Optimized Face Tracking in Adobe Flash

by

Kevin Lockard, B.A. Real Time Interactive Simulation

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Kevin Lockard

September 10, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Kevin Lockard

September 10, 2010

Acknowledgments

I want to thank Yann and John, whose help during this whole process was immensely appreciated, and my family for supporting my wacky idea to move to Ireland and learn to make video games.

Kevin Lockard

University of Dublin, Trinity College
September 2010

Optimized Face Tracking in Adobe Flash

Kevin Lockard
University of Dublin, Trinity College, 2010

Supervisor: Yann Morvan and John Dingliana

This dissertation aims to solve the problem of smooth face tracking for input. Face detection is a heavily researched problem in the field of computer vision, but much less attention is paid to smoothing the results to use as a form of input for applications. Face movement cannot be used as an alternative form of input until solutions are created to smoothly and accurately track a user's face over time. Existing solutions for face detection produce results that are often quite noisy and jitter a significant amount. The chosen platform is Adobe Flash, in order to take advantage of that platform's ease of distribution through the web and desktop applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Face Tracking

The task of determining whether or not an image contains a human face is one of the most basic problems in computer vision. The uses for such an algorithm vary widely, from photo album software that can automatically determine the people within a photograph to security software performing face recognition on security camera footage, but first must determine which part of the video stream represents a face. Because of this, face detection is one of the most widely studied problems in the field of computer vision. However, despite all of that work, little has been published on the matter of head tracking as a form of input. Previous works focus very heavily on the detection portion of the process, and very little is written about the task of getting a computer to track a user's head movement both smoothly and accurately. Head movement is a viable method of input for those unable to use other forms of input, but only if the tracking system used is both responsive and accurate. Augmented reality input options are also becoming mainstream in the video game industry. Alternative forms of input are available for every major game console. Both Sony and Nintendo have released motion sensing controllers for their consoles. Also, in November 2010 Microsoft will release the Kinect, an extension to the Xbox360 game console that implements face tracking along with face recognition, skeleton tracking, and voice recognition. These products all show the commitment that the video game industry has towards embracing

forms of input beyond traditional controllers.

1.1.2 Adobe Flash

Adobe Flash is one of the most widely installed pieces of software in the world, with Adobe claiming that it can be found on 99% of internet enabled computers, and is available on every major operating system. Due to this massive install base, Flash has gone relatively quickly from a simple program for vector graphics and animations to an advanced programming environment, complete with its own object-oriented language in Actionscript 3.0. But a program that can be found on such a wide variety of personal computers also has to make certain sacrifices in order to support them all. For example, the rendering pipeline of Flash was completely software-based until very recently when hardware acceleration was introduced into Flash Player 10.1. Just as Flash must make sacrifices in order to accomodate all the different machines it finds itself installed on, Flash applications must also be able to run on very low-end computers. The ability of Flash applications to reach such a massive audience - either through embedding them on websites or as desktop applications using Adobe Integrated Runtime or AIR - and the challenge of optimizing the tracking algorithm to be fast enough to run on a variety of computers are the main motivations for choosing Flash as the platform for this thesis.

1.2 Goals

The aim of the project was to develop an approach to face tracking which is accurate, fast, and smooth, and to implement that algorithm in Adobe Flash. During the course of research, two separate approaches were developed, each being based on a different underlying form of face detection. The developed algorithms are used to create an example application that is a three dimensional parallax application controlled by tracking the user's face. This application will provide the user with a view of the front square of Trinity College Dublin. The position of the user's face will be used as input to the application to determine the view point, therefore providing the user with the impression that they are looking into an actual scene within the computer screen, and by moving their head they can look around the scene.

Figure 1.1: Screenshot from the Parallax Viewing Application

The quality of the results will be benchmarked by a specially written application that measures the tracking algorithms on several different metrics of performance. Evaluating the results of these benchmarks will illuminate the proper situations in which the two approaches excel or fail.

1.3 Terminology

Since the terms are used quite often, it is important that the reader sees the difference between face detection and face tracking. For the purposes of this paper, face detection can be defined as using image processing techniques to find an area or areas within a static image that represent a human face. Face tracking is defined as using a face detection scheme in order to track the position of a face within a series of static images, or a video stream.

1.4 Document Layout

This thesis is presented in the following manner:

- Chapter 2 provides an overview of the current state of the art of face detection and tracking. It also gives a bit of background into the Actionscript language and the use of head tracking as a form of input.

- Chapter 3 outlines the design of the system created. It introduces the requirements for the ideal system, the algorithms chosen for investigation, and some detail into the benchmarking that will be performed in order to test the accuracy of the systems created.
- Chapter 4 delves into specific implementation details, problems faced, and the optimizations performed in order to get the tracking running as quickly as possible in Flash.
- Chapter 5 evaluates the results of benchmarking the algorithms using a few different metrics, focusing on the speed and accuracy of the developed algorithms. It also investigates the best and worst case scenarios for the developed algorithms, and benchmarks the different Actionscript design decisions made.
- Chapter 6 concludes the thesis and presents some ideas of possible future work.

Chapter 2

State of the Art

2.1 Face Detection

Face detection is a well-studied problem in the field of computer vision, and several different approaches to the problem have been put forth. Many solutions analyze the colors of an image to estimate where a face may be found. Another popular approach involves searching the image for areas of contrast that are often found in faces. This section will provide a brief overview of a few different approaches to face detection, and a bit more information on the methods that were chosen for the project.

2.1.1 Template Matching

Template matching is a face detection technique that attempts to determine if an area can be classified as a face by comparing it to images that are known to be faces. One of the earliest attempts at face detection through template matching was [4], but was later improved upon by such work as [10].

The first step of any template matching algorithm is to isolate the area of the image that should be tested against the template. This is done through skin color detection and segmentation. [4] suggest a histogram calibrated with a large number of samples of different types of skin color, which is then smoothed by a Gaussian distribution. This histogram will produce images representing the probability that the corresponding pixels represent skin, which can then be thresholded to extract regions most likely to represent skin. [10] suggests a more complex luminance-conditional distribution model

for skin pixel detection. The next step is to extract the areas that are possibly faces. The algorithm for this takes into account several different variables, such as orientation, center of mass, and the number of holes in a segment. Once a possible face has been extracted, it is tested against a face template. This template consists of an averaging of several different front views of faces, as shown in Figure 2.1. The template is scaled and rotated to match the segment as closely as possible, and the cross-correlation value between the two images is found. This value is then thresholded to determine if the area can be classified as a face or not.

Figure 2.1: Example of Template Used by Chang and Robles

Template matching can be a fairly accurate method of face detection, but suffers from several drawbacks. It is computationally more expensive than some of the alternatives. Since the template is generally comprised of forward facing faces, it can have trouble detecting rotated faces. Also, since the template is the main source of determining whether an area represents a face or not, any feature of the face that differs from the template can cause misses. This could include anything from the face being partially obscured to facial hair.

2.1.2 Feature Cascades

One method of face detection, set forth by [15], is to search an image for areas of contrast called features, which are then combined into a hierarchical structure called a cascade. This approach was developed as a general object detection algorithm, but given the right feature cascade it is quite effective at identifying faces. The hierarchical structure speeds up the algorithm significantly by allowing it break out of the search early if the given area does not appear to be what the algorithm is searching for.

Figure 2.2: Examples of Features Used in Feature-Based Face Detection

The features used in this algorithm represent areas of differing contrast that can be used to identify the object being searched for within the image. For example, when searching for a face, the eyes will often be darker than the cheeks below them. Features are often represented visually by rectangles that are part black, and part white, as shown in Figure 2.2. When placed on top of an image, the black area can be thought of as being weighted negatively, and the white areas weighted positively. Summing the values of all pixels within the rectangles, multiplied by their weights, will give you the value of that feature at the point in the image.

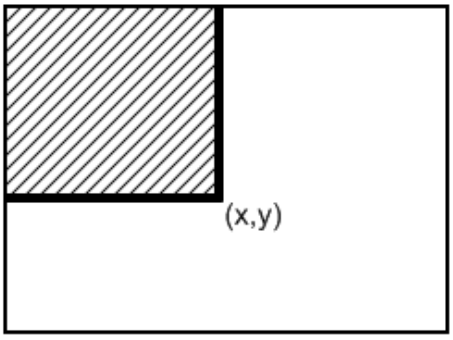

Figure 2.3: Example of an Integral Image

Calculating the values of features clearly is a bottleneck for this algorithm, and [15] introduces a new data structure to speed that process up significantly, called the integral image or summed area table (SAT). The integral image is defined as a version of an image where the value of any pixel (x,y) is defined as the sum of all pixels above and to the left of (x,y) in the original image, as shown in Figure 2.3.

This increases the calculation speed of finding the sum of all pixels in a given region from linear time to constant time. In Figure 2.4, the sum of the values within the rectangle created by points A,B,C, and D can be defined as

$$Sum(A, B, C, D) = I(A) - I(B) - I(C) + I(D)$$

Where $I(A)$ represents the value of the integral image at point A.

Besides speeding up the summing of rectangular areas by a large amount, the integral image is also quick to create, as it can be constructed with a single pass over the original image using the formula

$$F(x, y) = F(x, y - 1) + F(x - 1, y) + I(x, y) - F(x - 1, y - 1)$$

where $F(x, y)$ represents the value of the integral image at a given point (x, y), and $I(x, y)$ represents the original image. Any indexes outside of the bounds of the image are assumed to be equal to 0.

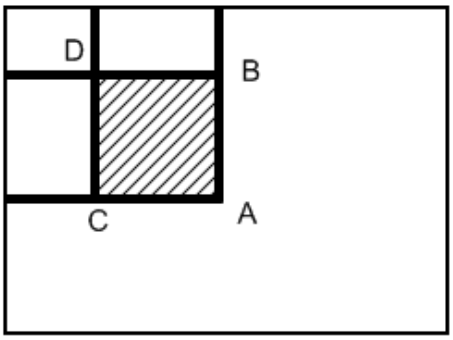

Figure 2.4: Finding the Sum of a Rectangular Area Using an Integral Image

The actual search for a face consists of using rectangular masks of varying scale. Searching at a wide variety of scales allows the algorithm to find faces of different sizes

within the image. When a scale is chosen, a rectangular mask of that size is moved across the image using a set step size. At every step, the portion of the image captured by the mask will be analyzed using the feature cascade. This search will either discover that the given rectangle within the image represents a face, or it does not. The results are recorded and the search continues after incrementing the position of the mask by the step size.

This algorithm runs quickly due to the hierarchical nature of the cascade, which eliminates a large amount of unnecessary searching. However it suffers from the necessity for a well-constructed feature cascade. Cascades can be constructed to detect many different sorts of objects, but they are limited in their scope. A cascade calibrated to detect forward facing faces will not detect a face if it is sufficiently tilted or rotated, which can prove to be a significant drawback in a tracking system.

Extended Feature Cascades

The feature cascade algorithm is extended in [11] with the addition of new types of features. The paper describes a method that allows for feature rectangles that are rotated 45°. In order to allow for rotated features, a new type of integral image had to be introduced, known as a rotated summed area table or RSAT.

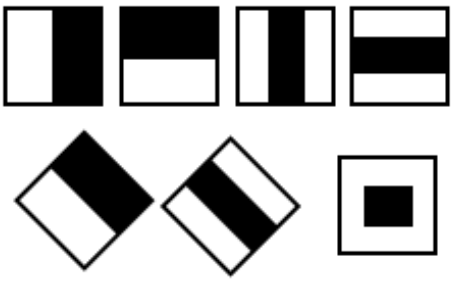

Figure 2.5: Extended Feature Set

A RSAT is used to calculate sums of areas within rotated rectangles, as shown in figure 2.6. Like the regular integral image, it is quick to calculate, requiring two passes over the original image. Using the function $R(x, y)$ to represent the value of the RSAT at the point (x, y), the first pass goes left to right and top to bottom calculating

$$R(x, y) = R(x - 1, y - 1) + R(x - 1, y) + I(x, y) - R(x - 2, y - 1)$$

where any pixel with a negative index is set to 0. The second pass goes from right to left and bottom to top calculating

$$R(x, y) = R(x, y) + R(x - 1, y + 1) - R(x - 2, y)$$

These advances can significantly improve the algorithm's results, with a minimal extra computation cost. However, it does not change the fact that a well-developed feature cascade must be created, and that the algorithm will have difficulty finding any object that does not specifically fit the criteria of objects that the cascade was created for finding.

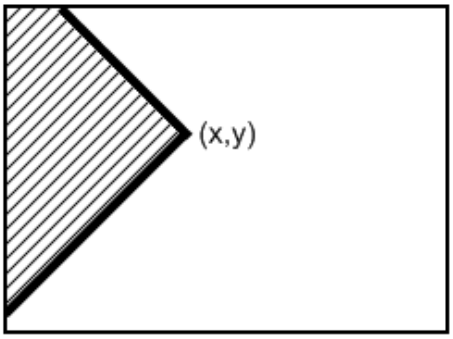

Figure 2.6: Example of Rotated Summed Area Table (RSAT)

2.1.3 Color-based Detection

Another common approach to face detection is to use a color-based approach. In [13] an algorithm is set forth that uses Bayesian analysis to determine the probability that any given pixel represents a face. The algorithm requires construction of two color histograms, one representing the entire image and another representing just the colors that can be found on the user's face. Blink detection or manual selection are recommended in obtaining the skin color histogram initially. This immediately presents two limitations of the algorithm: users with different skin colors cannot be tracked simultaneously, and the tracking may not be automatic if manual selection is selected as the method of obtaining a skin color histogram.

Once a suitable sampling of face colors has been collected, the algorithm passes

over the image and calculates a probability that any given pixel of normalized color value (r, g) represents a face. The desired probability is $P(skin|r, g)$, which can be discovered using Bayes' Theorem. Bayes' Theorem is an equation used in probability theory which states that

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

Plugging the desired probability into Bayes' Theorem gives the equation

$$P(skin|r, g) = \frac{P(r, g|skin) * P(skin)}{P(r, g)}$$

This equation can be solved using the two histograms $h_{skin}(r, g)$ and $h_{total}(r, g)$, and the total number of pixels contained in those histograms n_{skin} and n_{total}. Given those variables, estimates can be made on the values in the above equation. The probability of a color value (r, g) given that it is skin can be estimated by dividing the h_{skin} value for that color by the total number of colors in h_{skin} as shown:

$$P(r, g|skin) \approx \frac{h_{skin}(r, g)}{n_{skin}}$$

Similarly, the probability of a given color value can be estimated by dividing the h_{total} value for that color by the total number of pixels in h_{total}.

$$P(r, g) \approx \frac{h_{total}(r, g)}{n_{total}}$$

And finally, the probability of a skin pixel can be estimated by dividing the number of pixels in the skin histogram by the number of pixels in the histogram of the entire image.

$$P(skin) \approx \frac{n_{skin}}{n_{total}}$$

Plugging these values into Bayes' Theorem gives the equation

$$P(skin|r, g) \approx \frac{h_{skin}(r, g)}{h_{total}(r, g)}$$

for a pixel's probability of representing a face. Once a probability value has been found for every pixel in the image, the algorithm can lock onto any clusters that appear as probable faces.

This algorithm can be easily confused, namely by skin colored areas that are not faces within the image. False positives are avoided by assigning each pixel a weight, which is a Gaussian distribution centered on the last known position of a face. This weighting will eliminate false positives from appearing in areas of the image that do not contain a face. However, it also introduces a new issue where a particularly fast-moving face can be lost track of if it moves a large enough distance in between frames.

Another method for color-based detection, set forth by [3], involves using simple ratios between the R, G, and B values of a given pixel to determine whether it represents skin or not, followed by K-Means clustering to accurately classify the resulting clusters into discernible regions representing faces. Compared to some of the other approaches, the factors used to determine if a pixel represents skin or not is exceptionally simple. A few simple rules are followed:

- $R > G$
- $R > B$
- $G > B$
- The pixel cannot be near black, white, red, green, blue, yellow, magenta, or cyan

These simple rules form a fairly accurate filter for detecting skin colored pixels within an image, and work well on all different skin tones. However, this filter has no way of discerning if a skin colored pixel represents a face or another area of skin.

Color-based detection algorithms are in general, faster than other methods. They tend to require little calculation beyond a single pass over the target image, and possibly a clustering algorithm after that pass to classify the faces more accurately. However, this decrease in calculation has a drawback, in that color-based detectors are also generally more inaccurate and noisy in their results. Using color as the main input in determining if a pixel represents a face can produce a large amount of false positives. Given certain backgrounds that contain a lot of skin-colored areas, it can render the

algorithm almost useless. Color-based approaches also introduce the additional requirement that the image must contain color information, which renders them useless on grayscale images.

2.2 Face Tracking

The amount of work done on specifically tracking faces is significantly less than what has been done on face detection. Some of the reason for this comes from the fact that certain approaches allow face detection to be performed in real time. When tracking is not being used as a method of input, smoothness is not an issue and no solution for tracking is required beyond simply running a full face detection search on every frame of the video stream. This is a suitable solution for applications such as security software performing face recognition. They do not need a smooth tracking of the faces in a video, they simply need to identify the areas of the frame that represent faces so that they can perform further analysis on them. However, when being used for input purposes, a larger amount of work needs to be put into smoothing the results of the face detection phase, and making sure that the algorithm is also working fast enough to be responsive to the user.

[1] attempts to perform fast face tracking by using a combination of several different algorithms. Feature-based detection is used to initially find the face, with an additional step of a neural-network powered detector to verify the results of the feature-based detector. The verified face is then tracked using the Bayesian Mean-Shift algorithm. This is an iterative algorithm that analyzes confidence maps in an attempt to converge onto the peak of confidence within the image. Histograms of the background and of the area to be tracked are used to create this confidence map, in a process similar to the color-based detection set forth by [13]. Once the confidence map has been constructed, the Mean-Shift algorithm is used to find the area of peak confidence, which should represent the face being tracked. A Kalman filter is used to predict the movement of the face, in order to properly weight the confidence map towards the area which is believed to contain the face.

A Kalman filter is also used in [12] to predict the location of the face during tracking. However, this approach is not well suited to the task at hand. In these examples, the Kalman filter is used in order to predict the next location that a face will be found at,

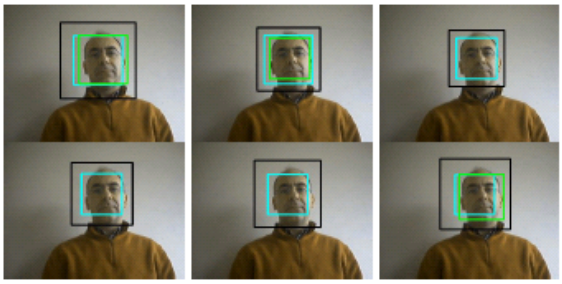

Figure 2.7: Series of frames from [1] showing tracking. The black rectangle represents the search area, the cyan rectangle is the predicted face position, and the green rectangle is the detected face position.

and restricting their search space to an area centered around that position and sized according to the diagonal values in the covariance matrix. This is used to limit the search space, and if a face is not detected in that area, then to provide an estimate of where the face is to give back to the application. This is not acceptable behavior for the task at hand because if the detector loses the location of the face, for example by the tilting of the user's head in a feature-based detection scenario, it could end up searching an entirely incorrect portion of the image. If this is the case, then it becomes quite difficult for the application to lock on to the user's face again. A similar approach, using the estimated position of the face in order to guide the search, is applicable to the problem researched here, and will be discussed further in Chapter 4.

2.3 Flash and Actionscript

Flash was originally released in 1996 as a simple vector graphics and animation application, evolving from a product called FutureSplash after it was aquired by Macromedia. It touted the large advantage of being able to produce .swf files, which could then be embedded on webpages. This made animations created using Flash able to be deployed to an enormous amount of users very easily, across several different platforms. Originally, only timelined animations could be created, but the release of Flash 2 in 1997 added a new element in the form of the scripting language Actionscript. In the beginning Actionscript could only be used to manipulate the timeline, and interactive applications were still impossible to create. But as they evolved, both Actionscript

and Flash grew into extremely important technologies, which now account for a large amount of applications, both embedded in webpages and on the desktop.

Actionscript started as a simple scripting language to control the timeline of keyframed animations. But as Flash grew, so did the language used to control it, and in 2000 Actionscript 1.0 was released with Flash 5. This provided only basic language constructs, such as loops, variables, and branch statements. It also provided for rudimentary object-oriented programming by allowing users to define objects as "prototypes," which could then be duplicated. Actionscript 2.0 was introduced in 2003 with Flash MX 2004. This implementation of the language added strong typing and an actual class syntax for more robust object-oriented programming. The most recent version, Actionscript 3.0, was released in 2006 with Flash Player 9. This was a complete object-oriented rehaul of the language. The inheritance system was re-created, with increased performance, and stricter type checking was introduced, in addition to a large amount of syntactical changes.

Actionscript is a relative newcomer in the world of programming languages, but due to the massive install base of Flash it is quickly becoming one of the most popular. Its simplicity and behind-the-scenes handling of advanced topics such as graphics make it relatively easy to learn. It is even set forth by [5] as being an ideal language for newcomers to computer science, due to the graphical nature of object creation and intrinsic support for operations such as drawing, which can be an immense task for beginners to program in a language like Java or C++. Despite all of this however, or possibly because of it, Actionscript is often not regarded as a "serious" programming language. It is considered intrinsically slow and ill suited for any sort of heavy computational task, such as image processing.

However, with the introduction of Actionscript 3.0, this is no longer the case. While Actionscript is certainly still higher level than some of the alternatives such as C/C++, with proper architecture in place it is absolutely capable of tackling a fairly advanced computer science task such as face tracking. Many available sources such as [8] focus on optimizing rendering performance, which was not an issue for this project. Fewer sources focus on the micro-optimizations that can be performed by implementing simple tasks in certain ways that, although less intuitive, run faster. This document will benchmark a few of the different optimizations made in the code, showing them to be significantly faster than alternative ways of achieving the same outcome.

2.4 Head Movement as Input

Many attempts have been made in the past towards making head and face tracking a viable alternative method of input. [14] introduces a system that uses advanced 3D model-based face tracking to use the translations and rotations of a user's head as a new method of mouse input. The authors also introduce a system to utilize mouth movements to trigger mouse events such as clicking. Their system can be used as a method of input for those with hand or motor disabilities as an alternate method for computer control.

[7] explores using face tracking as input for mobile devices. It introduces Mixed Interaction Spaces (MIXIS), in which the camera on a mobile device will track its own location in relation to a tracked object within view, and use that information as input for any number of applications. A few different challenges associated with this method of input are also described. For example, the authors found that users would often control the application by tilting the phone as opposed to moving themselves, especially in situations where the screen needed to remain visible. They also found that while lateral movement along the x and y planes was relatively simple to track, keeping track of movement in the z-axis and of face rotation proved much more difficult to accurately determine. This is especially true in any case where there is some occlusion, or the face is not entirely within the camera's view.

In [17], face tracking is presented as an enhanced method of input for some forms of video games. The paper describes experiments run by the authors that test players perception of games controlled through traditional means and by face tracking input. One game was a simple scuba diving game where the player had to collect oxygen, and the other was a modified first person shooter game that used face tracking to determining when the player's avatar should lean out of cover and attack. They found that for the simple diving game, players vastly preferred versions that used face tracking input. Testers felt that the experience was more immersive and challenging when their actual bodies were the controllers. Results were more mixed for the more complex first person shooter game, which combined both face tracking and traditional Xbox controller input. Players were not used to controlling their avatar through face tracking input, and most felt that the control scheme should be optional but not required. Interestingly enough, there was no significant difference in performance between the

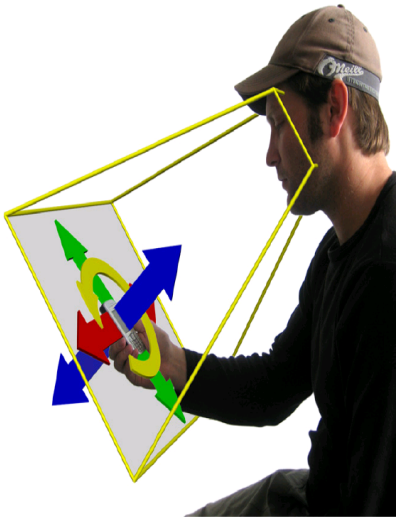

Figure 2.8: Example of Mixed Interaction Space (MIXIS) from [7]

two versions of the game. Also, the amount of horizontal movement in the players was about the same in both versions, indicating that players would subconsciously lean their bodies whether it affected the game or not.

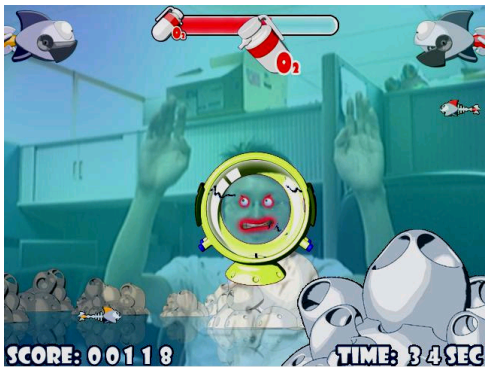

Figure 2.9: Screenshot of the Scuba Diving Game Presented in [17]

[2] also exhibits success modifying Quake 2 in order to use face tracking as the main method of input. Users are described as finding the experience enjoyable, although a small learning curve exists in which they must acclimate themselves to the new way of controlling the viewpoint.

Beyond the findings of [17] and [2], the release of Microsoft's Kinect in November

2010 show that the video gaming market is taking face and body tracking methods seriously as methods of input. The Kinect system uses a dedicated hardware accessory to perform whole body tracking, facial recognition, and voice recognition all in real time. Many game developers are taking advantage of this new technology to enhance player immersion in their games, and to create entirely new genres of games such as the exercise game.

Chapter 3

Design

3.1 Requirements

The goal of this project is to create a face tracking solution that is both accurate and fast enough to run in a Flash application, either embedded in a website or on the desktop using Adobe Integrated Runtime. Specifically:

- The algorithm has to run in real time (30FPS).
- The algorithm has to feel to the user that it accurately portrays the movements of the user's face.
- The algorithm has to be able to track the path of the user's face smoothly, minimizing any noise from the detection process.
- The algorithm must not at any point lose track of the user's face. It is acceptable to return an incorrect position if a face cannot be detected, but some position must always be returned.

Alternatively, there are a few requirements that are being overlooked for this project. Some things that this project does not aim to accomplish are:

- The algorithm does not have to match the user's movements identically. As long as the user feels that the algorithm is responding accurately, that is the most important aspect.

- The algorithm does not have to track multiple faces simultaneously.
- The algorithm does not have to track the size of the user's face, only the location.
- The algorithm does not have to do any sort of facial recognition, or facial feature tracking. Only the position and scale of the face as a whole are being monitored, nothing more.

3.2 Tracking Algorithm

When selecting an algorithm to pursue for the project, it was not immediately obvious which of the presented face detection paths should be taken in order to produce the desired results. A color-based tracker will most likely be faster, but may not be as accurate, and will also introduce more noise into the results. A feature-based cascade tracker will likely be more accurate and robust, but also more computationally expensive. Template matching or other approaches such as neural networks were immediately discarded as being too computationally expensive to be performed in real time, especially on machines that may not be state of the art.

It is for those reasons that we attempt to solve the problem using both a feature-based and color-based approach, in order to compare the results and attempt to find out which algorithm is better suited to the task of face tracking in Flash, and under what conditions they each excel.

3.2.1 Feature Cascade Algorithm

The feature cascade algorithm presented in [15] and [16] was chosen for feature-based face detection. The advantages of the extended set of features set forth in [11], while valuable, was deemed to be to much overhead for not enough gain in accuracy. The cost of creating another type of integral image to calculate what would end up being a small percentage of the feature rectangles was too much computation work to justify, when the original algorithm has an acceptable success rate by itself.

3.2.2 Color-Based Algorithm

The algorithm presented in [13] was chosen for color-based tracking. This algorithm has the advantage of tracking colors specifically found on the user's face, instead of simply searching for skin-colored pixels. This eliminates a large amount of work that is normally done by color-based detection schemes to determine which of the skin-colored regions could represent faces. It also accounts for all skin colors. In fact, since the colors are collected per user, any combination of skin color, facial hair, facial accessories, and even face paint or masks can be supported. The only drawback is that to do this, the algorithm requires a histogram of colors from the user's face. This presents a chicken and egg problem to the developer - you must know where the face is to get a sampling of the colors, but you need the sampling of colors in order to figure out where the face is. This problem was solved by initially using feature cascades to find the face, before switching to color-based tracking once an area has been determined to be a face for a sufficient amount of time.

3.3 Actionscript Optimization

When optimizing the code for Actionscript, a few major choices were made in order to increase the speed of the algorithm. These were not always intuitively faster than alternative methods of producing the same output, and in a few occasions were more complex. In order to justify the choices made in coding the algorithms, benchmarking was performed between the methods chosen and the alternative approaches. Some of the major design choices made were:

Vector vs. Array

In Flash Player 10, a new Vector class was introduced. This class represented a strongly typed list for the first time in Actionscript's history. Until Vector, the only option for creating lists was either a hand-rolled linked list implementation, or using the built in Array class, which is weakly typed. The Vector class was touted as having a significant performance increase over the Array class, at the cost of requiring the user to have Flash Player 10 installed. This cost was deemed acceptable for the added speed benefit of using the Vector class.

Events vs. Return Values

Flash uses a built-in event system to communicate between systems. It is quite robust, and often improves the readability of code significantly. A function can be set to be called whenever a given event is fired by an object. It will then be called at the correct time, and often with an accompanying object which contains information about the specific event. This is a very clean way of dealing with many different types of events, such as mouse or keyboard events, but is not as well suited to tasks which are trying to run as quickly as possible, such as the face detection portion of our tracking algorithm.

Preferring Inline Functions

Flash contains no built in support for inlining functions. This ends up becoming an issue in time-sensitive code, because the overhead of function calls can quickly become a major bottleneck of a significantly advanced algorithm. For this reason, whenever possible in the creation of these tracking algorithms, manual inlining was preferred over creating a function to perform a task. This approach has the disadvantage of decreasing the readability and maintainability of the code, so an attempt was made to limit the manual inlining to only simple tasks which could easily be performed without calling an external function. A good candidate for inlining would be something like finding the absolute value of a variable.

3.4 Face Tracking Benchmarking

More important than the design of either of the specific algorithms are the results they produce. It is for this reason that a robust benchmarking application was also created. This application uses motion captured head movement data to test the accuracy and speed of a given algorithm and classify it with four separate scores:

- Speed: The average time in milliseconds that it takes this algorithm to process an image and return the face position.
- Accuracy: A score based on the average amount of difference between the actual position of the face and the position that the algorithm returns.

- Responsiveness: A score based on the number of frames in which the algorithm returns a value which is moving in the correct direction. That is to say, if the actual face is moving to the left, and the algorithm returns a value which is either not moving, or is moving in the wrong direction, that will lower this value.
- Miss Percentage: A score added for comparisons to the base case, this represents the percentage of the time that the algorithm fails to find a face. To meet the requirements set forth in the beginning of this chapter, this value should always be 0.

Chapter 4

Implementation

4.1 Face Trackers

4.1.1 Overview

Both of the face tracking systems developed follow the same basic outline. The tracking system takes an Actionscript BitmapData object as its input, and outputs a two dimensional position representing the center of the face being tracked, or null if a face has not been locked on to yet. Both trackers employ an initial lock on period in which unchanged face detection is run on the image until a face has been detected in the same area of the image for a set amount of time. Once the system is locked on and the tracking begins, detection will be run on the input image. The values returned by the detector are then smoothed by the tracking system and returned to the application. The images given to the tracking system are assumed to be sequential video frames, and information from the previous frame is used to optimize the tracker's search.

4.1.2 Camera Input

The first issue involved in this problem is how to get a video stream from a webcam into a format that allows us to analyze it. This is done simply enough by using the built-in Camera class packaged with Flash. This class exposes a static function getCamera(), which when called will return a reference to the user's default camera device, if available. Flash will also automatically show the user a dialog box telling

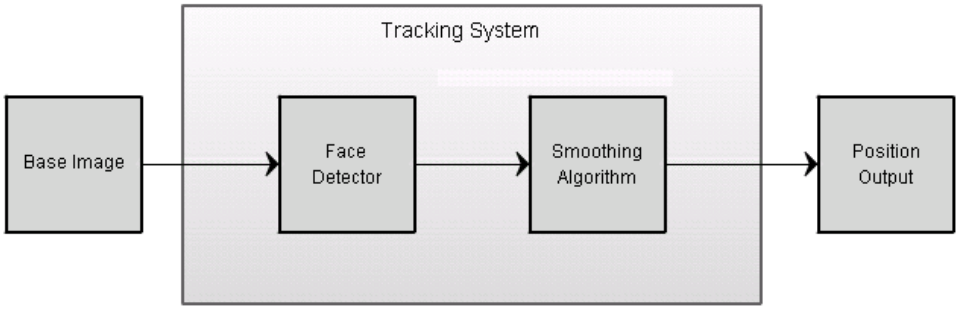

Figure 4.1: Flowchart of the System's Outline

them that the application requests access to the webcam, which they can reject. But if the user allows the application to access their webcam, this function will return a valid Camera object.

Once a Camera object has been obtained, it must be attached to a Video object. Video is a graphical class that will allow the video stream contained within the Camera object to be displayed. The Video.attachCamera() function can be used to attach the Camera to the Video object.

Lastly, for any sort of image processing to be performed on the video stream, it is necessary to have pixel access to the image. This is only possible if the graphic is within a BitmapData object. To get that, the Video object must be drawn into a BitmapData object, using the BitmapData.draw() function. After this has been performed the BitmapData object will contain the current frame of video, which can then be analyzed at a pixel level.

4.1.3 Feature-Based Tracker

Outline

The feature-based tracker utilizes the detection scheme described by [15]. This object detection algorithm is implemented within the OpenCV library, which is an open source library with a variety of useful functions relating to computer vision. The object detection section of OpenCV had previously been ported to Actionscript 3.0 by Ohtsuka Masakazu for the also open source Spark project (http://www.libspark.org). This code was later optimized for Actionscript 3.0 by Mario Klingemann (http://www.quasimondo.com/).

This optimized code embeds the feature cascade data within a class, which eliminates the need to load it from an external file, in addition to optimizing some of the data structures used. This code produced by Klingemann was used as the basis for the feature-based tracker.

The optimized tracking algorithm uses coherence between frames quite a bit in making decisions, so before the tracking is activated it is important that the previous frame information is quality. For this reason the algorithm runs using an initial period where it locks on to the user's face. During this lock on period, none of the tracking algorithm is used, and all that is running is the basic feature-based detection algorithm. Once the detection algorithm finds a face in the same general area of the video for a set amount of frames, it is then considered to be locked on, and the tracking algorithm begins.

Optimizations

Several optimizations to the algorithm have been included in order to increase the performance of this algorithm, and allow Flash to run it as quickly as possible. Most of these optimizations are related to assumptions that may or may not be applicable to a given situation, and it is advised that the reader carefully consider which assumptions are applicable to their application if attempting to reproduce this algorithm.

The main efforts put forth towards optimizing the base code, as opposed to the algorithm, follow the design decisions laid out in Chapter 3. First was to replace all instances of the Array class with the Vector class. None of the code made use of the ability of Array objects to hold different types of data in each index, so that was simple. The other change made was to switch the code from using the Flash event system to simply using return values. Originally, the code would fire an Event whenever a face was detected. This was changed to simply return the position of the face that was found from the trackFace() function.

In terms of optimizing the algorithm, the first and possibly most important optimization made is that when a valid face is found, the tracking algorithm immediately stops searching for further faces. This comes from an assumption put forth in the Requirements section of Chapter 4, which states that the tracking algorithm need only worry about tracking a single face at a time. If multiple faces were being tracked then

a thorough search of the entire image would be required, which would be extremely time consuming. Breaking out of the search early also allows for another class of optimization: since we know that we will stop searching after the first face is found, we can now focus on attempting to find that first face as quickly as possible.

Utilizing coherence between frames, we can attempt to find the face as quickly as possible in order to end the search. Some simple ways of doing this are to begin our search in the area of the image where the face was previously, and at the scale it was previously. Since we can assume that it has not moved too far in between frames, this will most likely result in finding the face fairly early on in the search.

Another approach to optimization is to try to detect when a face is not going to be found, and try to end searching as early as possible in this situation instead of exhaustively searching the entire image only to find nothing. This will often occur when the feature cascade is optimized for forward facing faces, and the user turns or tilts his or her head to the side. To prevent an exhaustive, time-consuming search when a face will most likely not be found, an optimization is put in place that only searches a subset of the possible scales. A full search will attempt to find faces at many different scale sizes, but by limiting the amount of scales searched to those surrounding the last known scale at which a face was found, quite a bit of unnecessary searching can be eliminated.

The last optimization is both an optimization method and increases the accuracy of the result. The basic detection algorithm chooses a scale rectangle to search, and proceeds to step that rectangle across the image in small increments. This means that the results will only fall on certain boundaries that are divisible by that step size. However, utilizing coherence between frames will give us an area where the face is most likely to appear. An adaptive step size can then be implemented, which uses very fine steps in the area where the face was last found, but large coarse steps in other areas of the image. This will both eliminate stepping artifacts in the results and decrease the amount of time spent searching areas where a face will most likely not be found.

Smoothing Techniques

The results of the face detection algorithm can be quite noisy between seemingly identical frames, due to slight changes in illumination and noise in the camera stream. For

this reason smoothing techniques must be employed to remove that noise, and in order to make the results smoothly track the user's face.

One smoothing technique is to keep track of a current location for the face; and when a new location is returned by the detection algorithm, interpolate our current position to the new one. This provides the benefit of never having our tracked location jump from one spot to another. The smooth interpolation from location to location provides much smoother feedback and a more natural response from the user's perspective.

Another obstacle to giving the user smooth reliable feedback is that sometimes the algorithm fails to find a face. This can occur if the user tilts his or her head or it is obscured for a brief moment. When the algorithm fails to find a face, this is dealt with by returning the last known position. This means that the tracking algorithm is guaranteed to return a location, which provides the benefit of giving the user feedback, and decreases the feeling that the computer has lost track of his or her face, even in cases where it has.

4.1.4 Color-Based Tracker

Outline

The color-based tracking implements a detection scheme quite similar to [13]. Similar to the feature-based tracker, this approach utilizes an initial lock on period. For the purposes of the color-based tracker, this lock on period is necessary to collect color information about the user's face before attempting to run the color-based detection scheme. Feature-based detection is used initially, and once the detection algorithm has found a face in the same area of the image for a set amount of frames, that area is used as the basis for the histogram containing a sample of face colors. Once the histograms of colors contained within the image and within the face are constructed, the algorithm can begin working as described in Chapter 2.

Optimizations

One of the most significant optimizations made on the color tracking algorithm is performing the analysis on a significantly down sampled version of the image. Decreasing the amount of pixels needed to test increases performance by a large margin while

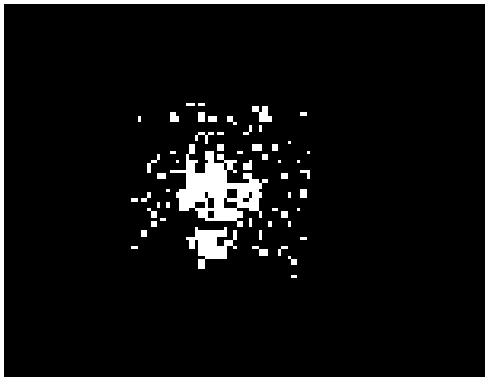

Figure 4.2: Example of Thresholded Image in Color-Based Detection

maintaining accuracy in detection. A small amount of down sampling actually has a positive effect on detection by reducing the amount of false positive pixels that will be found, and overall reducing the amount of noise in the thresholded image.

Another optimization made was to change the pixel weighting scheme from what was recommended in [13]. That paper recommends weighting pixels using a Gaussian distribution centered on the last known position of the face. This algorithm provided a slight increase in performance by simply using a linear weighting scheme based off of the distance from the current pixel to the last known face position. This provides similar results while decreasing the amount of calculation necessary to find the weight for any given pixel.

A simplification of the requirements for the algorithm also provide an implicit speed increase. The requirements laid out in the beginning of Chapter 3 say that the algorithms produced do not need to track the size of the faces, only the location. This simplifies the amount of work required to be performed on the thresholded image significantly. If the size of the face was a required output, then some sort of cluster detection algorithm would have to be performed, possibly in addition to erode and dilate operations that may be necessary to reduce noise and fill in gaps in the thresholded image. Since this was not required, a large amount of image processing was eliminated, which keeps the running speed manageable.

Finally, the method of determining the face position from the thresholded image is simplified from the original proposal. [13] proposes a grouping algorithm to find the position and extents of the face region. However, namely since the extents are not a

required output for our algorithm, this approach can be simplified to increase running speed. A simplification to their approach can be made by simply averaging the position of all the activated pixels in the thresholded image. Since outlying pixels are largely eliminated by using the weighting scheme, this provides a good approximation of face position.

Smoothing

A linear interpolation from the previous position to the current position is used as the smoothing mechanism for the color-based tracking scheme. Despite the results of color-based detection being fairly noisy, they tend to not stray too far from the position of the face. This means that simple interpolation for smoothing is actually provides a moderately stable position, with only a small amount of visual jitter when the face is held still. Further smoothing was deemed unnecessary, as it would have only made the algorithm run slower for a minimal increase in the smoothness of the tracking.

Unlike the feature-based approach, there is no need to plan for the situation where the detection algorithm does not detect a face. Due to the nature of the algorithm, some sort of value is always going to be returned. However, when no face is present within the image the results consist of an average position of all false positive pixels, which becomes an essentially random position. For this reason, a failure condition is included in the algorithm where if the percentage of positive pixels within the thresholded image drops too low, the detection scheme is deemed to have failed and the last known face position is returned. This, at the very least, prevents the algorithm from returning extremely varied, random results when no face is present.

4.2 Face Tracking Benchmarking

A proper method of testing the algorithms was necessary, which prompted the creation of the benchmarking application. This uses motion captured data of head movement to simulate a video stream, and check the tracking algorithm's results against the actual values.

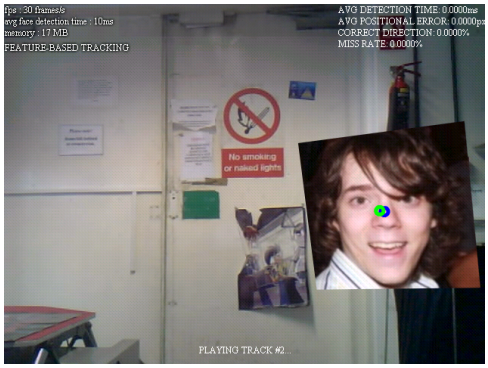

Figure 4.3: Screenshot of the Benchmarking Application

4.2.1 Motion Capture Data

The most accurate test would ideally be against an actual video of head movement, but it would be impossible to know the actual location of the face to compare the tracker's results against in that case. For this reason, motion capture data is used to simulate a video stream. Several different captures of varying head movements were recorded, and the information is read in and used to place and orient an image of a face in 3D space within the application. This is then passed into the tracker as though it was a video stream, and the resulting output can be compared to the motion capture data projected into 2D space in order to find the error produced by the tracker. There is also an option to display an image of a noisy background behind the face image, in order to more accurately simulate a real-life situation.

4.2.2 Performance Testing

The benchmarker tracks several different performance metrics from the tracking algorithms. For every different metric, the average value over the entire motion capture is recorded, in addition to the best and worst case values. The different values tracked are:

- Time per detection (ms): This measures the amount of time the tracker spends actually determining the location of a face, and performing any smoothing.
- Positional Error (px): This measures how far the position returned by the tracker is from the actual position of the face.

- Directional Error (%): This measures how often the tracker is moving in approximately the same direction as the actual face. The higher this percentage is, the more responsive and smooth the tracking will feel to the user. Directions being approximately equal is defined as an angle of less than 30 degrees in between the two motion vectors.

- Miss Rate (%): This measures how often the tracking algorithm fails to find anything. For the tracking algorithms put forth, this value should always be 0% since both algorithms have built-in mechanisms to handle the situation when the detector fails to find a face.

Chapter 5

Evaluation

This section presents and evaluates the results of the algorithms described in the paper. All results are tested on a desktop computer with a 2.67Ghz Intel Xeon processor, 3GB of RAM, and running Windows XP SP3. All tests were run in Flash Player 10.1 running in release mode. It is worth noting that this is a fairly good computer, and some of the timing results will seem fast. However, the aim of the project was to create algorithms that can run, using Flash, on a wide variety of hardware that may or may not be very powerful. For this reason the algorithms should be required to run even faster on fairly high-end machine.

5.1 Face Tracking Accuracy Results

5.1.1 Base Case

In order to adequately determine the effectiveness of the additional tracking logic and optimizations performed, a base case must be established. For this project, the base case is considered to be the feature cascade object detection code from Ohtsuka Masakazu and Mario Klingemann, which embeds the "haarcascade_frontalface_alt.xml" file included with OpenCV. The results against a black background run at a fair speed, averaging around 16.96ms per detection. It is also quite accurate, averaging only 7.56 pixels of error. However, for tracking purposes the results are fairly poor, with the directional error averaging at 24.65% and a 29.76% miss rate. Against a noisy background the results degrade even further. The average detection time increases to 21.89ms. The

average amount of error rises to 10.5 pixels. The correct direction percentage drops slightly to 22.64%. Interestingly, the average miss rate improves to 28.02%. This improvement can be attributed to the algorithm finding false faces in the background in situations where it does not detect the actual face, which is not an issue with the solid black background.

Motion Capture Track	**Average Detection Time (ms)**	**Average Positional Error (px)**	**Correct Direction Percentage**	**Miss Percentage**
1	16.4	7.49	19.94	29.14
2	17.68	7.561	26.42	31.33
2	16.79	7.654	27.6	28.82

Table 5.1: Benchmarked Results for Feature Cascade Detection, No Tracking, Black Background

Motion Capture Track	**Average Detection Time (ms)**	**Average Positional Error (px)**	**Correct Direction Percentage**	**Miss Percentage**
1	21.21	9.729	19.46	27.98
2	23.6	13.34	23.4	28.65
2	20.86	8.43	25.07	27.43

Table 5.2: Benchmarked Results for Feature Cascade Detection, No Tracking, Noisy Background

5.1.2 Feature-Based Tracking

The feature-based tracking significantly outperformed the base case in every field besides accuracy, and only was slightly behind in that field. On the solid black background, it averages 7.841ms per detection. The error averages at 15.43 pixels. The average correct direction percentage is 55.98%, a massive improvement over the base case. Finally, in keeping with the requirements of the algorithm, the miss percentage is a constant 0%, as even when the face is too tilted to detect, the algorithm will return the last known face position.

Results degrade only slightly when the test is performed against the noisy background. The average detection time rises a small amount to 9.97ms. The average error

remains almost the same at 15.68 pixels. The average correct direction percentage takes a small dip to 54.99%, and the average miss percentage remains 0%.

Motion Capture Track	**Average Detection Time (ms)**	**Average Positional Error (px)**	**Correct Direction Percentage**	**Miss Percentage**
1	7.822	13.23	51.61	0
2	8.191	16.38	51.4	0
2	7.51	16.67	64.95	0

Table 5.3: Benchmarked Results for Feature Cascade Tracking, Fully Optimized, Black Background

Motion Capture Track	**Average Detection Time (ms)**	**Average Positional Error (px)**	**Correct Direction Percentage**	**Miss Percentage**
1	10.36	15.48	48.82	0
2	10.82	17.37	50.72	0
2	8.742	14.2	65.45	0

Table 5.4: Benchmarked Results for Feature Cascade Tracking, Fully Optimized, Noisy Background

5.1.3 Color-Based Tracking

The color-based tracking algorithm also significantly outperformed the base case. On a black background, the color-based tracking averaged 9.366ms per detection. The average error was slightly higher than the other algorithms, averaging 16.24 pixels. The average correct direction percentage was the highest of the approaches at 75.2%. As with the feature-based tracking, the miss percentage remained at a constant 0%.

The color-based tracking was almost unaffected by the noisy background, which makes sense since the algorithm requires a constant amount of work per-pixel regardless of the background. The average detection time stayed approximately the same at 9.068ms. The average error was also very similar, at 17.25 pixels. The average correct direction percentage dropped slightly to 74.62%, and the average miss percentage remained at 0%.

Motion Capture Track	Average Detection Time (ms)	Average Positional Error (px)	Correct Direction Percentage	Miss Percentage
1	9.492	13.05	67.96	0
2	9.277	16.89	73.98	0
2	9.329	18.77	83.66	0

Table 5.5: Benchmarked Results for Color-Based Tracking, Fully Optimized, Black Background

Motion Capture Track	Average Detection Time (ms)	Average Positional Error (px)	Correct Direction Percentage	Miss Percentage
1	9.049	15.15	67.96	0
2	9.054	17.81	75.05	0
2	9.101	18.8	80.86	0

Table 5.6: Benchmarked Results for Color-Based Tracking, Fully Optimized, Noisy Background

5.1.4 Best/Worst Case Running Speeds

The averages of the various metrics are valuable, but it is also worth investigating the best and worst cases for running speed. These best and worst case values are recorded over runs of all three of the different motion capture tracks against the noisy background. As the results show, the color-based tracking has a very small amount of deviation from the average value. This is a reasonable result, as the color-based detection requires an almost identical amount of work per search, simply passing over every pixel of the downsampled image. The feature-based approaches show a much larger deviation from the average. This is also intuitive, as the feature-based search can spend a lot of time searching an area before determining that it does not represent a face. If there is a particular area of the background which fulfills the requirements of many of the features in the cascade, but not enough to fully qualify as a face, quite a bit of time can be wasted investigating the area. These results also show a hidden benefit of the feature-based tracker, which is that the best case scenario for that approach is significantly faster than any of the others. This scenario is when the face is not moving very much. Since the feature-based tracker begins its search at the previous face location, if the face is very close to that position it should be found

almost immediately, making the search take almost no time at all.

Method	**Best Case (ms)**	**Average (ms)**	**Worst Case (ms)**
Base Detection	12	21.89	40
Feature-Based Tracking	3	9.97	30
Color-Based Tracking	7	9.068	13

Table 5.7: Comparison of Best and Worst Case Running Times

5.2 Face Tracking Result Evaluation

The results largely followed the hypotheses set forth when choosing the two approaches: the feature-based tracking was slightly slower but more accurate, and the color-based tracking was faster but less accurate. In both solid and noisy background tests, the feature-based tracking performed approximately 2ms slower than the color-based tracking. Also on both backgrounds, the color-based tracking was approximately 5 pixels less accurate than the feature-based tracking. But in reality, neither of the algorithms fared particularly poorly in either field. 2ms is a very slight improvement to the running speed, and the average positional error in both cases is quite likely not enough to be detectable when using such an imprecise input method as face tracking. The results from these two metrics is not enough to show one approach to be a signficant improvement over the other.

However, the correct directional percentage results for the two approaches do differ significantly. The average correct direction percentage for feature-based tracking is 69.36% on the black background and 66.04% against a noisy background. The color-based tracking manages to score approximately 75% on both backgrounds. The results of having a high score in this field is face tracking that feels responsive and accurate. When using head movement as a form of input, the actual position returned is not nearly as important as accurately portraying any face movement, specifically the direction. Higher values in this field gives proof for the feeling gained by the author that the color-based tracking performs more accurately, even when results show that it actually does not.

As with any computer science problem, on approach cannot be deemed to be overall "better" than the other. The choice, as always, depends on the problem at hand. When using the face tracking as a form of input, the color-based tracking will often feel more responsive and is capable of handling tilted faces significantly better. However, when accuracy is truly valued and speed or responsiveness is not as large of a concern, the feature-based tracking will be a better choice. The feature-based approach will also be a good choice if the face is not expected to move very much, as the best case scenario for that tracker is significantly faster than any of the other approaches. Both manage to outperform the basic detection by a large margin, and that should only be considered when accuracy is of the utmost importance.

5.3 Actionscript Performance Results

Some of the major design decisions were laid out in Chapter 3, but before implementing them it was necessary to prove that those approaches were going to be faster than the alternatives. This was done by creating a simple timing application which compares the timing of simple examples of each approach. This program recorded the running time of two functions performing the same job using different approaches, over 500,000 iterations. The results, shown in Table 5.8, were used to guide the design decisions made while coding the algorithms.

Method	**Running Speed (ms)**
Vector.<T>	1276
Array	1421
Return Values	24
Events	495
Inline	2
Function	26

Table 5.8: Comparison of Running Speed of Design Options

Chapter 6

Conclusion

6.1 Future Work

6.1.1 Face Tracking Improvements

There are several improvements that can be made to the face tracking algorithms set forth by this document. Some examples include:

- This project completely ignores the aspect of tracking a face's size. Lateral movement was deemed to be the most important aspect of tracking, and tracking the size of a face was deemed to be too time consuming, especially in the case of color-based detection. However this is still an important aspect of a face detection scheme, and should be addressed.

- This project also includes no level-of-detail considerations. The algorithms presented in this paper aim to be one-size-fits-all solutions. However, a face tracking approach that improves or degrades its accuracy depending on the amount of time it is taking to run could be a very novel and scalable approach to this problem.

- Only two possible approaches of many are considered in this paper. It is entirely possible that one of the various other approaches to face detection could be adapted in order to make a faster, more accurate tracking scheme. In fact, this entire project revolves around modifying an existing face detection algorithm to perform smooth tracking. It may be the case that the slightly different problem

of smoothly tracking a face can be best solved by an entirely new algorithm, which is unsuited to basic face detection tasks.

- The algorithms developed make the assumption that only a single face is being tracked. This is a fair assumption when using the results to control a small application such as the parallax viewer, but for projects of a larger scale this assumption will most likely not be acceptable. A large area of research that uses face tracking is that of security applications attempting to perform face recognition. Trying to track a large amount of faces found on security camera footage is a much larger undertaking than what was proposed here, and would require a much different approach.

- The algorithms implement no sort of future position prediction for when a face is lost. Very accurate tracking could be achieved by developing an algorithm to predict where a face is going to end up based on previous positions. The head is a complex object that does not move predictably in all situations, so such an algorithm would be quite complex to create, but would be able to increase the accuracy of tracking significantly in situations where the detector is failing.

6.1.2 Flash

There has been very little critical research into the Flash platform. However, as it continues to gain acceptance in the computer science community, more tasks that were previously considered unsuitable for the platform are going to be ported over, and architectures that take advantage of Flash's intricacies are going to be developed. This will allow the applications created using such approaches to reach a much wider audience than they could originally. They will be able to span all major operating systems, and very soon mobile devices running Android, using a single codebase. Since Flash is a continuously developing platform, this research will hopefully illuminate areas in which the compiler and language developers can improve performance, such as function inlining.

6.2 Face As An Input Method

Using face movement as a method of input is still an idea in its infancy. Solutions exist for people with motor disabilities, but for general use face tracking is seen more as a novelty form of input than an accurate, acceptable solution. However, with the advent of camera-based controllers in the video game industry, specifically Microsoft's Kinect, alternate forms of input are receiving more and more attention. The success of Kinect could be the catalyst for an influx of research into the areas of alternative input methods. Hopefully this work will influence future research in the field of face detection to place more of an emphasis on obtaining smooth, accurate results in order to allow for face movement to be a viable form of input.

6.3 Conclusion from Algorithm Development

This project attempted to develop two algorithms that would provide fast, accurate face tracking solutions for applications. The algorithms developed were based on two very different approaches to face detection, one using a feature-based solution and the other analyzing the colors of the image. Both algorithms were optimized, both in terms of optimizing the approach towards detecting the users face and limiting the amount of time spent searching, and in terms of optimizing the code for the specific platform of Adobe Flash. The results from the detection phase were then smoothed in order to provide the user with smooth, accurate tracking of their face movements. Both algorithms ended up performing significantly better than the base case of a simple face detection scheme, both in running speed and in accuracy of tracking. Both of the algorithms meet the requirements set out at the beginning of Chapter 3, which include running in real time, minimizing the amount of noise in the results, and always providing the user with feedback even when the underlying detection scheme fails to find a face in the video stream.

The benchmarking application results show that the developed algorithms vastly outperform the base face detection. They are significantly more accurate, and the smoothing aspects allow them to accurately mirror the direction of user input, although they do fall short in terms of positional accuracy. However, when using a method of input as imprecise as head movement, it becomes difficult for a user to detect positional

inaccuracy. The most important aspect of tracking, for input purposes, is to match the user's movements in terms of direction and velocity - and the developed algorithms are able to do that. The parallax viewing application runs well and feels natural using either of the tracking schemes.

Bibliography

[1] Ognian Boumbarov, Strahil Sokolov, Plamen Petrov, Anatoly Sachenko, and Yuriy Kurylyak. Kernel-based face detection and tracking with adaptive control by kalman filtering. In *IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, 2009.

[2] Gary R. Bradski. Computer vision face tracking for use in a perceptual user interface. In *IEEE Workshop on Applications of Computer Vision*, 1998.

[3] Rob Byrd and Balaji Ranjani. Real time 2d face detection using color ratios and k-mean clustering. In *ACM Southeast Regional Conference*, 2000.

[4] Henry Chang and Ulises Robles. Face detection. Technical report, 2000.

[5] Stewart Crawford and Elizabeth Boese. Actionscript: A gentle introduction to programming. *Journal of Computing Sciences in Colleges*, 2006.

[6] Amir Faizi. Robust face detection using template matching algorithm. Master's thesis, University of Toronto, 2008.

[7] Thomas Riisgaard Hansen, Eva Eriksson, and Andreas Lykke-Olesen. Use your head exploring face tracking for mobile interaction. In *ACM Conference on Human Factors in Computing Systems*, 2006.

[8] Adobe Systems Incorporated. Optimizing performance for the flash platform. Technical report, 2010.

[9] Dong-gil Jeong, Yu Kyung Yang, Dong-Goo Kang, and Jong Beom Ra. Real-time head tracking based on color and shape information. *Image and Video Communications and Processing, Proc. of SPIE-IS&T Electronic Imaging*, 2005.

[10] Zhong Jin, Zhen Lou, Jingyu Yang, and Quansen Sun. Face detection using template matching and skin-color information. *Neurocomputing*, 2007.

[11] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *International Conference on Image Processing*, 2002.

[12] Paulo Menezes, Jose Carlos Barreto, and Jorge Dias. Face tracking based on haar-like features and eigenfaces. In *5th IFAC Symposium on Intelligent Autonomous Vehicles*, 2004.

[13] K. Schwerdt and J.L. Crowley. Robust face tracking using color. In *4th IEEE International Conference on Automatic Face and Gesture Recognition*, 2000.

[14] Jilin Tu, Thomas Huang, and Hai Tao. Face as mouse through visual face tracking. In *Proceedings of the Second Canadian Conference on Computer and Robot Vision*, 2005.

[15] Paul Viola and Michael J. Jones. Rapid object detection using a boosted cascade of simple features. In *Conference on Computer Vision and Pattern Recognition*, 2001.

[16] Paul Viola and Michael J. Jones. Robust real-time face detection. *International Journal of Computer Vision*, 2004.

[17] Shuo Wang, Xiaocao Xiong, Yan Xu, Chao Wang, Weiwei Zhang, Xiaofeng Dai, and Dongmei Zhang. Face tracking as an augmented input in video games: Enhancing presence, role-playing and control. In *Conference on Human Factors in Computing Systems*, 2006.