# Developing a Natural User Interface and Facial Recognition System With OpenCV and the Microsoft Kinect

Michael Gutensohn
Kennedy Space Center
Major: Computer Science
NIFS  Spring Session
Date:  13 04 2018

# Developing a Natural User Interface and Facial Recognition System With OpenCV and the Microsoft Kinect

Michael Gutensohn[1]

Rollins College, Winter Park, FL, 32789, USA

## Nomenclature

| | | |
|---|---|---|
| *AR* | = | Augmented Reality |
| *VR* | = | Virtual Reality |
| *NUI* | = | Natural User Interface |
| *Unity* | = | A C# based gaming and 3D graphics engine |
| *SDK* | = | Software Developer Kit |
| *Unity Package* | = | A collection of assets, scripts, and other Unity resources |
| *Facial Detection* | = | The process of determining if a face lies within an image |
| *Facial Recognition* | = | The process of determining the identity of a face from memory |
| *RGB-D* | = | Red, Green, Blue - Depth |
| *ROI* | = | Region of Interest in reference to an image |
| *FOV* | = | Field of View |
| *Wrapper* | = | software written to integrate otherwise incompatible software |

## I. Introduction

The task for this project was to design, develop, test, and deploy a facial recognition system for the Kennedy Space Center Augmented/Virtual Reality Lab. This system will serve as a means of user authentication as part of the NUI of the lab. The overarching goal is to create a seamless user interface that will allow the user to initiate and interact with AR and VR experiences without ever needing to use a mouse or keyboard at any step in the process.

## II. Requirements

a. Technical Requirements
   i. The system shall be compatible with the Microsoft Kinect or any other commercially available RGB-D camera of similar capability.
   ii. The system shall be compatible with Visual Studios C# .NET framework.
   iii. The system shall be compatible with the Unity Gaming Engine.
b. Budgetary Requirements
   i. Any resources used shall be open source.

## III. Previous Research

During the fall of 2017, a previous intern conducted the initial research for this project, examining different facial recognition software solutions. Unfortunately, none of the software identified in the intern's research met budgetary requirements. As a result, none could be used

[1] NIFS Intern, IT-C1, Kennedy Space Center, Rollins College

despite meeting technical requirements. It became clear that the only path forward would be to take a freely available set of software tools and adapt them to the needs of the project.

# IV. Approach

## A. Design

### I. Facial Detection

The first problem of facial recognition is facial detection. In simple terms, the conventional solution to facial detection is to use a standard digital camera and search the incoming frames for the features unique to the average human face. While this is an excellent use of existing technology, it is easy to spoof (exploit aspects of the software) by simply holding up a picture of human face, and thus not very secure.

In recent years, RGB-D camera sensors, such as the Microsoft Kinect, have become readily available to consumers. These cameras can function like normal cameras, but have added depth sensing technology built in. With this new found ability, facial detection becomes even easier to tackle, and significantly more difficult to spoof. The Microsoft Kinect first detects the body then the face of a person. Using its depth sensing ability, the Kinect not only searches for the two dimensional features of a face but also the unique terrain provided by the third dimension. This allows the Kinect to reliably distinguish a human face.

For this project, facial detection will be performed by the Kinect, and then the image data collected using the Kinect's built in camera will then be fed into our facial recognition software. This mitigates development requirements and allows us to focus on creating a fast and accurate facial recognition system.

### II. Facial Recognition

There are a number of facial recognition solutions available. As stated previously, many of these options are unavailable due to budgetary constraints. The most well documented, versatile, and freely available option is OpenCV[2], an open source library of computer vision functions.

OpenCV was initially developed by Intel. It is presently maintained by a community of developers. While created as a general purpose computer vision library for tasks such as image manipulation, object detection, and object recognition, OpenCV has a class of functions devoted specifically to facial recognition called FaceRecognizer[3].

The FaceRecognizer class is capable of implementing three different facial recognition algorithms, Eigen, Fisher, and Local Binary Patterns Histogram (LBPH), each of which have their advantages and disadvantages. The mathematics behind these algorithms are beyond my own understanding, but of the three algorithms, the LBPH algorithm is the most accurate[4]. The one disadvantage to LBPH is that it takes noticeably longer to train the classification model, but it is the only one that includes an update function for adding new faces, removing the need to create a new classification model every time a person is added to the system.

[2] "OpenCV Library," Open Source Computer Vision Library: https://opencv.org/

[3] Face Recognition with OpenCV: https://docs.opencv.org/3.0-beta/modules/face/doc/facerec/index.html

[4] "A Comparison of Facial Recognition Algorithms," Nicolas Delbiaggio: https://www.theseus.fi/bitstream/handle/10024/132808/Delbiaggio_Nicolas.pdf?sequence=1&isAllowed=y

# III. Implementation

The one drawback of OpenCV is that it is written in C++ and is not directly compatible with C#. The only way to implement OpenCV in C# is by using a library of wrapper functions. While OpenCV wrappers are freely available for C# .NET compatibility, any wrapper that provides both support for Unity as well as C# .NET exceeds budgetary constraints. Additionally, wrapping a function produces performance overheads in terms of processing due to the transition from managed (C#) to unmanaged (C++) code. These overheads are negligible for simple projects, but increases with each function call.

However, it is possible to mitigate these overheads by limiting the number of transitions made between C# and C++. This can be accomplished in this project by passing image data of the detected face from C# to C++, performing facial recognition using OpenCV in C++, then returning the name and identification number to C# if the face is recognized. Rather than making upwards of 50 transitions for every frame of the Kinect camera feed, the system makes 2-4 transitions depending on implementation.

## B. Development

The development process consisted of creating test beds in both Unity and C# .NET using the Kinect Unity package and the Kinect C# SDK[5]. Both test beds perform roughly the same tasks. They observe the environment using the Kinect. When a person walks within the FOV, the Kinect begins tracking their body and face. The testbeds collect the image data along with the ROI, a bounding box that within contains only the person's face. This information is then passed on to the facial recognition system.

The facial recognition system is a small library of functions. This library is referred to as AVRVision. This library of functions is written in C++ and compiled to a DLL, which is then wrapped in C#. The following is a summary of the most relevant functions of AVRVision:

### I. CheckFace()

The first function developed for the facial recognition system is CheckFace(). CheckFace() takes the raw image data and parameters of the frame provided by the Kinect's video feed, along with the tracking ID automatically assigned to the face, and the ROI containing the face. Once received, CheckFace() first processes the frame by converting it to a grayscale histogram, and then crops the image down to the face using the ROI provided by the Kinect. After processing the image, CheckFace() uses OpenCV's FaceRecognizer to check the image for a known face. If the face is known, CheckFace() will set the name, ID, and confidence level of its prediction. This data can be accessed through corresponding getter functions. If the face is not recognized or the face is obstructed, the name value is set to "Unrecognized." or "Face obstructed." and ID and confidence are set to -1.

When a face is unrecognized, the processed image of the face is saved to a directory named using the tracking ID provided by the Kinect in a temporary location. When enough images of an unrecognized face are stored in their folder,

[5] "Kinect for Windows" Microsoft Kinect SDK: https://developer.microsoft.com/en-us/windows/kinect

a final image is stored outside of the folder named with the face's tracking ID. This image will serve as a thumbnail when adding new users.

**II. CheckFace() version 2**

The original CheckFace() function, while functional, lacked accuracy due to the variance of angle and positioning of the face. If a user's face is not precisely upright, CheckFace() would be unable to recognize it despite already knowing it. To account for this issue it is necessary to normalize the position and angle of the face within the image.

Along with the bounding rectangle of the face, the Kinect also provides the coordinate positions of the eyes and nose within the image. CheckFace() version 2 takes these coordinates as integer arrays and uses them to adjust the angle and position of the face. Each eye acts as a point on a line, the angle of this line relative the rest of the image is taken, and the image is rotated so that the bottom and top of the image run parallel to the eye line. The nose is then used to center the face within the ROI created with the bounding box. With these changes, CheckFace() version 2 maintains accuracy despite the angle and position of the face within the original frame.

**III. Getter functions**

After the face has been analyzed, it is necessary to use GetName(), GetID(), and Confidence() to access the data set by CheckFace(). Each return the value their names imply.

**IV. AddFace()**[6]

When a new user is to be added to the system, the function AddFace() is called. AddFace() takes the tracking ID of the face to be added and the name of the user that the face belongs to. In the demo, the tracking ID is extracted from the file name of the thumbnail saved earlier by CheckFace() and the name is provided by the administrator. AddFace() moves the stored images of the new user's face to a permanent location, then updates FaceData.csv, a spreadsheet containing the path to each image and the ID that it corresponds with. FaceData is necessary when updating the facial recognition model as it allows the FaceRecognizer algorithm to map each image to a specific face using the given ID. AddFace() does not update the facial recognition model every time a new face is added. Rather, it moves the training images to a permanent location, updates the FaceData.csv file, and stores the names in a key-value map using the ID as the key and name as the value. This action allows the adding of multiple users at once.

**V. UpdateRecognizer()**

UpdateRecognizer() uses the data provided by AddFace() to update the facial recognition model. Of the three algorithms offered by FaceRecognizer in OpenCV, LBPH is the only one that supports and update function, allowing the addition of new faces to the model it generates without having to completely regenerate the model. Taking advantage of this fact saves time and processing.

[6] It is important to note that, due to the single threaded nature of Unity, the ability to add new users and update the facial recognition model is not supported in Unity.

# V. Results

The completed facial recognition system functions as expected. It accurately authenticates users and offers a simple interface for adding new users to its memory. I created both a Unity Package and .NET compatible library for use in future projects. It is important to note that the .NET library has more functionality than the Unity Package due to limitations caused by Unity's single threaded nature.

To demonstrate the effectiveness of the AVRVision library, I developed two simple applications called AVR Gateway and AVR Launch Services.

## A. AVR Gateway

VR Gateway uses the AVRVision library to authenticate and manage users, and once a user is authenticated, it displays a list of VR applications developed for the lab that the user has access to. There are additional tools to add apps as well as users. the data required to add a new user is collected during the authentication process, so all that is required of the admin is to select the face of the new user and provide a name. Any unselected faces will be deleted upon updating the recognizer.

In addition to the facial recognition capabilities provided by the AVRVision library, AVR Gateway implements a very basic form of speech recognition to mitigate the need for keyboard and mouse when accessing the lab.

## B. AVR Launch Service

The necessity of AVR Launch Services materialized as it was discovered that the power requirements of both a VR headset and the Kinect were too much for a single computer to handle simultaneously. The only way to use both at once is to have them each running on separate computers. AVR Launch Service runs in the background on the computer connected to the VR headset, listening for commands sent by AVR Gateway. When a user selects an app to launch, AVR Gateway sends the command to launch that specific app to AVR Launch Service, which then executes the command.

# VI. Acknowledgements

I'd like to acknowledge William Little for selecting me for this project, the Education Office for supporting the NIFS program, and USRA for sponsoring this internship.