Πανεπιστήμιο Θεσσαλίας
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

NodKit: A framework for implementing additional gestures on the iOS platform, utilizing head recognition and tracking technology

[Θεοδωρίδης Θ. Αθανάσιος]
attheodo@inf.uth.gr

Επιβλέποντες Καθηγητές
Αλκιβιάδης Ακρίτας
Δασκαλοπούλου Ασπασία

Βόλος, Ιούνιος 2013

In memory of my beloved father and irreplaceable mentor
1938-2013

Table Of Contents

Acknowledgments

I wish to express my deepest gratitude to my father, Theodoros Theodoridis, for his enormous overall support and acceptance he showed throughout his living years towards my passion for programming and engineering. The unfortunate health condition that led to his extensive kinetic problems and inability to speak, laid the foundations of the research that helped implement NodKit, as part of a broader Augmentative and Alternative Communication software aid. He'll be missed and remembered.

My grateful thanks are also extended to my mother, Theodora, for the invaluable support and care she showed towards her husband, and my father, during the last and difficult times of his time. Her solid personal effort towards his wellbeing-ness inarguably helped me shape the mental stability required to proceed with implementing NodKit and writing this very thesis.

Moreover, I would like to sincerely thank my thesis supervisor, professor Alkiviadis Akritas, for his enthusiastic encouragement towards the iOS platform and Apple technologies as well as for his overall critique and guidance during my work on this and other projects.

Last but not least, I would like to extend my heartfelt thanks to my girlfriend and close friends for the good and bad moments we lived together that still serve as a great pool of valuable life experiences and inspiration for work and personal advancement.

ATHANASIOS THEODORIDIS
VOLOS, 2013

Abstract

NodKit is a re-usable and extensible framework, in the form of a compiled static library, targeted for the iOS platform, that provides third party developers with the option of utilizing the front facing camera of devices and head-tracking technology, as an additional source of user interaction input in their applications.

It is based on advanced middle and low-level native technologies of the iOS platform such as the `AVFoundation`, an advanced Cocoa framework for manipulating video frames in realtime, `Grand Central Dispatch` (GCD), an Apple implementation of task parallelism based on the thread pool pattern, `Blocks`, a C-level syntactic and runtime feature for passing code segments to functions as if they were values, and finally the `Core Image` library for processing raw video frames and looking for face related features in them.

In this thesis we will describe how the aforementioned framework components can come together in an efficient and scalable way to eventually form a high-level API for receiving and acting on events generated by the movement and direction of the user's head. Finally, we will assess how NodKit can be used to create new input actions by combining touch gestures and head gestures - or nods - and how these can be utilized in various ways to create new software solutions and aids for a plethora of industries.

Keywords: head tracking, iOS, iPhone, iPad, OSX, face detection, accessibility, framework, objective-c, head gestures

Chapter 1
Introduction

1.1 Prologue and Goals

The way people interact with tools and machines has always been an interesting topic of extensive research for professionals like visual designers, product engineers and user interface experts. From punch cards, levers, buttons and switches to more sophisticated mediums like the keyboard, mice, trackpads, touch surfaces and most recently voice recognition mechanisms, user-machine interaction and the accompanying sciences have gone through "hell and high water" in order to facilitate the way people work with machines.

Throughout the past decade, the technological evolution that allowed the engineering of smaller and computationally more capable devices has triggered a transition from the traditional desktop workstation to more portable personal computers like tablets and smartphones. In an effort to increase the portability of these devices, the various input peripherals like the mouse and the keyboard have nowadays been completely abandoned in favor of the touchscreen.

The same reasons that assisted touch-aware technologies to progress so much in the last years, also contributed to a more rapid evolution of voice recognition and speech as a medium to provide input to computing devices. It is certainly safe to assume that people nowadays prefer to interact with machines in more human-centric ways, mimicking to the feasible extend, the way they interact and communicate with each other.

Human-to-human interaction, besides speech also involves body language in which, besides the body stance and hand gestures, the head and facial expressions also play a quite important role. NodKit firstly aims to demonstrate how body language, and specifically the movements of the head, can create a new gesture interaction protocol with machines and eventually reach a production-ready level where it can be incorporated in various applications such as mobile operating systems, games, assistive communication aids for disabled people and so on. Purpose of this thesis is to follow a bottom up approach in describing how NodKit prototype is implemented, starting from the lowest core frameworks of the iOS platform all the way up to a fully working mock-up iPhone application demonstrating the various possibilities of the technology.

1.2 Structure of this paper

In **Chapter 2** there is a brief outline of the iOS platform and hardware as well as the developing tools and methodology used to produce NodKit's proof of concept.

In **Chapter 3** we provide a detailed description of the `AVFoundation` framework components that NodKit uses to capture raw video from the video camera of the device and feed it frame by frame to the face and features detector.

Chapter 4 provides some background on face detection technologies and algorithms and `CIDetector`, a Core Image class that natively and hardware-accelerated helps us easily detect faces and face features.

NodKit's details, configuration and delegation API is covered in **Chapter 5**.

Finally, in **Chapter 6**, we have enlisted some conclusions on the subject of head-based interaction gestures and notes on possible future work and upgrades.

Chapter 2
Platform Description

2.1 iOS Hardware

NodKit prototype was developed using the Apple iPhone 4S smartphone as the hardware of choice. A summary of the device's technical specifications of interest are listed in Table 2.1 and an annotated floor-plan of it's A5 SoC on Figure 2.1

Display	Type	LED-backlit IPS LCD, capacitive touchscreen, 16M colors
	Size	640 x 960 pixels, 3.5 inches (~330 ppi pixel density)
Memory	Type	Samsung DDR2 RAM
	Size	512MB
Storage	Type	Toshiba THGVX1G7D2GLA08 24 nm MLC NAND flash memory.
	Size	16 GB
CPU	Type	Dual-core 1 GHz Cortex-A9
GPU	Type	PowerVR SGX543MP2
Camera	Primary	8 MP, 3264x2448 pixels, autofocus, LED flash
	Secondary	0.3MP, VGA, 480p@30fps

Table 2.1 - Apple iPhone 4S Technical Specification

Although the iPhone model described above is not at the time of this writing the flagship in Apple's model range, it certainly proved more than capable of executing the intensive computations required by NodKit. We should also mention that NodKit framework runs equally well on the iPad 2 hardware and it's newest version, as well as on the iPod Touch (5th gen) or newer. Finally, it's worth mentioning that the front-facing camera mainly utilized by the framework, has been recently revamped to a more capable 720p@30fps "Facetime HD" camera. Essentially, that means that face features detection on newer iOS hardware will be easier and more precise, specifically in low light conditions.

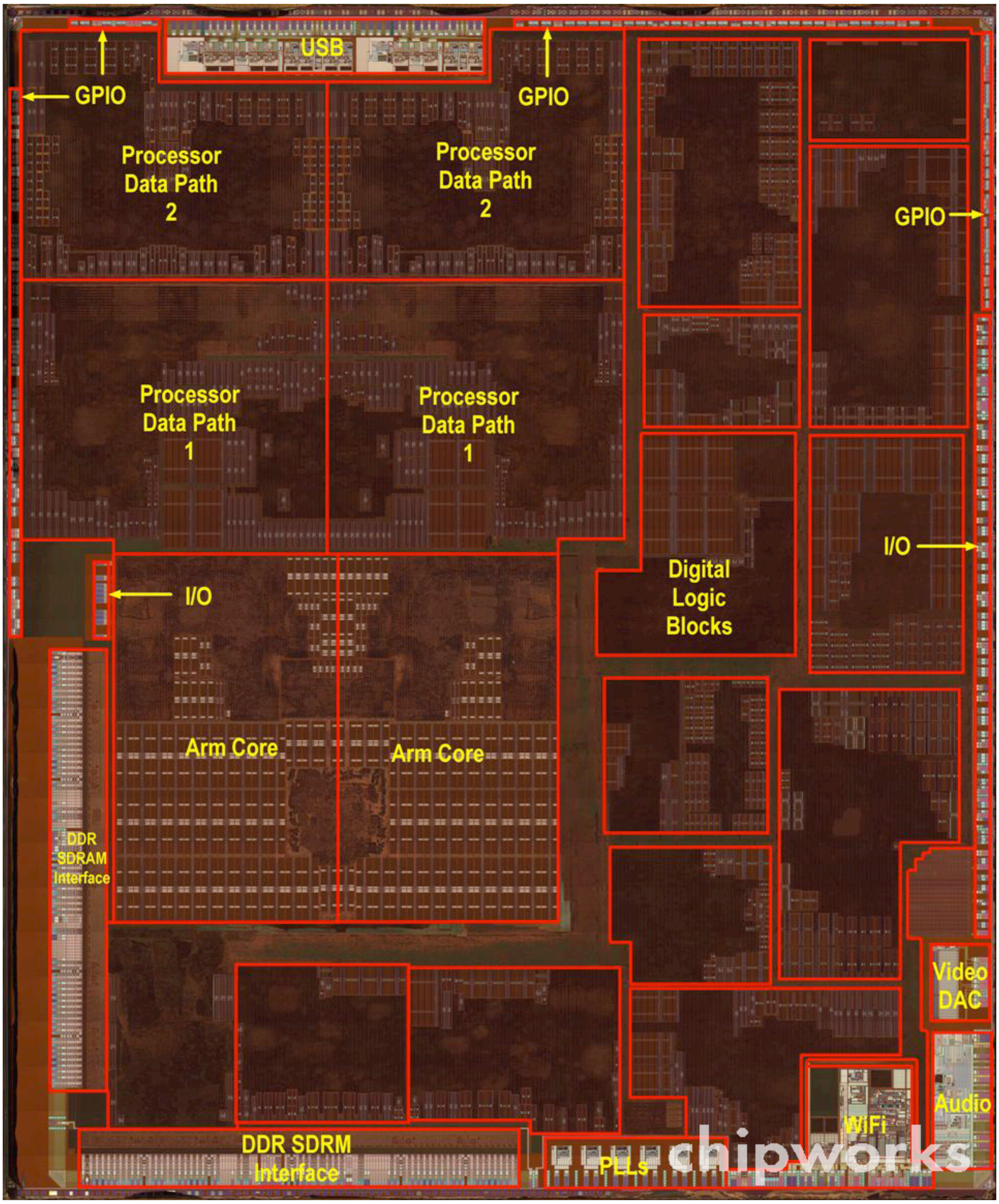


Figure 2.1 - Apple A5 SoC (System on a Chip)

2.2 iOS Operating System

The iOS operating system is a mobile operating system developed and distributed by Apple and ships preinstalled on the iPhone, iPad, iPod Touch and Apple TV platforms since originally released in 2007. iOS derives from Mac OSX, Apple's desktop operating system, which itself derives from the Darwin project and shares many of it's POSIX and BSD heritage like the process model, the network stack and virtual file system. This advanced operating system manages the devices hardware and provides all the technology stacks required to implement native applications.

Native applications on the iOS platform are built using the core iOS system frameworks and Objective-C language and directly run on the operating system in a restricted, sandboxed, environment. The iOS Software Development Kit (SDK) contains all the tools and interfaces needed to develop, install, run and test native applications on the platform. The iOS architecture itself is layered and provides all the necessary abstractions to transparently talk to the underlying hardware through a set of well-defined system interfaces that eliminate the problem of frequent hardware changes.

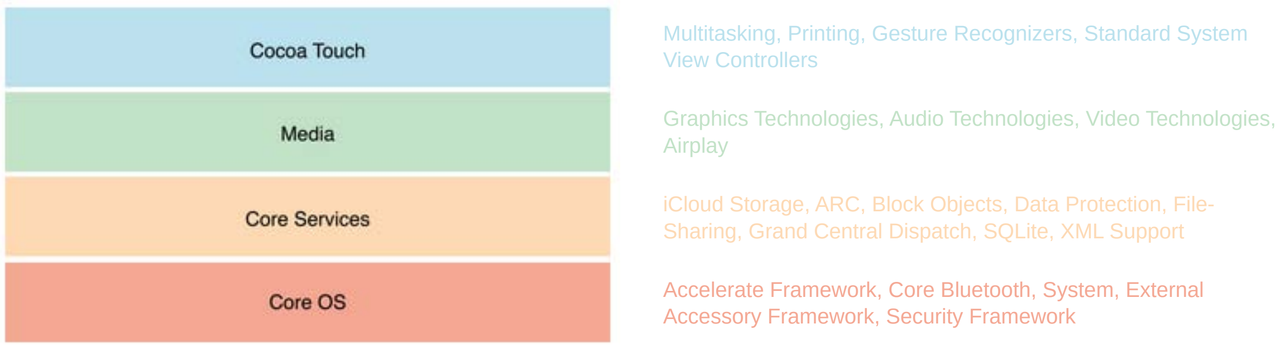

Cocoa Touch
Multitasking, Printing, Gesture Recognizers, Standard System View Controllers
Media
Graphics Technologies, Audio Technologies, Video Technologies, Airplay
Core Services
iCloud Storage, ARC, Block Objects, Data Protection, File-Sharing, Grand Central Dispatch, SQLite, XML Support
Core OS
Accelerate Framework, Core Bluetooth, System, External Accessory Framework, Security Framework

Figure 2.2 - Layers of iOS platform

2.2.1 The concept of Delegation

Delegation is a simple and powerful pattern used extensively in iOS development. Essentially, in delegation one object in the program acts on behalf of, or in coordination with, another object. The object responsible for the delegation keeps a reference to the other object, the delegate, and when time is appropriate sends a message to it. In such way the delegate is informed of an event that the delegating object is about to handle or has just handled. Upon receiving such a message, the delegate may respond to that message by updating a UI appearance, it's own or other object's state and in some cases even return a value that affects how an event is handled.

2.2.2 Grand Central Dispatch (GCD)

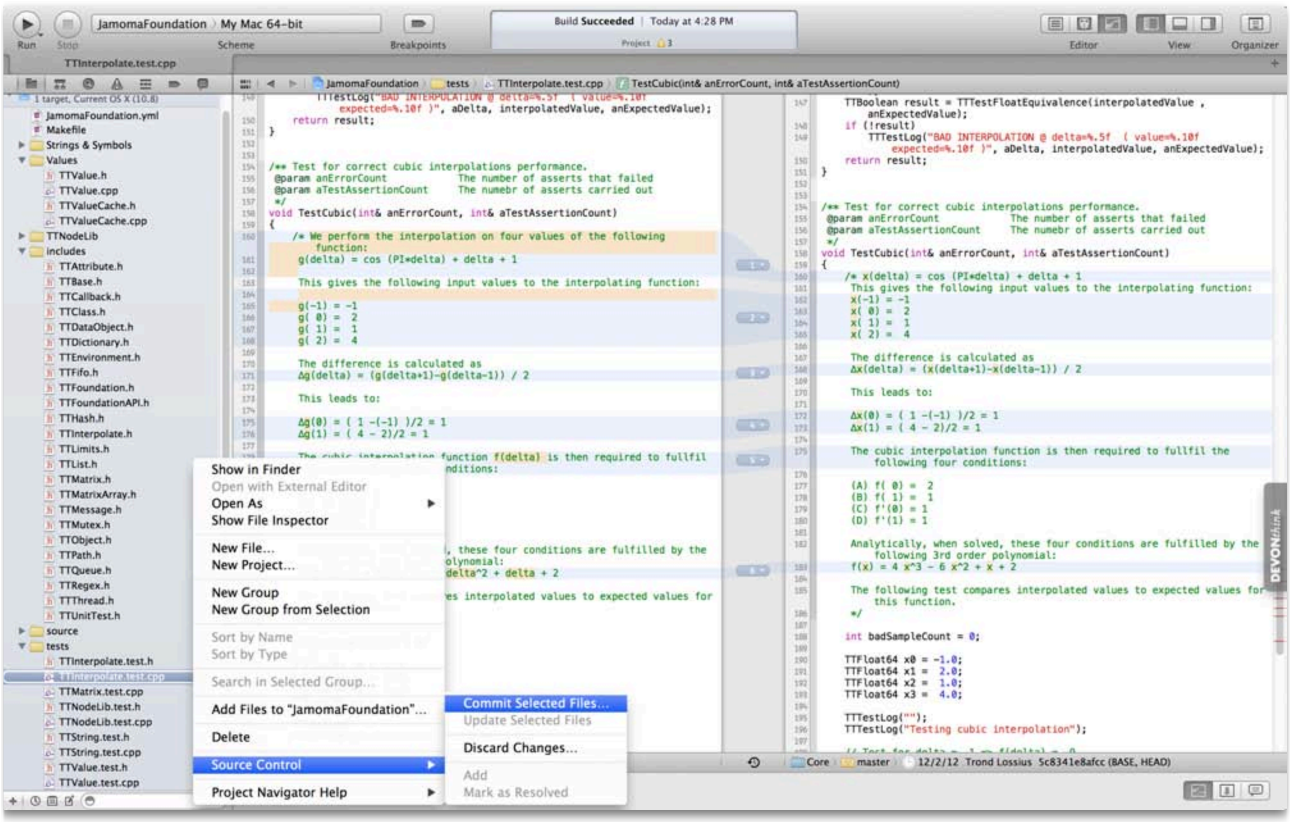
Introduced in iOS4.0, Grand Central Dispatch or GCD is a technology down to the BSD-level of the operating system that provides a robust and convenient alternative solution to threading. Furthermore, GCD can be used for many other low-level tasks such as reading and writing file descriptors, implementing timers and monitoring signals and process events. By defining the work that needs to be performed as task and adding it to an appropriate *dispatch queue*, GCD takes care of creating the needed threads pool and of scheduling the tasks to run on those threads.

Dispatch queues are first-in/first-out object-like structures that manage the tasks submitted to them. NodKit extensively uses GCD and dispatch queues in many places like for example to set the video sample buffer delegate of the video frames captured. In this occasion, a serial dispatch queue is preferred to guarantee that the video frames will be delivered and analyzed in the order they are captured.

2.3 Xcode, Apple's Software Development IDE

Xcode is Apple's LLVM based Integrated Development Environment for designing, developing, debugging and profiling applications designated for the MacOSX and iOS platforms. With twice as fast compilation and run times than *gcc*, Apple's *llvm* compiler used in the latest Xcode, is built from the ground up as a set of highly optimized libraries, easy to extend and optimize and designed to run on today's modern chip architectures like iPhone's ARMv7. Besides support for languages like C, C++, Objective-C and Objective-C++, Xcode can be also used as a programming environment for languages like Java, Applescript, Python and Ruby with the same ease.

The Xcode IDE is the main tool used to develop and build NodKit framework for release and testing on the iOS platform. Apart from the basic program editing features it provides, like built-in documentation, Assistant Editor, code completion and other inline context-sensitive information, Xcode is also used to debug NodKit using the graphical debugger that serves as a front-end to LLDB as well as to perform static code analysis. Furthermore, Xcode's support for various source versioning tools proved very helpful for dealing with the git SCM (Source Control Management), for NodKit's source control.



```
        TTTestLog("BAD INTERPOLATION @ delta=%.5f  ( value=%.10f
            expected=%.10f )", aDelta, interpolatedValue, anExpectedValue);
    return result;
}


/** Test for correct cubic interpolations performance.
 @param anErrorCount               The number of asserts that failed
 @param aTestAssertionCount        The numebr of asserts carried out
 */
void TestCubic(int& anErrorCount, int& aTestAssertionCount)
{
    /* We perform the interpolation on four values of the following
        function:
     g(delta) = cos (PI*delta) + delta + 1

     This gives the following input values to the interpolating function:

     g(-1) = -1
     g( 0) =  2
     g( 1) =  1
     g( 2) =  4

     The difference is calculated as
     Δg(delta) = (g(delta+1)-g(delta-1)) / 2

     This leads to:

     Δg(0) = ( 1 -(-1) )/2 = 1
     Δg(1) = ( 4 - 2)/2 = 1

     The cubic interpolation function f(delta) is then required to fullfil
                                 nditions:
```

```
                                 , these four conditions are fulfilled by the
                                 olynomial:
                                 delta^2 + delta + 2

                                 es interpolated values to expected values for
```

```
    TTBoolean result = TTTestFloatEquivalence(interpolatedValue ,
        anExpectedValue);
    if (!result)
        TTTestLog("BAD INTERPOLATION @ delta=%.5f  ( value=%.10f
            expected=%.10f )", aDelta, interpolatedValue, anExpectedValue);
    return result;
}


/** Test for correct cubic interpolations performance.
 @param anErrorCount               The number of asserts that failed
 @param aTestAssertionCount        The numebr of asserts carried out
 */
void TestCubic(int& anErrorCount, int& aTestAssertionCount)
{
    /* x(delta) = cos (PI*delta) + delta + 1
     This gives the following input values to the interpolating function:
     x(-1) = -1
     x( 0) =  2
     x( 1) =  1
     x( 2) =  4

     The difference is calculated as
     Δx(delta) = (x(delta+1)-x(delta-1)) / 2

     This leads to:

     Δx(0) = ( 1 -(-1) )/2 = 1
     Δx(1) = ( 4 - 2)/2 = 1

     The cubic interpolation function is then required to fullfil the
        following four conditions:

     (A) f( 0) =  2
     (B) f( 1) =  1
     (C) f'(0) = 1
     (D) f'(1) = 1

     Analytically, when solved, these four conditions are fulfilled by the
        following 3rd order polynomial:
     f(x) = 4 x^3 - 6 x^2 + x + 2

     The following test compares interpolated values to expected values for
        this function.
     */

    int badSampleCount = 0;

    TTFloat64 x0 = -1.0;
    TTFloat64 x1 =  2.0;
    TTFloat64 x2 =  1.0;
    TTFloat64 x3 =  4.0;

    TTTestLog("");
    TTTestLog("Testing cubic interpolation");
```

Figure 1.3 - Xcode 4 IDE

Finally, Xcode comes bundled with Instruments; a special front-end to the DTrace tool developed by Sun Microsystems. Instruments is a performance, analysis and testing tool that allows for dynamically tracing and profiling code on the OSX and iOS platforms. At a glance, using Instruments the developer can examine how one or more processes behave, record a sequence of user actions and replay them reliably reproducing the exact same events, create custom DTtrace instrumentation to analyze certain system aspects and do performance analysis and automated-testing of certain code parts or whole applications. Instruments proved to be invaluable to the development of NodKit, in the form of examining how the framework performed in real-life scenarios on the device and particularly how it consumed computational and memory resources.

Chapter 3
The Audio-Visual Framework (AVFoundation)

3.1 AVFoundation Introduction

AVFoundation is one of the many frameworks the iOS platform provides to play and create time-based audio-visual media. The AVFoundation framework is high performance, hardware accelerated and based on asynchronous processing. NodKit is heavily based on this framework since it is used to capture the input video stream from the device's front-facing camera and manipulate the frames in realtime.

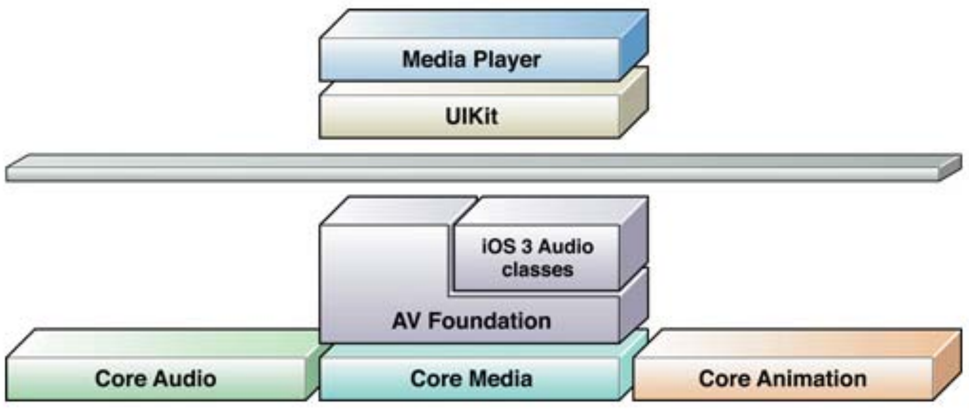


Figure 3.1 - AVFoundation Framework Architecture

3.2 Capturing Video Frames

NodKit framework starts the procedure by capturing video frames from the device's front facing camera. In order to coordinate the flow of data from the video camera and perform real-time capturing, an `AVCaptureSession` object must be instantiated and appropriate inputs and outputs must be configured. Following the instantiation of the session object, we need to set the quality level of the output to the minimum VGA quality using the constant named `AVCaptureSessionPreset640x480`. Although this resolution setting is low, it proved to be reasonably adequate for the post processing of the raw video frames.

When the session object is setup in place, we need to select the front facing video device and create a device input capture. An `AVCaptureDevice` object represents a physical capture device and the properties associated with it. The `AVCaptureDevice` object is required in order to configure the various properties of the underlying hardware. By instantiating such an object, using `+defaultDeviceWithMediaType:AVMediaTypeVideo` we make sure the front-facing camera of the device is selected as input.

In the next step, we need to provide a video data output that will produce the uncompressed video frames we are going to use, in order to detect the head position, orientation and movement. Such an output is configured by instantiating an `AVCaptureVideoDataOutput`. Extensive care should be given to the `alwaysDiscardsLateVideoFrames` flag, that ensures that any late arriving and out of order video frames will be dropped and discarded, so that they don't interfere with any previous head detection processing.

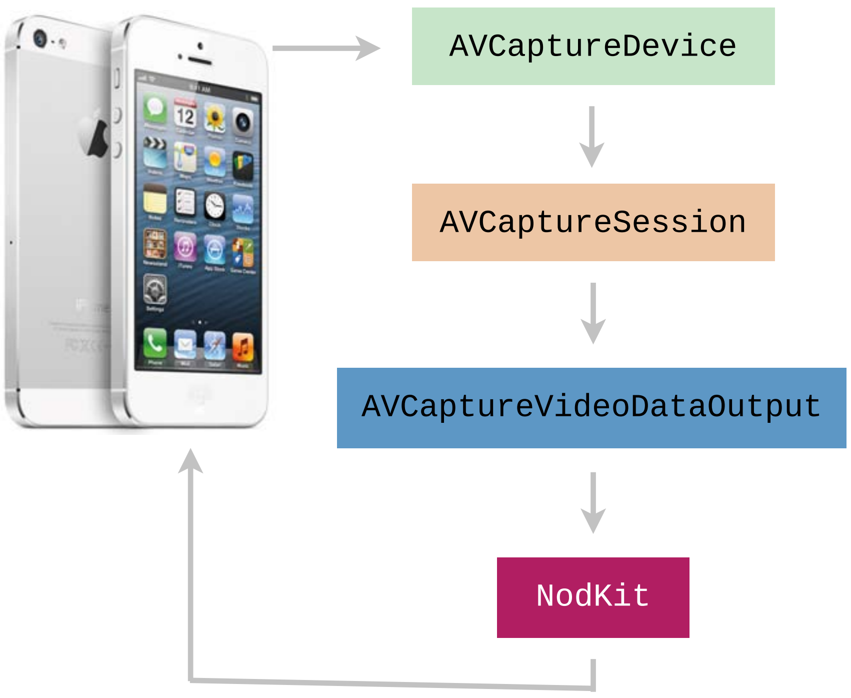


Figure 3.2 - Video capturing architecture

Finally, we need to create and initiate a serial dispatch queue via the `dispatch_queue_create()` method of the GCD (Grand Central Dispatch) framework. It's important to use a serial type dispatch queue in order to guarantee that the video frames will reach the delegate method in the order they were captured from the device. After we publish the `startRunning` message to our session object, the `AVCaptureVideoDataOutput` will send every video sample buffer captured to the sample buffer delegate, namely `captureOutput:(AVCaptureOutput *)captureOutput didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer fromConnection:(AVCaptureConnection *)connection` where NodKit logic takes place.

Chapter 4
Face Detection

4.1 Introduction to Face Detection practices

Face detection is the computer technology that determines and identifies the location and geometry of human faces and it's facial characteristics in digital images or video frames. In essence, it is regarded as a specific case of the more generic object-class detection, the task of which is to find the locations and sizes of particular objects in an image that belong to a certain given class. Examples of object classification include eyes, upper torsos, pedestrians, cars, license plates and other type of objects. Face detection should not be mistaken for "Face Recognition" where beyond detecting the human face we are able to compare it to a precomputed dataset or image, in order to precisely identify the presence of certain characteristics.

Face detection technologies are of high technological importance being the first step for innovating on fields such as Human Computer Interaction, the very field we are tackling in this thesis, Expression Recognition, Emotional State Recognition, applications related to Advanced Surveillance systems as well as Automatic Target Recognition (ATR). The evolution of these technologies have created a wide range of possibilities for commercial and law enforcement applications.

Besides the fact that the current state of face detection algorithms is already pretty advanced, there are numerous challenges to overcome before a unique and robust approach is standardized. Some of them include but are not limited to problems such as out-of-plane rotation of the subject, the presence of facial hair like beards or hair, unique facial expressions, occlusions by long hair and conditions that affect the image quality such as lighting, distortion, noise and compression.

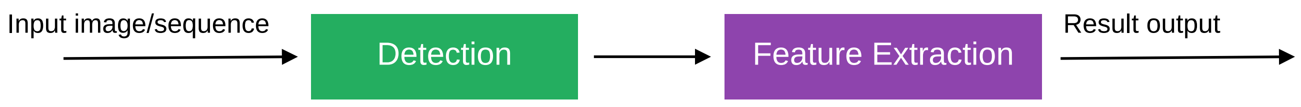


Figure 4.1 - The basic flowchart of face detection

In terms of the underlying modeling process used, the approaches to face detection fall into two broader categories: a) local feature-based ones; b) global methods. In the former ones, prominent features such as the eyes, nose and mouth are first detected and located. By taking various measurements and performing calculations on these facial components, the detecting methods construct feature vectors on which they rely the detection and characterization of the facial features and their geometrical relationships. On the other hand, global methods take a more holistic approach towards face detection without explicitly looking for facial features. Instead, they involve encoding the whole facial image and treating the resulting digital footprint as a point in a high-dimensional space while at the same time assume that all faces are constrained to particular positions, orientations as well as scales.

4.2 Haar-like features and the Viola-Jones method

Haar-like features are certain pre-classified features present in digital imagery and are widely used in object recognition and classification. They own their name to their similarity with Haar wavelets and are extensively used in real-time face detecting applications. It was Viola and Jones that back in 2001 modified the idea of using Haar wavelets in feature extraction and developed the so called Haar-like features and eventually the Viola-Jones object detection framework in 2001, the first object detection framework that provided competitive object detection rates in real-time.

A Haar-like feature takes into consideration neighboring rectangular regions at a certain location inside a detection window, sums up the pixel intensities in each of these regions and calculates the difference between these sums. If this particular difference is above a threshold set during the learning phase, then the feature is said to be present in the image.

$$\frac{\sum(pixelsInWhiteRegion) - \sum(pixelsInBlackRegion)}{totalPixelsInBothRegions}$$

Eq 4.1 - Pixel difference used for categorization

Let's consider a human face. It is very easy to observe that the horizontal region of the eyes is comparably darker than the region of the cheeks. Based on this observation, a common haar feature for face detection is a set of two adjacent rectangles which are located above the eye and the cheek region. Similar assumptions can produce other haar-like features useful for detecting the mouth and nose position.

The Viola-Jones method for detecting objects in images combines four key concepts: a) Haar features b) An Integral Image for rapid feature detection c) The AdaBoost machine-learning technique and d) a cascaded classifier used to combine many features in an efficient manner.

In order for the detection framework to determine the presence or absence of hundreds of Haar-like features at every possible image location and at several scales rapidly, Viola and Jones proposed the usage of summed area tables, or Integral Images. Generally speaking, "integrating" means assembling small units together which in the case of digital imagery these small units are actually pixels. The integral value of a pixel is defined as the sum of all pixels above it and to it's left. By starting from the top left of an image and traversing all the way to the right and down, the whole digital image can be integrated with only a few integer operations per pixel encountered. The resulting Integral Image can then be used as a two-dimensional lookup table in the form of a matrix which has the same size of the original image. Each element in this integral image contains the sum of all pixels located on the up-left region of the original image like we described before. Using this information we can compute the sum of rectangular areas in the image at any position or scale using only four lookups as described in Equation 4.2 below.

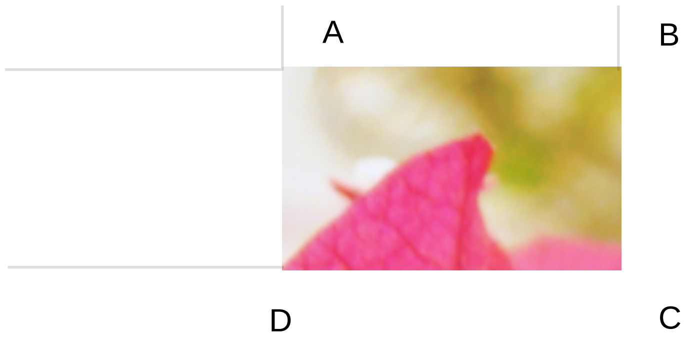


Figure 4.2 - Finding the sum of an area within the original image

Let A, B, C and D be the values of the integral image at the corners of a rectangle, then the sum of the original image values within this rectangle, or any other rectangle, can be computed with just three integer operations. Each Haar-like feature may need more than four lookups in the Integral Image depending on how exactly it is defined. Most of Viola and Jone's 2-rectangle features require six lookups.

$$\sum_{(x,y)\in ABCD} i(x,y) = ii(D) + ii(A) - ii(B) - ii(C)$$

Eq 4.2 - Sum of a rectangular area defined by four points.

In order for the detection framework to select the specific Haar-like features to use and to set the approximate threshold levels, the AdaBoost machine learning method is utilized. This method combines many "weak" classifiers to create one "strong" classifier. "Weak" here means that the classifier only gets the correct answer a little more often than a random guessing would, which is obviously not acceptable. However, if all these "weak" classifiers were to drive the final answer more and more towards the correct direction, we'd have a "strong" classifier capable of delivering the desired results. AdaBoost achieves this by a selecting a set of weak classifiers, the features, assigning a weight to each one of them and chaining them to produce the "strong" classifier. It then combines these strong classifiers into a chain of filters which forms the cascade classifier. If an image subregion makes it through every stage of this chain, then it's classified as "face" and ultimately the face location has been detected.

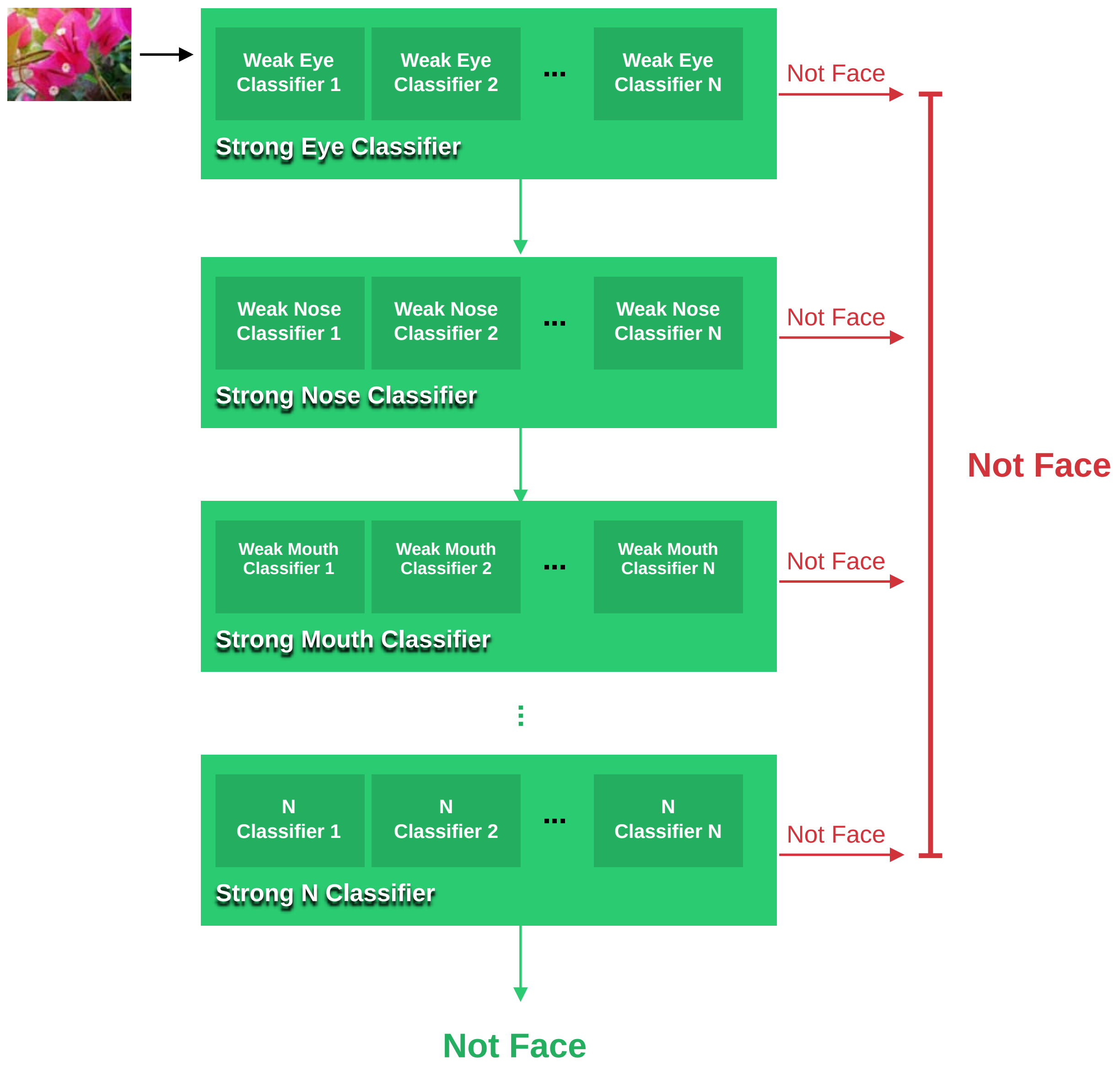


Figure 4.3 - Cascade Architecture of Viola-Jones method

In the above cascade architecture, lets assume:

- The Strong Eye Classifier consists of 1 weak Eye Classifier which achieves 100% detection rate and about 50% false positive rate.
- The Strong Nose Classifier consists of 5 Weak Nose Classifiers which achieve 100% detection rate and 40% false positive rate, or 20% cumulative.
- Finally, the last Strong Mouth Classifier with 20 Weak Mouth Classifiers achieves 100% detection rate with 10% false positive rate.

It's clear that overall, the cascade classifier produces a 2% cumulative false positive rate.

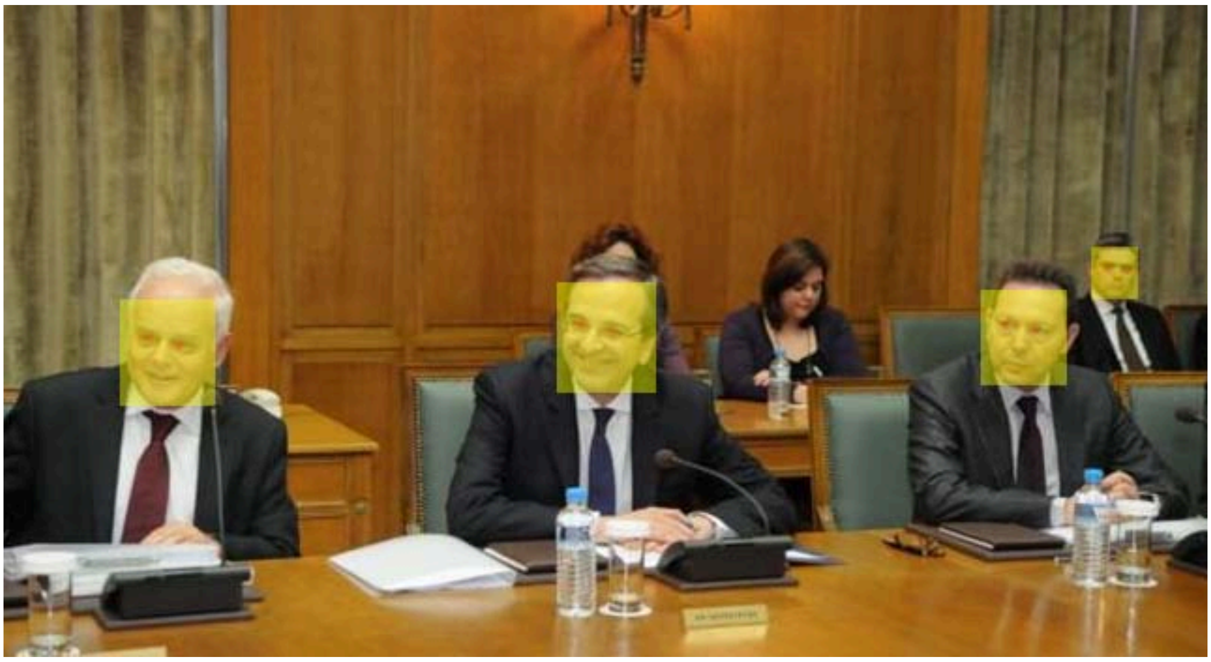
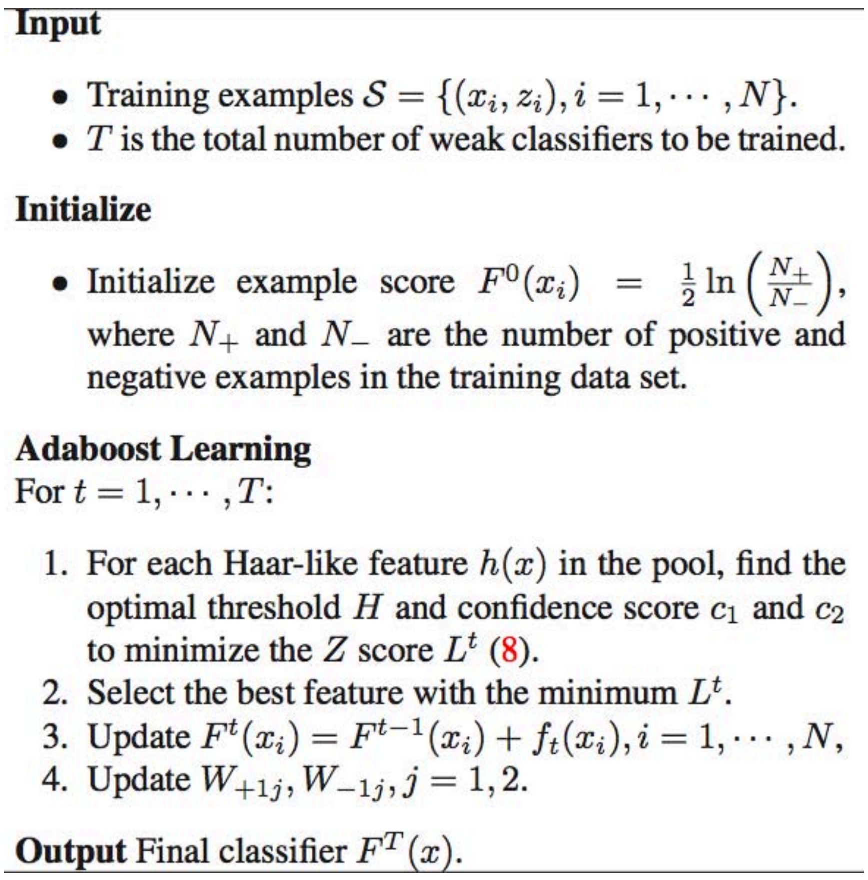


Figure 4.4 - Output of Nodkit's Face Detector on Test Images



```
Input
    • Training examples $\mathcal{S} = \{(x_i, z_i), i = 1, \cdots, N\}$.
    • $T$ is the total number of weak classifiers to be trained.

Initialize
    • Initialize example score $F^0(x_i) = \frac{1}{2}\ln\left(\frac{N_+}{N_-}\right)$,
      where $N_+$ and $N_-$ are the number of positive and
      negative examples in the training data set.

Adaboost Learning
For $t = 1, \cdots, T$:
    1. For each Haar-like feature $h(x)$ in the pool, find the
       optimal threshold $H$ and confidence score $c_1$ and $c_2$
       to minimize the $Z$ score $L^t$ (8).
    2. Select the best feature with the minimum $L^t$.
    3. Update $F^t(x_i) = F^{t-1}(x_i) + f_t(x_i), i = 1, \cdots, N$,
    4. Update $W_{+1j}, W_{-1j}, j = 1, 2$.

Output Final classifier $F^T(x)$.
```

Algorithm 4.1 - AdaBoost learning pseudo code

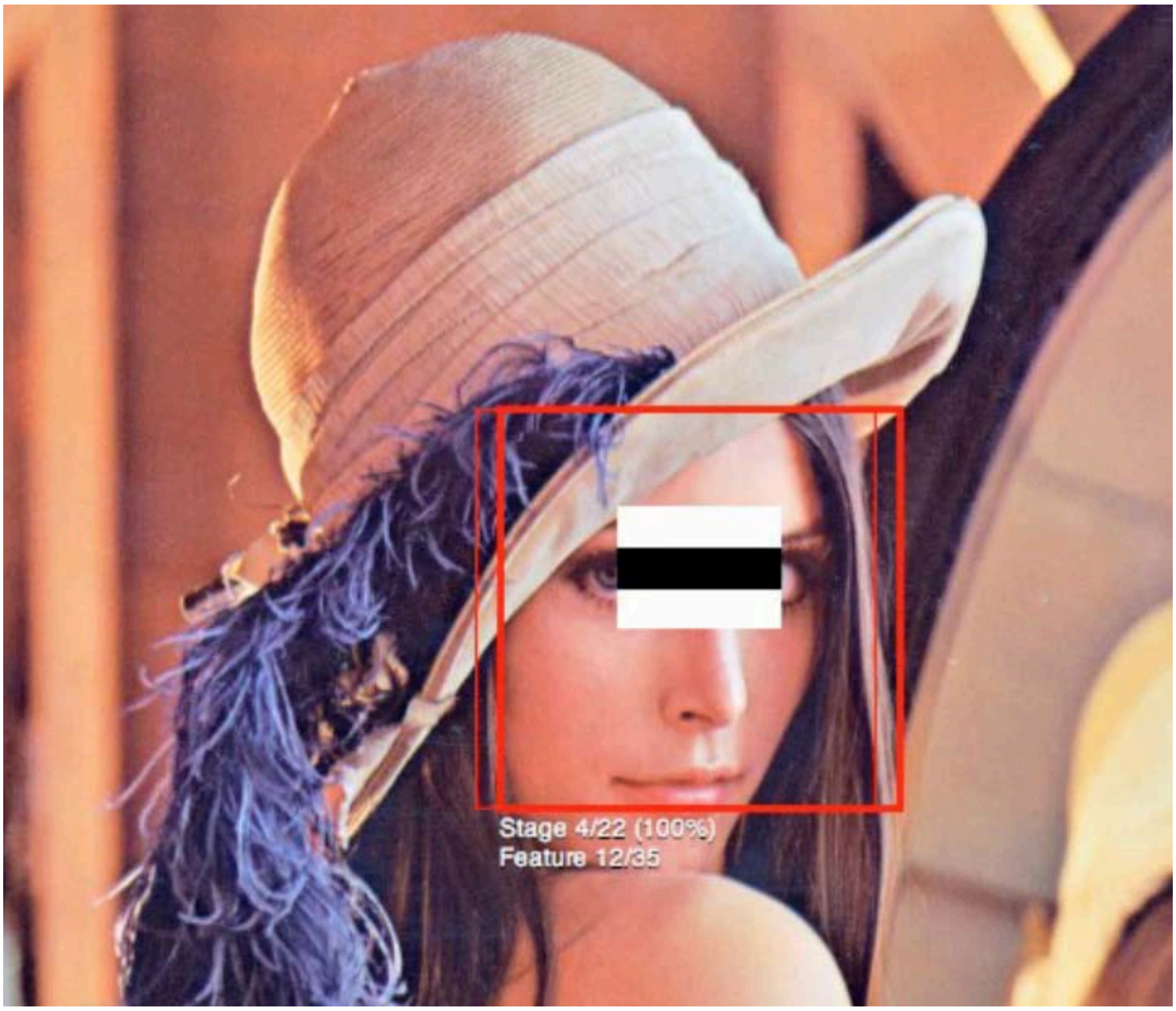


Figure 4.5 - Haar-like feature for mouth detection running on Lenna

This attentional cascade classifier is constructed manually. To be specific, the total number of the weak classifiers and the decision threshold for rejecting early at each node are both specified by hand, which is not a trivial task. If the decision thresholds were to be set too carelessly, the final face detector might be fast but the overall detection rate and accuracy will be limited. On the other hand, if the decision threshold were to be set in a very strict and conservative way, the resulting detector might end up being unbearably slow. Truth to be said, a good fast and reliable face detector can take a lot of work and time of fine-tuning to produce the desired results. As an advancement to the original method, L.Bourdev and J.Brandt proposed a method that starts with a relatively small set of training examples, and adds to it new samples at each stage that the current classifier produces wrong output. Of course, the number of the new non-faces added at each training cycle severely affects the "focus" of AdaBoost during the training phase. If this number grows too large, AdaBoost won't be able to catch up with it and the false positive rate will get higher. If the number is too small, the cascade classifier will contain too many of the "weak" classifiers in order to reach a comprehensive false positive rate. In another approach, C.Zhang and P.Viola, proposed to use "importance sampling" to help address the huge data set problem. The training positive or negative data sets are being resampled periodically to ensure feasible computation in restricted time frames.

4.3 Face Detection on the iOS Platform

The first prototype of NodKit on the iOS platform, contained custom code based on the open-source OpenCV library for face detection using the Viola Jones method described above. Luckily, Apple on September 2010 acquired the Swedish face-recognition firm Polar Rose and it's proprietary face detection and recognition technology later made it as a part of the Core Image framework with the release of iOS5 on October 2011.

Although there is limited public information on how exactly Polar Rose's face detection method works, we've came to the assumption that it converts the two-dimensional images into a pseudo-3d artifact and then applies a modified Viola Jones method with custom Haar-like features. This 3D conversion technique greatly improves the matching ability of the algorithm by helping to eliminate pose and lighting conditions in order to build more generic face attributes. Furthermore, Polar Rose used to provide the so-called FaceCloud service that stored hundreds of thousands of images containing faces, a data set which was undoubtedly severely used to constantly train their proprietary cascade classifiers.

Now a part of the CoreImage framework on iOS, Polar Rose's technology is abstracted behind the `CIDetector` class object. By initializing such an object via the `+(CIDetector *)detectorOfType:(NSString*)type context:(CIContext*)context options:(NSDictionary*)options` and setting a detector of type `CIDetectorTypeFace` we can easily perform the aforementioned image processing methodologies in order to detect faces and their features in image contexts. In addition to that, a special option defining the detector accuracy must be passed to the detector. For the time being, the accuracy options are limited to just "low" or "high" and several tests proved that the "low" option is quite adequate for our purposes without sacrificing much of the device's resources and performance.

Like we described in Chapter 3.2, the `AVCaptureVideoDataOutput` object sends the captured video frames buffer to the setup delegate, so at that point we create the image context from the video frames and run the face features detector on it, in order to detect the face and determine it's position. However, before we start making gesture decisions based on the head posture and motion we need to detect the orientation of the device because it affects the way video is captured and in accordance, the way image contexts are created. The device's orientation will eventually lead to the image's EXIF orientation as this is defined in the EXIF specifications. To be specific the device has four possible orientations:

1. `UIDeviceOrientationPortraitUpsideDown` - The device is oriented vertically and the home button is on the top
2. `UIDeviceOrientationPortrait` - The device is oriented vertically and the home button is on the bottom. This is the default orientation for iPhones.
3. `UIDeviceOrientationLandscapeLeft` - The device is oriented horizontally and the home button is on the right.
4. `UIDeviceOrientationLandscapeRight` - The device is oriented horizontally and the home button is on the left.

According to the EXIF specification and bearing in mind that the front-facing camera is utilized we have for each of the aforementioned orientations for:

1. The 0th row is on the left and 0th column is on the bottom.
2. The 0th row is on the right and the 0th column is on the top.
3. The 0th row is on the bottom and the 0th column is on the right.
4. The 0th row is on the top and the 0th column is on the left.

With each image's frame exif orientation setting we can now directly call the `featuresInImage` instance method of the face detector with the `CIDetectorImageOrientation` set correctly in order to collect the face features of every video frame in an `NSArray` and feed it to NodKit's decision-making algorithm.

Chapter 5
NodKit Internals

5.1 - Calibration phase and the Idle Zone

Like we described in the previous chapter, the `CIDetector` returns a `NSArray`, essentially an array of structures, that contains the detected face features in each of it's slots. Each of these slots contain a `CIFaceFeature` object with the following properties:

```
struct CGRect bounds;
struct CGPoint leftEyePosition;    // position of the left eye
struct CGPoint rightEyePosition;   // position of the right eye
struct CGPoint mouthPosition;      // position of the mouth
BOOL hasLeftEyePosition;           // Face feature is the left eye
BOOL hasRightEyePosition;          // Face feature is the right eye
BOOL hasMouthPosition;             // Face feature is the mouth
```

For example, if the first feature in the features array is the mouth, the `hasMouthPosition` boolean property would be set to `YES` and the bounds `CGRect` struct would encapsulate the x/y origin of the feature in a virtual cartesian pane as well as it's height and width in it. By combining the positioning and size information of these facial features we are able to construct what we call "Head Pointer". The Head Pointer can be thought of a pointing cursor directed by the size, orientation, posture and movement of the user's head. The position of the Head Pointer is easily calculated as the center of the circle defined by the three points; the left eye, the right eye and the mouth using the cartesian equation of the circle and analytic geometry.

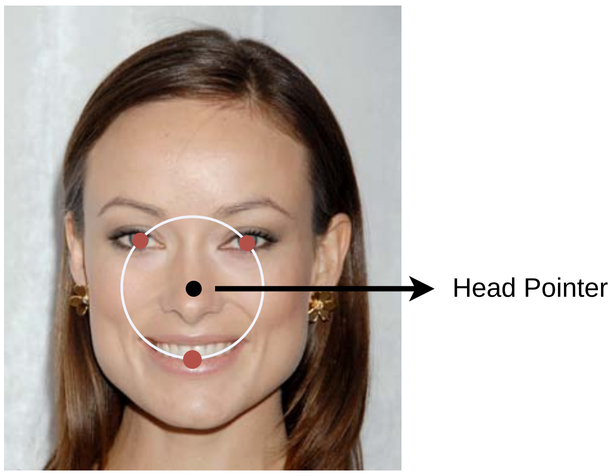


Figure 5.1 - Calculated Head Pointer on Olyvia Wilde's portrait image

The following algorithm demonstrates a fairly easy way to calculate the Head Pointer using geometry by joining two pairs of points in order to create two chords. The perpendicular bisectors of a chord always passed through the center of the circle.

```
- (CGPoint)calculateHeadPointFromLeftEye:(CGPoint)leftEyePosition
                                  rightEye:(CGPoint)rightEyePosition
                                     mouth:(CGPoint)mouthPosition
{

   CGPoint headPoint;

   float yDelta_leftEye = rightEyePosition.y - leftEyePosition.y;
   float xDelta_leftEye = rightEyePosition.x - leftEyePosition.x;
   float yDelta_rightEye = mouthPosition.y - rightEyePosition.y;
   float xDelta_rightEye = mouthPosition.x - rightEyePosition.x;

   float leftEyeSlope = yDelta_leftEye / xDelta_leftEye;
   float rightEyeSlope = yDelta_rightEye / xDelta_rightEye;

   headPoint.x = (leftEyeSlope * rightEyeSlope * (leftEyePosition.y - mouthPosition.y) +
rightEyeSlope * (leftEyePosition.x + rightEyePosition.x) - leftEyeSlope * (leftEyeSlope.x +
mouthPosition.x) ) / (2 * (rightEyeSlope - leftEyeSlope));

   headPoint.y = -1 * (headPoint.x - (leftEyePosition.x + rightEyePosition.x)/2) / leftEyeSlope +
(leftEyePosition.y + rightEyePosition.y) / 2;

   return headPoint;
}
```

Algorithm 5.1 - Function for calculating the Head Pointer from eyes and mouth positions

After detecting the Head Pointer, NodKit initiates what we call the Calibration Phase. During the Calibration Phase, a total amount of `CALIBRATION_SAMPLES` is collected by sequentially keeping the position values of the Head Pointer in the virtual cartesian pane. If the user is not moving abruptly and holds his head fairly steady during the calibration phase, these calibration values of sequential Head Pointer positions accumulate and when the total is reached, the average of the last five values is calculated and set as the Calibration Center Point. On the other hand, if one of the sampled values greatly varies in distance from the previous one, all the previous samples are discarded and the Calibration Phase restarts. When the Calibration Phase is completed successfully, a circular Idle Zone is assumed around the Calibration Center Point and NodKit calls it's required delegate function `-(void)nodKitController:(NodKitController *)controller didStartTrackingHead;` informing the calling controller that the framework has locked the user's head and starts capturing nods and gestures.

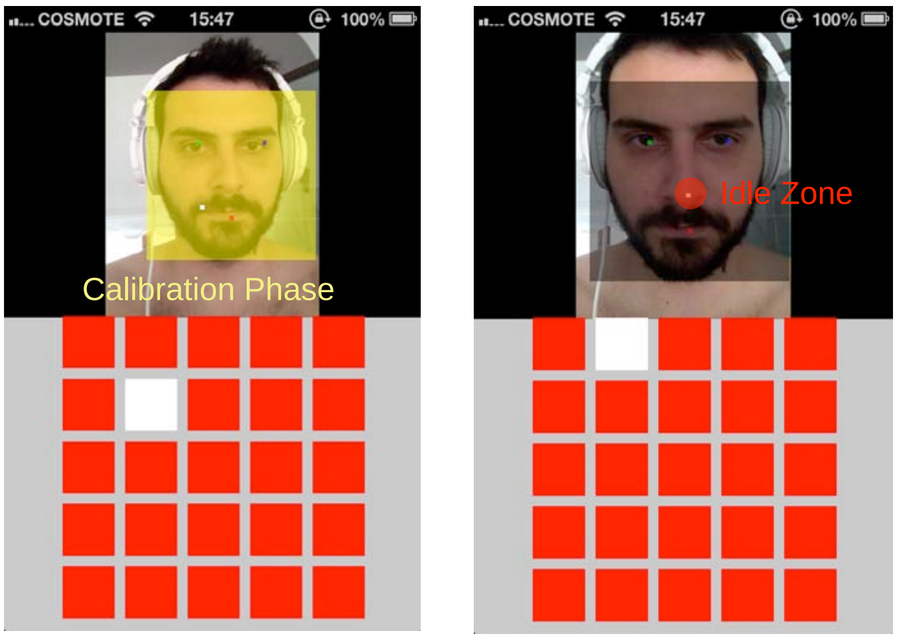


Figure 5.2 - Screenshots demonstrating the Calibration Phase on the left and the Idle Zone on the right right after the Calibration Center Point is calculated (in white)

The purpose of the Idle Zone is to eliminate the accidental movement of the Head Pointer due to the slight natural head movement that varies but is always present to an extend, as well as any potential shaking of the device due to hand movement, and thus the camera that is utilized by the framework. The diameter of the Idle Zone in the virtual cartesian pane is manually set via a preprocessor definition namely `IDLE_ZONE__THRESHOLD`. Finally, whenever the Head Pointer enters or stays in the Idle Zone, a global boolean flag named `isInIdleZone` is set to `YES`.

5.2 - Movement detection and gestures

Whenever the state of NodKit is not in the Calibration Phase and the Head Pointer is not in the Idle Zone, the video frames that arrive are investigated in order to detect possible movement of the Head Pointer and thus the user's head. The process starts by calculating the `verticalOffset` and `horizontalOffset` values which are essentially the distance difference between the Calibration Center Point and the current position of the Head Pointer. If this particular difference is smaller than the radius of the Idle Zone then NodKit ignores the movement and proceeds to the next frame. If the distance is greater than the Idle Zone's radius but smaller than the special threshold `DIRECTION_DISTANCE_THRESHOLD`, then NodKit also ignores the movement of the head. The two previous cases along with fine tweaking the aforementioned configuration thresholds in accordance with the specific application, can effectively discard false and accidental movements of the user's head.

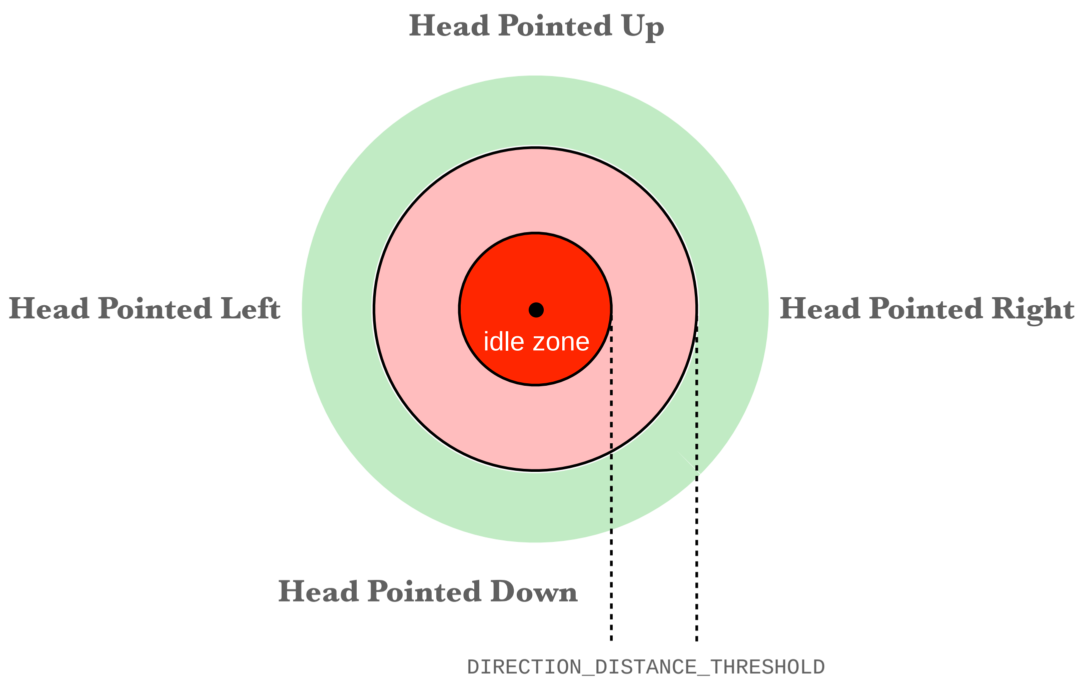


Figure 5.3 - Areas in which the Head Pointer is classified as movement or not

When the Head Pointer's distance from the Calibration Center Point is greater than the Idle Zone's radius as well as the `DIRECTION_DISTANCE_THRESHOLD` then NodKit classifies the distance difference as a valid head movement and depending on the sign of the calculation result an up/down or left/right direction is decided. NodKit is able to handle two different kind of movements: a) continuous movements and b) discrete movements, depending on the property set during the initialization or during the runtime.

Continuous movements are the ones where the head moves towards a direction and holds there for a worthy amount of time before returning back into the Idle Zone. Detecting continuous movements with NodKit is as easy as implementing any of the following optional delegate methods available:

```
-(void)nodKitController:(NodKitController*)controller
didStartLookingLeftWithDistance:(CGFloat)distance;
-(void)nodKitController:(NodKitController*)controller didStopLookingLeft;

-(void)nodKitController:(NodKitController*)controller
didStartLookingRightWithDistance:(CGFloat)distance;
-(void)nodKitController:(NodKitController*)controller didStopLookingRight;
```

```
-(void)nodKitController:(NodKitController*)controller
didStartLookingUpWithDistance:(CGFloat)distance;
-(void)nodKitController:(NodKitController *)controller didStopLookingUp;

-(void)nodKitController:(NodKitController*)controller
didLookDownWithDistance:(CGFloat)distance;
-(void)nodKitController:(NodKitController *)controller didStopLookingDown;
```

For example, when the user's head starts looking down and holds there for a while, when the Head Pointer goes past the Idle Zone and the `DIRECTION_DISTANCE_THRESHOLD,` NodKit calls the `didStartLookingDownWithDistance:` delegate method and stores the distance from the Calibration Center Point into the `distance` variable. When the user stops looking down and reverts to a position near it's starting one, or in NodKit terms the Head Pointer goes somewhere back in the Idle Zone, then the `didStopLookingDown` delegate method is called. It is safe to assume that between these delegate calls, the user's head is steadily looking towards a direction nowhere near the Idle Zone or the Calibration Center Point.

On the other hand, a movement is registered as discrete when the user looks towards a direction and then returns to the Idle Zone and only then the appropriate delegate method is called. The optional available delegate methods for capturing discrete movement events are the following:

```
/* Discrete Nods to one direction */
-(void)nodKitController:(NodKitController*)controller
didLookLeftWithDistance:(CGFloat)distance;

-(void)nodKitController:(NodKitController*)controller
didLookRightWithDistance:(CGFloat)distance;

-(void)nodKitController:(NodKitController*)controller
didLookUpWithDistance:(CGFloat)distance;

-(void)nodKitController:(NodKitController*)controller
didLookDownWithDistance:(CGFloat)distance;
```

Again, these delegate methods are called only when the user's head makes a movement and returns back to the Idle Zone. The `distance` variable like in the continuous movement paradigm holds the Head Pointer's distance from the Calibration Center Point.

Finally an optional, more abstract and generic delegate method is available for any other use someone utilizing NodKit framework might think of.

```
-(void)nodKitController:(NodKitController *)controller didMoveFromPoint:
(CGPoint*)startingPoint toPoint:(CGPoint*)endingPoint
withIdleAreaCenterPoint:(CGPoint *)centerPoint;
```

This delegate method, when implemented, is called every time a frame is processed and as the name prototype clearly specifies, the starting point of the Head Pointer is set to the `startingPoint` variable, the ending point of the Head Pointer is set to the `endingPoint` variable and the Calibration Center Point is set to the `centerPoint` variable. In fact, NodKit implements both continuous and discrete movement detection by using a method with an interface definition almost identical to this delegate's one.

5.3 - NodKit runtime initialization & configuration

Like we described in the preface sections of this thesis, NodKit is distributed in the form of a compiled static library or framework. In order for someone to utilize NodKit's capabilities the MZNodKit.h header file and the MZNodKitLib.a must be included in the iOS project's build phase. Beyond that point, NodKit framework uses the Singleton design pattern that restricts the instantiation of a class to one distinct object thats is required to coordinate the appropriate actions across the system. In order to initialize NodKit's singleton manager the `sharedManager` message should be sent to a `NodKit` object. Beyond that point using the singleton object, the developer must set the delegate where the required delegate methods are implemented and can then access or set various properties during the runtime like the `faceDetectionAccuracy`, the `gestureDetectionRate` or the `calibrationCenterPoint` and `idleAreaRadius`. These properties along with the shared manager singleton object and the optional and required delegates described previously constitute NodKit's programmatic interface.

Chapter 6
Epilogue and Conclusions

6.1 Conclusions

In this thesis we prototyped and described a new approach for interface input using innovative head tracking technologies and advanced face detection techniques. We overviewed how using an integrated development kit like the one Apple provides for the iOS platform along with the advanced native frameworks on the iOS platform can help us build production ready software for integration by third party developers. In addition to that, we described how we took advantage of the hardware's capabilities, the video infrastructure of the operating system and a mashup of the Viola Jones method and analytical geometry to invent a solution that augments the user's interaction with machines in a way that a few years ago looked like a distant future idea. The numerous real-life "field" tests of the framework in demo applications on actual devices, clearly proved that the technology performs remarkably well and the overall approach is undoubtedly on the right track.

6.2 Future Work

NodKit is a starting effort in a very promising field. Despite the fact that the prototype is near production readiness that are still various limiting factors to be considered before incorporating it in large scale distributions. For example, although the code is optimized and accelerated to an extend, the continuous heavy computations required for face detection and the constant camera hardware utilization drains the device's power battery a lot quicker than in normal moderate use cases. Or, although NodKit is a perfect user experience enhancement add-on in use cases such as gaming, the resources required to support the head gestures may overlap with the graphics and rendering requirements thus sacrificing the overall performance of the system.

However, at todays technological progression pace, mobile devices like the iPhone and the iPad will get considerably more capable and powerful in the near future. The incorporation of multicore processors, higher RAM availability and better graphic chipsets along with the advancement in battery and power-source industries will allow developers use more and more of the underlying hardware capabilities without having todays performance challenges. Furthermore, the device platform itself may incorporate new sensors such as proximity or infrared sensors, that will not only allow head detection with far less computations but also in all three axes much like Microsoft's Kinect technology that can create and process 3D-models of the user's figure in real-time. Although at this point NodKit sufficiently proves the concept of using your head to robustly send input to a computer device, it can undoubtedly benefit from some of the aforementioned platform abstractions in order to shape into a commercial and widely adopted technology that will not only make our everyday devices smarter and easier to use but will also help people with disabilities to utilize technology in ways that will aid them to increase their living standards.

Bibliography & Sources

[1] http://en.wikipedia.org/wiki/Face_detection
[2] http://en.wikipedia.org/wiki/Haar-like_features
[3] http://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework
[4] http://en.wikipedia.org/wiki/IOS
[5] http://en.wikipedia.org/wiki/Macosx
[6] A Survey of Face Detection, Extraction and Recognition, Yongzhong Lu, Jingli Zhou, Shengsheng Yu June 2002
[7] A Survey of Recent Advances in Face Detection, Cha Zhang and Zhengyou Zhang June 2010
[8] Face Recognition Using Long Haar-like Filters, Y. Higashijima1, S. Takano1, and K. Niijima, Department of Informatics, Kyushu University, Japan.
[9] An Introduction to Face Detection and Recognition, Ziyou Xiong
[10] http://developer.apple.com/library/ios/#documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/00_Introduction.html
[11] Cocoa Programming, Apple Developer Portal http://developer.apple.com/library/ios/#documentation/General/Conceptual/CocoaEncyclopedia/Introduction/Introduction.html
[12] Core Image Programming Guide, Apple Developer Portal http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/CoreImaging/ci_intro/ci_intro.html#//apple_ref/doc/uid/TP30001185
[13] Concurrency Programming Guide, Apple Developer Portal http://developer.apple.com/library/ios/#documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html
[14] CIFeature Class Reference, Apple Developer Portal http://developer.apple.com/library/ios/#documentation/CoreImage/Reference/CIFeature_Ref/Reference/Reference.html
[15] XCode Concepts, Apple Developer Portal https://developer.apple.com/library/ios/#featuredarticles/XcodeConcepts/Concept-Targets.html