Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) | Major Qualifying Projects

April 2013

Real Time Person Tracking and Identification using the Kinect sensor

Matthew Fitzpatrick
Worcester Polytechnic Institute

Nikolaos Matthiopoulos
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation

Fitzpatrick, M., & Matthiopoulos, N. (2013). *Real Time Person Tracking and Identification using the Kinect sensor*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/2642

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

WORCESTER POLYTECHNIC INSTITUTE

Real Time Person Tracking and Identification using the Kinect sensor

Major Qualifying Project in Electrical & Computer Engineering

Matthew Fitzpatrick, Nikolaos Matthiopoulos

4/25/2013

Contents

Contents 1
Table of Figures 2
Abstract 4
Intro to smart houses / Definition of problem 5
Derivation of system requirements 7
Evaluation of potential solutions 9
  Command interface 9
    Computer terminal 9
    Remote control / Smartphone app 9
    Voice commands 10
  User identification 10
    Password 10
    RFID chip 10
    Biometrics 11
    Facial recognition 12
  User tracking 12
    Video tracking 12
    Active/Passive Infrared Tracking 13
    Ultrasonic Sensors 14
    Thermographic Imaging 15
    Footstep tracking 16
  The Kinect & why it's awesome 16
System Design 19
  Broad system overview 19
  System modules 20
  Hardware overview – system architecture 21
    Central Processor 21
    Room Computers 21
    Sensors 22
  Software overview 23

`MainWindow` Class 24
`Location` Class 25
`Room` Class 25
`Person` Class 25
Facial Recognition 26
Tracking in detail 26
Facial Recognition in detail 28
System Testing & Analysis 31
Tracking Testing 31
Static Testing 31
Motion Testing 32
Facial Recognition Testing 33
Integrated Testing 34
Conclusion & Future Work 35
Bibliography 37
Appendix 39
MainWindow.xaml 39
MainWindow.xaml.cs 40
Utilities.cs 54
Location.cs 64
Person.cs 66
RecognizedFace.cs 70
References included in the Project 71
OpenCV libraries imported 72
EigenObjectRecognizer.cs (imported from EgmuCV Library) 73

Table of Figures

Figure 1: mControl v3 GUI view from computer terminal (TopTenReviews, TopTenReviews, 2013) 9
Figure 2: Graafstra's glass-encased RFID Chip implant process (Greenberg, 2011) 11
Figure 3: Example of facial Recognition (Johnson, 2012) 12

Figure 4: (a) Video input (n.b. color image, shown here in grayscale). (b) Segmentation. (c) A 2D representation of the blob statistics. (Wren, Azarbayejani, Darrell, & Pentland, 1997) 13
Figure 5: Ultrasonic sensor area of coverage 15
Figure 6: Connection of the hardware used for the footstep capture system. (Rodríguez, Lewis, Mason, & Evans, 2008) 16
Figure 7: Effective horizontal view range of the Kinect sensor according to Microsoft ™ 18
Figure 8: System Block Diagram 20
Figure 9: Hardware Overview Diagram 21
Figure 10: UML Graph of Software 23
Figure 11: Track Association 28
Figure 12: Some eigenfaces from AT&T Laboratories Cambridge 29
Figure 13: 2-meter Static Test Figure 14: 4-meter Static Test 31
Figure 15: 2-meter Walking Test Figure 16: 4-meter Walking Test 32
Figure 17: Diagonal Test Figure 18: Approach Test 33

Abstract

The objective of this project was to design and implement an automated tracking and user identification system for use in a smart home environment. Most existing smart home systems require users to either carry some sort of object that the house can identify or provide some sort of identification when they issue a command. Our project seeks to eliminate these inconveniences and enable users to issue commands to their smart home through simple, nonintrusive voice prompts. Microsoft's Kinect sensor unit was chosen as our design platform for its functionality and ease of use.

Intro to smart houses / Definition of problem

As computers become smaller and cheaper, they are being integrated into a broader range of devices, especially home appliances. This ubiquity has led to a trend towards integration. So-called “smart houses” allow residents to control various aspects of the home environment through a central computer system. Users may be able to control temperature, music selection, and other functions through a wall-mounted console or even a smartphone or tablet application. The problem with these implementations is that, although they are more convenient and centralized than traditional home systems, they still require users to interact with a physical device and provide some sort of authentication, whether that is a password, some sort of biometric reading, or comes from a carried device, such as an RFID chip. We propose that the user interface of such a system can be simplified to an intuitive voice interface, where users need only issue spoken commands preceded by a keyword.

Home automation controllers are usually judged upon the features they offer, the compatibility with other systems for control and the help/support for the product. The communities of customers in websites such as amazon.com seem to be very satisfied with a few of the systems out there but bring up as a main concern the user interface offered by the system. Most potential buyers of systems as such are looking for a system that has the control features they need and is easily accessible. Easy access tends to be through any possible device that a user might have in the house with internet access, such as a smartphone, a tablet pc, a laptop, or a desktop. Some of the available products support the addition of microphones for voice commands which is a popular feature, but some users raise concerns for its security. Home automation controllers and software range from $50 to $220 dollars for a one fits all purpose system (TopTenReviews, TopTenREVIEWS We Do the Research So You Don’t Have To., 2013).

Home automation software runs on a standalone device or on a computer in the house. The system provides a wireless network or connects to an existing network in the household. It is able to communicate with a variety of different appliances that are also connect to the same network. Each homeowner gets to decide how many things and actions this system controls in the household by purchasing and installing the appropriate compatible devices that can communicate with the system. These devices can be as intricate as a central security system of video surveillance or outlets with a wireless controller. Therefore, the cost of installing the system varies vastly by the choice of features that the homeowner implements. Homeowners that install home automation system are willing to spend a few extra dollars to replace existing infrastructure or add new permanent fixtures as long as it doesn’t alter the space significantly. It is also a key component that an average homeowner can install the

system on his own with the use of an instruction manual. If that is not the case, then efficient technical support is the first thing that customers of this market demand.

The system this project attempts to implement does not control home automation. What it attempts to develop is a set of sensors that will need to be installed in every room, which will provide a secure line of communication between the user and the home automation system. Home automation is managed through a separate automation controller system. What has not been attempted yet, to the extent of our knowledge, is a command interface that will remove the necessity of a device, such as a smartphone, tablet or computer terminal, and will still be able to provide the same security. The output of the system will be a command that will be handed over to the home automation system and that will in turn execute the command. Currently, the only alternative to a device terminal for a controller interface is simple speech recognition though a microphone. This provides security risks because voice recognition does not provide unique user identification, so anyone in near proximity to the microphone can gain access to the system.

Additionally, the home automation systems in commercial availability do not support actions based on where the residents of the house are at the time. Adding tracking of the users into the system can provide easier command association to the users for permissions and allows the industry of home automation system to expand towards an unexplored aspect of household automation action.

Derivation of system requirements

In order to assess if the system created in this project is a successful solution, it needs to fulfill specific criteria. The nature of the problem and the environment in which it is implemented set some baseline specifications that any system of this sort needs to meet.

First, we need to know how many people our system needs to be able to handle at one time. According to the 2010 census, the average American household size was 2.58 persons per household (Lofquist, Lugaila, O'Connell, & Feliz, 2012). That means that our system needs to be able to track and identify at the very least 3 people at the same time. Tracking one person moving through a space is easy, but it becomes more difficult when multiple people interact in a room. When this happens, the system needs to take into account the close proximity the occupants might have and any blind spots created when users walk in front of each other. A system acceptable for the purposes of this project should be unaffected by the blind spots and close interactions, or it should be able to recover quickly enough for these errors not to be noticed by the users. Along with these requirements comes the drawback that the system should be able to uniquely identify each one of those users. This means that the system needs to be powerful enough to perform many calculation per second and the more people the system is able to track and identify at the same time, the more powerful the system needs to be. One way to handle the amount of processing needed is to distribute it across multiple computers. For example, installing a computer in each room to handle data pertinent to that room, and installing another computer to act as a central processor for information applicable to multiple rooms means that each computer can be significantly less powerful, and therefore cheaper.

Secondly, our system must be capable of handling a wide variety of home layouts and room geometries, as no two houses are alike. We felt that approaching the project with a room-based model made the most sense for this, especially from a software standpoint. This allows the system to be customized for each room, while keeping details pertaining to idiosyncrasies in a specific room hidden from the central system by one or more layers of abstraction.

To ensure that our market is as broad as possible, our system must be easily installable in both new and existing houses. Aesthetics comes into play here as well, as potential customers may be turned off of a system that results in unsightly wires running throughout their house. To accomplish both of these goals, we want to maximize the area that can be covered by one sensor unit.

One of our goals with this system is to have a database of registered users and be able to recognize them in the most unobtrusive way possible. This means the

system needs a way to not only identify registered users, but needs somewhere to store the identifying information for each user.

Our system must be able to accurately track users through a variety of social situations, including those that involve close interaction. For example, if two people hug, the system must either continue to differentiate them through the hug or recognize that they have come too close together and attempt to re-identify them. Additionally, the system should be able to distinguish between two people engaged in normal conversation, and not interpret this as coming too close. We measured the distance between a number of New England college students engaged in normal conversation and found the minimum distance between them was approximately 0.6m. In order to account for this distance plus an additional buffer, our system must be able to distinguish people standing no less than 0.5 m apart.

Though our system does need to be able to identify its users, user tracking means this identification need not happen continuously. However, our system should have an identity for a user when that user gives a command. Therefore identification should occur as soon as possible after the user is first tracked. Additionally, the user should be re-identified if he or she interacts with another user in such a way that they might become confused. Therefore, while the system should be able to maintain an identity at times when checking it is impossible, it should be able check users' identities frequently enough that mix-ups are resolved as quickly as possible.

Evaluation of potential solutions

Command interface

Computer terminal

Perhaps the easiest way to implement a command interface for a smart home is to install a computer terminal in every room. From a coding standpoint, all it requires beyond command execution is a GUI. People looking into smart home systems are also likely to be frequent computer users, meaning that a terminal interface should have a very low learning curve, making it easy to use. Convenience, however, is the main drawback of this interface. Though it is more convenient for users to be able to adjust things like a thermostat from any room, it is still preferable for users to do things like listen to music using a portable music player than control music from a static terminal, which severely affects the benefit of having a smart home.

Figure 1: mControl v3 GUI view from computer terminal (TopTenReviews, TopTenReviews, 2013)

Remote control / Smartphone app

One way to take the terminal concept and make it more convenient is to use a remote control instead. Though this is more complicated to code than a dumb terminal as it needs to communicate with the house somehow, it is vastly more convenient to use, though this is somewhat counteracted by the inconvenience of having to carry the remote control and the possibility of losing it. An even more convenient solution is to allow control of the house through a smartphone app, if the residents happen to own

smartphones. Users are likely to carry their smartphones with them almost all the time, so they will not need to carry anything more than they usually do. These systems do raise security concerns, however, so designers will need to ensure that whatever sort of wireless communication they use prevents anyone but the intended users from operating the system or accessing any information about the house or occupants. These systems, then, are more difficult to implement than they first appear, as they require a solid background in network security.

Voice commands

The previous solutions are inconvenient because they impose restrictions on their use. A terminal requires the user to go to a specific place, while a remote control requires the user carry a specific object. Therefore, it is clear that the ideal solution will impose as few restrictions as possible on the user. The user should be able to issue commands to the system from anywhere in the house without having to carry anything. This is possible by implementing a voice control interface. Users need only speak a command, generally preceded by a keyword to ensure the system only executes intended commands. Though voice command systems are easy and convenient to use, they present a number of coding difficulties. First of all, speech-to-text conversion is difficult, though it does not need to be implemented from scratch. The biggest problem with voice commands is source attribution, determining which user issued a command.

User identification

Password

One possible solution for user identification is assigning each user a unique password, which they then use when they want to access the system. Though passwords are commonly used in a variety of computer applications, they are impractical for smart homes for a variety of reasons. First of all, passwords only work with an interface that allows the user to enter the password secretly. With regard to the interfaces we examined, this excludes voice commands, the only interface that fulfills our system goals of location and object independence. Additionally, users may find having to remember their passwords inconvenient, and may write them down somewhere easily accessible. This all but eliminates the security provided by a password system if an unregistered user finds the password record.

RFID chip

RFID chips provide a physical object to gain access to the system. The concept of an RFID chip resembles that of a door key. The user needs to hold the object in his hand to be identified and granted access to the system, which provides more security since someone without that “key-object” will not be granted access. There are two ways of carrying an RFID chip on a person, as an object, such as a card or a keychain, or as

an implant. As an object, this requires the users to carry an object with them, which is a lot of the time undesirable by people in their own homes. As an implant it is a controversial idea with a lot of opposition and a lot of fear surrounding it. Lastly, RFID chip requires a device to induce current in the chip, in order for the chip to transmit an ID. This process requires the RFID chip to be within close proximity to the reading device, which does not free the user's position in the room in order to give a command. Such an implementation would require the users to "check-in" to the system every so often.

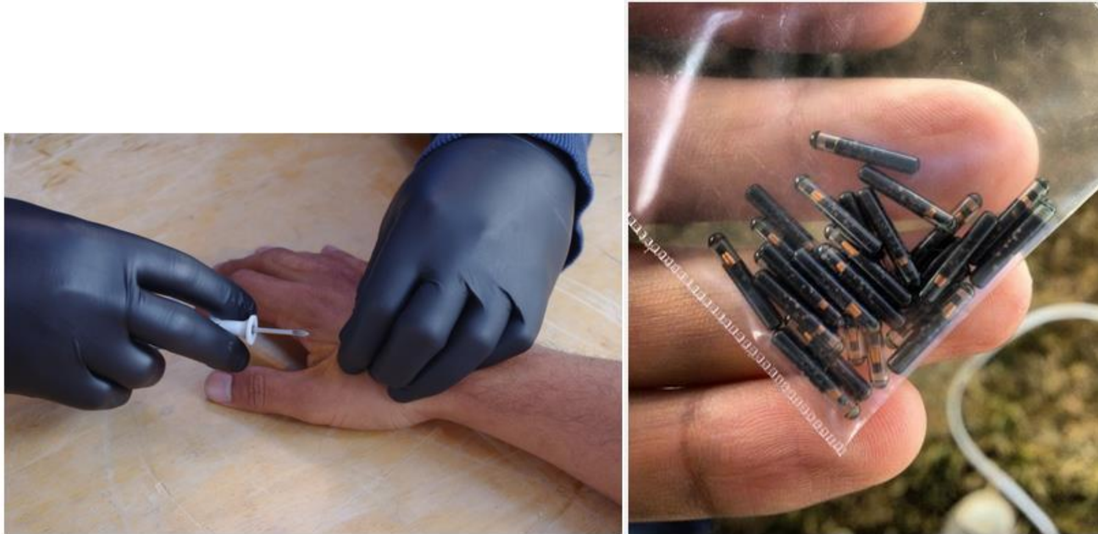


Figure 2: Graafstra's glass-encased RFID Chip implant process (Greenberg, 2011)

Biometrics

There are a number of physical characteristics that can be used to uniquely identify any person, such as fingerprints, the patterns of blood vessels behind the retina of the eye, and the texture of the iris. Each of these features is unique for each person, and the methods for detecting them have very high accuracy rates. Implementation for these can be used in much the same way as RFID tags, but because they are measurements of a person's natural features, do not require the user to either carry on object or have a chip implanted. However, these systems do maintain many of the other drawbacks of RFID tracking, as they are only effective within a very limited range. Fingerprint scanners require physical contact with the device in order to work. This means that the scanners must be either statically located within the house or incorporated into some sort of portable device which users would need to carry. This rules them out for our purposes, as our goals are to eliminate those inconveniences. Retinal scanners do not require physical contact, however they also have a very short range (up to 12 inches in a laboratory setting) (Hill, 1999), and require the user to cooperate with the scanner, meaning this type of identification cannot be done

passively. Therefore, we determined that biometric scanning would not be a viable solution for our system's user identification module.

Facial recognition

Facial recognition is a one way of identifying people that has been in development for many years. It allows a system to identify someone at a long range which is limited mainly by the resolution of the camera used rather than anything else. This solution does come with a couple of downsides, though. The addition of video surveillance is necessary to the system and even if it is explicitly stated that the images are processed and not stored, it still raises concerns that may make some homeowners uncomfortable. This deters people from installing the system. Another downside to the system is that facial recognition software is not very reliable when it comes down to identifying two siblings who look similar, but new software is being developed constantly, which allows the opportunity for improvement to the system without the need to install new hardware. This technology seems to be the only feasible solutions to our system that comes at the low cost of a camera and provides the long range identification desired.

Figure 3: Example of facial Recognition (Johnson, 2012)

User tracking

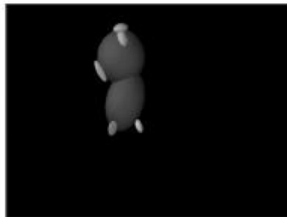
Video tracking

Video tracking usually works with an array of video cameras laid out around the house's rooms that capture images of the space. When an image that is different than the previous is captured, it is processed with object recognition software which determines whether that object is a human or not. When that object is classified as a human, then using depth simulation in 2D a relative position to the camera is calculated.

This method has the drawback that it requires two image capturing devices close to each other, since depth perception is vastly inaccurate in monocular vision[1]. Additionally, image processing is very dependent on resolution and highly intensive in processing power. So the more accurate the tracking is, the system processing the images needs to be exponentially more powerful, which leads to high costs. Lastly, video surveillance is something a lot of homeowners feel wary of and would be reluctant to install in their homes.

(a)

(b)



(c)

Figure 4: (a) Video input (n.b. color image, shown here in grayscale). (b) Segmentation. (c) A 2D representation of the blob statistics. (Wren, Azarbayejani, Darrell, & Pentland, 1997)

Active/Passive Infrared Tracking

A variety of infrared (IR) sensors can be used to track users. These come in two primary flavors, passive IR sensors and active IR sensors. Passive IR sensors are frequently used in security systems, and work by detecting heat within their range. When a warm object (such as a person) moves within range of a passive IR sensor, the sensor is tripped. Active IR sensors, on the other hand, consist of two units: an emitter and a detector. The emitter shines a beam of infrared light onto the detector, which sends a signal when the beam is interrupted. For a precise tracking application

[1] Vision obtained from a single point sensor.

such as ours, the application of both of these types of systems would be essentially the same, in one of two formats. For the first format option, each room would be divided into a grid, with either a passive IR sensor with a Fresnel lens or an IR emitter/detector pair for each grid line. The second format places the sensors at each door to detect movement between rooms. Each of these formats presents problems, however, the first with multiple persons in a room, the second with command association.

To show how a grid system struggles with multiple persons, assume Person 1 and Person 2 are standing in a room at positions (X1, Y1) and (X2, Y2), respectively. The room controller knows there are people on X1, X2, Y1, and Y2, but has no way to associate the coordinates into pairs, as having persons at positions (X1, Y2) and (X2, Y1) would yield the same results. Additionally, a third person at either (X1, Y2) or (X2, Y1) would not change the sensor readings, nor would a fourth at the other point. In fact, for any given sensor reading, the number of possible people in the room ranges from the greater of #X or #Y to #X*#Y, where #X and #Y indicate the number of X or Y values sensed. If the system only needs to be able to handle two or three people in a room at a time, this might be able to be addressed by keeping track of previous positions, however, this becomes increasingly complex as the number of tracked people grows, and since our system needs to be able to track up to six persons per room (meaning up to 36 possible locations), this format is really not feasible.

A door-based IR system does not suffer from scaling the way a grid system does, however, it does not allow the system to identify which user is issuing a command, which is the goal of a tracking system. Because this setup can only determine when a person enters or exits a room, it is really only useful for a simple system that does little more than turn lights off in unoccupied rooms. Since we want our system to be capable of more than that, this is not a feasible solution.

Ultrasonic Sensors

The use of sound mapping is widely used in robotics projects, nautical underwater mapping, archeology, medical devices and many other applications. It provides an accurate measurement of distance from the sensor to the objects in front of it. By choosing the right wavelength emitted, the sound wave will ignore objects of lower density than the intended target, providing a small layer of transparency, as well as controlling the range of accuracy of the sensors.

The applications for object mapping with sound waves are vast, but the technology seems to run into a lot of problems when it encounters the human body as a surface. Humans are made mostly of water, and the curvature of the body disperses sound unevenly, which reduces the accuracy of the sensors. A human will echo to the sensor a signal which is too small to detect within the accuracy of a reasonable cost effective sensor. Mass purchase of an ultrasonic sensor within our range of interest is in

the price range of $30 each (Bonar, 2012). Ultrasonic sensors for human tracking is used in a combination of two sensors spaced from each other covering the same range, which doubles the cost and narrows the range of the covered area.

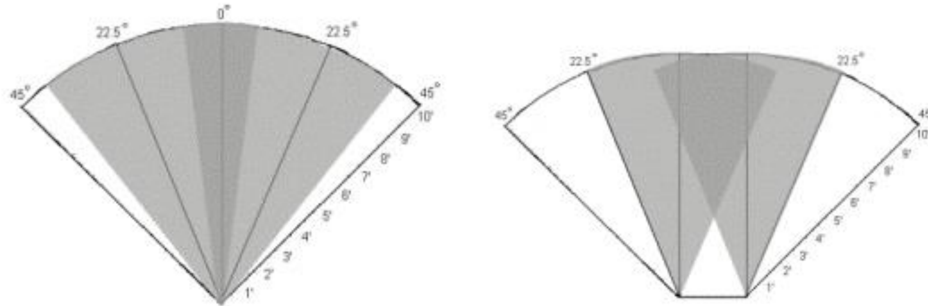


Figure 5: Ultrasonic sensor area of coverage

Those specifications force the use of very sensitive ultrasound sensors. High sensitivity might be an advantage in most cases, but when it comes to sound, it can lead to a lot of errors and false readings. The readings a high sensitive ultrasonic sensor makes can be easily altered. Most major problem with ultrasonic person tracking is the effect different clothing has on the readings. Different clothes will reflect sound differently, sending back significantly different waves when a person is for example wearing a jacket or wearing any metallic objects on them (Bonar, 2012). Additionally, it is feasible to adjust a wavelength to ignore “softer” objects but it is not feasible to ignore denser targets such as a table or a couch. Taking that into account, it becomes a major problem when trying to eliminate certain wavelengths from the equation in order to reduce noise or false readings. Thus, ultrasonic seems to serve room mapping well but shows major drawbacks in the detection of people, especially in a dynamic environment such as a home.

Thermographic Imaging

Using thermographic imaging as a tracking method works in much the same way as video tracking. Thermographic cameras are installed throughout the house, and software is used to evaluate where users are in the image and from there where they are in the room. The benefit that thermographic imaging has over video processing is that thermographic cameras detect wavelengths of light in the infrared band of the spectrum, while video cameras deal with visible light. This is helpful because users have warm bodies and emit infrared light in a fairly narrow band, whereas clothing comes in every color in the visible spectrum, and is often inconsistent as well. This means that people are significantly easier to detect in thermographic images compared to video. Additionally, facial recognition using thermographic images has been shown to be possible and very accurate, meaning one device can be used for both tracking and identification, as is the case with a video camera system.

Footstep tracking

One of the least error-prone technologies currently on the market for indoor applications is footstep signal tracking. One system, developed at Institut Eurecom, uses piezoelectric cables laid under the floor to form a grid (Rodríguez, Lewis, Mason, & Evans, 2008). These wires send a signal to a central processor when weight is applied to them as a voltage potential drop. The system is then able to accurately determine the location of each person in the area, and was successfully used to track 35 persons taking 160 steps each. The average error rate over the testing of 3500 was 13% which is more than enough to track people in a space.

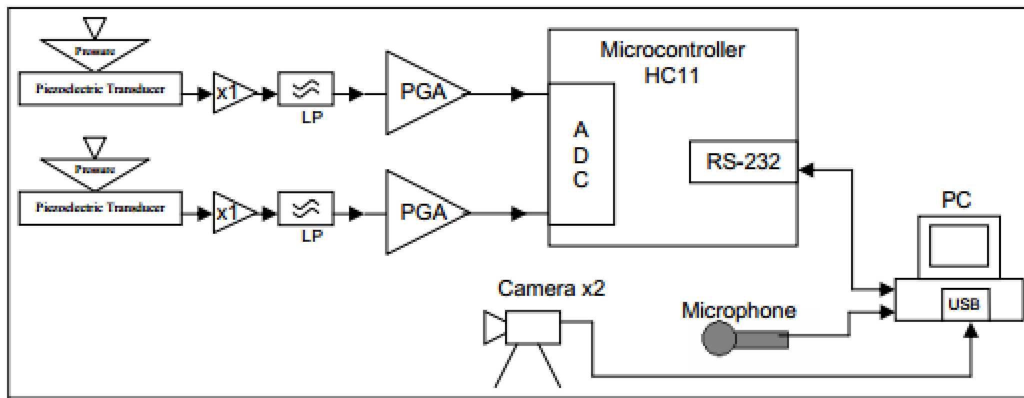


Figure 6: Connection of the hardware used for the footstep capture system. (Rodríguez, Lewis, Mason, & Evans, 2008)

While footstep tracking systems may work well for hallways or conference rooms, a study at Georgia Tech concluded that these systems have serious scalability drawbacks. The piezoelectric cable is not the cheapest of materials and when laid in a grid under a floor adds up to great lengths making both the material cost and installation cost significantly large.

Additionally, heavily furnished rooms pose a problem, as the furniture will exert constant pressure on the wires, possibly yielding false positive readings or overriding a legitimate signal. The final problem with these systems is that they are not easily installed in existing homes. Because the wires need to run under the floors, putting them into existing homes requires tearing up and reinstalling the floors wherever the system is put in place. The significant overhead associated with this makes these kinds of systems impractical for our purposes.

The Kinect & why it's awesome

During the research of available technologies, there was a specific platform suggested by multiple sources for development on. The Microsoft Kinect sensor is a platform that was developed by Microsoft for use in games but was quickly embraced by the community of individual developers as a powerful cheap platform with a lot of ready-

to-use functionalities. It has a video camera that can capture images of 1280-by-960 pixels at a rate of 12 frame per second or 640-by-480 pixels at a rate of 30 frames per second. Along with the camera, it uses an active IR array to detect depth in the environment in front of the Kinect at accuracy of 4 cm at the maximum distance. Furthermore, the platform has a microphone array of four microphones which provides noise cancellation and sound source angle determination.

The API that Microsoft provides for use with the Kinect provides functionality for both user tracking and voice commands, though these need to be adapted for use in our implementation. The tracking methods in the API do not maintain continuous tracks of users. Rather, the API simply returns an array of Skeleton objects for each data frame, and Skeletons for one person are not guaranteed to be in the same array index between one frame and the next. Therefore, in order to maintain an identity for each user, we needed to implement a way to match the incoming Skeletons to previously detected Skeletons that had been matched with user identities. Similarly, Microsoft's API has no way to uniquely identify users. However, we were able to find the third-party EmguCV API that provides facial recognition capability for the Kinect. EmguCV is a C# wrapper for the well-known OpenCV video-processing library, both of which are open-source and freely licensed.

The disadvantages of using the Kinect are that it is not a viable solution for a final product because it is owned by Microsoft. This makes it not only significantly more expensive than buying the hardware we would need to make our own sensor system, but would require us to work out a deal with Microsoft before we even thought about selling the system. Additionally, because we are using Microsoft's closed-source library to do tracking, we would need to figure out another method of tracking if we were to use anything other than the Kinect. However, the existence of the Kinect proves that these things are possible, and we felt that our limited time would be better spent working on ensuring that our system can uniquely identify and track users rather than building and debugging a complex hardware system that somebody else has already proven is possible.

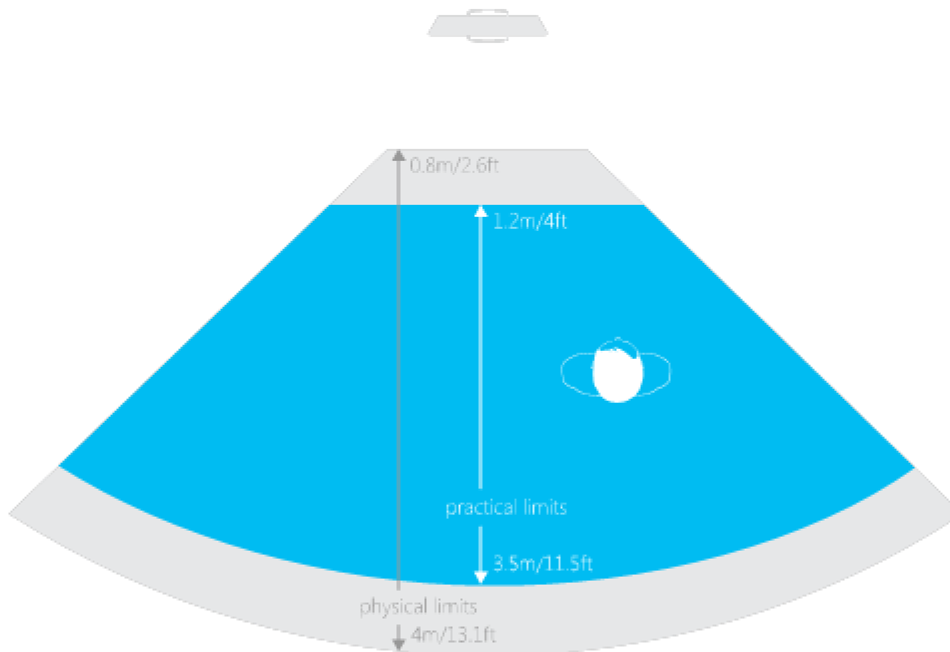


Figure 7: Effective horizontal view range of the Kinect sensor according to Microsoft TM

Kinect sensors are sold at the cost of $150 dollars each, and its terms of use do not allow a system to use the Kinect as a package in the application we intend it to be used. This means that it is not the final solution sensor for the software. It is though, the most suitable option for this project since it has all of the available hardware, ready to use out of the box, its hardware is not expensive to make, but expensive to purchase in the Kinect package. The scope of this project is to prove that the solution suggested is feasible, and the use of the Kinect brings the project a few steps closer to the desired solution. The total cost of the hardware inside the Kinect package is $56 which is not an unreasonable price for each sensor package (Graft, 2012).

System Design

Broad system overview

Having decided that the Kinect would be our best choice for a proof-of-concept prototype, we needed to determine exactly how we were going to fulfill our system requirements. Because we decided to take a modular approach, using an object-oriented programming language seemed like the most obvious course of action. Fortunately, Microsoft provides API's for the Kinect in several languages, including C++ and C#. We decided to work in C#, though we now realize that C++ may have been a better option.

User tracking was done through Microsoft's Kinect API, which was specifically designed for that purpose. Unfortunately, the Microsoft algorithms that detect joints (which are the basis of the tracking system) are proprietary and hidden from users of the API. Therefore, we were unable to dig into those to figure out how they worked. This is problematic, as anyone who wants to fully implement our system will either have to use the Kinect as their sensor array or develop their own person-detection software.

Facial recognition is done by collecting images from the Kinect color image stream, then processing them to recognize the faces in each picture. This is a multistage process. First it converts the image to grayscale; it then finds the faces in image. It crops out and sizes down the faces in the image to speed up the recognition and sends it to the facial recognition component in the mainframe computer. There it compares each recognized face against the known faces and returns the tag associated with the closest match. If a face is not closely related to a trained face then it returns a blank tag.[2]

The command interface we plan to use with our system will rely on voice commands. Microsoft's API for the Kinect includes voice-to-text functionality, which should allow us to read in commands as if they were typed strings. Though implementation of command execution is outside the scope of this project, simple string parsing and manipulation should make command recognition and interpretation easy. Matching commands to users will be done in much the same way that we match recognized faces to users, as the Kinect has the capability to determine the angle at which a command was heard. Once a command has been interpreted, the command and the identity of the user who issued it may be sent to the central house-wide processor, which will then determine if the room is capable of executing the command, and if the user has permission to issue it.

[2] See section "Facial Recognition" for more details.

System modules

Figure 8 illustrates the various modules of the system, and how they work together to identify users and the commands they issue. The sensor array, a Microsoft Kinect in the prototype, consists of a microphone array, a depth sensor, and a visible-light camera. First, each room's computer uses the depth sensor and video camera together to determine the location of each person in the room. This data is combined with data obtained previously to generate a track history for each person, meaning the system does not have to identify each person every time it receives new data. At the same time, the room computer is examining the video data and trying to identify human faces. When it finds a face, it pulls it out of the video frame, standardizes its size, and sends it to the central computer for identification. Once the face has been identified, the identity is sent back to the room computer, where it is associated with the corresponding track history, completing the system's representation of a person. When the microphone detects an incoming command, it checks the angle of origin of the command against the location of each person in the room in order to determine who issued the command. The command issued and the identity of the user who issued it are sent to the central computer, where they are checked against the user permissions database. This database holds a listing of each registered user and the commands they are allowed to issue. If the command is deemed permissible, the system proceeds to execute it, otherwise, it is ignored.

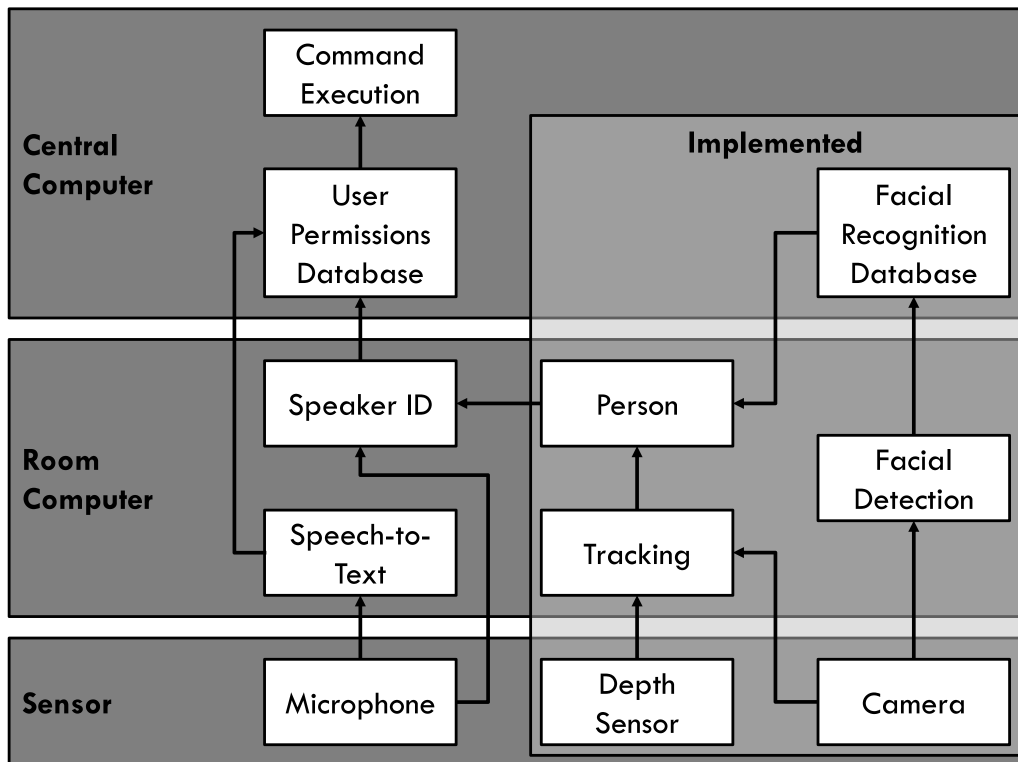


Figure 8: System Block Diagram

Hardware overview – system architecture

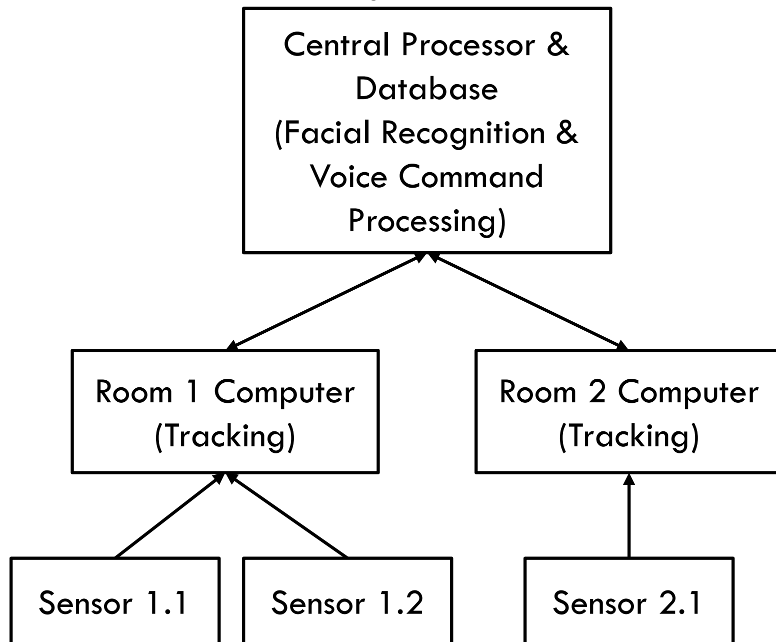


Figure 9: Hardware Overview Diagram

Central Processor

Each house our system is installed in will require a central, house-wide computer responsible for storing information that each room might need and performing general-purpose operations. The most basic functionality this computer must be able to handle is determining user identity based on facial data, and execution of user commands. Facial identification should be run on the central computer as running it on each room computer would necessitate duplication of the facial recognition database in each room. Centralizing facial identification also ensures that the list of recognized faces is consistent across every room in the house. System maintenance is also easier when the data is all located in the same place. If some of the facial data becomes corrupted and a user is no longer properly identified, the corrupted data only needs to be corrected on the central computer, rather than on every room computer. Command processing should occur on the central computer for much the same reason. There is no need to have a separate copy of the command parsing algorithm or the user permission database in each room, especially since commands may span multiple rooms, or affect user profiles, which would be stored centrally.

Room Computers

Just as the central computer should handle certain things so that they do not need to be repeated on each room computer, there are some things that the central computer does not need to know, and should be handled by the room computers. For example, the central computer does not need to know the precise location of each

person in the house. However, each room does need to know where the people within it are located so that it can correctly identify and associate commands with them. Additionally, though facial identification occurs on the central computer, each room should run facial detection on the video data it receives from its cameras. Doing this means that the central computer does not need to process every frame of video from every camera in the entire house looking for faces. Instead, if each room computer finds the faces in its video frames, it can send just the faces to the central computer, which then only needs to identify them. Though this will slow down frame rates for tracking, the improvement in response time for identification from the central computer will be well worth it.

Sensors

Each room will contain at least one Kinect, but may need more depending on its size and shape. Though our prototype only uses one Kinect, adding more should not cause many difficulties. In order for that to work, the room computer would need to be able to tell which Kinect it was receiving data from and the location and orientation of each Kinect. Knowing that, it should be simple to convert coordinates relative to any Kinect onto a room-wide coordinate system.

Though our prototype consists of one computer and one Kinect, many rooms in the houses that our system may be installed in will either be too large or otherwise shaped such that one Kinect will not be sufficient to offer tracking throughout the room. Although the Kinect is only a development platform, we expect that a similar, custom-built sensor array will have a similar effective range. Therefore our room computers must be able to process and correlate data from multiple sensor arrays. The best way to do this would be to define one of the sensor arrays as the basis for the coordinate system for the room. Data coming in from other sensors must be translated into the global coordinates, but some basic trigonometry provides a means for this by these equations:

$$x = x_0 + \sqrt{x_1^2 + y_1^2}\,\sin\left(\theta_0 + \tan^{-1}\frac{x_1}{y_1}\right)$$

$$y = y_0 + \sqrt{x_1^2 + y_1^2}\,\cos\left(\theta_0 + \tan^{-1}\frac{x_1}{y_1}\right)$$

Equation 1: Coordinate System Transfers

where (x, y) is the position of the point in the global coordinate system, (x_0, y_0) is the origin of the local coordinate system, θ_0 is the angle between the y-axis of the global system and the y-axis of the local system, and (x_1, y_1) is the position of the point in the local coordinate system.

Software overview

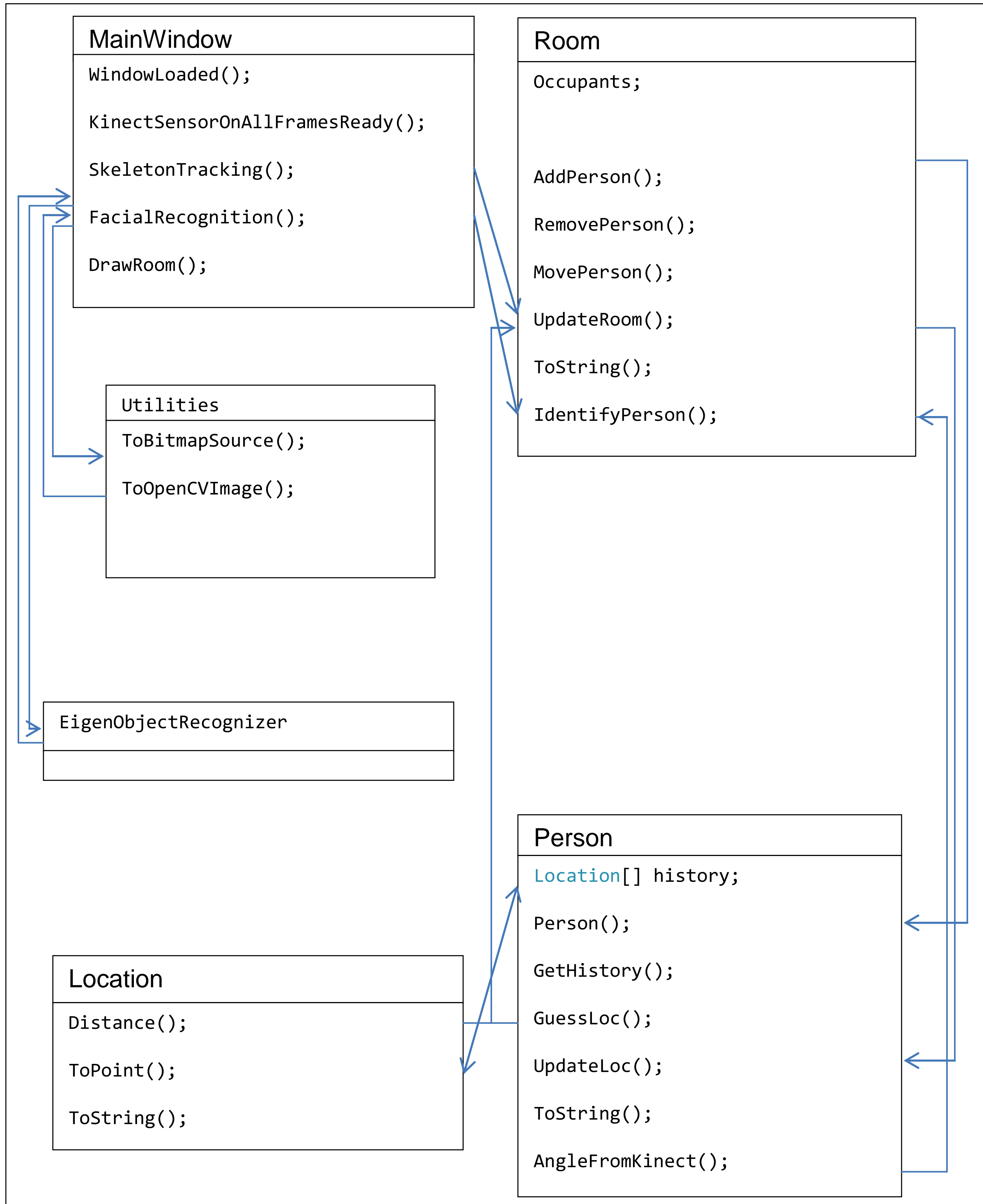


Figure 10: UML Graph of Software

MainWindow Class

The MainWindow class holds the generic functionality of the software. It holds a set of global variables for the proper functionality of the code and methods for the primary functions of the software. When the software starts, it executes the MainWindow() method which initializes the components of the GUI (for testing purposes), then loads the eigenfaces[3] and the trained faces that the facial recognition is going to run against. It also creates the instance of the room that is going to be observed.

When the MainWindow() method is complete, the MainWindowLoaded() runs. This allocates the images in the GUI (for demonstration and testing), locates the Kinect sensor and instantiates it as an object.

FacialRecognition() method is an event handler that is fired whenever there is a new image captured by the Kinect's ColorStream[4] and the previous execution of this method is already complete. The EgmuCV library can only process images in a format native to EmguCV called IImage. The image captured by the Kinect is originally stored as a bitmap image and then converted to an IImage using the ToOpenCVImage() method in the Utilities class. Once that occurs, the rest of the method sends the IImage to the EigenObjectRecognizer to detect and identify faces. In the end it stores the ID of the faces recognized and their position in the picture and sends the data to the IdentifyPerson() method in the Room class to be assigned to tracked persons appropriately.

The SkeletonTracking() method is another event handler that executes whenever the Kinect's SkeletonStream generates a new SkeletonFrame. The SkeletonStream is an abstraction that uses the Kinect's camera and rangefinder to determine where each person in the Kinect's range is located. Each SkeletonFrame generated by the SkeletonStream consists of an array of Skeleton objects, which in turn contain three-dimensional coordinates for each joint the Kinect recognizes. The SkeletonFrame array does not guarantee that the first Skeleton will be located at array index 0, the second in array index 1, and so on, so our method needs to go through the array and determine which Skeletons are actually valid before using those Skeletons to generate an array of Locations, which is then passed to the UpdateRoom() method in the Room class.

---

[3] Look in "Facial Recognition" for explanation
[4] Frame by frame video image capture as bitmap input.

The `KinectSensorOnAllFramesReady()` method is an event handler that is triggered whenever there is a new frame of data from the skeleton detection and the image captured by the `ColorStream`. This event is a joined version of the `FacialRecognition()` and the `SkeletonTracking()` methods combined.

`Location` Class

Because our system is designed to track people in 2D space, we designed the `Location` class to represent a point in 2D space. `Locations` can either be constructed by providing an x- and a y-coordinate, or from a `Skeleton` object, which is obtained from the Kinect's `SkeletonStream`. The `Location` class also has a method to determine the distance between two `Locations`, which is an important part of our tracking algorithm.

`Room` Class

In most ways, our Room class is just a glorified array of Person objects. However, it also contains the tracking algorithm, which associates the Skeletons from the Kinect's SkeletonStream with the Persons in the Room. Rooms are also created with boundaries, which are used to determine when Persons leave the Room and should be removed.

The `Room` class in our system forms the base that the rest of the software is built upon. Each `Room` is defined by a `size` (how many `Persons` it can hold in its `occupants` array), `boundaries` (the set of `Locations` that are considered within the `Room`), and its `MAX_MISSED` value, which defines the number of frames of `SkeletonStream` data an `occupant` of the `Room` can go without receiving a new `Location` from a detected `Skeleton`. `Occupants` are removed from the `Room` either when they move to a `Location` outside of the `Room's boundaries`, or if they miss more frames in a row than the `MAX_MISSED` value.

The `Room` class contains methods for adding and removing `Persons` from the `Room`, for moving `occupants` to other `Rooms`, and for determining whether the `Room` is fully updated. `Rooms` are also responsible for running our track association algorithm, which matches the `Persons` in the `Room` with the incoming `Skeleton` data. This is examined in more detail later.

`Person` Class

Our `Person` class is focused on maintaining the information our system needs in order to correctly associate commands with users. This basically consists of keeping a location and an identity for each person and ensuring that updating either one does not result in the loss of the other. Because we cannot guarantee that facial recognition data for each person is available in each frame, we need to keep track of where each person

has been and use that tracking information to associate incoming data from the `SkeletonStream` with existing tracks. Exactly how this is done will be addressed when we look at the tracking module in detail, however, knowing that we need to keep track of not only where each person is, but also where they have been and are likely to be helps explain some of the choices we made both in the `Person` class and the `Room` class.

Each `Person` object stores an array of `Locations` (`history`) corresponding to the last 15 `Locations` it has occupied previously, as well as the number of frames of `SkeletonStream` data in a row for which this `Person` has not been matched to an incoming `Skeleton`. This number is used by the `Room` class to identify when the `Person` has been lost to tracking and should be removed.

The primarily interesting methods in the `Person` class are the `guessLoc()` method and the two `updateLoc()` methods. `guessLoc()` uses a `Person's` previous history to anticipate where it is going to be next. It takes the `Person's` current `Location`, adds the difference between its current `Location` and its previous `Location` (its approximate velocity) and the average change in its displacement across its entire `history` (its approximate acceleration). When a `Room` object is trying to associate incoming `SkeletonStream` data with the `Persons` it contains, it uses the `Location` anticipated by `guessLoc()` as the center of the search for each `Person`. If the `Room` finds an incoming `Skeleton` within range of the expected `Location`, it calls the `updateLoc()` method that takes a `Location` argument, passing in the `Skeleton's Location`. This method updates the current `Location` to match the incoming `Location`, and clears the missed frame counter. If there is no `Skeleton` within range, however, the `updateLoc()` method with no arguments is called. This updates the current `Location` of the `Person` to the `Location` provided by `guessLoc()` and increments and returns the missed frame counter.

Facial Recognition

Our facial recognition module operates separately from the tracking algorithm, and utilizes the Kinect's color camera. To match a recognized face to a Person, we use the face's x-position in the video frame to determine its angle from the Kinect. We can then calculate the angle for each Person in the Room, and attach the face to the one with the angle closest to the one calculated for the face.

Tracking in detail

Our system's tracking process works in two steps, person detection and track association. Person tracking takes the input data from the camera and rangefinder and determines where people are located in the room, and track association takes that data

and uses it to update, create, or remove track histories for individuals. Person detection comes as part of Microsoft's API for the Kinect, and the exact mechanisms for it are hidden. Track association, however, we had to implement ourselves.

We interact with the `Skeleton` tracking portion of the API by enabling a `SkeletonStream`, which alerts us whenever it has compiled a new frame of data. The `SkeletonFrames` generated by the `SkeletonStream` each contain an array of `Skeletons`, as well as some additional information not relevant to our project. The Kinect can fully track up to two `Skeletons`, meaning that it provides accurate position data for each of 20 joints on those two, though it can also provide accurate position data for center-of-mass for up to four additional `Skeletons`. As our project is only interested in determining the location of each person in a room, we ignore most of the data provided by the fully-tracked `Skeletons`, and only pull information on their centers-of-mass. Having obtained all the location data it needs, our program enters its track association algorithm.

Track association refers to the method by which incoming position data is associated with existing tracking data, and has been frequently studied with regard to radar and other tracking systems. Because the data our system receives from the Kinect is similar to data provided by radar systems, we were able to utilize similar techniques to those we studied. Figure 11 illustrates how our system works in a typical example. Once a new frame of `Skeletons` has been obtained, each `Skeleton` is converted into a `Location` using the x- and z-coordinate of its center of mass as the x- and y-coordinate of the new `Location`. These are represented as the large dots in the figure. The black traces beginning at 1_0, 2_0, and 3_0 represent the `histories` of three `occupants` of the `Room`. Each `Person` calls `guessLoc()`, and their anticipated positions are shown as the small dots at the end of each trace. When looking for potential new `Locations`, each `Person` only looks within a limited range, represented by the circles centered on the `guessLoc() Locations`. This prevents existing tracks that should not have a new `Location` from attaching themselves to `Locations` that should become new `Persons`. If this was not implemented, track 3 in the figure would be updated with `Location` 4_1 instead of 3_1, which should clearly not happen. If multiple `Locations` fall within the range, the `Person` chooses the `Location` nearest its `guessLoc() Location`, as shown with track 1. If a `Person` has no `Locations` within range, it moves to its `guessLoc() Location`, increments its missed frame counter, and reports the number of frames it has missed to the `Room`. If this exceeds the limit set by the `Room`, the `Person` is removed. Once all `Persons` in the `Room` have updated their `Locations`, unclaimed `Locations` are turned into new `Persons` and start a new track, as would happen with `Location` 4_1 in the figure.

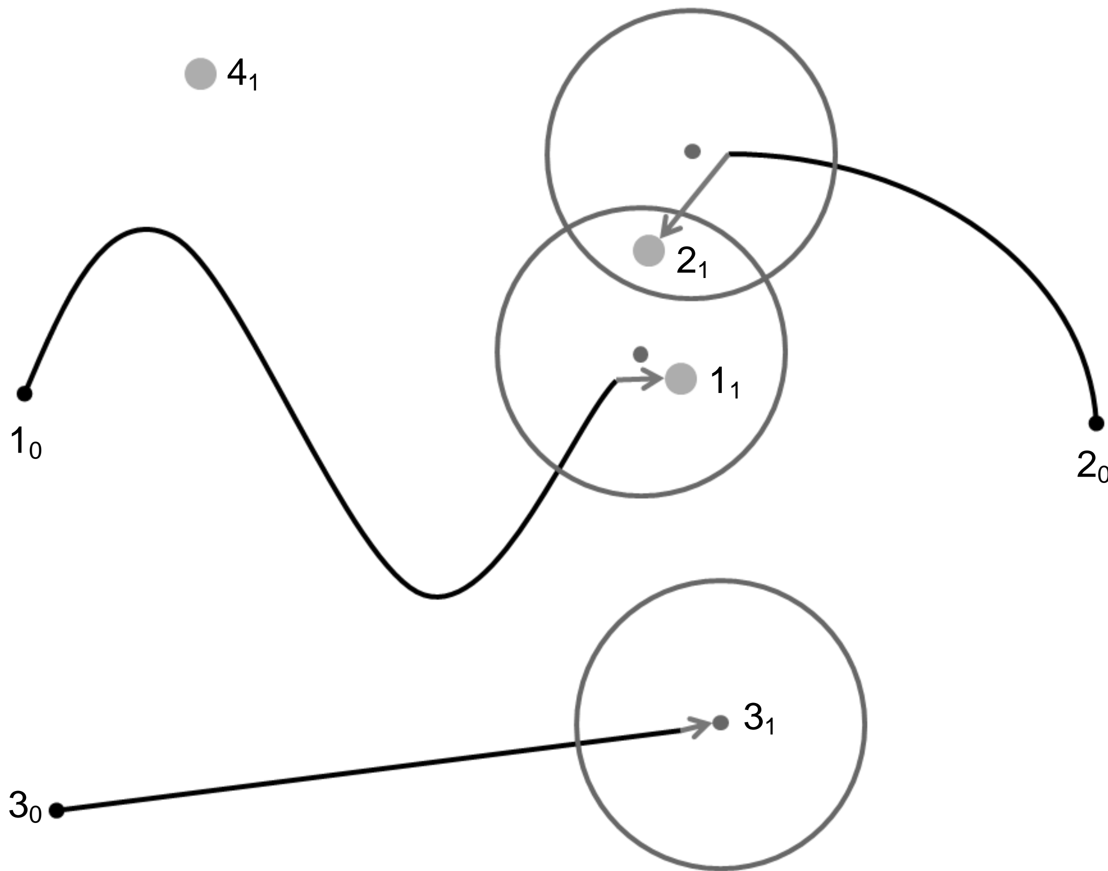


Figure 11: Track Association

Facial Recognition in detail

Facial recognition in our system works by reading in the images from the video camera on the Kinect and processing them using the eigenobject recognizer in the OpenCV library. The OpenCV library is accessed through the EmguCV wrapper which allows the OpenCV library, which is written in C++, to be accessed by .NET languages such as C#.

The EmguCV library's facial recognition algorithm is based on eigenfaces, a technique that takes advantage of the fact that human faces have a fairly uniform structure, and that defining features have consistent locations. This means that human faces can be represented as a linear combination of images designed to highlight these differences. These generalized images, which are essentially eigenvectors used to map images into a vector space of human faces, are the eigenfaces, and any human face can (in theory) be uniquely identified by determining the eigenvalue for each eigenface. In addition, the eigenface method can tell if a given image is not a face, as its eigenvalues will be far removed from the average.

A grayscale image can be regarded as a two dimensional matrix, with dimensions being the vertical position and the horizontal position, with the shade of each pixel being the value held in the element. Each vector in a space can be defined as a combination of scalar values multiplied by a set of dimensional vectors, called eigenvectors. For example, the standard vectors used to define a vector in a two dimensional space are x and y. An eigenface is a combination of eigenvectors and defines an image as a combination of values multiplied by “standardized face ingredients”, which are a statistical analysis’s of large numbers of pictures of faces. This analysis is done through a technique called Principal Component Analysis (PCA) (Jolliffe, 2002). PCA is a mathematical algorithm used to reduce the number of interrelated data points to a set of uncorrelated value that retain all the variety of the original data. This allows the facial recognition to run faster without becoming notably less accurate. The EigenObjectRecognizer method of the EmguCV library also uses this technique as well, to reduce the amount of data in the trained images and in order to perform facial recognition faster and with less processing power.

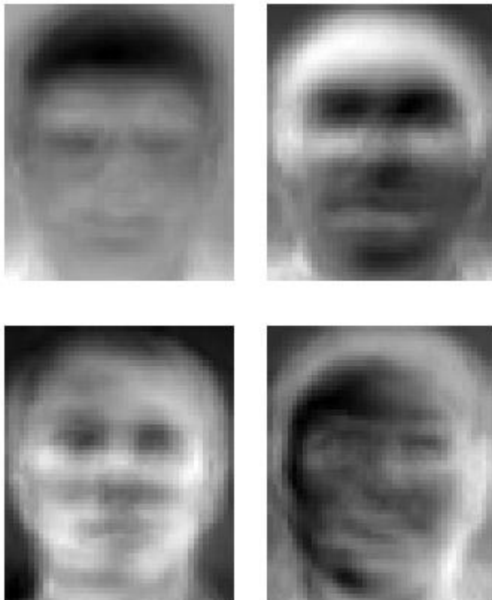


Figure 12: Some eigenfaces from AT&T Laboratories Cambridge

The system is designed to take in 12 greyscale frames of 1280-by-960 pixels every second and process whether it detects an object of close eigenvalues to a standard face. When an area of the image has a detected face, it copies the region in which the face was detected and crops it to a new picture frame of 20 pixels by 20 pixels. It then takes that new picture and sends it to a different function, in the mainframe computer to be processed for identification.

The trained faces of the system are loaded when the system starts. Each of the trained faces is a single grayscale image of 20-by-20 pixels and is associated with a label which is the ID of the user. For development and testing purposes, that ID was the name of the person in the picture but the system is not bound to that.

Once the image of the unidentified person is sent to the facial identification portion of the software, it is compared against each of the trained faces in a linear fashion. When an object is detected to be within a certain eigenvector “distance” from the trained image, the facial recognition will consider it successful, stop further comparisons and return the label/ID of the recognized face. If the comparison fails for all trained faces, then the method returns an empty string. When the recognition is successful, the software picks up the center of the position of the face in the picture and the ID of the person in that picture. Then it divides the number of pixels on the width of the picture by the angle of the horizontal view range of the Kinect sensor (57°) to figure out the relative angle of the face detected to the Kinect sensor. Once that angle is taken in by the rest of the software, it is compared against the relative angle of the tracked people in the room and if it falls within 3° of a person, that person’s ID is updated to the recognized face’s ID.

One of the major drawbacks of eigenfaces, especially for our system, is that the technique is extremely sensitive to variations in orientation and lighting. These can drastically change the appearance of a face, making it difficult for the system to correctly identify a person, and possibly preventing it from recognizing a face as a face in the first place. However, as we do not need to continuously keep track of our users’ faces, we can wait for optimal conditions to identify them.

System Testing & Analysis

Tracking Testing

Testing the tracking module of our system consisted of three parts. In the first, we gathered data for one person standing motionless in front of the Kinect in order to determine the precision of our system at various ranges. In the second, we examined how our system fared when it had to track a person moving in a number of patterns. Finally, we examined how well the system dealt with multiple people in a small space. The system was configured for textual output of the positions of all detected persons for all of these trials.

Static Testing

The results of our static testing revealed that although the system does lose some precision at longer ranges, it remains well within acceptable limits. The results for our 2- and 4-meter tests are illustrated in Figure 13 and Figure 14, respectively, clearly demonstrating the discrepancy in precision. At 2 meters, our 244 data points fall within a box 2.6 cm x 1.6 cm. The 363 data points at 4 meters fall within a box 11.8 cm by 7.9 cm. Having determined that people generally do not get closer than 60 cm, we felt that the system was unlikely to confuse two stationary people having a conversation, even at its maximum range.

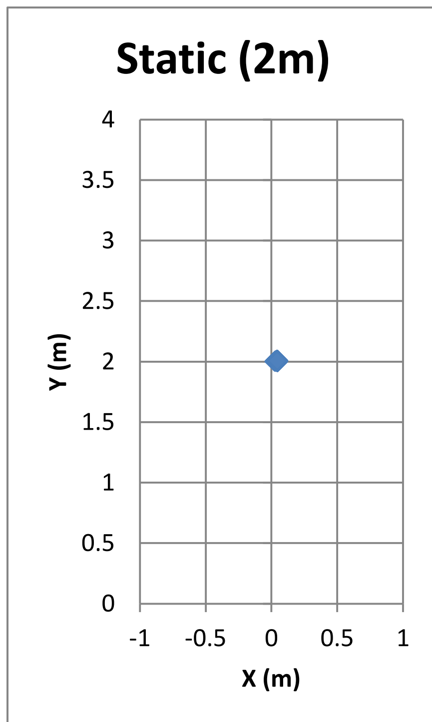


Figure 13: 2-meter Static Test

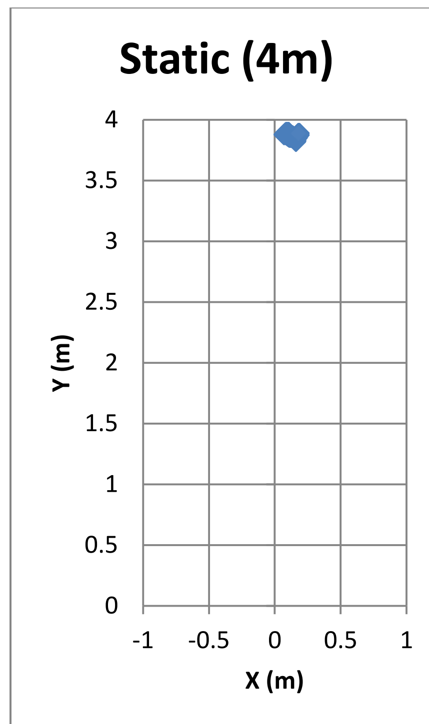


Figure 14: 4-meter Static Test

Motion Testing

As with the static tests, the tests with a moving person indicate that our system works as anticipated. Results for four of the tests are show in Figure 15 through Figure 17. For these tests, the volunteer walked at a normal pace from one designated point to another. These tests gave us useful information on both the Kinect's range and average walking speed. The data gathered from these tests showed that average distance moved per frame was just over 1 cm.

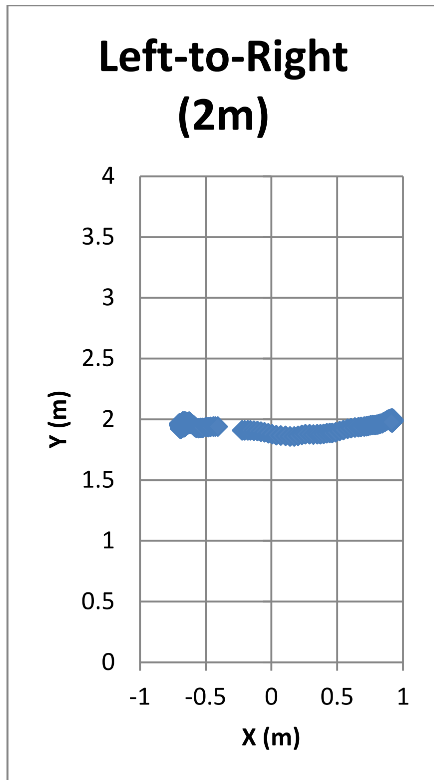


Figure 15: 2-meter Walking Test

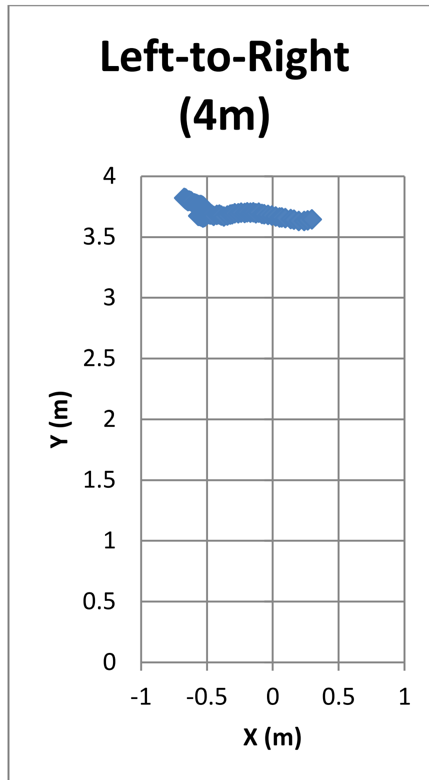


Figure 16: 4-meter Walking Test

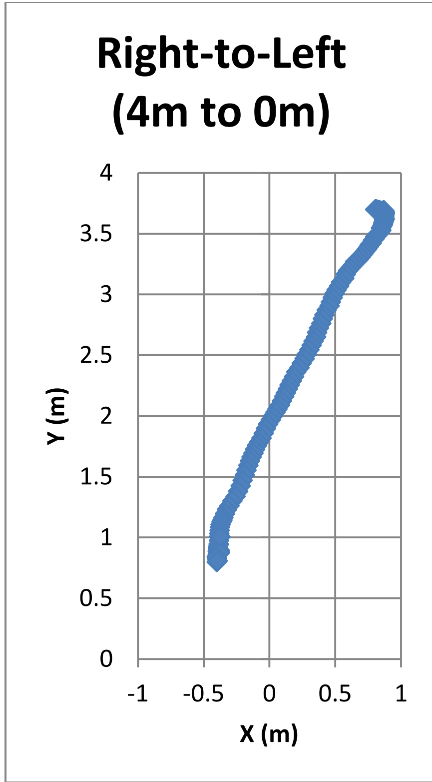


Figure 17: Diagonal Test

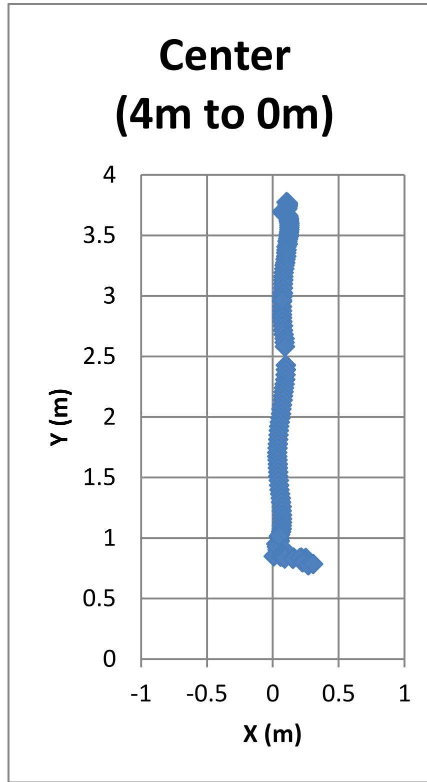


Figure 18: Approach Test

Facial Recognition Testing

After implementation of the facial recognition software through the Kinect sensor the software was unable to run in real time. The issue observed is that the GUI updating the display of the image froze and was unresponsive to user input. Debugging and running the software under performance profiling software showed that the software was taking too much time within the EgmuCV `EigenObjectRecognizer` function and particularly in an error handling task. After a non-constant period of time in the function, the `EigenObjectRecognizer` would recognize the face in the picture and return the tag associated with the face, but would not update the display. Going through facial recognition step-by-step in the debugger showed no errors and updated the display image with the detected face and recognized tag appropriately. Due to time constraints we were unable to figure out the issue.

For real-time testing purposes, we wired the facial recognition software to a laptops integrated camera and used CyberLink YouCam® software to feed the image stream. The software was tested to detect up to 6 faces, which is the maximum number of people our system is designed to track at the same time. The maximum distance that a face can be detected was measured to be 2 meters away from the camera, on a capturing resolution of 720-by-480 pixels of the integrated HP TrueVision HD Webcam. The Kinect is capturing at a higher resolution of 1280-by-960 pixels, which should

theoretically slow down the facial detection and increase the range of detection, when it is able to run real time.

In order to test facial recognition, we trained the software with a gradually increasing number of images per person. The system did not show a significant increase of processing time per trained image up to 50 trained images added. The system is weak in recognizing different lighting and orientation as predicted, but the more trained faces are added for each person, with different orientations and under different lighting, the face recognition increases drastically. The minimum suggested number of trained images of each person was found to be approximately 10. When there were fewer than 10 trained images of each person, the software seemed to misidentify faces or more often than not return no matches.

Integrated Testing

Because the facial recognition alone did not work in real-time, we were unable to ascertain whether the full system would work in real-time. However, as with the facial recognition alone, stepping through the code in debug mode gave promising results. Not only did the two parts continue to function separately, indicating that there were no conflicts between them, the system correctly matched the identified faces with `Person` tracks. We anticipate that solving the problems with facial recognition will result in a fully-functional, integrated system.

Conclusion & Future Work

Despite the fact that our prototype did not meet all the requirements we set out to fulfill, we believe that it could be developed into a fully functioning system without too much additional effort. Our tracking module is fully functional and runs in real time, and the issues with our facial recognition seem to be easily addressable. In fact, when run step-by-step, both the facial recognition and tracking work together successfully, demonstrating that they do not interfere with one another. Microsoft's API supports the use of voice commands with the Kinect, and will allow our system to associate the commands with users, resulting in the full realization of our goal.

For further improvement of the system, it is suggested to rewrite the existing code in C++. This is not a hard feat to accomplish since the Microsoft Kinect API can be programmed in C++ and the OpenCV library is natively in C++. This should allow a much better understanding of the issues that the system currently has, and since C++ is a language which allows memory allocation and a lot more control over the software, it should be easier to tackle the issues in that language. Doing so also takes the EmguCV API out of the process, which is a simple wrapper of the library, which should allow the software to run smoother and potentially avoid bugs in the wrapper.

Another suggested development is the addition of a security system. Since the hardware of a camera is in place, video surveillance of every room is available. That would allow the homeowners access to a video stream of any room they desire. It is also possible to integrate the facial recognition to alert the homeowner if there is an individual in the house that it is unable to recognize for a long period of time, or the face detected is far too different than the trained faces in the system.

Adding the voice command portion with a permission based system is one of the first things that can be added to bring the system closer to completion. The implementation of such a feature is pretty straightforward. The Kinect sensor has a microphone array that can detect speech, cancel out noise and determine the angle of origin of the sound. The Microsoft Kinect API holds a set of functions that give the angle of origin in degrees from the center and Microsoft also provides a Speech-to-text API for the Kinect sensor. Once a user issues a command to the system, it can convert the command to text and associate the command to the user who issued it in a much similar manner as the facial identification. Then it can check whether the user has the permission to issue said command and if he is, then the system can output the command in a form that a home automation system can receive.

A more complicated, but much needed addition to the system, is the use of multithreading. The software has to perform a lot of parallel tasks and some of them, such as the skeletal tracking cannot be interrupted. This leads to the need of a very powerful processor, which is costly. By splitting the processing of different tasks to

different processors, the system can run much more smoothly and it will cut down the cost. This can be done by the use of *ROS (Robot Operating System)* (ROS.org, 2013) which is licensed under an open source, BSD license. ROS is an API that can be used to transfer data from one device to another for processing and the devices can share resources. Each processor can be dedicated to each task and that is expected to help alleviate the real time issues the current software is faced with.

Bibliography

Bonar, T. (2012, 10 7). *Ultrasonics and People Detection*. Retrieved December 3, 2012, from MaxBotix High Performance Ultrasonic Sensors: http://www.maxbotix.com/articles/037.htm

Graft, K. (2012, November 10). *Report: Kinect Parts Cost Around $56 Total*. Retrieved April 24, 2013, from Gamasutra The Art of Buisness & Making Games: http://www.gamasutra.com/view/news/31517/Report_Kinect_Parts_Cost_Around_56_Total.php

Greenberg, A. (2011, August 13). *Want An RFID Chip Implanted Into Your Hand? Here's What The DIY Surgery Looks Like* . Retrieved March 12, 2013, from Forbes: http://www.forbes.com/sites/andygreenberg/2012/08/13/want-an-rfid-chip-implanted-into-your-hand-heres-what-the-diy-surgery-looks-like-video/

Hill, R. (1999). Retina Identification. In A. K. Jain, R. Bolle, & S. Pankanti, *Biometrics: Personal Identification in Networked Society.* Kluwer Academic Publishers.

Johnson, C. (2012, September 11). *EMGU Multiple Face Recognition using PCA and Parallel Optimisation.* Retrieved January 30, 2013, from Code Project For those who code: http://www.codeproject.com/Articles/261550/EMGU-Multiple-Face-Recognition-using-PCA-and-Paral

Jolliffe, I. (2002). *Principal Component Analysis, Second Edition.* New York: Springer-Verlag New York, Inc.

Kelly, N. (2009). *A Guide to Ultrasonic Sensor Set Up and Testing, Instructions, Limitations, and Sample Applications.* Michigan: Michigan State University.

Lofquist, D., Lugaila, T., O'Connell, M., & Feliz, S. (2012, April 1). *United State Census Bureau.* Retrieved April 22, 2013, from U.S. Department of Commerce United State Census Bureau: http://www.census.gov/prod/cen2010/briefs/c2010br-14.pdf

Rodríguez, R. V., Lewis, R. P., Mason, J. S., & Evans, N. W. (2008). Footstep Recognition for a Smart Home Environment. *International Journal of Smart Home*, 95-110.

ROS.org. (2013, April 23). *ROS.org*. (I. Saito, Editor) Retrieved April 24, 2013, from ROS.org: http://www.ros.org/

TopTenReviews. (2013). Retrieved April 23, 2013, from TopTenReviews: http://home-automation-systems-review.toptenreviews.com/mcontrol-review.html

TopTenReviews. (2013). *TopTenREVIEWS We Do the Research So You Don't Have To.* Retrieved April 23, 2013, from TopTenReviews: http://home-automation-systems-review.toptenreviews.com/mcontrol-review.html

Wren, C. R., Azarbayejani, A., Darrell, T., & Pentland, A. P. (1997). Pfinder: Real-Time Tracking of the Human Body. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 780-785.

Appendix

MainWindow.xaml

```
<Window x:Class="WpfApplication3.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="700" Width="1050" Loaded="WindowLoaded">
    <Grid>
        <Image Name="TrackerDisplay" HorizontalAlignment="Left" Height="480"
VerticalAlignment="Top" Width="640" Margin="392,10,0,0"/>
        <Image Name="ColorImage" HorizontalAlignment="Left" Height="240"
VerticalAlignment="Top" Width="320" Margin="10,250,0,0"/>
        <Button Name="FullStart" Content="FullStart!" HorizontalAlignment="Left"
Margin="10,10,0,0" VerticalAlignment="Top" Width="205" Click="FullStart_Click"
Height="35"/>
        <Button Name="DryCamStart" Content="Dry Camera Start" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="100" Margin="10,60,0,0" Click="DryCamStart_Click"/>
        <Button Name="DryCamStop" Content="Dry Camera Stop" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="100" Margin="10,87,0,0" Click="DryCamStop_Click"/>
        <Button Name="StartSkeleTrack" Content="Start Track" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="100" Margin="115,60,0,0" Click="StartSkeleTrack_Click"/>
        <Button Name="StopSkeleTrack" Content="Stop Track" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="100" Margin="115,87,0,0" Click="StopSkeleTrack_Click"/>
        <Button Name="FacialRecStart" Content="Start Facial Rec"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="100" Margin="10,127,0,0"
Click="FacialRecStart_Click"/>
        <Button Name="FacialRecStop" Content="Stop Facial Rec" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="100" Margin="10,154,0,0" Click="FacialRecStop_Click"/>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Globalization;
using System.IO;
using System.Linq;
using System.Windows;
using System.Windows.Forms;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using Emgu.CV;
using Emgu.CV.CvEnum;
using Emgu.CV.Structure;
using Microsoft.Kinect;
using Microsoft.Kinect.Toolkit;

namespace WpfApplication3
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private static readonly int bgr32BytesPerPixel = (PixelFormats.Bgr32.BitsPerPixel
+ 7) / 8;
        /// <summary>
        /// Width of output drawing
        /// </summary>
        private const float RenderWidth = 640.0f;

        /// <summary>
        /// Height of our output drawing
        /// </summary>
        private const float RenderHeight = 480.0f;

        /// <summary>
        /// Thickness of clip edge rectangles
        /// </summary>
        private const double ClipBoundsThickness = 10;

        private readonly KinectSensorChooser sensorChooser = new KinectSensorChooser();
        private bool currentFrameFlag = false;

        /// <summary>
        /// Active Kinect sensor
        /// </summary>
        private KinectSensor sensor;

        /// <summary>
        /// Drawing group for skeleton rendering output
        /// </summary>
        private DrawingGroup drawingGroup;

        /// <summary>
        /// Drawing image that we will display
        /// </summary>
```

```
        private DrawingImage imageSource;

        /// <summary>
        /// Drawing group for skeleton rendering output
        /// </summary>
        private DrawingGroup facialDrawingGroup;

        /// <summary>
        /// Drawing image that we will display
        /// </summary>
        private DrawingImage facialImageSource;

        // MF 01/31/2013
        private Room room;

        RecognizedFace[] facesInPic = new RecognizedFace[6];

        //Declararation of all variables, vectors and haarcascades
        Image<Bgr, Byte> currentFrame;
        HaarCascade face;
        MCvFont font = new MCvFont(FONT.CV_FONT_HERSHEY_TRIPLEX, 0.5d, 0.5d);
        Image<Gray, byte> result = null;
        Image<Gray, byte> gray = null;
        List<Image<Gray, byte>> trainingImages = new List<Image<Gray, byte>>();
        List<string> labels = new List<string>();
        List<string> namePersons = new List<string>();
        int contTrain, numLabels;
        string name = null;

        private Bitmap bmap = new Bitmap(1280, 960,
System.Drawing.Imaging.PixelFormat.Format32bppRgb);

        Image<Bgr, Byte> displayFrame;
        private WriteableBitmap colorImageWritableBitmap;
        private byte[] colorImageData;
        private ColorImageFormat currentColorImageFormat = ColorImageFormat.Undefined;

        EigenObjectRecognizer recognizer;

        public MainWindow()
        {
            this.InitializeComponent();

            this.face = new HaarCascade("haarcascade_frontalface_default.xml");

            this.room = new Room();

            try
            {
                //Load of previous trainned faces and labels for each image
                string labelsinfo =
File.ReadAllText(string.Format("{0}/TrainedFaces/TrainedLabels.txt",
System.Windows.Forms.Application.StartupPath));
                string[] labels = labelsinfo.Split('%');
                this.numLabels = Convert.ToInt16(labels[0]);
                this.contTrain = this.numLabels;
                string loadFaces;
```

```
                for (int tf = 1; tf < this.numLabels + 1; tf++)
                {
                    loadFaces = string.Format("face{0}.bmp", tf);
                    this.trainingImages.Add(new Image<Gray,
byte>(string.Format("{0}/TrainedFaces/{1}", System.Windows.Forms.Application.StartupPath,
loadFaces)));
                    this.labels.Add(labels[tf]);
                }

                //////////////////////////////////////
                //TermCriteria for face recognition with numbers of trained images like
maxIteration
                MCvTermCriteria termCrit = new MCvTermCriteria(this.contTrain, 0.001);

                //Eigen face recognizer
                this.recognizer = new EigenObjectRecognizer(
                    this.trainingImages.ToArray(),
                    this.labels.ToArray(),
                    3000,
                    ref termCrit);
                ///////////////////////////////////////
            }
            catch (Exception)
            {
                //MessageBox.Show(e.ToString());
                System.Windows.Forms.MessageBox.Show("Nothing in binary database, please
add at least a face(Simply train the prototype with the Add Face Button).", "Trained
faces load", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
            }
        }

        /// <summary>
        /// Execute startup tasks
        /// </summary>
        /// <param name="sender">object sending the event</param>
        /// <param name="e">event arguments</param>
        ///
        private void WindowLoaded(object sender, RoutedEventArgs e)
        {
            // Create the drawing group we'll use for drawing
            this.drawingGroup = new DrawingGroup();

            // Create an image source that we can use in our image control
            this.imageSource = new DrawingImage(this.drawingGroup);

            this.TrackerDisplay.Source = this.imageSource;

            // Create the drawing group we'll use for drawing
            this.facialDrawingGroup = new DrawingGroup();

            // Create an image source that we can use in our image control
            this.facialImageSource = new DrawingImage(this.facialDrawingGroup);

            this.ColorImage.Source = this.facialImageSource;

            // Look through all sensors and start the first connected one.
            // This requires that a Kinect is connected at the time of app startup.
            // To make your app robust against plug/unplug,
```

```
            // it is recommended to use KinectSensorChooser provided in
Microsoft.Kinect.Toolkit
            foreach (var potentialSensor in KinectSensor.KinectSensors)
            {
                if (potentialSensor.Status == KinectStatus.Connected)
                {
                    this.sensor = potentialSensor;
                    break;
                }
            }

            if (null != this.sensor)
            {
                // Turn on the skeleton stream to receive skeleton frames
                this.sensor.SkeletonStream.Enable();

this.sensor.ColorStream.Enable(ColorImageFormat.RgbResolution1280x960Fps12);

                // Start the sensor!
                try
                {
                    this.sensor.Start();
                }
                catch (IOException)
                {
                    this.sensor = null;
                }

                this.sensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Default;
            }

            // MF 01/31/2013
            // Initialize our Room
            this.room = new Room();
        }

        /// <summary>
        /// Clears objects and event handlers when the window is closed.
        /// </summary>
        /// <param name="sender">The sender.</param>
        /// <param name="e">The <see cref="System.EventArgs" /> instance containing the
event data.</param>
        private void WindowClosed(object sender, EventArgs e)
        {
            if (null != this.sensor)
            {
                this.sensor.Stop();
            }

            this.sensorChooser.Stop();
        }

        /// <summary>
        /// Handles the Click event of the FullStart control. Starts skeleton tracking
and facial recognition events.
        /// </summary>
        /// <param name="sender">The source of the event.</param>
```

```
        /// <param name="e">The <see cref="System.Windows.RoutedEventArgs" /> instance
containing the event data.</param>
        private void FullStart_Click(object sender, RoutedEventArgs e)
        {
            this.sensor.AllFramesReady += KinectSensorOnAllFramesReady;
        }

        /// <summary>
        /// Kinects the sensor on all frames ready.
        /// </summary>
        /// <param name="sender">The sender.</param>
        /// <param name="allFramesReadyEventArgs">The <see
cref="Microsoft.Kinect.AllFramesReadyEventArgs" /> instance containing the event
data.</param>
        private void KinectSensorOnAllFramesReady(object sender, AllFramesReadyEventArgs
allFramesReadyEventArgs)
        {
            Skeleton[] skeletons = new Skeleton[0];

            //Everytime there is a new skeleton fram this event is fired.
            using (SkeletonFrame skeletonFrame =
allFramesReadyEventArgs.OpenSkeletonFrame())
            {
                if (skeletonFrame != null)
                {
                    //transfer the skeleton data to a global array of skeletons
                    skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];
                    skeletonFrame.CopySkeletonDataTo(skeletons);
                }
            }

            //When the image frame is displayed, the this portion of the code repeats
itself to update
            //the room data and display the new position of tracked Persons
            using (DrawingContext dc = this.drawingGroup.Open())
            {
                // Draw a transparent background to set the render size
                dc.DrawRectangle(System.Windows.Media.Brushes.Black, null, new
System.Windows.Rect(0.0, 0.0, RenderWidth, RenderHeight));

                // Update Room with current skele frame
                bool[] skelesTracked = new bool[skeletons.Length];
                int numTracked = 0;

                for (int i = 0; i < skeletons.Length; i++)
                {
                    if (skeletons[i].TrackingState == SkeletonTrackingState.NotTracked)
                    {
                        skelesTracked[i] = false;
                    }
                    else
                    {
                        skelesTracked[i] = true;
                        numTracked++;
                    }
                }

                Location[] skeleLocs = new Location[numTracked];
```

```
                for (int i = 0; i < skeletons.Length; i++)
                {
                    int j = 0;

                    if (skelesTracked[i])
                    {
                        skeleLocs[j] = new Location(skeletons[i]);
                        j++;
                    }
                }

                this.room.UpdateRoom(skeleLocs);
                //this.DrawRoom(room, dc);
                Console.Write(this.room);

                // prevent drawing outside of our render area
                this.drawingGroup.ClipGeometry = new RectangleGeometry(new
System.Windows.Rect(0.0, 0.0, RenderWidth, RenderHeight));
            }

            using (var colorImageFrame = allFramesReadyEventArgs.OpenColorImageFrame())
            {
                if (colorImageFrame == null)
                {
                    return;
                }

                this.currentFrame =
Utilities.ImageToBitmap(colorImageFrame).ToOpenCVImage<Bgr, Byte>().Resize(320, 240,
Emgu.CV.CvEnum.INTER.CV_INTER_CUBIC);

                this.gray = this.currentFrame.Convert<Gray, Byte>();

                //Face Detector
                MCvAvgComp[][] facesDetected = this.gray.DetectHaarCascade(
                    this.face,
                    1.2,
                    2,
                    Emgu.CV.CvEnum.HAAR_DETECTION_TYPE.DO_CANNY_PRUNING,
                    new System.Drawing.Size(20, 20));

                foreach (MCvAvgComp f in facesDetected[0])
                {
                    this.result = this.currentFrame.Copy(f.rect).Convert<Gray,
byte>().Resize(100, 100, Emgu.CV.CvEnum.INTER.CV_INTER_CUBIC);
                    //draw the face detected in the 0th (gray) channel with blue color
                    this.currentFrame.Draw(f.rect, new Bgr(System.Drawing.Color.Red), 2);

                    int i = 0;
                    if (this.trainingImages.ToArray().Length != 0)
                    {
                        //TermCriteria for face recognition with numbers of trained
images like maxIteration
                        MCvTermCriteria termCrit = new MCvTermCriteria(this.contTrain,
0.001);

                        //Eigen face recognizer
```

```
                        EigenObjectRecognizer recognizer = new EigenObjectRecognizer(
                            this.trainingImages.ToArray(),
                            this.labels.ToArray(),
                            3000,
                            ref termCrit);

                        this.name = recognizer.Recognize(this.result);

                        //Draw the label for each face detected and recognized
                        //this.currentFrame.Draw(this.name,
                        //    ref this.font,
                        //    new System.Drawing.Point(f.rect.X - 2, f.rect.Y - 2),
                        //    new Bgr(System.Drawing.Color.LightGreen));

                        this.facesInPic[i] = new RecognizedFace(this.name, (f.rect.X +
(f.rect.Size.Width / 2)), (f.rect.Y + (f.rect.Size.Height / 2)));

                        i++;
                    }
                }

                foreach (RecognizedFace fa in this.facesInPic)
                {
                    if (fa != null)
                    {
                        this.room.IdentifyPerson(fa);
                    }
                }
                //Show the faces procesed and recognized
                //this.ColorImage.Source = Utilities.ToBitmapSource(this.currentFrame);
            }

            using (var image = this.displayFrame)
            {
                if (image != null)
                {
                    this.ColorImage.Source = Utilities.ToBitmapSource(image);
                }
            }

            using (DrawingContext dc = this.drawingGroup.Open())
            {
                this.DrawRoom(this.room, dc);
            }
        }

        /// <summary>
        /// Handles the Click event of the StartSkeleTrack control.
        /// Adds skeleton frame event handler.
        /// Event is triggered when there is a new skeleton frame detected.
        /// </summary>
        /// <param name="sender">The source of the event.</param>
        /// <param name="e">The <see cref="System.Windows.RoutedEventArgs" /> instance
containing the event data.</param>
        private void StartSkeleTrack_Click(object sender, RoutedEventArgs e)
        {
            this.sensor.AllFramesReady += SkeletonTracking;
        }
```

```
        /// <summary>
        /// Handles the Click event of the StopSkeleTrack control.
        /// Removes skeleton frame event handler
        /// </summary>
        /// <param name="sender">The source of the event.</param>
        /// <param name="e">The <see cref="System.Windows.RoutedEventArgs" /> instance
containing the event data.</param>
        private void StopSkeleTrack_Click(object sender, RoutedEventArgs e)
        {
            this.sensor.AllFramesReady -= SkeletonTracking;
            this.TrackerDisplay.Source = null;
        }

        /// <summary>
        /// Skeletons tracking event handler.
        /// Everytime there is a new skeleton fram this event is fired.
        /// </summary>
        /// <param name="sender">The sender.</param>
        /// <param name="allFramesReadyEventArgs">The <see
cref="Microsoft.Kinect.AllFramesReadyEventArgs" /> instance containing the event
data.</param>
        private void SkeletonTracking(object sender, AllFramesReadyEventArgs
allFramesReadyEventArgs)
        {
            Skeleton[] skeletons = new Skeleton[0];

            using (SkeletonFrame skeletonFrame =
allFramesReadyEventArgs.OpenSkeletonFrame())
            {
                if (skeletonFrame != null)
                {
                    skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];
                    skeletonFrame.CopySkeletonDataTo(skeletons);
                }
            }

            using (DrawingContext dc = this.drawingGroup.Open())
            {
                // MF 01/31/2013
                // Update Room with current skele frame
                bool[] skelesTracked = new bool[skeletons.Length];
                int numTracked = 0;

                for (int i = 0; i < skeletons.Length; i++)
                {
                    if (skeletons[i].TrackingState == SkeletonTrackingState.NotTracked)
                    {
                        skelesTracked[i] = false;
                    }
                    else
                    {
                        skelesTracked[i] = true;
                        numTracked++;
                    }
                }

                Location[] skeleLocs = new Location[numTracked];
```

```
                int j = 0;

                for (int i = 0; i < skeletons.Length; i++)
                {
                    if (skelesTracked[i])
                    {
                        skeleLocs[j] = new Location(skeletons[i]);
                        j++;
                    }
                }

                this.room.UpdateRoom(skeleLocs);
                this.DrawRoom(this.room, dc);
                Console.Write(this.room);

                // prevent drawing outside of our render area
                this.drawingGroup.ClipGeometry = new RectangleGeometry(new
System.Windows.Rect(0.0, 0.0, RenderWidth, RenderHeight));
            }
        }

        /// <summary>
        /// Handles the Click event of the FacialRecStart control.
        /// Adds Color frame event handler.
        /// Event is triggered when there is a new color image frame detected.
        /// </summary>
        /// <param name="sender">The source of the event.</param>
        /// <param name="e">The <see cref="System.Windows.RoutedEventArgs" /> instance
containing the event data.</param>
        private void FacialRecStart_Click(object sender, RoutedEventArgs e)
        {
            this.sensor.AllFramesReady += FacialRecognitionLight;
        }

        /// <summary>
        /// Handles the Click event of the FacialRecStop control.
        /// Removes color image frame handler
        /// </summary>
        /// <param name="sender">The source of the event.</param>
        /// <param name="e">The <see cref="System.Windows.RoutedEventArgs" /> instance
containing the event data.</param>
        private void FacialRecStop_Click(object sender, RoutedEventArgs e)
        {
            this.sensor.AllFramesReady -= FacialRecognitionLight;
        }

        /// <summary>
        /// Facials recognition event handler.
        /// </summary>
        /// <param name="sender">The sender.</param>
        /// <param name="allFramesReadyEventArgs">The <see
cref="Microsoft.Kinect.AllFramesReadyEventArgs" /> instance containing the event
data.</param>
        private void FacialRecognition(object sender, AllFramesReadyEventArgs
allFramesReadyEventArgs)
        {
            using (var colorImageFrame = allFramesReadyEventArgs.OpenColorImageFrame())
            {
```

```
                if (colorImageFrame == null)
                {
                    return;
                }

                // Make a copy of the color frame for displaying.
                if (this.currentColorImageFormat != colorImageFrame.Format)
                {
                    this.currentColorImageFormat = colorImageFrame.Format;

                    this.bmap = new Bitmap(colorImageFrame.Width,
                        colorImageFrame.Height,
                        System.Drawing.Imaging.PixelFormat.Format32bppRgb);
                }

                //resize and reformat frame for processing
                this.currentFrame =
Utilities.ImageToBitmap(colorImageFrame).ToOpenCVImage<Bgr, Byte>().Resize(320, 240,
Emgu.CV.CvEnum.INTER.CV_INTER_CUBIC);
                this.displayFrame = this.currentFrame;

                this.gray = this.currentFrame.Convert<Gray, Byte>();

                //Face Detector
                MCvAvgComp[][] facesDetected = this.gray.DetectHaarCascade(
                    this.face,
                    1.2,
                    2,
                    Emgu.CV.CvEnum.HAAR_DETECTION_TYPE.DO_CANNY_PRUNING,
                    new System.Drawing.Size(20, 20));

                foreach (MCvAvgComp f in facesDetected[0])
                {
                    this.result = this.currentFrame.Copy(f.rect).Convert<Gray,
byte>().Resize(100, 100, Emgu.CV.CvEnum.INTER.CV_INTER_CUBIC);
                    //draw the face detected in the 0th (gray) channel with blue color
                    this.displayFrame.Draw(f.rect, new Bgr(System.Drawing.Color.Red), 2);

                    int i = 0;
                    if (this.trainingImages.ToArray().Length != 0)
                    {
                        //TermCriteria for face recognition with numbers of trained
images like maxIteration
                        MCvTermCriteria termCrit = new MCvTermCriteria(this.contTrain,
0.001);

                        //Eigen face recognizer
                        EigenObjectRecognizer recognizer = new EigenObjectRecognizer(
                            this.trainingImages.ToArray(),
                            this.labels.ToArray(),
                            3000,
                            ref termCrit);

                        this.name = recognizer.Recognize(this.result);

                        //Draw the label for each face detected and recognized
                        this.displayFrame.Draw(this.name,
                            ref this.font,
```

```
                                new System.Drawing.Point(f.rect.X - 2, f.rect.Y - 2),
                                new Bgr(System.Drawing.Color.LightGreen));

                            this.facesInPic[i] = new RecognizedFace(this.name, (f.rect.X +
(f.rect.Size.Width / 2)), (f.rect.Y + (f.rect.Size.Height / 2)));
                            i++;
                        }
                    }

                    //Show the faces procesed and recognized
                    this.ColorImage.Source = Utilities.ToBitmapSource(this.displayFrame);
                }
            }

        // FacialRecognitionLight is an attempt to fix the GUI freezing issue and is
otherwise identical to FacialRecognition
        private void FacialRecognitionLight(object sender, AllFramesReadyEventArgs
allFramesReadyEventArgs)
        {
            using (var colorImageFrame = allFramesReadyEventArgs.OpenColorImageFrame())
            {
                if (colorImageFrame != null)
                {
                    this.currentFrame =
Utilities.ImageToBitmap(colorImageFrame).ToOpenCVImage<Bgr, Byte>().Resize(320, 240,
Emgu.CV.CvEnum.INTER.CV_INTER_CUBIC);
                    this.currentFrameFlag = true;
                }
            }

            using (DrawingContext dc = this.facialDrawingGroup.Open())
            {
                if (this.currentFrameFlag == true)
                {
                    this.gray = this.currentFrame.Convert<Gray, Byte>();

                    //Face Detector
                    MCvAvgComp[][] facesDetected = this.gray.DetectHaarCascade(
                        this.face,
                        1.2,
                        2,
                        Emgu.CV.CvEnum.HAAR_DETECTION_TYPE.DO_CANNY_PRUNING,
                        new System.Drawing.Size(20, 20));

                    foreach (MCvAvgComp f in facesDetected[0])
                    {
                        this.result = this.currentFrame.Copy(f.rect).Convert<Gray,
byte>().Resize(100, 100, Emgu.CV.CvEnum.INTER.CV_INTER_CUBIC);

                        int i = 0;
                        if (this.trainingImages.ToArray().Length != 0)
                        {
                            this.name = this.recognizer.Recognize(this.result);

                            //Draw the label for each face detected and recognized
                            FormattedText occupantName = new FormattedText(this.name,
                                CultureInfo.GetCultureInfo("en-us"),
                                System.Windows.FlowDirection.LeftToRight,
```

```
                                new Typeface("Verdana"),
                                30,
                                System.Windows.Media.Brushes.Blue);

                            dc.DrawText(occupantName, new System.Windows.Point(f.rect.X -
2, f.rect.Y - 2));

                            this.facesInPic[i] = new RecognizedFace(this.name, (f.rect.X
+ (f.rect.Size.Width / 2)), (f.rect.Y + (f.rect.Size.Height / 2)));
                            i++;
                        }
                    }

                    this.currentFrameFlag = false;
                    //Show the faces procesed and recognized
                    //this.ColorImage.Source =
Utilities.ToBitmapSource(this.currentFrame);
                }
            }
        }

        private void DryCamStart_Click(object sender, RoutedEventArgs e)
        {
            this.sensor.AllFramesReady += VgaCapture;
        }
        private void DryCamStop_Click(object sender, RoutedEventArgs e)
        {
            this.sensor.AllFramesReady -= VgaCapture;
            this.ColorImage.Source = null;
        }
        //Simple vga camera capture for testing
        private void VgaCapture(object sender, AllFramesReadyEventArgs
allFramesReadyEventArgs)
        {
            using (var colorImageFrame = allFramesReadyEventArgs.OpenColorImageFrame())
            {
                if (colorImageFrame == null)
                {
                    return;
                }

                // Make a copy of the color frame for displaying.
                var haveNewFormat = this.currentColorImageFormat !=
colorImageFrame.Format;
                if (haveNewFormat)
                {
                    this.currentColorImageFormat = colorImageFrame.Format;
                    this.colorImageData = new byte[colorImageFrame.PixelDataLength];
                    this.colorImageWritableBitmap = new WriteableBitmap(
                        colorImageFrame.Width, colorImageFrame.Height, 96, 96,
PixelFormats.Bgr32, null);
                    this.ColorImage.Source = this.colorImageWritableBitmap;
                }

                colorImageFrame.CopyPixelDataTo(this.colorImageData);
                this.colorImageWritableBitmap.WritePixels(
                    new Int32Rect(0, 0, colorImageFrame.Width, colorImageFrame.Height),
                    this.colorImageData,
```

```
                    colorImageFrame.Width * bgr32BytesPerPixel,
                    0);
                }
            }

            /// <summary>
            /// Draws the room.
            /// Displays tracked Person information of the room for demonstration purposes.
            /// </summary>
            /// <param name="r">The r.</param>
            /// <param name="drawCont">The draw cont.</param>
            private void DrawRoom(Room r, DrawingContext drawCont)
            {
                float[] roomDims = r.FitDimensionsToDisplay(); //loads the dimensions of the
room
                //Draw a box for room walls
                drawCont.DrawRectangle(System.Windows.Media.Brushes.Black, null, new
Rect(0.0, 0.0, (int)roomDims[0], (int)roomDims[1]));
                drawCont.DrawRectangle(System.Windows.Media.Brushes.White, null, new
Rect(2.0, 2.0, (int)roomDims[0] - 4, (int)roomDims[1] - 4));

                //Draw grid for reference points
                for (int i = 1; i < 11; i++)
                {
                    drawCont.DrawLine(new
System.Windows.Media.Pen(System.Windows.Media.Brushes.Black, 1),
                        new System.Windows.Point(0, (i * (int)roomDims[0] / 10)),
                        new System.Windows.Point((int)roomDims[1], (i * (int)roomDims[0] /
10)));
                    drawCont.DrawLine(new
System.Windows.Media.Pen(System.Windows.Media.Brushes.Black, 1),
                        new System.Windows.Point((i * (int)roomDims[1] / 10), 0),
                        new System.Windows.Point((i * (int)roomDims[1] / 10),
(int)roomDims[0]));
                }

                Person[] occupants = r.GetOccupants();//Load person data from room
                bool[] iDs = r.GetIDs(); //load IDs

                for (int i = 0; i < r.GetSize(); i++)  // loop through occupants
                {
                    if (iDs[i]) // if we've got a valid Person
                    {
                        System.Windows.Point point0, point1; // endpoints for current line

                        //Draw history line for each Person
                        for (int j = 0; j < (Person.HistorySize - 1); j++) // loop through
history
                        {
                            try
                            {
                                point0 = occupants[i].GetHistory(j).ToWPoint(roomDims[2]);
                                point1 = occupants[i].GetHistory(++j).ToWPoint(roomDims[2]);
                                drawCont.DrawLine(new
System.Windows.Media.Pen(System.Windows.Media.Brushes.Blue, 4), point0, point1);
                            }
                            catch (Exception)
                            {
```

```
                        break;
                    }
                }

                //Draw the label for each face detected and recognized
                FormattedText occupantName = new
FormattedText(occupants[i].PersonName,
                    CultureInfo.GetCultureInfo("en-us"),
                    System.Windows.FlowDirection.LeftToRight,
                    new Typeface("Verdana"),
                    30,
                    System.Windows.Media.Brushes.Blue);

                //Draw the label for each face detected and recognized
                drawCont.DrawText(occupantName,
occupants[i].GetHistory(0).ToWPoint(roomDims[2]));
            }
        }
    }
}
}
```

Utilities.cs

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Media.Imaging;
using System.Linq;
using System.Runtime.InteropServices;
using System.Windows;
using Emgu.CV;
using Microsoft.Kinect;



namespace WpfApplication3
{

    public static class Utilities // MF 01/31/2013 - Holds various Utility methods
    {
        /// <summary>
        /// Delete a GDI object
        /// </summary>
        /// <param name="o">The poniter to the GDI object to be deleted</param>
        /// <returns></returns>
        [DllImport("gdi32")]
        private static extern bool DeleteObject(IntPtr o);

        //public class SafeHBitmapHandle : SafeHandleZeroOrMinusOneIsInvalid
        //{
        //    [SecurityCritical]
        //    public SafeHBitmapHandle(IntPtr preexistingHandle, bool ownsHandle)
        //        : base(ownsHandle)
        //    {
        //        SetHandle(preexistingHandle);
        //    }

        //    protected override bool ReleaseHandle()
        //    {
        //        return GdiNative.DeleteObject(handle) > 0;
        //    }
        //}

        /// <summary>
        /// Convert an IImage to a WPF BitmapSource. The result can be used in the Set
Property of Image.Source
        /// </summary>
        /// <param name="image">The Emgu CV Image</param>
        /// <returns>The equivalent BitmapSource</returns>
        public static BitmapSource ToBitmapSource(IImage image)
        {
            using (System.Drawing.Bitmap source = image.Bitmap)
            {
                IntPtr ptr = source.GetHbitmap(); //obtain the Hbitmap

                BitmapSource bs =
System.Windows.Interop.Imaging.CreateBitmapSourceFromHBitmap(
                    ptr,
                    IntPtr.Zero,
```

```
                    Int32Rect.Empty,
                    System.Windows.Media.Imaging.BitmapSizeOptions.FromEmptyOptions());

                DeleteObject(ptr); //release the HBitmap
                return bs;
            }
        }

        /// <summary>
        /// Converts IImage to a WritableBitmap.
        /// </summary>
        /// <param name="image">The image.</param>
        /// <returns>WriteableBitmap</returns>
        public static WriteableBitmap ToWritableBitmap(IImage image)
        {
            using (System.Drawing.Bitmap source = image.Bitmap)
            {
                IntPtr ptr = source.GetHbitmap(); //obtain the Hbitmap

                BitmapSource bs =
System.Windows.Interop.Imaging.CreateBitmapSourceFromHBitmap(
                    ptr,
                    IntPtr.Zero,
                    Int32Rect.Empty,
                    System.Windows.Media.Imaging.BitmapSizeOptions.FromEmptyOptions());

                DeleteObject(ptr); //release the HBitmap

                WriteableBitmap wb = new
System.Windows.Media.Imaging.WriteableBitmap(bs);
                return wb;
            }
        }

        /// <summary>
        /// Converts a bitmap image to an open CV IImage.
        /// </summary>
        /// <param name="bitmap">The bitmap.</param>
        /// <returns></returns>
        public static Image<TColor, TDepth> ToOpenCVImage<TColor, TDepth>(this Bitmap
bitmap)
            where TColor : struct, IColor
            where TDepth : new()
        {
            return new Image<TColor, TDepth>(bitmap);
        }

        /// <summary>
        /// ColorImageFrame to bitmap.
        /// </summary>
        /// <param name="image">The image.</param>
        /// <returns></returns>
        public static Bitmap ImageToBitmap(ColorImageFrame image)
        {
            byte[] pixeldata = new byte[image.PixelDataLength];
            image.CopyPixelDataTo(pixeldata);
            Bitmap bmap = new Bitmap(image.Width, image.Height,
System.Drawing.Imaging.PixelFormat.Format32bppRgb);
```

```
            BitmapData bmapdata = bmap.LockBits(
                new Rectangle(0, 0, image.Width, image.Height),
                ImageLockMode.WriteOnly,
                bmap.PixelFormat);
            IntPtr ptr = bmapdata.Scan0;
            Marshal.Copy(pixeldata, 0, ptr, image.PixelDataLength);
            bmap.UnlockBits(bmapdata);
            return bmap;
        }
    }
}
```

Room.cs

```
using System;
using System.Drawing;
using Emgu.CV.Structure;

namespace WpfApplication3
{
    public class Room // MF 01/31/2013
    {
        static readonly Location minBound = new Location(-2, 0);
        static readonly Location maxBound = new Location(2, 4);
        static readonly Location[] defaultBounds = { minBound, maxBound };// should be
approximate detection area of Kinect

        /// <summary>
        /// Maximum amount of error for track association
        /// </summary>
        const float TrackErr = 0.07F;

        /// <summary>
        /// Maximum number of frames a Person be untracked before being removed
        /// </summary>
        readonly int maxMissed;

        /// <summary>
        /// number of occupants the Room can hold
        /// </summary>
        readonly int size;
        /// <summary>
        /// Local coordinates of room corners (assuming rectangular room)
        /// </summary>
        readonly Location[] boundaries;
        /// <summary>
        /// Persons currently in the Room
        /// </summary>
        readonly Person[] occupants;

        /// <summary>
        /// Initializes a new instance of the <see cref="Room" /> class.
        /// </summary>
        /// <param name="bounds">The bounds.</param>
        /// <param name="size">The size.</param>
        /// <param name="maxMissed">The max missed.</param>
        public Room(Location[] bounds, int size = 6, int maxMissed = 15)
        {
            this.boundaries = bounds;
            this.size = size;
            this.maxMissed = maxMissed;
            this.occupants = new Person[size];
            this.IDs = new bool[size];
            for (int i = 0; i < size; i++)
            {
                this.IDs[i] = false;
            }
        }
```

```
/// <summary>
/// Initializes a new instance of the <see cref="Room" /> class.
/// </summary>
/// <param name="size">The size.</param>
public Room(int size = 6)
{
    this.boundaries = defaultBounds;
    this.size = size;
    this.occupants = new Person[size];
    this.IDs = new bool[size];
    for (int i = 0; i < size; i++)
    {
        this.IDs[i] = false;
    }
}

/// <summary>
/// Gets or sets the IDs.
/// </summary>
/// <value>The IDs.</value>
public bool[] IDs { get; set; }

/// <summary>
/// Fits the dimensions of the room to display.
/// </summary>
/// <returnsa>Array of floats containing the x dimension, y dimension of the room
///             and ratio of real size of room to dimension of display</returns>
public float[] FitDimensionsToDisplay()
{
    float ratio = 1;
    float xDim;
    float yDim;

    float x = this.boundaries[1].X - this.boundaries[0].X;
    float y = this.boundaries[1].Y - this.boundaries[0].Y;

    if (((x * ((float)2)) >= (y * ((float)3))) && (x > ((float)640)))
    {
        ratio = (((float)640) / x);
    }
    else if (((x * ((float)2)) < (y * ((float)3))) && (y > (float)480))
    {
        ratio = (((float)480) / y);
    }
    else if ((x * ((float)2)) >= (y * ((float)3)))
    {
        ratio = (((float)640) / x);
    }
    else
    {
        ratio = (((float)480) / y);
    }

    xDim = ratio * x;
    yDim = ratio * y;

    float[] dimensions = new float[3] { xDim, yDim, ratio };
```

```
        return dimensions;
    }

    /// <summary>
    /// Gets the size of the room.
    /// </summary>
    /// <returns></returns>
    public int GetSize()
    {
        return this.size;
    }

    /// <summary>
    /// Determines whether the given Location is in the room
    /// </summary>
    /// <param name="loc">Location to be analyzed</param>
    /// <returns>Whether loc is within this Room</returns>
    public bool Contains(Location loc)
    {
        if (loc.X < this.boundaries[0].X)
        {
            if (loc.X < this.boundaries[1].X)
            {
                return false;
            }
        }
        if (loc.X > this.boundaries[0].X)
        {
            if (loc.X > this.boundaries[1].X)
            {
                return false;
            }
        }
        if (loc.Y < this.boundaries[0].Y)
        {
            if (loc.Y < this.boundaries[1].Y)
            {
                return false;
            }
        }
        if (loc.Y > this.boundaries[0].Y)
        {
            if (loc.Y > this.boundaries[1].Y)
            {
                return false;
            }
        }

        return true;
    }

    /// <summary>
    /// Gets the occupants.
    /// </summary>
    /// <returns></returns>
    public Person[] GetOccupants()
    {
        return this.occupants;
```

```
        }

        /// <summary>
        /// Gets the I ds.
        /// </summary>
        /// <returns></returns>
        public bool[] GetIDs()
        {
            return this.IDs;
        }

        /// <summary>
        /// Add a Person to the Room and assign it an ID
        /// </summary>
        /// <param name="p">Person to add</param>
        /// <returns>the ID of the Person</returns>
        public int AddPerson(Person p)
        {
            int i = 0;
            while (i < this.size && this.IDs[i])
            {
                i++; // looks for an unassigned ID
            }

            if (i < this.size)
            {
                this.occupants[i] = p;
                this.IDs[i] = true;
                Console.WriteLine(string.Format("New Person {0} found at {1}", i,
p.GetLoc()));
                return i;
            }
            else
            {
                return -1; // should put some error handling here instead
            }
        }

        /// <summary>
        /// removes the Person from the Room by freeing its local ID#
        /// </summary>
        /// <param name="p">Person to remove</param>
        public void RemovePerson(Person p)
        {
            this.IDs[p.ID] = false;
            Console.WriteLine(string.Format("Person {0} removed", p.ID));
        }

        /// <summary>
        /// Moves the person to another room.
        /// </summary>
        /// <param name="p">The Person.</param>
        /// <param name="r">The new Room.</param>
        public void MovePerson(Person p, Room r)
        {
            this.RemovePerson(p);
            p.ID = r.AddPerson(p);
        }
```

```
        /// <summary>
        /// Updates the Location of all Persons in the Room
        /// </summary>
        /// <param name="skeles">Locations of all Skeletons detected in the Room</param>
        public void UpdateRoom(Location[] skeles)
        {
            // tracks whether each Skeleton has been associated with a Person
            bool[] skeleTracked = new bool[skeles.Length];

            if (skeles.Length != 0)
            {
                for (int i = 0; i < skeles.Length; i++)
                {
                    skeleTracked[i] = false;
                }

                // tracks wheter each Skeleton is close enough to be a candidate for each
Person

                bool[,] closeEnough = new bool[this.size, skeles.Length];
                for (int i = 0; i < this.size; i++)
                {
                    for (int j = 0; j < skeles.Length; j++)
                    {
                        closeEnough[i, j] = false;
                    }
                }

                // tracks anticipated Location for each Person
                Location[] newLocs = new Location[this.size];

                // anticipate Location for all tracked Persons
                for (int i = 0; i < this.size; i++)
                {
                    if (this.IDs[i])
                    {
                        newLocs[i] = this.occupants[i].GuessLoc();
                    }
                }

                // determine which Skeletons fall within tracking radius for each person
                for (int i = 0; i < this.size; i++)
                {
                    if (this.IDs[i])
                    {
                        for (int j = 0; j < skeles.Length; j++)
                        {
                            try
                            {
                                if (newLocs[i].Distance(skeles[j]) <= TrackErr)
                                {
                                    closeEnough[i, j] = true;
                                }
                            }
                            catch (Exception)
                            {
                                break;
                            }
```

```
                }
            }
        }

        // attach each Skeleton to the closest Person that wants it
        for (int j = 0; j < skeles.Length; j++)
        {
            float minDist = 1000000;
            int index = -1;

            for (int i = 0; i < this.size; i++)
            {
                if (this.IDs[i] && !this.occupants[i].IsUpToDate() &&
skeles[j].Distance(newLocs[i]) < minDist)
                {
                    index = i;
                    minDist = skeles[j].Distance(newLocs[i]);
                }
            }

            if (index >= 0)
            {
                if (skeles.Length >= index)
                {
                    this.occupants[index].UpdateLoc1(skeles[j]);
                    skeleTracked[j] = true;
                }
            }
        }

        // allocate new Persons for each untracked Skeleton
        for (int j = 0; j < skeles.Length; j++)
        {
            if (!skeleTracked[j])
            {
                new Person(skeles[j], this);
            }
        }

        // update remaining Persons and remove any "lost" ones
        for (int i = 0; i < this.size; i++)
        {
            if (this.IDs[i] && !this.occupants[i].IsUpToDate())
            {
                if (this.occupants[i].UpdateLoc() > this.maxMissed)
                {
                    this.RemovePerson(this.occupants[i]);
                }
            }
        }
    }
}

/// <summary>
/// Returns a string that represents the current object.
/// </summary>
/// <returns>A string that represents the current object.</returns>
public override string ToString()
```

```
        {
            try
            {
                string r = "";
                for (int i = 0; i < this.size; i++)
                {
                    if (this.IDs[i])
                    {
                        r += string.Format("{0}\n", this.occupants[i].ToString());
                    }
                }

                //if (r == "") r = "Room is empty";

                return r;
            }
            catch (Exception)
            {
                return "";
            }
        }

        /// <summary>
        /// Identifies a person.
        /// Attaches the recognized tag of a person to the
        /// </summary>
        /// <param name="face">The face recognized.</param>
        public void IdentifyPerson(RecognizedFace face)
        {
            double faceAngle = 500;

            if (face != null)
            {
                faceAngle = (face.XPosition - 640) / 57.0; //calculates relative angle of
detected face to the kinect orientation

                foreach (Person p in this.occupants)
                {
                    if (p != null)
                    {
                        if (p.AngleFromKinect() >= (faceAngle - 1.5) &&
                            p.AngleFromKinect() <= (faceAngle + 1.5) &&
                            p.PersonName != "")
                        {
                            p.PersonName = face.Name;
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

Location.cs

```
using System;
using System.Drawing;
using Microsoft.Kinect;

namespace WpfApplication3
{
    public class Location // MF 01/31/2013 - 2D Location used for tracking
    {
        public float X;//Displacement of Person left to right
        public float Y;//Displacement of Person in depth

        /// <summary>
        /// Initializes a new instance of the <see cref="Location" /> class.
        /// </summary>
        /// <param name="x">The x.</param>
        /// <param name="y">The y.</param>
        public Location(float x, float y)
        {
            this.X = x;
            this.Y = y;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="Location" /> class.
        /// </summary>
        /// <param name="s">Skeleton.</param>
        public Location(Skeleton s)
        {
            this.X = s.Position.X;
            this.Y = s.Position.Z;
        }

        /// <summary>
        /// Finds the Cartesian distance between two Locations
        /// </summary>
        /// <param name="loc">Location to find distance to</param>
        /// <returns>Cartesian distance between this and loc</returns>
        public float Distance(Location loc)
        {
            try
            {
                return (float)System.Math.Sqrt((this.X - loc.X) * (this.X - loc.X) +
(this.Y - loc.Y) * (this.Y - loc.Y));
            }
            catch (Exception)
            {
                return 0;
            }
        }

        /// <summary>
        /// Translates parameters of Location to System.Drawing.Point
        /// </summary>
        /// <param name="ratio">The ratio of the room dimensions to pixels.</param>
        /// <returns></returns>
        public Point ToPoint(float ratio)
```

```
        {
            float xPos = ((float)320) - ( ((float)3) * (this.X * ratio));
            float yPos = ((float)480) - (this.Y * ratio);
            return new Point((int)(xPos), (int)(yPos));
        }

        /// <summary>
        /// Translates parameters of Location to System.Windows.Point
        /// </summary>
        /// <param name="ratio">The ratio of the room dimensions to pixels.</param>
        /// <returns></returns>
        public System.Windows.Point ToWPoint(float ratio)
        {
            float xPos = ((float)240) - (this.X * ratio);
            float yPos = ((float)480) - (this.Y * ratio);
            return new System.Windows.Point((int)(xPos), (int)(yPos));
        }

        /// <summary>
        /// Returns a string that represents the current object.
        /// </summary>
        /// <returns>A string that represents the current object.</returns>
        public override string ToString()
        {
            try
            {
                return string.Format("({0}, {1})", this.X.ToString(), this.Y.ToString());
            }
            catch (Exception)
            {
                return "";
            }
        }
    }
}
```

Person.cs

```
using System;

namespace WpfApplication3
{
    public class Person // MF 01/31/2013 - Person object used for tracking
    {
        /// <summary>
        /// This Person's identity, updated by facial recognition software
        /// </summary>
        public int ID;

        public static readonly int HistorySize = 15;// number of frames to store in the
history

        /// <summary>
        /// List of Locations this Person has been previously.  Includes current Location
        /// </summary>
        readonly Location[] history = new Location[HistorySize];

        /// <summary>
        /// history[current] is this Person's current Location
        /// </summary>
        int current = 0;

        /// <summary>
        /// counts the number of frames since this Person's last been tracked
        /// </summary>
        int missedFrames = 0;

        /// <summary>
        /// Whether this Person's current Location has been updated this frame
        /// </summary>
        bool upToDate = false;

        public Person(Location loc, Room r)
        {
            for (int i = 0; i < HistorySize; i++)
            {
                this.history[i] = loc; // ensures that attempts to recall positions prior
to detection return earliest known position
            }

            this.ID = r.AddPerson(this);

            this.PersonName = "Unknown";

            if (this.ID >= 0)
            {
                this.upToDate = true;
            }
        }

        public string PersonName { get; set; }

        /// <summary>
        /// Gets the current Location
```

```
        /// </summary>
        /// <returns>the current Location</returns>
        public Location GetLoc()
        {
            return this.history[this.current];
        }

        /// <summary>
        /// Gets a past Location occupied by this Person
        /// </summary>
        /// <param name="i">time steps to go back (i=0 returns current location)</param>
        /// <returns>Person's Location i frames ago, or earliest know Location if i is
outside history buffer</returns>
        public Location GetHistory(int i)
        {
            if (i > HistorySize)
            {
                i = HistorySize; // prevents attempted access to frames outside the
history buffer
            }

            int frame = this.current - i;

            if (frame < 0)
            {
                frame += HistorySize; // wrap around if we're out of the array
            }

            return this.history[frame];
        }

        public bool IsUpToDate()
        {
            return this.upToDate;
        }

        /// <summary>
        /// Guesses current Location based on history
        /// </summary>
        /// <returns>Guessed Location</returns>
        public Location GuessLoc()
        {
            try
            {
                float[] xDis = new float[HistorySize - 1];
                float[] yDis = new float[HistorySize - 1];
                float[] xAccel = new float[HistorySize - 2];
                float xAccelAvg = 0;
                float[] yAccel = new float[HistorySize - 2];
                float yAccelAvg = 0;

                int next = (this.current + 2) % HistorySize;
                int last;

                for (int i = 0; i < HistorySize - 1; i++)
                {
                    last = next - 1;
                    if (last < 0)
```

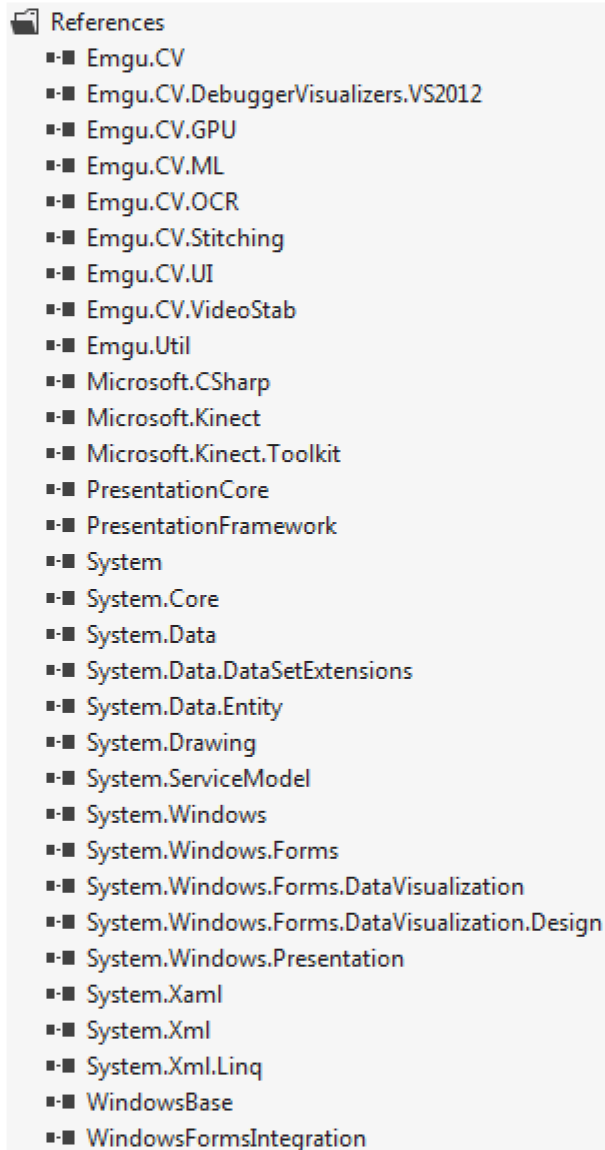
```
                    {
                        last = HistorySize - 1;
                    }

                    xDis[i] = this.history[next].X - this.history[last].X;
                    yDis[i] = this.history[next].Y - this.history[last].Y;

                    next++;
                    next %= HistorySize;
                }

                for (int i = 0; i < HistorySize - 2; i++)
                {
                    xAccel[i] = xDis[i + 1] - xDis[i];
                    xAccelAvg += xAccel[i];

                    yAccel[i] = yDis[i + 1] - yDis[i];
                    yAccelAvg += yAccel[i];
                }

                xAccelAvg /= HistorySize - 2;
                yAccelAvg /= HistorySize - 2;

                this.upToDate = false;

                float newX = this.history[this.current].X + xDis[xDis.Length - 1] +
xAccelAvg;
                float newY = this.history[this.current].Y + yDis[yDis.Length - 1] +
yAccelAvg;

                return new Location(newX, newY);
            }
            catch (Exception)
            {
                return this.history[this.current];
            }
        }

        /// <summary>
        /// Updates current Location
        /// </summary>
        /// <param name="loc">New current Location</param>
        public void UpdateLoc1(Location loc)
        {
            this.current++;
            if (this.current >= HistorySize)
            {
                this.current = 0;
            }
            this.history[this.current] = loc;
            this.upToDate = true;
        }

        public int UpdateLoc()
        {
            this.current++;
            if (this.current >= HistorySize)
            {
```

```
                this.current = 0;
            }
            this.history[this.current] = this.GuessLoc();
            this.upToDate = true;

            this.missedFrames++;
            return this.missedFrames;
        }

        public override string ToString()
        {
            try
            {
                return string.Format("{0}: {1}", this.ID.ToString(),
this.history[this.current].ToString());
            }
            catch (Exception)
            {
                return "";
            }
        }

        /// <summary>
        /// Angles from kinect.
        /// </summary>
        /// <returns> the relative angle angle of the person from the orientation of the
Kinect</returns>
        public double AngleFromKinect()
        {
            return Math.Atan(this.history[this.current].X /
this.history[this.current].Y);
        }
    }
}
```

RecognizedFace.cs

```
using System;
using System.Linq;

namespace WpfApplication3
{
    public class RecognizedFace
    {
        /// <summary>
        /// Initializes a new instance of the <see cref="RecognizedFace" /> class.
        /// </summary>
        /// <param name="n">The name.</param>
        /// <param name="x">The x position.</param>
        /// <param name="y">The y position.</param>
        public RecognizedFace(string n, int x, int y)
        {
            this.Name = n;
            this.XPosition = x;
            this.YPosition = y;
        }

        /// <summary>
        /// Initializes a new instance of the <see cref="RecognizedFace" /> class.
        /// </summary>
        public RecognizedFace()
        {
            this.Name = "";
            this.XPosition = 0;
            this.YPosition = 0;
        }

        public string Name { get; set; }

        public int YPosition { get; set; }

        public int XPosition { get; set; }
    }
}
```

References included in the Project



OpenCV libraries imported

- cublas64_50_35.dll
- cudart64_50_35.dll
- cufft64_50_35.dll
- cvextern.dll

- npp64_50_35.dll
- opencv_calib3d249.dll
- opencv_contrib249.dll
- opencv_core249.dll
- opencv_features2d249.dll
- opencv_ffmpeg249_64.dll
- opencv_flann249.dll
- opencv_gpu249.dll
- opencv_highgui249.dll
- opencv_imgproc249.dll
- opencv_legacy249.dll
- opencv_ml249.dll
- opencv_nonfree249.dll
- opencv_objdetect249.dll
- opencv_photo249.dll
- opencv_stitching249.dll
- opencv_video249.dll
- opencv_videostab249.dll

EigenObjectRecognizer.cs (imported from EgmuCV Library)

```
using System;
using System.Diagnostics;
using Emgu.CV.Structure;

namespace Emgu.CV
{
   /// <summary>
   /// An object recognizer using PCA (Principle Components Analysis)
   /// </summary>
   [Serializable]
   public class EigenObjectRecognizer
   {
      private Image<Gray, Single>[] _eigenImages;
      private Image<Gray, Single> _avgImage;
      private Matrix<float>[] _eigenValues;
      private string[] _labels;
      private double _eigenDistanceThreshold;

      /// <summary>
      /// Get the eigen vectors that form the eigen space
      /// </summary>
      /// <remarks>The set method is primary used for deserialization, do not attemps to
set it unless you know what you are doing</remarks>
      public Image<Gray, Single>[] EigenImages
      {
         get { return _eigenImages; }
         set { _eigenImages = value; }
      }

      /// <summary>
      /// Get or set the labels for the corresponding training image
      /// </summary>
      public String[] Labels
      {
         get { return _labels; }
         set { _labels = value; }
      }

      /// <summary>
      /// Get or set the eigen distance threshold.
      /// The smaller the number, the more likely an examined image will be treated as
unrecognized object.
      /// Set it to a huge number (e.g. 5000) and the recognizer will always treated the
examined image as one of the known object.
      /// </summary>
      public double EigenDistanceThreshold
      {
         get { return _eigenDistanceThreshold; }
         set { _eigenDistanceThreshold = value; }
      }

      /// <summary>
      /// Get the average Image.
      /// </summary>
      /// <remarks>The set method is primary used for deserialization, do not attemps to
set it unless you know what you are doing</remarks>
```

```
      public Image<Gray, Single> AverageImage
      {
         get { return _avgImage; }
         set { _avgImage = value; }
      }

      /// <summary>
      /// Get the eigen values of each of the training image
      /// </summary>
      /// <remarks>The set method is primary used for deserialization, do not attemps to
set it unless you know what you are doing</remarks>
      public Matrix<float>[] EigenValues
      {
         get { return _eigenValues; }
         set { _eigenValues = value; }
      }

      private EigenObjectRecognizer()
      {
      }


      /// <summary>
      /// Create an object recognizer using the specific tranning data and parameters, it
will always return the most similar object
      /// </summary>
      /// <param name="images">The images used for training, each of them should be the
same size. It's recommended the images are histogram normalized</param>
      /// <param name="termCrit">The criteria for recognizer training</param>
      public EigenObjectRecognizer(Image<Gray, Byte>[] images, ref MCvTermCriteria
termCrit)
         : this(images, GenerateLabels(images.Length), ref termCrit)
      {
      }

      private static String[] GenerateLabels(int size)
      {
         String[] labels = new string[size];
         for (int i = 0; i < size; i++)
            labels[i] = i.ToString();
         return labels;
      }

      /// <summary>
      /// Create an object recognizer using the specific tranning data and parameters, it
will always return the most similar object
      /// </summary>
      /// <param name="images">The images used for training, each of them should be the
same size. It's recommended the images are histogram normalized</param>
      /// <param name="labels">The labels corresponding to the images</param>
      /// <param name="termCrit">The criteria for recognizer training</param>
      public EigenObjectRecognizer(Image<Gray, Byte>[] images, String[] labels, ref
MCvTermCriteria termCrit)
         : this(images, labels, 0, ref termCrit)
      {
      }

      /// <summary>
```

```
      /// Create an object recognizer using the specific tranning data and parameters
      /// </summary>
      /// <param name="images">The images used for training, each of them should be the 
same size. It's recommended the images are histogram normalized</param>
      /// <param name="labels">The labels corresponding to the images</param>
      /// <param name="eigenDistanceThreshold">
      /// The eigen distance threshold, (0, ~1000].
      /// The smaller the number, the more likely an examined image will be treated as 
unrecognized object. 
      /// If the threshold is &lt; 0, the recognizer will always treated the examined 
image as one of the known object. 
      /// </param>
      /// <param name="termCrit">The criteria for recognizer training</param>
      public EigenObjectRecognizer(Image<Gray, Byte>[] images, String[] labels, double 
eigenDistanceThreshold, ref MCvTermCriteria termCrit)
      {
         Debug.Assert(images.Length == labels.Length, "The number of images should equals 
the number of labels");
         Debug.Assert(eigenDistanceThreshold >= 0.0, "Eigen-distance threshold should 
always >= 0.0");

         CalcEigenObjects(images, ref termCrit, out _eigenImages, out _avgImage);

         /*
         _avgImage.SerializationCompressionRatio = 9;

         foreach (Image<Gray, Single> img in _eigenImages)
             //Set the compression ration to best compression. The serialized object can 
therefore save spaces
             img.SerializationCompressionRatio = 9;
         */

         _eigenValues = Array.ConvertAll<Image<Gray, Byte>, Matrix<float>>(images,
             delegate(Image<Gray, Byte> img)
             {
                return new Matrix<float>(EigenDecomposite(img, _eigenImages, _avgImage));
             });

         _labels = labels;

         _eigenDistanceThreshold = eigenDistanceThreshold;
      }

      #region static methods
      /// <summary>
      /// Caculate the eigen images for the specific traning image
      /// </summary>
      /// <param name="trainingImages">The images used for training </param>
      /// <param name="termCrit">The criteria for tranning</param>
      /// <param name="eigenImages">The resulting eigen images</param>
      /// <param name="avg">The resulting average image</param>
      public static void CalcEigenObjects(Image<Gray, Byte>[] trainingImages, ref 
MCvTermCriteria termCrit, out Image<Gray, Single>[] eigenImages, out Image<Gray, Single> 
avg)
      {
         int width = trainingImages[0].Width;
         int height = trainingImages[0].Height;
```

```
         IntPtr[] inObjs = Array.ConvertAll<Image<Gray, Byte>, IntPtr>(trainingImages,
delegate(Image<Gray, Byte> img) { return img.Ptr; });

         if (termCrit.max_iter <= 0 || termCrit.max_iter > trainingImages.Length)
            termCrit.max_iter = trainingImages.Length;

         int maxEigenObjs = termCrit.max_iter;

         #region initialize eigen images
         eigenImages = new Image<Gray, float>[maxEigenObjs];
         for (int i = 0; i < eigenImages.Length; i++)
            eigenImages[i] = new Image<Gray, float>(width, height);
         IntPtr[] eigObjs = Array.ConvertAll<Image<Gray, Single>, IntPtr>(eigenImages,
delegate(Image<Gray, Single> img) { return img.Ptr; });
         #endregion

         avg = new Image<Gray, Single>(width, height);

         CvInvoke.cvCalcEigenObjects(
             inObjs,
             ref termCrit,
             eigObjs,
             null,
             avg.Ptr);
      }

      /// <summary>
      /// Decompose the image as eigen values, using the specific eigen vectors
      /// </summary>
      /// <param name="src">The image to be decomposed</param>
      /// <param name="eigenImages">The eigen images</param>
      /// <param name="avg">The average images</param>
      /// <returns>Eigen values of the decomposed image</returns>
      public static float[] EigenDecomposite(Image<Gray, Byte> src, Image<Gray, Single>[]
eigenImages, Image<Gray, Single> avg)
      {
         return CvInvoke.cvEigenDecomposite(
             src.Ptr,
             Array.ConvertAll<Image<Gray, Single>, IntPtr>(eigenImages,
delegate(Image<Gray, Single> img) { return img.Ptr; }),
             avg.Ptr);
      }
      #endregion

      /// <summary>
      /// Given the eigen value, reconstruct the projected image
      /// </summary>
      /// <param name="eigenValue">The eigen values</param>
      /// <returns>The projected image</returns>
      public Image<Gray, Byte> EigenProjection(float[] eigenValue)
      {
         Image<Gray, Byte> res = new Image<Gray, byte>(_avgImage.Width,
_avgImage.Height);
         CvInvoke.cvEigenProjection(
             Array.ConvertAll<Image<Gray, Single>, IntPtr>(_eigenImages,
delegate(Image<Gray, Single> img) { return img.Ptr; }),
             eigenValue,
             _avgImage.Ptr,
```

```
            res.Ptr);
         return res;
      }

      /// <summary>
      /// Get the Euclidean eigen-distance between <paramref name="image"/> and every
other image in the database
      /// </summary>
      /// <param name="image">The image to be compared from the training images</param>
      /// <returns>An array of eigen distance from every image in the training
images</returns>
      public float[] GetEigenDistances(Image<Gray, Byte> image)
      {
         using (Matrix<float> eigenValue = new Matrix<float>(EigenDecomposite(image,
_eigenImages, _avgImage)))
            return Array.ConvertAll<Matrix<float>, float>(_eigenValues,
                delegate(Matrix<float> eigenValueI)
                {
                   return (float)CvInvoke.cvNorm(eigenValue.Ptr, eigenValueI.Ptr,
Emgu.CV.CvEnum.NORM_TYPE.CV_L2, IntPtr.Zero);
                });
      }

      /// <summary>
      /// Given the <paramref name="image"/> to be examined, find in the database the
most similar object, return the index and the eigen distance
      /// </summary>
      /// <param name="image">The image to be searched from the database</param>
      /// <param name="index">The index of the most similar object</param>
      /// <param name="eigenDistance">The eigen distance of the most similar
object</param>
      /// <param name="label">The label of the specific image</param>
      public void FindMostSimilarObject(Image<Gray, Byte> image, out int index, out float
eigenDistance, out String label)
      {
         float[] dist = GetEigenDistances(image);

         index = 0;
         eigenDistance = dist[0];
         for (int i = 1; i < dist.Length; i++)
         {
            if (dist[i] < eigenDistance)
            {
               index = i;
               eigenDistance = dist[i];
            }
         }
         label = Labels[index];
      }

      /// <summary>
      /// Try to recognize the image and return its label
      /// </summary>
      /// <param name="image">The image to be recognized</param>
      /// <returns>
      /// String.Empty, if not recognized;
      /// Label of the corresponding image, otherwise
      /// </returns>
```

```
        public String Recognize(Image<Gray, Byte> image)
        {
            int index;
            float eigenDistance;
            String label;
            FindMostSimilarObject(image, out index, out eigenDistance, out label);

            return (_eigenDistanceThreshold <= 0 || eigenDistance < _eigenDistanceThreshold
) ? _labels[index] : String.Empty;
        }
    }
}
```