Augmented reality filters: facial feature tracking and augmentation

Group 9
João Carlos Pereira Fernandes
jooc@student.chalmers.se

Abstract

In a generation taken over by technology, the virtual world is becoming more and more real. Advancements such as Augmented Reality (AR) and Virtual Reality (VR) have caused a worldwide impact, whether it is purely for entertainment, leisure or social media, but also in fields such as education and medicine. AR and VR technologies can be used to carefully plan surgeries of high risk or to engage students and create dynamical learning environments.

With this project, we aim to dive into recognized methods of face and facial feature detection, give the theoretical intuition behind them and put them to use to create basic AR filters. Every prominent algorithm used in the scope of this project is carefully explained throughout the report and obtained results are displayed afterwards.

1 Introduction

The task of identifying faces and facial features using computer vision is a fundamental piece for several technologies nowadays and with constantly evolving solutions.

The goal of this project can be divided into 3 parts: identification, tracking and augmentation. The principles used for the first and second parts are detailed throughout this paper, while the latter consists primarily of experiments with image manipulation and the 'statistics' obtained from the previous parts.

For training and testing purposes, it will be used the YouTube Faces data set (which can be extracted here). It contains over 3400 videos, already broken down in .jpg frames, of nearly 1600 people, giving plenty of test images to fine tune the model for this project.

Given the variety in face structure, orientation, even in background noise (sometimes multiple people can be seen in a video) this dataset is suitable to test all sort of different environments. Note that this is executed using unlabeled data, which means the precise locations of face and facial features are not given.

2 Theoretical methods

In this section, a brief overview of the algorithms used for the completion of this project and preceding theory is presented. Each subsection contains the explanation of the algorithm along with its utility for this paper.

2.1 Viola-Jones algorithm

Consider we have a grayscale image $f = \{f_{ij}\}_{i=1:n}^{j=1:m}$ where $0 \leq f_{ij} \leq 255$. A key element of the Viola-Jones algorithm is to calculate the **integral image** I_f, with the same size of f, where each pixel is calculated as

$$I_{ij} = \sum_{p=1}^{i} \sum_{q=1}^{j} f_{pq}$$

that is, the sum of every pixel above and to the left. We then use this new image to compute **Haar features**. Haar features work similarly to convolutional filters to detect edges, corners, lines and other features but are simplified in the sense of merely calculating differences between sum of pixels in certain regions.

The calculation of a single Haar filter through the entire (original) image would require expensive computational work, which is cut down to constant time complexity with the integral image: the calculation of each convolution is found by only calculating sum and differences of key pixels in this new image.

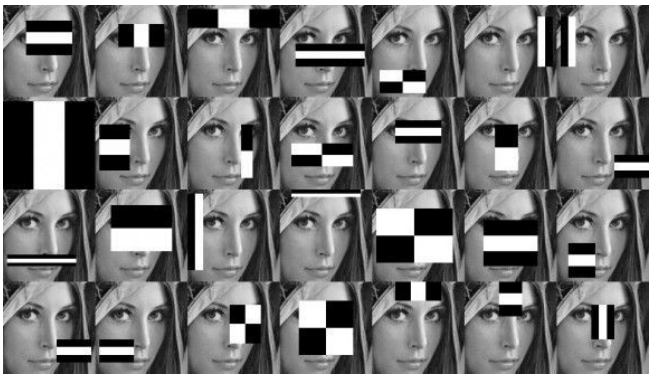


FIGURE 1. Example of Haar features calculation

The object detection procedure can compute an immensely large number of features before determining if a face exists or not. Therefore, the Viola-Jones uses an **AdaBoost** reweighting step in order to sample a smaller number of features. The *MATLAB* implementation of this algorithm contains already pretrained classifiers for specific tasks, but an intuitive approach to its functioning is presented below.

Assume we have a set of n labeled images $(\mathbf{I}_1, \mathbf{y}_1), ..., (\mathbf{I}_n, \mathbf{y}_n)$ where $\mathbf{I}_k$ represents an image and $\mathbf{y}_k$ its label, $\mathbf{y}_k = 1$ if $\mathbf{I}_k$ contains a face (or whatever object the classifier is being trained for) and $\mathbf{y}_k = 0$ otherwise. We also have a set of m features and we want to select a smaller

subset (let's assume with cardinality b) of the best classifying features.

For $i = 1 : m$, we define h_i as a (weak) classifier using only the i^{th} feature. Each classifier is specified by a **threshold** θ_i, a **polarity** p_i ($\in \pm 1$, used to invert weak classifiers) and a **weight** α_i. The algorithm is as follows:

1. Initialize all weights to $\alpha_i = \frac{1}{n}$
2. Repeat b times:

   Normalize all weights s.t. $\sum_i \alpha_i = 1$

   For each feature f_j, train a (weak) classifier h_j in a way that $h_j(\mathbf{I}_i) = 1$ if $p_j f_j > p_j \theta_j$ and 0 otherwise.

   Evaluate the error $\varepsilon_j = \sum_i \alpha_i (h_j(\mathbf{I}_i) - \mathbf{y}_i)$ and choose the feature j with lowest error.

   Update the weights $\alpha_i \longleftarrow \alpha_i \, \beta_{j^*}^{1-e_i}$, where j^* is the index of the minimal error feature, $e_i = 0$ if $h_{j^*}(\mathbf{I}_i) = \mathbf{y}_i$ and 1 otherwise and $\beta_{j^*} = \frac{\varepsilon_{j^*}}{1-\varepsilon_{j^*}}$
3. Assemble the final classifier as $h(\mathbf{I}_i) = \text{sgn}\left(\sum_{j=1}^{b} \alpha_i \, h_j(\mathbf{I}_i)\right)$

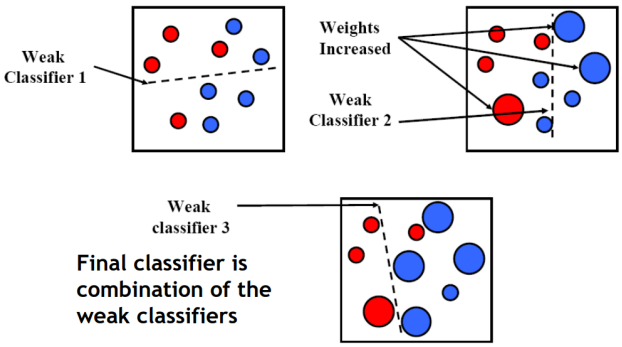


FIGURE 2. Intuitive diagram for the combination of weak classifiers

In order to avoid false positive detections, the Viola-Jones algorithm is constructed in a cascade architecture. This means that the weak classifiers are chained with progressively increasing complexity and if any weak classifier indicates a negative result, the evaluation of the global classifiers halts and defaults to a negative result. Intuitively, this helps the algorithm speed up evaluations while also avoiding passing potentially false positive results.

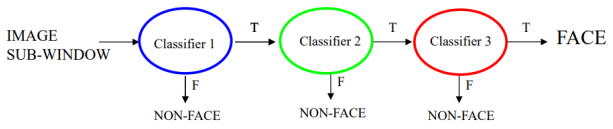


FIGURE 3. Intuitive diagram for the cascade of classifiers

The Viola-Jones algorithm is used in this project to, firstly, identify a face within a given frame and, in case of retrieving a face, identifying other facial features such as eyes, nose and mouth.

This algorithm is directly implemented in *MATLAB* in the Computer Vision toolbox, but we need to go deeper than the actual program and calculate precise statistics about the features found: area, inclination, distance from one another, in order to most accurately apply the AR filters we want. To do so, we are going to make use of different features that can be computed in the image, namely, **SURF** features, **BRISK** features, **MinEigen** features and **MSE regions**. A brief explanation of each follows over the next subsections.

2.2 SURF features

SURF stand for *Speeded-Up Robust Features*. It is a features extraction algorithm that follows the previous work of **SIFT** (Scale-Invariant Feature Transform) descriptor. Like the Viola-Jones algorithm, the identification of SURF features on a figure f relies on its integral image which we denote by I_f.

The location and scale of features is discovered using the determinant of a Hessian matrix obtained by derivating the convolution of the integral image I_f with a Gaussian filter. Let $\mathbf{x} = (x, y)^{\text{T}}$ be a coordinate from the image. We denote by $L(\mathbf{x}, \sigma)$ the convolution of I_f with a Gaussian filter with parameter σ at the coordinate $\mathbf{x}$. We can then calculate a Hessian matrix:

$$H(\mathbf{x}, \sigma) = \begin{bmatrix} \frac{\partial^2}{\partial x^2} L(\mathbf{x}, \sigma) & \frac{\partial^2}{\partial y \partial x} L(\mathbf{x}, \sigma) \\ \frac{\partial^2}{\partial x \partial y} L(\mathbf{x}, \sigma) & \frac{\partial^2}{\partial y^2} L(\mathbf{x}, \sigma) \end{bmatrix}$$

The determinant of the Hessian matrix is used as a measure of local change around the point. Thanks to the integral image, the box filters convolution can be calculated in constant time. The flagged points as SURF are the ones that attain maximal determinant values for H.

2.3 BRISK features

BRISK stands for *Binary Robust Invariant Scalable Keypoints*. It is a feature point detection algorithm whose main advantages is scale and rotation invariance. It is widely used to identify matching features in two different pictures where one of them may be transformed (translated, rotated, rescaled, or from different perspectives).

The BRISK features come from extensive research and only a succint presentation of how to obtain them is included in this paper, but it can be seen in full detail in [1].

The image is transformed into a 'pyramid' where each layer incresases the scale of the picture. These layers are denominated **octaves**. Potential interest points are analyzed in the original image as well as the copies with higher and lower scale. The image saliency is measured at each scale dimension, and afterwards a one-dimensional interpolation of the obtained values indicates the 'maximal saliency' point. The image below depicts the intuition behind the algorithm.

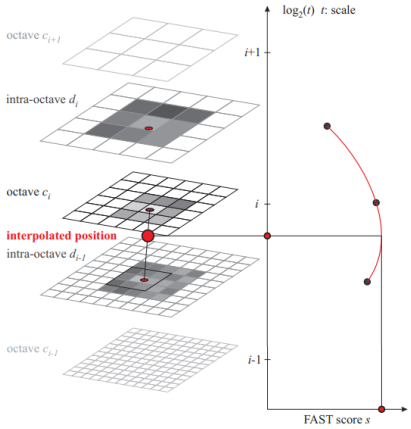


FIGURE 4. BRISK algorithm

2.4 MinEigen features

The **Minimum Eigenvalue** algorithm, developed by Shi and Tomasi, follows the preceding work by Harris in corner detection. Given a subframe I and a coordinate point $\mathbf{x} = (x, y)^T$, consider the following matrix:

$$M = \begin{bmatrix} \frac{\partial^2}{\partial x^2} I_{\mathbf{x}} & \frac{\partial^2}{\partial y \partial x} I_{\mathbf{x}} \\ \frac{\partial^2}{\partial x \partial y} I_{\mathbf{x}} & \frac{\partial^2}{\partial y^2} I_{\mathbf{x}} \end{bmatrix}$$

Harris idealized that a corner point can be detected in I by assigning a score (denoted by R) to this matrix and flagging a subframe as a corner if this score goes above a certain threshold. The score is given by:

$$R = \det M - k \operatorname{tr} M$$

where k is a parameter of the Harris algorithm. Shi and Tomasi developed a simpler yet more effective score function, given by:

$$R = \min\{\lambda_1, \lambda_2\}$$

where λ_1, λ_2 are the eigenvalues of the matrix M. From an algebraic standpoint, this simply means that if the variation along the principal components of the subframe is higher than a certain parameter, it is flagged as a corner.

2.5 MSE regions

MSER stands for *Maximally Stable Extremal Regions*. An **extremal region** is defined as a connected subset of pixels $Q \subseteq f$ such that for every pixel $q \in Q$ and every pixel $r \in \partial Q$ (frontier of Q) we have $f_q > f_r$ or $f_q < f_r$. Let Q_i denote an extremal region where every pixel $q \in Q$ is such that $f_q < i$. Q_i is a **MSER** if the function denoted by

$$g_\varepsilon(x) = \frac{|Q_{x+\varepsilon} \backslash Q_{x-\varepsilon}|}{|Q_x|}$$

where $\varepsilon > 0$ (it's a parameter defined in the algorithm; each value can yield different results) and $|Q|$ denotes the cardinality of set Q, achieves a local minimum at $x = i$.

MSE regions can be used to help calculate statistics about the extracted features. As an intuitive example, the pixels in a person's face will have an approximately equal intensity as opposed to their background, hair and other facial features, so it allows for an estimation of the area of the face.

Now that we have explained how the identification of facial features work, we must know how to track them over moving frames. The following algorithm is the main foundation for this process.

2.6 KLT algorithm

The **Kanade-Lucas-Tomasi algorithm** is one of the most widely used feature tracking mechanism. It works as an optimization problem, where the algorithm must find a suitable 'displacement' between two images that minimizes an error function.

Suppose we have two consecutive frames, f and g (with same dimensions), and a coordinate point $\mathbf{x} = (x, y)^{\mathrm{T}}$. The goal is to find a displacement vector $\mathbf{h} \in \mathcal{R}^2$ that minimizes the error function given by:

$$E(\mathbf{h}) = \sum_{\mathbf{x}} (f_{\mathbf{x}+\mathbf{h}} - g_{\mathbf{x}})^2$$

By using a Taylor expansion series of $f_{\mathbf{x}+\mathbf{h}}$

$$f_{\mathbf{x}+\mathbf{h}} \approx f_{\mathbf{x}} + \mathbf{h} \left(\frac{\partial}{\partial \mathbf{x}} f_{\mathbf{x}} \right)^T$$

and derivating the error function with respect to $\mathbf{h}$, we obtain:

$$\mathbf{h} \approx \left(\sum_{\mathbf{x}} (g_{\mathbf{x}} - f_{\mathbf{x}}) \frac{\partial}{\partial \mathbf{x}} f_{\mathbf{x}} \right) \left(\sum_{\mathbf{x}} \frac{\partial}{\partial \mathbf{x}} f_{\mathbf{x}} \left(\frac{\partial}{\partial \mathbf{x}} f_{\mathbf{x}} \right)^T \right)^{-1}$$

This procedure is executed for every trackable feature within a neighbourhood of itself in order to determine to most plausible position of the feature in the next frame. The algorithm fails to track a feature if the "displacement" goes above a certain threshold (parameter).

3 Procedure

3.1 Detecting face

Firstly, we use the Viola-Jones algorithm to detect a face and immediately plot SURF, BRISK and MinEigen features.

FIGURE 5. Face detection and feature extraction

Then, using the face as region of interest, repeat the Viola-Jones algorithm to detect eyes, mouth and nose. The features obtained in the first are then separated by each of these facial features.

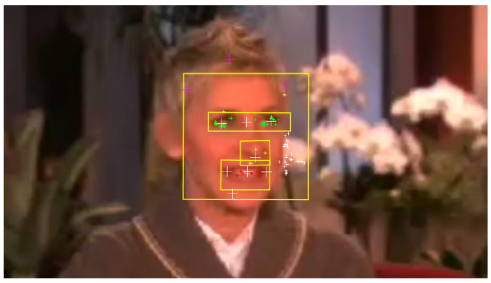


FIGURE 6. Facial feature detection and feature separation

The way the code was implemented for this report allows for the computer to not detect the specific facial features, as long as it detects a face. An example can be seen for the image below. It detects face, but no other facial feature. In this scenario, the computer will keep searching for any 'missing' facial features until it is able to track it.

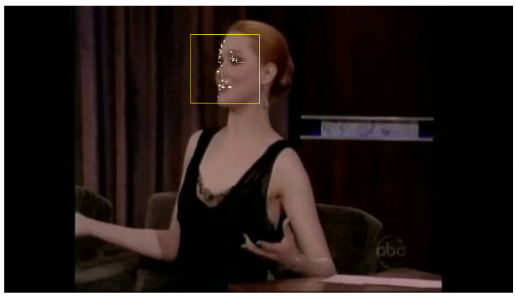


FIGURE 7. Example of frame with facial features not found

3.2 Tracking feature points

Next, the KLT algorithm allows to create tracking objects in charge of not only tracking the specific feature points, but as well as estimating the best transformation between consecutive frames in order to maintain the most accurate area of each facial feature.

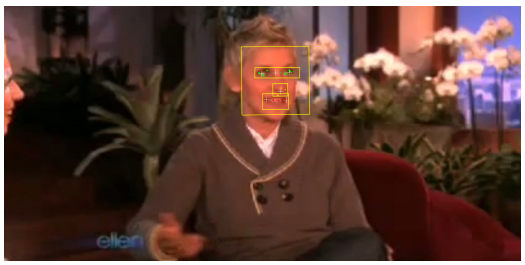


FIGURE 8. First frame of video with tracked objects

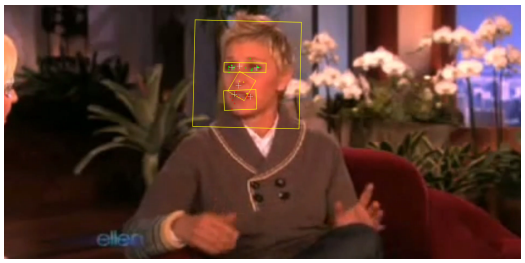


FIGURE 9. Final frame of video with tracked objects

The program used for this report includes one tracking object for the face as a whole (using all the feature points obtained in the first step), a tracking object for each facial feature identified (eyes, mouth, nose), each using the respective features that were separated in the second step, and an additional tracking object for specific keypoints obtained from all the tracked features. These keypoints are the main foundation for the construction of the filters in the last step.

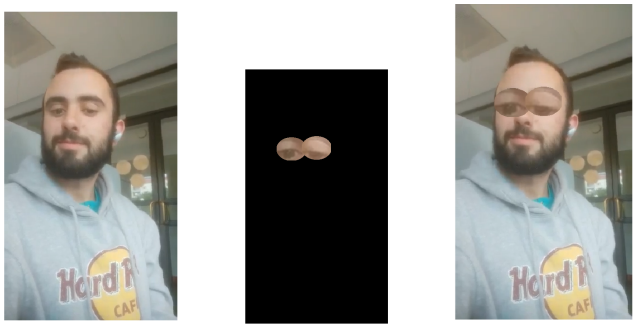
The program is also prevented against tracking failures. Often times, the algorithm fails to find a proper transformation between the feature points of consecutive frames. When this happens, the program 'deletes' the presence of that facial feature and as described above, keeps searching for it in the frames that follow. An example of tracking failure can be seen below.

aBefore losing track bAfter redetection

FIGURE 10. Set of frames separating a tracking failure

3.3 Writing a filter

After tracking the features throughout the video, we obtain an array of structures that contain the needed information to 'compute' these filters. Simply put, these filters are created by creating a dummy image, that can contain deformations of the original picture or even other pictures at the specified locations, depending on the type of filter we're trying to create. This dummy image then gets pasted on top of the original image to create the filter.



aOriginal bDummy cFiltered

FIGURE 11. Example of filter creation

These filters get created frame by frame, using a structure containing information about key points in each frame. From then, we calculate useful measures using some 'generalizations' to make computation run faster. For example, we can calculate the distance between the eyes by clustering the eye features, computing their means to obtain the center, and using that distance to get an approximation of the width of the face. Similarly, we can compute the angle at which the eyes lie in order to calculate the tilt. An example of the structure used is presented below.

```
struct with fields:

    cent_eye1: [164.0043 220.0105]
    cent_eye2: [228.4530 220.9165]
  mouth_edge1: [164.2578 289.0217]
  mouth_edge2: [224.7011 320.0501]
   mouth_cent: [193.9925 304.1543]
    nose_cent: [199.2676 271.7623]
      nose_up: [196.2590 220.5193]
         chin: [187.4699 352.8962]
```

FIGURE 12. Structure used to compute measures

The code also prevents crashes in the case of unidentified facial features. In the event of not having all the needed feature locations and measures, the algorithm will simply return the original frame without any change. Since every frame contains its own structure with measures, it is possible to have the filter fall in and out of the picture during the final video.

4 Results/experimentations

Because this project was done using unlabeled data, it is hard to obtain accuracy measures since it would imply looking at each case individually. However, one way to evaluate performance is to track the number of tracked features throughout the video. Considering the features found during the first step and using the partition of features done in the second step, we can graph the number of tracked features in function of the frame number.

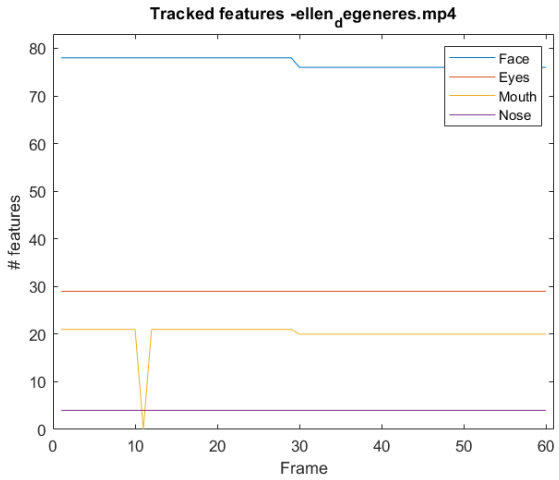


FIGURE 13. Number of tracked features

In this example, we notice that the tracking objects work fairly well, mantaining an almost constant number of tracked features, meaning almost all of them got properly tracked. The spike noticed in the 'Mouth' graph around the 10^{th} frame, shows that the algorithm could not compute a transformation between the previous frame and therefore 'deleted' all the mouth features and redetected them in the following frame.

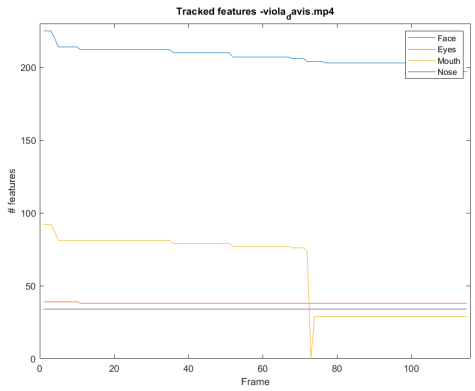


FIGURE 14. Number of tracked features

In the above example, we have reason to believe the detection did not go as planned. It's noticeable that there is a spike in the number of 'Mouth' features, similar to the last example. However, after redetection, the number of features decreases significantly, which is not to expect from nearly consecutive frames in a continuous video. In fact, the graph shown is the one obtained from the set of images in 10. Upon a closer look, after the tracking failure the mouth gets redetected in the spot as the nose, since the actual position of the mouth is too covered up to get a proper detection.

5 Conclusion and future work

The ideas used in this project extend to the widest range of applications: from facial reconstruction, security, or simple interactive and entertainment purposes as depicted here.

It is also worth mentioning that the pretrained algorithms implemented in *MATLAB* are not perfect, sometimes displaying inaccurate detections (not detecting certain features that exist or detecting them in the wrong place - a recurrent misclassification was detecting a mouth in the eye socket) but these can further improved upon by feeding labeled pictures into the model. The Computer Vision toolbox contains a program that allows the user to create their own 'Viola-Jones'-based classifier with labeled images.

With this project, we delved into prominent object detection and tracking algorithm, putting them to use to create several simple AR filters. A lot of the work was done based of locations of key points identified in the face but it is possible to create much more complex feats with these mechanisms. Further advancement on this project would include extending the filter application using more and more accurate key points and allowing live camera feed.

---

1. R. S. Stefan Leutenegger, Margarita Chli, BRISK: Binary Robust Invariant Scalable Keypoints (2011)
2. D. Tyagi, Introduction to SURF (Speeded-Up Robust Features) (2019)
3. L. Shapiro, Recognition Part II: Face Detection via AdaBoost (2016)
4. A. Ward, Facial Detection — Understanding Viola Jones' Algorithm (2020)
5. D. L. Joss Fong, How Snapchat's filters work (2016)
6. K. H. M. Therasa, S.M. Poonkuzhali and S. Raja, Face Recognition using Unsupervised Feature Learning (UFL) Approach (2016)
7. P. Juma, Face Detection using Viola-Jones Algorithm in Matlab (2021)
8. a. A. MathWorks, Face Detection and Tracking Using the KLT Algorithm (2022)