. interchanged. To solve this problem, we tried using weighted average predictions but it sometimes leads to wrong information being displayed for a longer duration.

- The application takes three inputs (depth data, image detection buffer and the machine learning model data), combines them all to process the output simultaneously to show results in the screen is bit too quick. The application can handle heavy throughput and still deliver required results. However some performance issues are observed if the app is kept open for more than 8 mins.

- Stabilization of the video and the generation of 360° imagery is now done as a separate desktop project. This work should be on-boarded onto the iPhone to have an end to end connectivity.

6. Code References

In CameraKit:

CKFPreviewView.swift : This is a custom view for showing the frames/record the video. This file also is responsible for drawing bounding boxes.

CKFSession.swift : This file initiates the AVCaptureSession and returns the camera device input. It is also responsible for start and stop of the session

CKFVideoSession.swift : This file is responsible for capturing depth and the movie frame output buffer. It is then passed to the Controller using the delegate methods present in the VideoCaptureDelegate protocol.

CameraKit.h : In this header, you should import all the public headers of your framework using statements like #import <CameraKit/PublicHeader.h>

In CameraKitDemo:

VideoViewController.swift : This file is responsible for controlling the camera UI and the camera preview.
models - This folder contains the coreml model files that are used for predictions in the app.

7. Improvements (Alternatives)

Below are some of the challenges/implementations that we discussed earlier and potential workarounds/alternative technologies to look for future implementation:

- One addition is to try and check if any additional metadata can be added to the frame, in which we can encode the predictions and other sensor details to each frame metadata, so that they can be used during post processing. This metadata can be further utilized in future if there are any additional components that get built.

- One of the problems we faced with CoreML is that it has two different kinds of delegates, one which can return the bounding box and the other which only returns the predictions. So, that is the reason we have to use two two separate machine learning models. - A single model that does both prediction and bounding box can be trained but it requires some additional effort in data augmentation. Also, one further research can be done in the area of Attention models.

- The problem we found with the model, that it gets a little confused between the right and left side predictions as both the images look symmetrical. If combining the symmetrical classes into a general class like “front corner” instead of “left-front corner” and “right-front corner” is feasible, the information can be more relevant.

- As of now the depth data is calculated by taking the average of depth data from the pixels situated in the centre of the frame. - This can be further improved if we try to get the depth from the pixels inside the bounding boxes provided by the model in which scenario we know exactly that the average depth that is being displayed is from the automobile.

- The problem with directly computing the velocity where there is a minor delay in updation of the speed values makes the app the updates bit slow can be rethought with different sensors such as Pedometer or use of different inbuilt functionality such as CoreMotion Activity.

- We have tried almost five different software stabilization algorithms like Point Feature Matching, Advanced Eyes Articulate Mechanism, Mathematical transformations using OpenCV, videostab in OpenCV, Optical Flow using OpenCV but none of them are that efficient in completely stabilizing the video, one way to achieve the best stabilization is to use a hardware stabilization accessory like gimbal.

References

[1] CameraKit [github] https://github.com/CameraKit/camerakit-ios
[2] CoreML [Apple Documentation] https://developer.apple.com/documentation/coreml
[3] Vision [Apple Documentation] https://developer.apple.com/documentation/vision
[4] CoreMotion [Apple Documentation] https://developer.apple.com/documentation/coremotion
[5] MeshFlow [github] https://github.com/sudheerachary/Mesh-Flow-Video-Stabilization
[6] Video Stab [github] https://github.com/AdamSpannbauer/python_video_stab
[7] jQuery Reel [github] https://github.com/pisi/Reel
[8] AV Foundation [Apple Documentation] https://developer.apple.com/av-foundation/
[9] AVKit [Apple Documentation] https://developer.apple.com/documentation/avkit/
[10] Keras Documentation [Website] https://keras.io/
[11] OpenCV Documentation [Website] https://opencv.org/
[12] Using CoreML model [Website] https://heartbeat.fritz.ai/how-to-fine-tune-resnet-in-keras-and-use-it-in-an-ios-app-via-core-ml-ee7fd84c1b26
[13] AVCaptureVideoDataOutput + AVCaptureMovieFileOutput use is not supported. Using AVAssetWriter to write the recording output to the buffer. https://stackoverflow.com/questions/4944083/can-use-avcapturevideodataoutput-and-avcapturemoviefileoutput-at-the-same-time