DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Using AR to Help Learn Songs on a Physical Piano

Linus Seidler

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Using AR to Help Learn Songs on a Physical Piano

Augmented-Reality Anwendung, die Noten über der Klaviatur einblenden kann, um dem Nutzer beim Üben zu Helfen

Author:	Linus Seidler
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Christian Eichhorn, M.Sc.
Submission Date:	17.06. 2019

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 17.06. 2019 Linus Seidler

Abstract

Augmented Reality (AR) as an emerging technology provides a novel way to represent and transfer information. In particular, AR can and has been utilized to help a user learn songs on a piano. In this thesis, the design and implementation of an AR piano learning application prototype are described which is intended to run on a smartphone-powered Head-Mounted Display (HMD). A unique approach for the detection of piano keys by analyzing the keyboard's natural features is presented, which lifts some restrictions of previous approaches. It utilizes a novel monochromatic marker detection specifically designed for this task. To achieve this, the tracking algorithm was implemented using native OpenCV and deployed as a self-sufficient library while the application prototype was created using Unity3D. For the integration of the library into the prototype, low-level means of communication were implemented. Finally, the system was successfully deployed on Android. It is shown that a tailored natural feature tracking algorithm can be superior in terms of accuracy and performance compared to a generic, purely marker-based, solution.

Contents

1 Introduction

Augmented Reality (AR) as an emerging technology, which is predicted to grow at an accelerated rate [1], has gradually become more consumer accessible over the past few years. No longer is such technology exclusively confined to be employed in the industry or by people who can afford expensive prototypes. Instead, cheaper Head-mounted displays (HMD) targeted at the consumer market have been developed and even more devices are under development. Thanks to these devices especially, the share of the population owning an augmented-reality device is expected to multiply in the next couple of years. This enables a rising number of people to benefit from augmented reality applications with unique advantages over non-AR-applications.

Music is present in virtually every society and indisputably plays an important role in their cultures. It is also a fundamental form of human expression and a core human experience [2]. Naturally being able to play an instrument is still a desirable and essential skill. Additionally, it has been shown statistically that learning an instrument improves cognitive skills and is associated with higher conscientiousness, openness, and ambition [3].

When learning to play a new piece on the piano, one of the biggest and most common difficulties for a beginner is reading the sheet music. A system displaying the score information directly on the keyboard using augmented reality technology will be very helpful for any piano player who can not read sheet music very well. It could ultimately enhance the player's practising experience and even facilitate and accelerate their learning progress. It has already been shown that the use of augmented reality in music education, although challenging, has several benefits [4].

Computer-based music tutoring systems have been researched since the 1960s and developed further into working applications since the late 1980s [5]. When augmented reality technology emerged, its feasibility for music tutoring system was soon tested. A number of applications have demonstrated the possibilities of AR like the *Augmented Reality Bass Guitar* in 2003 [6], *Music Education using Augmented Reality with a Head Mounted Display* in 2013 [7] and *HoloKeys* in 2017 [8] just to name a few. The benefits of such systems have been shown and proven, nevertheless they are very poorly distributed at present. This in turn can be attributed mainly to the poor end-user distribution of high-grade AR-HMDs, on which most said systems operate. Cheaper, lower-grade displays developed for the consumer market, like many different smartphone-based systems, are much more prevalent. An augmented-reality-based piano tutoring application developed or adapted to run on such a HMD would make the system affordable and reach a wider consumer base. Consequently, testing the applicability

of smartphone-powered HMDs for such an application is of interest.

When reviewing existing applications, it is noticeable that a central focus lies on teaching aspects. Those questions of how to best motivate a player, analyze their weaknesses and mistakes and provide useful feedback and guidance can be considered rather psychological or pedagogical problems, which can be separated from many of the technological challenges. Because of their focus on tutoring, applications may have technological shortcomings regarding the applicability of the system or the visual quality of the augmentations. The biggest technological challenge is undoubtedly the tracking of the piano keyboard, which determines the visual accuracy of augmentations. All of the reviewed applications with the exception of a single proposed system, namely *Piano AR* by Huang et al. [9], rely solely on markers placed on the piano for tracking. Most of them even only use a single marker, resulting in two major downsides: first and foremost, the marker is required to be fully contained in every frame of the tracking camera, which is unsuitable for full-sized pianos. Second, state-of-the-art marker detection is not precise enough for establishing a homography on an object far larger than the marker itself. Virtual objects can not be mapped to physical objects far away from the marker with great accuracy. Huang's markerless piano detection algorithm requires the whole piano to be visible in every frame, which is also unsuitable when tracking keys of a fullsized keyboard.

A detection algorithm capable of tracking a keyboard's natural features could achieve a better accuracy across the entire width of a grand piano and work well with narrower camera captures. Designing an algorithm tailored to this task could allow for improvements in performance, too. It could act as the basis of a future system as well as an improvement for existing applications. It is therefore of importance to explore if and how such a detection algorithm can be implemented and exploited to lift current technological limitations. The goal of this thesis is (1) to develop a natural feature tracking algorithm for piano keys as a self-sufficient module, (2) to develop a simple prototype for a piano tutoring application for a smartphone-powered HMD on Android and (3) to integrate the tracker module into the application and thereby demonstrate how the tracker could be used in other applications.

2 Related Work

2.1 First Computer-based Music Teaching Systems

Music tutoring software has been researched early on since the increased distribution of personal computers. One of the first systems, called the *Piano Tutor*, was presented in 1990 by Dannenberg et al. [10]. It is a highly interactive system which targets beginning piano students. Concepts are conveyed with video presentations which cover a small amount of new material with each new presentation. The student will then be given a piece to play with the sheet music displayed on the computer screen in musical notation. The student is required to play on a MIDI-keyboard connected to the computer, so the application can analyze and compare what was actually played.

The authors aim at providing very detailed feedback, which is also transmitted by voice, instead of just listing the errors made by the student. As an example, it is laid out that the program can speculate about a wrong note and derive a cause, such as difficulty playing a skip. The Piano Tutor also detects when the user has mastered a piece and is ready to move on, and in that case presents the next content. Each block of content, called a lesson, includes a presentation and a performance which the student has to master. The lessons are not organized in a strict sequence but instead focus on different concepts and can be worked through in different orders based on the students interest. The final concept and conclusions of the Piano Tutor were presented in 1993 after the project was completed [1]. The feedback from users testing the system was consistently positive and the progress could be validated by a human tutor.

The usability of augmented reality for visual representation in a music teaching application has already been tried and tested early on, . *An Augmented Reality Based Learning Assistant for Electric Bass Guitar* was presented in 2003 by Cakmakci et al. [6]. In the application, a marker is placed upon the bass guitar and onto a finger of the user. The bass part of a score is played one note at a time, and each note is marked visually with a red mark on the fingerboard. The system waits for the user to put his finger on that note and then continues. Thereby, the user's searching process for individual notes can be eliminated, as information is placed right where it is needed. In contrast to Dannenberg et al., the system does not try to convey a complete musical education. Instead, a goal stated by the authors is "to accelerate the process of associating a musical score, sounds represented by that score and the fingerboard". They value an accelerated motor development more than a logical understanding of musical

[1] Different, variously detailed, reports on the piano tutor project can be found on the following website: `https://www.cs.cmu.edu/~rbd/bib-ptutor.html`

concepts in their system.

2.2 Augmented Reality based Piano Tutoring Systems

As AR has become more popular during the years, a variety of proposed systems using this technology have emerged. Notable systems using a HMD were presented in *Music Education using Augmented Reality with a Head Mounted Display* (2013) by Chow et al. [7], *Music Everywhere – Augmented Reality Piano Improvisation Learning System* (2017) by Glickman et al. [11], *HoloKeys: An Augmented Reality Application for Learning the Piano* (2017) by Hackl and Antes [8] and *Piano AR* (2011) by Huang et al. [9]. Rogers et al. presented a system managing without a HMD in their paper *P.I.A.N.O.: Faster Piano Learning with Interactive Projection* (2014) [12].

Chow et al. address the problem of lacking student interest and motivation and state that beginner students are the largest user group, and would likely benefit most from an affordable system which is fun to use. They were the first to describe the use of a HMD for a consumer oriented system. Their setup consists of a video-see-through HMD with an attached camera and a MIDI capable keyboard which are both connected to an external computer for processing. Hackl and Antes present a streamlined, fully untethered system running on the Microsoft Hololens, which results in increased usability. They allow the user to input custom musical pieces [2]. Glickman et al. propose a similar system running on the Hololens aswell but additionally make use of a MIDI-over-Bluetooth-enabled electric piano in order to detect key presses. They also include concepts associated with computer games. Huang et al. describe an application which functions without any marker placement in the scene. Rogers et al. present a system notable for displaying augmentations without a HMD. It requires a fixed setup with a video projector mounted above the piano, which has a flat panel mounted behind the keyboard on equal height. The projector displays augmentations directly onto the keyboard and the panel. Key presses can be detected via a MIDI-interface and are also used for user input. A system setup like this has several significant advantages over the use of HMDs with the current technology. It has no limitations regarding the field of view (FOV), it does not require the user to wear anything and it can provide a much better resolution for augmentations. Additionally, tracking of the piano keyboard is not necessary. On the other hand, the system is much more complicated to setup and requires more space.

All of the applications aim at helping a user to learn to play a full piece on the piano by conveying the information of the sheet music, which is written in musical notation, in more intuitive ways.

[2] New pieces can be loaded by the user as MIDI files and are processed using the open-source library 'C# Synth Project' (`https://archive.codeplex.com/?p=csharpsynthproject`)

2.2.1 Visualization Approaches

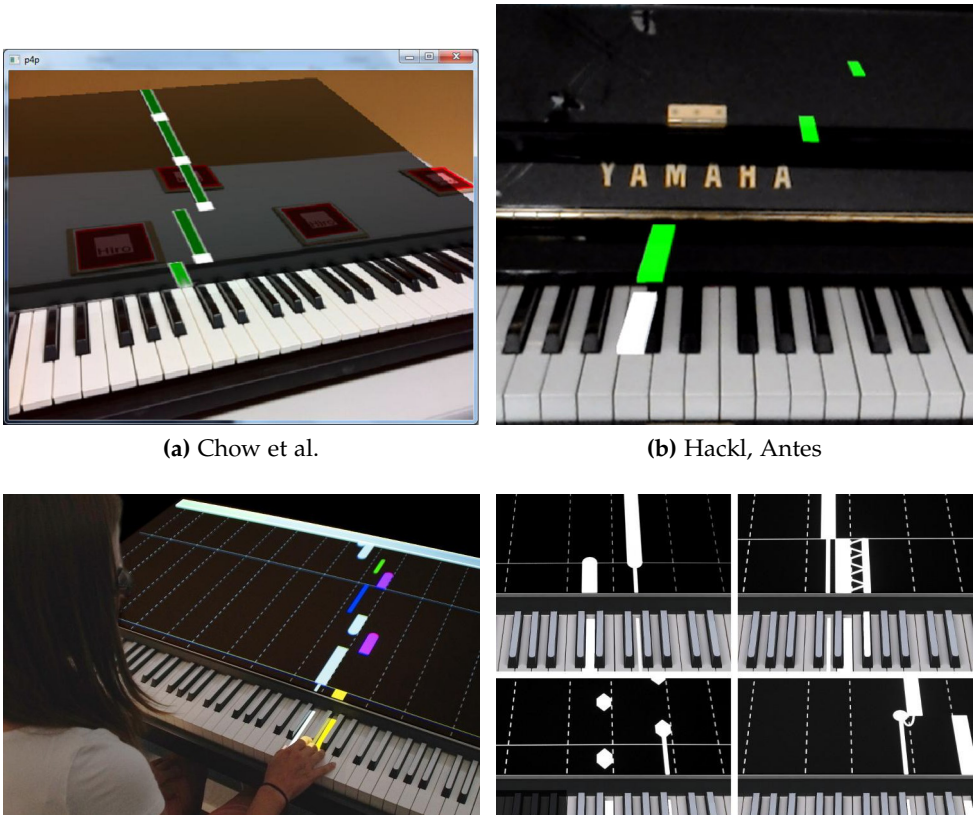
Chow et al., who were among the first to use a HMD in a piano tutoring application, acknowledge the difficulty of translating a note from a written score to the physical key on the keyboard for beginner players. As Cakmakci et al. had already recognized, in augmented reality the information regarding a note can be displayed directly where needed. Therefore it is of importance to compare the systems at hand based on the way they visually convey information about the music.

Chow et al. chose a representation where note objects come towards the user and should be played when they reach the piano (see Figure 2.1a). Virtual notes do not overlay the key but are consumed instead upon reaching the keyboard. This representation is also known from the game series *Beatmania*[3], which was first released in 1997, and *Synthesia*[4], which was first released in 2006. Both applications do not make use of augmented reality but feature 2D representations of keyboard and notes. Chow et al. place notes in a 3D environment, where the virtual notes have roughly the same size as physical keys and are placed in the same plane. Huang et al. place cylinders which should represent virtual fingers above the keys (see Figure 2.2b). Glickman et al. mark individual notes by placing a "mirror key" overlay perpendicular to the physical key. They state, that the advantage of this representation as being unobstructed while still connecting directly to the indicated keys. Additionally, the user's focus is directed above the keyboard, which is a long term desire of learning to play the piano. Two different styles of "mirror key" augmentations exist: one is capable of indicating timings and duration's, the other is a static overlay of the same size as the physical key (see Figures 2.2c and 2.2d).

Hackl and Antes explored multiple visualization methods in their application. What they call the "Beatmania Approach" closely resembles the visualization method used by Chow et al. (see Figure 2.1b). However, approaching keys are scaled in size according to their distance to the physical key they are approaching. Keys that need to be played sooner are therefore larger and draw in more of the user's attention. Besides, unlike Chow et al., virtual keys completely overlay the physical keys the moment they should be played and only disappear after the key has been played. On the "Instant Approach", keys are only highlighted at the exact time and for the exact duration they should be played (see Figure 2.2a). Hackl and Antes comment that the "Instant Approach" is not feasible for new players, as it is very difficult to learn a new piece, but could be useful for advanced players. It is noted that a limited FOV is less of a problem here. The authors claim that the "Beatmania Approach" is much more useful especially for newer players as they can anticipate the upcoming notes. They claim that learning a piece with the "Beatmania Approach" should be at least equally efficient as learning from sheet music.

As opposed to others corrresponding to the "Beatmania Approach" discussed earlier, the

[3] `https://en.wikipedia.org/wiki/Beatmania`
[4] `https://www.synthesiagame.com/`



(a) Chow et al.

(b) Hackl, Antes

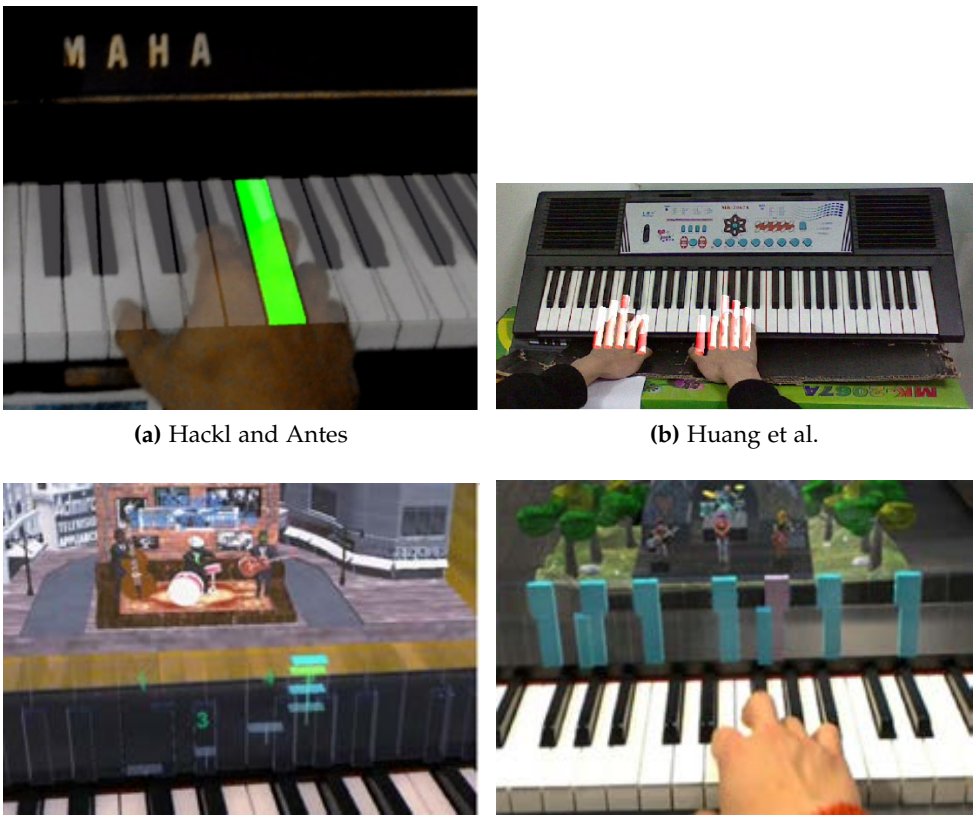
(c) Rogers et al. (1)

(d) Rogers et al. (2)

Figure 2.1: Visualization Approaches (1)
In this figure, visualization styles comparable to the style known from the game *Beatmania* are shown. Chow et al. [7] chose to never overlay the physical keyboard and shrink the virtual keys once they have reached it **(a)**. Hackl and Antes [8] take an opposite approach **(b)**. Rogers et al. [12] use colors to support correct fingering **(c)** and visualize desired accents in play **(d)**.

visualization style proposed by Rogers et al. is enhanced to discern the music's different articulations. Basic notes are displayed as rectangles. Notes to be played legato have round endings and notes to be played staccato have pointed endings. Trill and Grace notes feature special representations, too. To support correct fingering, notes corresponding to important finger switches are colored differently. Fixed colors are used for every finger.

A user study on the application from Chow et al. revealed that the speed for the "Beatmania Approach" on Visualization needs to be carefully chosen. A majority of users were intimidated by approaching notes if they could not keep up with them, which caused them to miss even more notes. Furthermore, additionally providing sheet music seems needless or even distracting as users had problems focusing on the sheet music when augmentations are present at the same time. Although these results were obtained from a single system, they



(a) Hackl and Antes

(b) Huang et al.

(c) Glickman et al. (1)

(d) Glickman et al. (2)

Figure 2.2: Visualization Approaches (2)
Hackl and Antes [8] propose an "Instant Approach", in which every key is highlighted as long as it should be pressed **(a)**. Huang et al. [9] introduced virtual fingers represented by cylinders **(b)**. Glickman et al. [11] bring forth an approach in which notes are falling down **(c)**. In Figure **(d)**, indicated keys belong to a harmony.

can certainly be conveyed to any other application using the same visual representations.

2.2.2 Keyboard Tracking

In order to display the augmentations, knowing the positions of piano keys in the frame is inevitable. The visual accuracy of the augmentations most importantly depends on the accuracy of the tracking used. One can differentiate between two fundamentally separate approaches. With the marker-based approach, a fiducial marker is positioned in the scene with a physically fixed relation to the keyboard. By tracking the marker, the positions of keys can be derived. With the second approach, the video is analyzed directly in order to detect natural features of the keyboard and identify the keys or compute their positions.

In the application proposed by Chow et al., keyboard detection is handled by a marker-based solution using the ARToolkit software. As the camera might not always capture the

full keyboard, multiple markers are used. The authors noticed that the performance was significantly better when identical markers were used. Therefore, a pattern was devised with unique relative distances between every two markers in order to identify the markers. They encountered jittering and shaking of the augmented reality overlay caused by noise in the markers' pose detection. The problem was attenuated by applying a moving average filter to the camera matrix which projects the world space coordinates of the keyboard to the camera space. It is noted that a larger filter kernel size results in a more stable overlay, however with a reduced response time when compared to a smaller kernel, and that the optimal kernel size will depend on the camera quality.

Hackl and Antes make use of a single, big fiducial marker for keyboard tracking in their application. A full virtual keyboard used for displaying augmentations is aligned with the physical keyboard accordingly, so that when the user moves his head, the marker pose can be applied to the virtual keyboard. This approach requires a manual calibration by the user, who has to set the position and scale of the virtual keyboard and is therefore subject to inaccuracies. Glickman et al. do not specify the tracking used, but it is assumed that a similar marker-based tracking approach was used. Huang et al. present a marker-less approach to tracking of a piano keyboard. Their system is especially notable for being the only marker-less keyboard tracking algorithm capable of processing frames from a moving camera in real-time. Therefore it will be described in detail below in section 2.3. Rogers et al. have no need for tracking in their application, since the projector producing the augmentations is in a physically fixed relation to the keyboard. It only needs to be calibrated once.

2.2.3 Feedback on User Performance

Chow et al., like Dannenberg et al., attached great importance to giving feedback to the users according to their musical performance. To implement complex feedback, they decided to use recorded midi files of the music pieces which contain not just the notes and timing, but also velocity as reference models against the user's midi output. They give accuracy feedback solely by different colors in note visualization for the following reasons: colors are perceived preattentively [13], they have an intuitive meaning (most likely strongest with red and green), they do not use extra screen space (as opposed to size and shape) and they are less distracting than changes of other visual attributes (such as shape). The purposeful use of colors for conveying information intuitively is outstanding in this system. At the end of a performance, additional statistical feedback regarding the player's accuracy and timing is given. However, unlike in the *Piano Tutor* application, no reasons for errors are explored.

The application proposed by Hackl and Antes does not feature user feedback, as the keys which were played cannot be detected. Glickman et al. state that they want to focus on encouraging correct play rather than highlighting mistakes made. Nevertheless, their technical implementation enables feedback. The system described by Rogers et al. focuses on correctness of key presses in terms of performance feedback. Immediately when a key is pressed, it turns red or green depending on whether or not it was correct. At the end of a

piece, a detailed feedback screen shows all played notes and differentiates between correct notes, notes with an incorrect duration and missed notes.

2.2.4 User Menu and Input

The application by Hackl and Antes implements a 2D user interface which is placed in the 3D scene using world-stabilized coordinates for navigation within the application's features. The user interface consists of three scenes: in the main menu, the user can select a musical piece and playback speed and in playback mode the user sees the actual augmentations along with a timeline that shows the current playback position. User input works with gaze-based interaction as well as gestures, which are not further specified. The gaze direction is not determined by the eyes but by the orientation of the head, e.g. the HMD. In the system by Rogers et al., the user menu is projected onto the keyboard directly. It is activated with a foot pedal and users can change settings via key presses.

2.2.5 Features

Chow et al. allow the user to manually adjust the tempo of a piece to suit their ability. Additionally, a *Note Learning Mode* was added which pauses each note on arrival and waits for the user to press the key. Glickman et al. describe a mode of their application which encourages the user to improvise. The application plays background chords and highlights a range of keys which will sound harmonious when played along. This feature is notable as it encourages the musical creativity of the user. The background chords can be played by an animated virtual band positioned above the keyboard, and different musical styles can be selected. Another unique feature is the ability to display a virtual animated hand that plays specific musical pieces in order to demonstrate correct hand positioning and advanced playing techniques[5]. The virtual hand is obtained by capturing the performance of a real pianist beforehand, which is done using a modified Leap Motion Sensor[6].

The application by Rogers et al. features a "Listen Mode", in which the music piece plays by itself while highlighting the corresponding keys. Notably, this mode was revealed to be important following feedback from a psychologist and a piano teacher contacted by the researchers. In the settings menu, users can set the tempo of a piece and scroll through its visualization.

2.2.6 User Evaluation

Rogers et al. compared their application with a non-AR piano teaching application as well as learning a piece using sheet music by performing statistical analysis. They observed that

[5]It has to be noted that in the paper describing the system, features are only mentioned but not described in detail and no information regarding the implementation is shared. It can therefore not be determined how well the features work and what could be improved.

[6]`https://www.leapmotion.com/`

their system exceeded the alternative in both percentage of correct notes played and note duration accuracy. However, users of the system also played more incorrect notes since they tried to play more notes overall. Rogers et al. concluded that their system improves initial learning performance but also enables a steeper learning curve. Most users commented that they enjoyed using the system. During a user study performed by Chow et al., all participants expressed that they enjoyed their system.

2.3 Keyboard Detection Approaches

2.3.1 Huang et al.

Huang et al. present a markerless keyboard tracking algorithm in their article *Piano AR: A Markerless Augmented Reality Based Piano Teaching System* [9]. It was written in OpenCV and C++.

For preparation, camera parameters and keyboard geometry, e.g. distances, widths and lengths are measured and stored. The process of recognizing and tracking the keyboard can be divided into the following steps: *(1) Image Preprocessing*, *(2) Contour Extraction and Analysis*, *(3) Keyboard Recognition*, *(4) Geometrical Feature Gaining*. The complete system architecture can be seen in Figure 2.3. During *Image Preprocessing*, the original image is binarized since

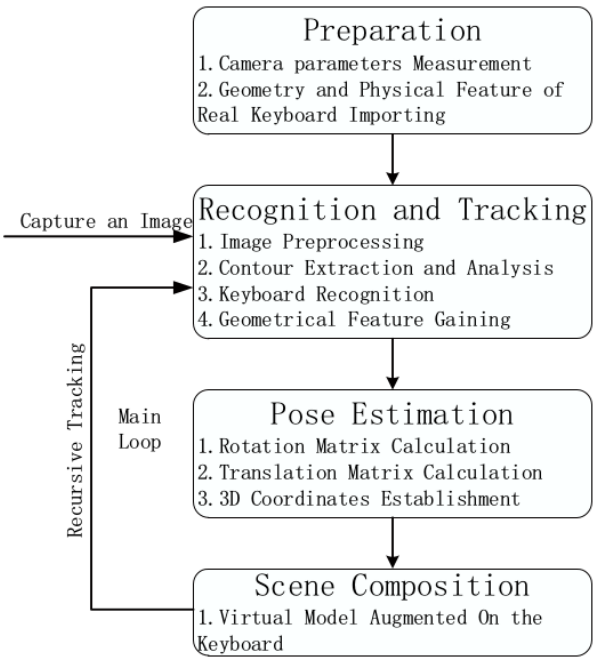


Figure 2.3: The Architecture of Huang et al.'s AR System
The computer vision processing is done in *Recognition and Tracking*
Source: [9]

only two colors, namely black and white, are associated with the keyboard anyways. A

small threshold for binarization is preferred in order to minimize the black contours in the image that don't belong to black keys, which will reduce the computations in the following steps. Possible small black lines between white keys will be eliminated using a morphological closing operation.

In *Contour Extraction and Analysis*, all external contours are extracted from the binary image. Contours which don't properly fit are discarded by checking their length against an upper and a lower threshold. Hereafter, the convex hull of each contour is calculated and further approximated with a polygon using the Douglas-Peucker (DP) algorithm[7]. Only those contours approximated by a quadrangle are kept as keyboard candidates.
Keyboard Recognition starts by finding the top edge of each of the quadrangles using a heuristic algorithm developed by Huang. A scan along a line, parallel to but below the top edge, is performed. The resulting array is checked for sections of continuous black pixels which correspond to black keys. If the number of keys is equal to the predefined count, the keyboard contour is considered found and all corner points A, B, C, D are assigned. For Pose Estimation, a coordinate system is established on the keyboard and the matrix transforming from camera space to keyboard space is computed using the detected corner points of the contour. The stated accuracy of the algorithm is 1.97 pixels. Keyboard recognition and pose estimation took a combined time of 50 milliseconds during testing on a computer with 2GB memory and a 2.33 GHz processor.

2.3.2 Visual Music Transcription Systems

The usefulness of detecting individual keys on a piano has also been explored in context of other systems than music teaching applications. One concept which has gained attention is the implementation of a visual piano music transcription system. Instead of highlighting keys that should be pressed, this system rather tries the opposite; to detect key presses from a video stream of an advanced musician playing the piano in order to create the according sheet music in musical notation. Although none of the algorithms described can be used for detecting a keyboard in a video stream from a moving camera, it is still useful to evaluate, if some of the concepts described can be adapted.

Akbari and Cheng: claVision (2015)

claVision, presented by Akbari and Cheng in 2015 [14], is a piano music transcription system notable for running in real time without any markers. It accomplishes this by capturing a full 88 key keyboard with a static camera that is fixed in the physical world. Therefore the initial detection of the keyboard, which is most computationally intensive, only needs to be performed once and eventually updated later. The initial detection algorithm, as described in the paper, will be further explained as similar concepts could be applied in a piano tracking algorithm with different requirements.

[7] `https://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm`

First of all, the proposed algorithm performs Hough line transform to extract lines appearing in the image. Then all 4-combinations of the extracted lines are evaluated based on their intersection points. Each set of lines with at least 4 intersections, forming a quadrilateral, is transformed to a rectangle using homogeneous transformation and is considered a candidate. From the structure of piano keyboards, a rectangle can be regarded as the keyboard if it has the maximum brightness in the lower-third and contains the most black keys in the upper two-thirds part. The first condition is tested by comparing all intensities in the lower third of the candidates. The black keys are counted using a connected components labelling algorithm which extracts and counts individual objects in the image. In order to differentiate black from white keys, the Otsu thresholding method is used. The white keys are finally estimated from the position of the black keys, assuming standard piano key dimensions [8].

The algorithm for detecting keys pressed will not be discussed further as it works by subtracting images taken from the exact same position and can not be applied to a moving camera stream. During tests on a laptop with an Intel Core i7-4510U CPU (2.00 GHz) and 16 GB DDR3 RAM, initial detection of the keyboard's outline was successful for all videos and took an average of 691.9 milliseconds. Detection of the individual keys took and average of 16.6 milliseconds while achieving an accuracy of 95,2%.

Suteparuk: Detection of piano keys pressed in video

A similar approach was presented by Suteparuk in the paper *Detection of Piano Keys Pressed in Video* [16]. He assumes that the keyboard is surrounded by a thick black border and can be easily detected and extracted from the image. In the extracted region, the black keys' outlines are detected with a Sobel operator. Small regions, which are caused by noise and to be removed as well as the black keys, are detected using a region labelling algorithm. Suteparuk noted that a Hough Transform applied to the region would correct the rotation of the keyboard very well because the orientation of the piano keys is so dominant.

2.4 ChromaTag Marker Detection

The *ChromaTag* marker detection by DeGol et al. [17] stands out for its speed and for using its own colored marker design alongside a very clever utilization of the CIELAB color space's features (see Section 3.2.3). A comparable marker design and algorithm for grayscale images is *AprilTag*[9] which is popular and commonly used in applications and beyond similar to *ArUco*[10]. The principles behind the marker design and the algorithm as well as technical details about the detection algorithm will be further explained in detail here. The implementation was done using OpenCV and C++.

[8] The key dimensions used were taken from the article "The Size of the Piano Keyboard" [15]

[9] `https://april.eecs.umich.edu/software/apriltag`

[10] `https://www.uco.es/investiga/grupos/ava/node/26`

Marker Design

ChromaTag markers are colored markers and require the processing of a 3-channel image. The authors state that as one of their main motivations, they wanted to prove that utilizing the information contained in a colored image can actually improve the performance of algorithms in computer vision compared to using grayscale images, where only a single channel needs to be evaluated in every operation. The marker design is based on a high contrast of red and green which is rarely found in natural scenes [17], thus reducing the number of marker candidates found in a scene. It consists of three square rings: an outer white ring, a black border and an inner green ring, and furthermore a red square in the center (see Figure 2.4). The red and green color values correspond to the most extreme values in the alpha channel

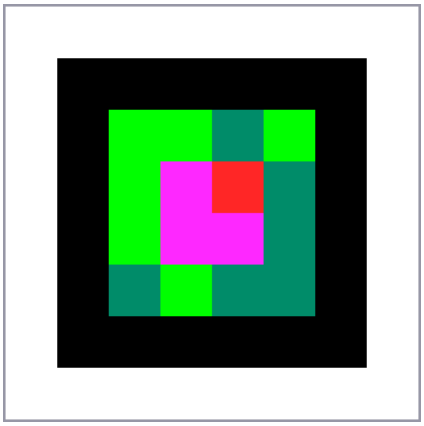
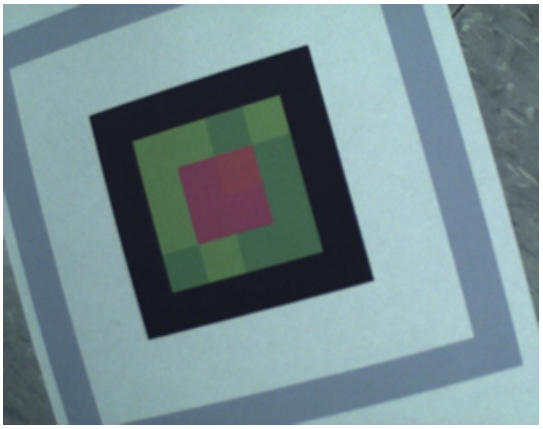
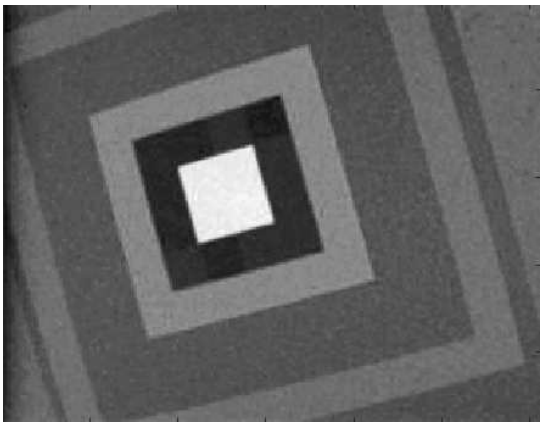


Figure 2.4: A *ChromaTag* Marker (16H5 Family)
Source: [18]

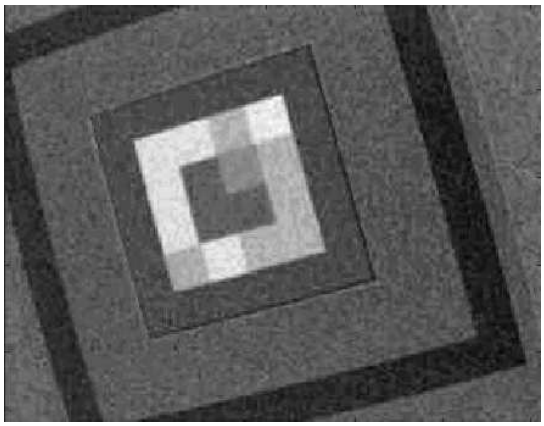
of the CIELAB color space. The gradient strength of this kind of edge can therefore be determined by only looking at a single variable. The marker's tag ID is encoded within the beta channel of the marker and is encoded the same way as in *AprilTag* markers. Because chrominance has a lower effective resolution than luminance due to the Bayer pattern filters used in common cameras, the marker is designed with an outer black and white border for precise localization of the corners of the marker square. The separation of individual channels for marker pose detection and tag encoding can be seen in Figure 2.5. The Figure clearly shows how pixels can have the same alpha channel value but completely different beta channel values. The principles behind the marker detection algorithm are based on the following assumptions: (1) Most of the time wasted in marker detection is usually wasted on computations on marker candidates which are rejected later. By rejecting false candidates as early as possible, detection speed can be improved by a great factor. (2) A high contrast of red and green can rarely be found in natural scenes, thus reducing the number of marker candidates that will be found in a scene.



(a) Image in RGB



(b) Alpha Channel



(c) Beta Channel

Figure 2.5: CIELAB Channels of a Marker Image
In Figures **(b)** and **(c)** it can clearly be seen how the marker design encodes different information in different image channels. Additionally, in Figure (b) the high gradient between red and green can be seen. The marker tag in Figure (c) is comparable to black and white square marker designs.
Source: [17]

Detection Algorithm

The first step of the algorithm is to find potential tag locations in the frame. It it assumed that the tag cell of each marker is at least $\mathcal{N}/2$ pixels wide in the frame and thereby sufficient to search over pixels on a grid of every $\mathcal{N}$ rows and columns. Each pixel is converted from BGR, which is the color space the frames are provided in, to CIELAB, and compared against previous pixels in terms of alpha channel intensity. The pixel-wise conversion is also done in following scans. Because the conversion is rather slow by default, a pregenerated hashtable is used, which allows for a reduction of conversion time by orders of magnitude. If the difference is greater than a fixed threshold, the surrounding area is marked and detection commences. Pixels detected this way will lie on the border of the tag's red square and green

ring. To reject false detections as early as possible, the next step is to determine if the pixel really belongs to a marker. From the pixel's location, scans in all four directions are performed until a red-to-green border is found. If not every scan terminates successfully within the area's range, the area is abandoned. Otherwise, in case of success, a procedure of multiple scans is performed which, given a fixed accuracy, converges when the tag's center point is found. The same scans are used to find the marker's green-to-black border. Next, using a successive polygon building algorithm, a surrounding polygon is fit for every ring of the marker and further approximated by a quadrilateral. The estimated outer corners are further refined and tag decoding is initialized. A homography matrix can be estimated from the black-to-white corner points. Using this homography, the pixels encoding the marker tag are extracted and retrieved regarding their beta channel values in CIELAB. If looking at the beta channel, tags are equivalent to *AprilTag* tags in greyscale images and decoded and identified as well as rejected the same way.

Performance and Drawbacks

On a sample set of images generated by the authors with a resolution of 752 x 480 pixels, the algorithm was able to process 709 frames per second (fps) while detecting a single or more than one marker, or 2616 fps if no marker was contained within the image. This was tested using a 3.5 GHz Intel i7 Ivy Bridge processor. Frame rates of more than 500 fps could be reproduced using images provided for testing on a laptop with a 2.2 GHz i7 Sandy Bridge processor. The only clearly stated drawback of the algorithm is that it is less robust against steep viewing angles when compared to AprilTag and will fail frequently in those conditions.

2.5 Related Technologies

2.5.1 Augmented Reality Tracking Frameworks

When integrating AR technology into an application, it is usually preferred to use frameworks which provide ready-made solutions. They feature state-of-the-art detection capabilities and can accelerate the development by a great factor. Regarding the implementation of the proposed detection algorithm, natural features of generic piano keyboards need to be detected and analyzed in order to derive positions of individual keys. The frameworks have been reviewed concerning their capabilities to perform or help with such a task. A necessary feature was the capability of detecting features based on their color or shape and arrangement within the image in different environments. Direct extraction of all feature coordinates was preferred over the return of an origin point with a homography.

In this section, an overview over popular AR-frameworks will be given. All of the presented frameworks can be used in conjunction with Unity but have not been selected therefore.

Wikitude SDK

Wikitude[11] is capable of tracking images, objects, scenes and geographical locations. It can also render its own augmentations. Applications can be deployed on android, ios and windows. Wikitude can be used as a standalone development environment as well as be integrated into Unity3D with its own plugin[12], but it should be noted that the plugin does not support all features of the Wikitude engine. It utilizes both ARCore and ARKit for faster processing dependent on the target hardware and therefore achieves above-average performance on mobile platforms. The scene recognition and tracking works with an implementation of the well-known SLAM algorithm and stands out in comparison to similar platforms. Tracking objects is handled through the use of feature point clouds. The developer must upload a video or a couple of images showing the object in different poses which are then used for generating a custom file containing a feature point cloud for this particular object. The file will be loaded at runtime and enable the platform to search for similar features and establish a tracking from the camera's image stream. Object tracking only allows for the use of exactly the same real object as a marker object. It is not able to track generic objects which share similarities. Additionally, this tracking will only return a single anchor point with a homography for the placement of virtual objects.

A unique feature of the Wikitude SDK is the possibility of developing plugins for the engine itself. Plugins can be written in C++, Java or ObjectiveC and make use of all the power of Wikitude's native API. The plugin API however aims at similar application areas as the engine by itself and is less powerful than a full computer vision library as OpenCV. The use of Wikitude's own plugins in conjunction with the Unity plugin is not supported. The Wikitude Engine is free for private use and has pricing options for enterprises.

ARCore

ARCore[13] has been developed by google and supports android and iOS. It can be used within Unity, the Unreal Engine and Android Studio with Sceneform. It is capable of tracking markers, images, scenes and planes, but the same limitations as with Wikitude are present. It is feature-wise not more powerful than Wikitude, but is notably completely provided for free. While it would be possible to perform computer vision operations on captured images as shown in the arcore-unity-sdk examples which are available on github, doing so would require a significant amount of work and therefore not justify the use of ARCore.

ARKit

As ARKit was developed for iOS exclusively, it will not be considered further. The features are comparable to ARCore, although ARKit is expected to perform better due to being tailored to iOS.

[11] `https://www.wikitude.com/`
[12] `https://www.wikitude.com/products/extensions/unity/`
[13] `https://developers.google.com/ar/`

Vuforia

Vuforia[14] has been around for a longer time and is one of the most popular augmented reality development platforms. It has been directly integrated into the Unity3D game engine in 2017[15] and since then, it is possible to obtain a free license. The engine supports a wide variety of virtual and augmented reality devices, including the HoloLens. Using Unity with Vuforia is probably the fastest way to create an AR application, as many features work well out of the box. Vuforia can track objects based on their shape if an according 3D-model is given. It also provides image, object and environment tracking comparable to Wikitude's tracking capabilities.

Usability of the Frameworks

All of the mentioned frameworks work well for what they are designed to do. However, they all miss the ability to further analyze the camera images and extract individual feature points from an image. Marker tracking in all frameworks will not be faster than a similar tracking done in native OpenCV using C++.

2.5.2 Smartphone-Powered Optical See-Through HMDs

For displaying computer generated graphics over real world images, two techniques exist: video see-through and optical see-through. With video see-through goggles, the real world is captured via camera and the image is then shown on a screen after being enhanced with the computer generated content. The advantage of this technique arises from the fact that no augmented reality content can be generated instantaneously. Therefore, a slight delay, even if it is only a hundredth of a second, is inevitable. By only displaying the captured images when the enhancement is done, real world and augmentations will always add up exactly. The delay is passed to the whole video stream, which is less noticeable for the user. As a downside however, common side effects known from Virtual Reality (VR) headsets like motion sickness may arise. Using Optical see-through HMDs (OHMD) on the other only the actual augmentations are displayed, while the real world can be seen as is. They have no means of avoiding the delay resulting from computation time. But the view of the real world can of course not be matched by any video display. Motion sickness is much less of a problem, if at all, as the user can orient himself on the real world. The advantages become more significant the faster the augmentations are computed and the less intrusive the augmentations are placed within the image.

When designing a system which aims at teaching or helping the user at playing the piano, is it important to only enhance but not interfere with the user's natural learning progression. One of the key elements of learning progress when it comes to instruments are the student's motor developments. This can be described as the process of associating sounds with the

[14] `https://developer.vuforia.com/`

[15] `https://library.vuforia.com/articles/Training/getting-started-with-vuforia-in-unity.html`

physical actions performed by the student. The better the piano player, the more complex the relation between sounds and keystrokes will become until extreme precision and timing is developed. Now if a user learns to play piano while always wearing a video see-through device, his brain will adapt to the delay and establish a relation between sounds and finger movement which includes the presence of the delay. Then, if a user stopped wearing the HMD, he might have to adapt his motoric abilities to the presence of no delay and could lose some of his learning progress. It is therefore preferred to use an optical see-through device for the application as it interferes least with the natural way of learning the piano.

In the last years, VR headsets powered by smartphones have become very popular. The google cardboard and related googles[16] based on the same framework, headsets can cost as little as $7 or can even be built at home. Products exist within every price range up to professional solutions. This enables everyone owning a compatible smartphone to obtain a VR device very easily, and many people will get a headset just to try out the technology. Unfortunately, few of these devices are capable of augmented reality without hacks like cutting a hole into the headset. Additionally, they would all be video see-through which is not optimal as described previously. Smartphone-powered OHMDs exist, but are not yet as refined as their VR counterparts, since they are more complex to construct. Various devices have been released but were still in a rather prototype-like stage of development. Additionally, available devices are more expensive and less easy to buy in all parts of the world. Standalone augmented reality goggles which can be purchased at the time of writing are much too expensive for regular consumers to buy. Famous examples are the Microsoft HoloLens costing$3,000 [17] and the Magic Leap One costing$2,295.00 [18]. Below, an overview of already released smartphone-powered devices, alongside devices stated to be released in the near future will be given.

Aryzon[19] is a headset very similar to the google cardboard, except that it is visual see-through. It was also marketed as the augmented reality counterpart of the google cardboard. Two versions are available: a hands-free version costing $30 and a handheld version costing $15. An open-source SDK is available and developing apps using Unity3D is possible and encouraged. The framework supports both IOS and Android smartphones. The headset is more useful for testing and exploring the technology than for real applications, due to the very limited FOV.
Mira Prism[20] is yet to be released to consumers, but development editions have been available since 2017. It is made out of hard plastic and is rather sturdy, and it has been stated that it is a bit heavy. The headset has a larger FOV than the HoloLens with a 60 degrees diagonal. It is available for preorder costing $99, the final price is stated to be $149.

[16] `https://vr.google.com/intl/de_de/cardboard/get-cardboard/`
[17] `https://www.microsoft.com/en-us/p/microsoft-hololens-development-edition/8xf18pqz17ts?activetab=pivot`
[18] `https://shop.magicleap.com/en/Categories/Devices/Magic-Leap-One-Creator-Edition/p/M9001`
[19] `https://www.aryzon.com/`
[20] v2 available for preorder at `https://www.mirareality.com/developers`

Device	*Release Date*	*FoV in degrees*	*Tracking Capabilities*	*Weight*
Aryzon	2018	35	na	381g
Mira Prism	2017	60 diagonal	6DOF	na
Tesseract Holoboard	2018	82	3DOF, positional tracking	185g
Lenovo Mirage AR	2017	60	6DOF, positional tracking	477g

Table 2.1: Specifications of smartphone-powered OHMDs

Tesseract Holoboard[21] The first version of this headset was distributed in 2018, being priced at $149, but is now assumed to be sold out. It features a very large FOV of 80 degrees while being quite lightweight. A successor is in development but uses built in hardware instead of utilizing a smartphone and will be priced higher than its predecessor.
Mirage AR[22] The Mirage headset developed by Lenovo was originally sold for $300 and marketed as a complete experience in conjunction with a Star Wars game developed specifically for the headset. It features a large FOV of 60 degrees and contains two cameras placed close to the user's eyes. Colored luminous marker devices can be detected and used as reference. On the flipside, it weights 477g, which is much more than other headsets. Unfortunately, Lenovo did not design a framework, that could have been employed by developers utilize the headset. It's usage is therefore very restricted.

[21] v2 (standalone) available for preorder at `https://myholo.io/`
[22] `https://www.lenovo.com/us/en/arvr/#starwars`

3 Application Design

3.1 System Architecture

The full piano tutoring application consists of the keyboard tracking module on the one hand and the actual application that the user will be presented with on the other. It is desired that both system parts, although dependent on each other, are self-contained. This way, the tracking module could be integrated into other applications but also swapped for another detection method in the prototype. The tracking module's goal is very straightforward: detect the positions of all keys, given a frame containing a keyboard. The main goal of the prototype is to show that such a system can run on an Android smartphone used for powering a HMD. Therefore, a possible integration of the tracker module into a Unity application must be demonstrated. The prototype must include the basic functionalities of loading a song and visually conveying the note information; it should also be possible to set the playback speed, as this is an extremely important feature teaching-wise. Besides, the prototype should be easily extendable regarding the user interface and visualization techniques. It would be best if the prototype could be deployed on other platforms as iOS and other HMDs with minimal adjustments.

Implementation of the tracking algorithms as a self-sufficient module is most reasonably done using a computer vision library. For this task, OpenCV has been chosen, as it is quite powerful and fast, convenient to use and freely available. For development of the Android application, the Unity game engine has been chosen as the most suitable development platform. It can handle complex visualization and user input much better than OpenCV. Unity is very popular and widely distributed and has good support for AR including build options for various HMDs. Being designed for deploying the same application on multiple platforms with minimal adjustments, it suits this design request very well, too. To satisfy the self-sufficiency guideline, it is preferred to provide the video frame for the tracking module instead of letting it handle the camera stream on its own, the main reason for this being that it can not be guaranteed that the tracking module can open a video stream with the right settings on every device. By manually providing the frame instead, Unity or custom solutions can be used for camera capture. For performance reasons and better separation of concepts, the information flow between both systems should be minimized. The flow of information is illustrated in Figure 3.1.

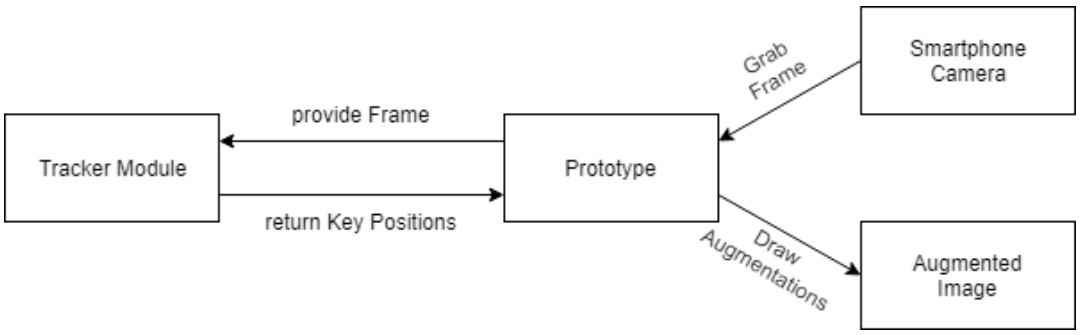


Figure 3.1: The Flow of Information in the Piano Tutoring System

3.2 Keyboard Detection

In order to display augmentations with the greatest possible visual accuracy, it is desirable to obtain the exact corner points of every key in the frame captured by the camera. All of the reviewed systems, except the one presented by Huang et al., use exclusively fiducial markers for this task. However several issues arise from this approach, which are especially present on wide keyboards. If using a single marker, the camera's FOV may be too small to contain both the marker and the part of the keyboard which should be augmented. Additionally, marker detection is often not precise enough to produce stable augmentations due to the size of the overlay. Chow et al. also noted instabilities in their overlay due to marker detection imprecisions and in the system by Hackl and Antes, it can be seen that the augmentations lack accuracy. Using markers that have to be placed outside the keyboard, as all reviewed marker-based approaches have, exerts a restriction on the physical composition of the piano. It might not have a place where the marker can be put to satisfy the given requirements, like a sufficiently large flat area aligned with the keyboard at an acceptable angle. Those restriction can be avoided by placing the marker or markers directly on the keyboard. In return, placing markers on the keyboard itself puts considerable restrictions, most importantly in terms of size, on the marker design and tracking algorithm used. This in turn puts considerable restrictions, most importantly in terms of size, on the marker design and tracking algorithm used.

The algorithm proposed by Huang et al. requires the whole keyboard to be visible in each frame. While that may be feasible for smaller keyboards, a full-sized piano keyboard will not be contained completely in frames captured with a camera attached to a piano player's head, if not using special wide-angle camera lenses. The algorithm is therefore not suitable for tracking wide keyboards. Instead, a new tracking algorithm will be developed which does not rely on detecting the keyboard's outer contour and can work on a section of the piano. However, if the keyboard is not fully visible in the frame, there will be no natural way of telling which octave a key belongs to. In order to solve this uncertainty, the use of markers for marking the octave is suggested. Markers should be spaced at such distance that at least one marker is always visible. By knowing the key each marker is placed on, octave and note

for all keys can be derived. Markers will be placed on the keyboard to avoid the restrictions described previously.

As the detection of such small markers will not be accurate enough for establishing an overlay, all the piano keys have to be tracked separately. This will be done using a natural feature tracking algorithm which recognizes the keyboard's structure and detects the keys individually. In order to develop the tracking algorithm, the composition of keyboards needs to be examined. For defining the marker requirements, possible keyboard dimensions must be known as well.

3.2.1 Keyboard Dimensions

The design of pianos apart from classical pianos can vary greatly but the keyboard is almost always a common feature. Usually, any instrument considered a piano follows the same guidelines about width, spacing and color of the keys. Some guidelines have emerged for structural reasons of building and aligning the mechanisms behind each key in analog pianos with strings, but digital stage pianos use the same guidelines for reasons of consistency in user experience. Other instruments such as keyboard controllers, which are used in digital music production, do not have the same key mechanism as pianos and their keys can differ in size, spacing, alignment, form and color. I will focus only on piano keyboards and keyboard controllers with a comparable design. All keyboards implement the chromatic scale.

When looking at different implementations across manufacturers, the overall keyboard size is directly proportional to the octave span[1]. A typical octave span on a modern keyboard is 164mm to 165mm. For mechanical reasons, is is desirable to have an equal spacing between all keys, black and white, at the rear end. For best playing experience, it is desirable that all black and all white keys have the same width respectively. Given the familiar layout of an octave, it is not mathematically possible to have both and because manufacturers resolve this conflict differently, different keyboard structures exist [19]. A widespread solution is to have four white keys with a width of 23mm and three white keys of 24mm width and black keys with a width of 14mm, resulting in a rear width of either 13mm or 14mm for each key [15]. The spacing and alignment can be seen in Figure 3.2. Now, those key widths add up to exactly 164mm per octave. The space between keys is set as small as possible and can be distributed equally, counting towards the keys mathematical width but substracting from the physical width. Furthermore, smaller pianos following the same guidelines exist which have smaller keys but the same size relations between keys. Black keys often run a bit narrow at the top.

Considering all of this in terms of marker size, a maximum width of 2cm for a white key and 1cm for a black key at the top are the requirements. As the rear end of white keys is as wide as the black keys, a marker width of 1cm will be assumed to allow for a flexible

[1]The octave span denotes the width of a complete octave

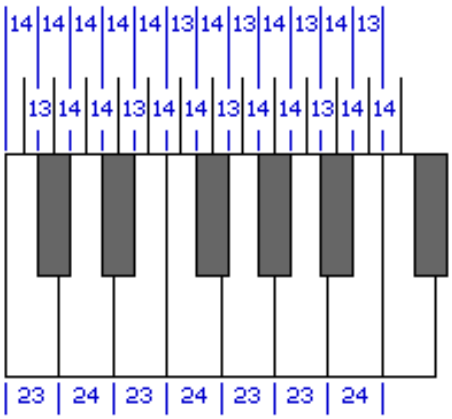


Figure 3.2: Keyboard Dimensions and Spacing
The figure shows a common solution for key dimensions and spacing of the mechanical levers. It is desired that all keys are equally wide and all levers are equally spaced, which is not possible. Therefore, different designs and dimensions exist.
Source: [15]

marker placement. The length of individual keys is not essential for piano construction nor for playing experience and therefore varies more strongly. A length of 9cm for black keys and 14cm for white keys was measured on a normal piano and may serve as a rough assessment. Therefore, the marker length is not especially restricted by the length of keys and can be chosen according to other considerations.

3.2.2 Marker Design

In order to prevent occlusion of the markers by the piano players fingers, the markers will be placed on the keys upper part. Following the requirements imposed by the keyboard dimensions, the marker can not be wider than 1 cm. Different marker detection algorithms have been tried for this task. On a full-size keyboard with 88 keys, no more than 8 markers will be needed, as the video frame can be considered insufficient if it shows less than one octave of keys. We consider the following exemplary setup: the piano player is sitting relatively upright and the piano's keyboard is situated at about the height of the player's waist. If the camera's FOV is not especially narrow, we can expect more than two octave spans to be visible in each frame. This was evaluated using a camera with 60° FOV but cameras in HMDs or smartphones can have a FOV of up to 120° or more, further reducing the number of markers required.

As the detection algorithm will crucially depend on markers for the previously explained reasons, it becomes a necessity to use as much information as the markers can provide in order to make assumptions and simplify the key detection algorithm as far as possible. Through clever marker placement, information about the keyboard positioning and orientation as well

as the key dimensions can be obtained. Each marker will be placed to align with the rear end of its key and therefore the rear end of the keyboard. Its width should correspond as accurately as possible to its key's width and the relation between key length and marker length needs to be known. The marker orientation will serve as a rough estimate of the keyboard's orientation.
As each marker need not encode any information, the following requirements for marker design arise:

- It is possible to create 4 to 8 distinguishable markers in the same scene.
- A marker of a width of 1cm at a distance of 1m at most, captured in a frame of 1280x720 pixels or less, can be detected comfortably.
- The marker orientation shall be computed as precisely as possible without hindering performance greatly.

Most popular fiducal markers are square and not designed for such a small size, so the corresponding detection algorithms will fail frequently. The marker size in pixels can be increased by simply using a larger frame, but this comes with a large computational cost as doubling the marker size in both dimensions will quadruplicate the number of image pixels. Additionally, the great majority of markers are designed to contain high black-to-white pixel gradients, which can be detected easily by scanning the image. An example would be the *ArUco* marker design which is widely used with detection algorithms ready to use in the OpenCV library[2]. But a piano keyboard will also contain very strong black-to-white gradients, so those marker designs do not seem like a good fit for the proposed application. Furthermore, the detection of ArUco markers is rather computationally expensive. A feasible marker detection algorithm capable of efficiently detecting a marker of such a small size could not be found.

Custom Marker Design

Following considerations about a special marker design tailored to the problem, it was determined that a mono-colored rectangular marker would best fit the requirements. A colored marker can be detected by the color gradient between the marker's outline and the surrounding area, but in contrast to black-and-white markers, it can also be detected by performing a color thresholding operation on the image. This can be easier to implement and will work especially well in the colorless keyboard environment. In order to distinguish the different markers, different colors will be used. The colors are chosen based on their detection performance.

As mentioned above, the marker's width is set to the key's width at the rear end it is placed on. The marker height is set to be half the height of a black key, measured at its base. Using a rectangular marker is advantageous in this case as its orientation can be computed more precisely compared to an equivalent square marker.

[2] `https://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html`

3.2.3 Color Spaces

Detection performance of a monochrome marker can, depending on the approach, be greatly affected by the chosen color space. Although all color spaces represent the same information, they separate it differently. A pixel color can be converted from one color space to the other using mathematically defined conversion algorithms. When using a sufficiently large bit depth, no information is lost during the conversion. The color spaces RGB, HSL or HSV respectively and CIELAB will be looked at because of their popularity and significant advantages over other representations for the task at hand.

RGB/ BGR

The RGB color space is the most widely used color space due to its simplicity and the high distribution of RGB image sensors. The BGR color space is only a permutation of the RGB color space and therefore identical in theory. The R, G and B values correspond to the red, green and blue chromaticities of the color, and the final color is defined by their additive combination. A major disadvantage of the RGB color representation is that is does not separate the actual chromacities and brightness of a color.

HSV/ HSL

The HSV and HSL color models define colors by saturation, hue and value (brightness) or lightness respectively. Hue is usually represented by an angle from 0 to 360 degrees in a hue wheel, the other two channels are linear. Brightness can be understood as the amount of light, which can be any color, while lightness can be understood as the amount of white, as illustrated in Figures 3.3 and 3.4. The saturation value in both color spaces is scaled

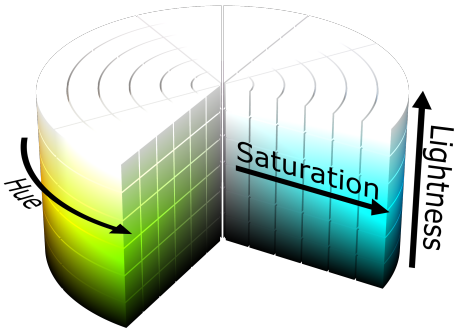


Figure 3.3: The HSL Color Space
Source: [20]

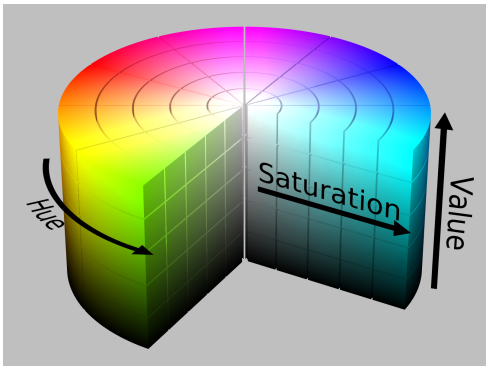


Figure 3.4: The HSV color space
Source: [21]

differently to fit the value or lightness definition. However, the same concepts apply to both color spaces. Defining any color in the HSV color space is very intuitive, because the concept of this color representation resembles the way humans perceive color very closely. Using HSV is therefore beneficial when defining colors manually. It is also widely utilized in the

field of computer vision as the separation of lightness and chroma makes it much easier to distinguish or threshold colors under different lighting conditions. Bora et al. [22] have come to the conclusion that the HSV color space is most suitable for the segmentation of noisy color images, although it should be noted that this is hard to determine unambiguously.

CIELAB

The CIELAB color space, also known as L*a*b*, was derived from the CIE XYZ color space and defined by the International Commission on Illumination (CIE) in 1976. L* represents the lightness of a color while a* and b* are chromatic components representing a point in a gradient with a* (alpha) as a red to green gradient and b* (beta) as a yellow to blue gradient (see Figure 3.5). The CIELAB model is device independent with respect to a given

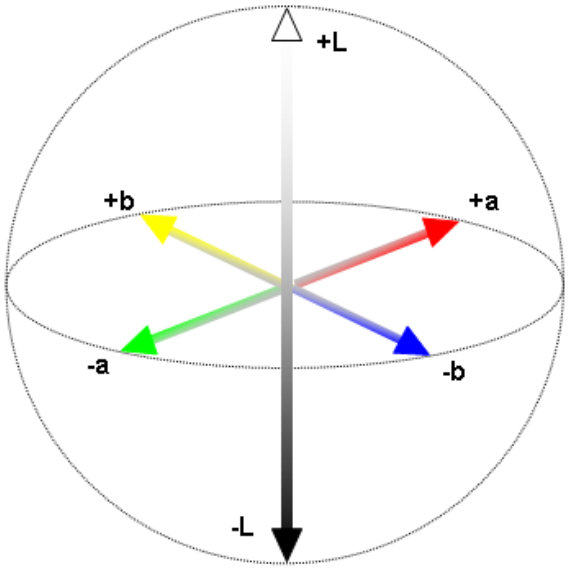


Figure 3.5: The CIELAB Color Space
Source: [23]

whitepoint. It is designed to represent a 'uniform' color space: for any two points in the defined coordinate system the relative perceptual differences can be approximated by the euclidean distance between the points. That assures a more linear visual color progression when compared to HSV. The properties of CIELAB can be exploited to detect red-to-green or blue-to-yellow gradients very easily in images, because it is sufficient to only look at a single color channel of the image. Additionally, it has been stated that this color space is usually suited much better for detecting gradients in images compared to the RGB color space. It can drastically improve edge detection as it removes most edges on an object while preserving edges separating objects [17]. The benefits of separating lighting and chroma information as in HSV apply to CIELAB the same.

Color Conversion Performance in OpenCV

In OpenCV, images and camera streams are usually provided or loaded in BGR format. To benefit from the advantages of another color space, each frame has to be converted by running the conversion algorithm for every single pixel. Therefore, potential conversion performance differences would be important. To test conversion performance, a simple test Application was developed which loads an image in BGR format. The image is then converted to the CIELAB and HSV color spaces using OpenCV's function *cvtColor(...)* and the conversion time is measured. Results are averaged over a 1000 performed conversions and the same test was run multiple times to account for different background CPU load. Running the application on a computer with 2.9 GHz using an image of size of 1600x900 led to the following results in Table 3.2.3. It can be concluded that there is no pivotal difference in conversion performance.

Color Space	*Conversion Time in ms*	*Frames per Second*
HSV	0.018	56
CIELAB	0.015	67

Table 3.1: Color Conversion Performance from BGR using *cv::cvtColor*

The performance of the color space itself is considered much more important than the small performance difference present.

3.2.4 Marker Detection Algorithm

A naive approach for a colored marker detection could look like this:

- Blur the image to eliminate noise.
- Convert the image to a color space that is well suited for thresholding operations.
- Perform a 3-channel thresholding on the image using an upper and a lower threshold to gain a binary image than can be processed more easily (*cv::inRange*).
- Try to find contours using a contour finding algorithm (*cv::findContours*).
- Filter contours by size and discard contours too small or too large to be a marker.
- Try to approximate each contour with a polygon, and keep only those with 4 corners.
- Discard quads that do not have the correct predefined width/height ratio.

This approach would already yield good results but can be improved significantly in terms of performance. An especially computationally-intensive operation is *cv::findContours*, but the color space conversion as well as the thresholding are also relatively costly because they have to be performed on every pixel of the image. To achieve a better performance, as few as possible operations should be performed on the full frame. The following approach is

proposed: first, a strongly downsampled image is created and used to roughly identify the regions which could contain a marker, before then performing a precise and more costly detection algorithm. This reduces the amount of pixels to be processed to the size of the downsampled image and the size of all regions considered for further processing. The benefit may be small in a colorful environment, as many regions might be derived from detecting non-marker-related colors, but if most of the frame consists of the colorless keyboard, the number of non-marker regions will be low.

Major considerations have been made about the color space best suited for the thresholding operations. HSV and CIELAB are both color spaces suitable for the task with a similar color detection performance and conversion time. There is no great performance difference regarding the *cv::inRange* operation, which will lead to an imprecise region proposal. But for the most precise detection of a marker inside a small given region, different techniques can be used. The CIELAB color space is chosen for this task for the following reasons: In the CIELAB color space, high and low values in the alpha and beta channel are only present in pixels belonging to specific colors. Therefore, the presence of a color can be determined by evaluating only one channel. Additionally, a single color channel can be treated as a grayscale image. If for example the alpha channel is taken, the brightest pixels will correspond to red pixels of the full frame and the darkest to green pixels. Now, areas of red or green pixels can be detected using grayscale thresholding algorithms like Otsu's thresholding, which delivers much better results than the global thresholding *cv::inRange*. This greatly simplifies detection of the four extreme colors red, green, blue and yellow specifically and increases detection efficiency.

3.2.5 Key Detection

A naive approach for detecting the piano's keys might try to binarize the image and detect the contours of every key. This could be done because of the great contrast between black and white keys. Noisy contours could be discarded based on their size, shape and distance relations to the other contours. However, if the keyboard's top border is black as well, the algorithm will fail to detect the contours of the black keys and even if this can be avoided it would still be a source for imprecision. Additionally the borders between the white keys might not always be visible in the camera feed. So adjusting the tracking algorithm for all possible variations as well as establishing the relation between all found contours identified as keys would be very challenging. Lastly, *cv::findContours* is a computationally-intensive operation whose execution on the full frame should be avoided.

It shall be demonstrated that it is possible to make use of the special arrangement of keys on the keyboard of a piano in order to improve detection speed and accuracy in comparison with the naive approach. An algorithm which needs to evaluate far fewer pixels can be developed. Direct detection of the keys' outline and of the outer contour of the keyboard itself can be avoided.

Detecting the Black Keys

In the keyboard shape, all keys are vertically parallel to each other, all keys share the same upper boundary and all black and white keys have the same lower border, respectively. The upper and lower border of the white keys are the upper and lower border of the keyboard itself as well. As the marker provides a vague estimate of the keyboard's location and orientation in space, it is possible to draw a horizontal line roughly parallel to the keyboard's upper border which intersects with some or all of the black keys, dependant on the estimation accuracy. The intersections between the line and the black keys can be detected and treated as key positions in a single dimension. The positions on its own are meaningless, but if combined with positions from another parallel but vertically shifted line, key borders in 2D can be derived. Of course, lines which are scanned along should intersect with the same keys if possible. In a perfect frame, assuming perfect subpixel precision, every two intersection points of a key's border will produce the same vector. Given imprecision however, two points will effectively produce a higher resolution vector if further apart which results in a higher accuracy of derived key borders. The distance between the scan lines should therefore be maximized and is made relative to the detected marker height, so that it is, too, relative to the size of the keys in the frame. The task of combining intersection points from multiple scan lines to form vertical borders for the keys can be done with a simple matching. It is known which two points belong to intersections with the same key in the scan line as white-to-black and black-to-white gradients can be differentiated. If two pairs of points belong to the same key, a line drawn between their center points will contain only black pixels, disregarding reflection that might occur. This line's direction vector will also be referred to as the key's direction. Since all keys have the same direction vector in theory, matching could be handled based on this vector, too.

The front end of a black key can be calculated from a single line scan. As the horizontal end line is parallel to the keyboard's horizontal borders and orthogonal to the key's direction, knowing a single intersection point on the key's front end line is enough to derive the line segment. But since all those intersection points lie on a straight line in theory, a more robust approach is also possible. A line fit through all the black key's front end points will be treated as the mathematically correct representation of all front end line segments. The front corner points will be computed as intersections between this line and each key's vertical borders. The end points are denoted by a strong black-to-white gradient between the black and white keys and therefore can be found. It is not possible to directly determine the black keys' rear end visually. The reason for this is that the keyboard's frame is assumed to consist of a rather dark material, as is the case with most pianos. Therefore, the intensity gradient between black keys and the frame usually is too small to be detected but the intensity gradient between white keys and the frame is much higher and can be detected instead. As all rear end points of the black and white keys lie on the same line, it is sufficient to know a rear end point for enough of the white keys so that a line can be derived. The rear half of a white key is set as the part between two black keys with the same direction vector. A line scan upwards, opposite of the white key's direction, will find the intersection point with the

keyboard's frame and the line fitted through all those intersection points is taken as the rear end of all keys. The black keys can be described by their four corner points, which can now all be computed as the intersections between the key's vertical border, the line denoting all key's rear end points and the line formed by the black key's front end points.

Detecting the White Keys

The white keys' outlines are much harder to detect visually because their edges are rarely clearly visible in the frame. Instead, knowing the layout of a keyboard, all corner points of the white keys can be computed if the keyboard's lower border, which corresponds to the line formed by all white keys' front end points, is known. This line will be computed in the same way as keyboard's upper border. The white keys, unlike the black keys, have different shapes based on their note. Because of this, it is essential to assign notes to every black key before the next steps, so when looking at a pair of two subsequent black keys, it is known which shape they have and also whether one or two white keys lie in between.

At first, the black key which is closest to the leftmost detected marker is computed. This is done by finding the shortest euclidean distance between the marker's center point and all black keys' center points. The note of this key is known as it is denoted by the marker. The note of the black key to the left or to the right depends on whether one or two white keys lie in between. This can be determined by comparing the distance between those keys with the marker's width, which serves as an estimate for the width of a single key. Another way is to take a look at the piano scale and check which keys are expected and only assign sharp notes to the black keys. If all black key's were detected correctly, the second approach is more elegant and robust. The first approach however could be adapted to detect irregularities regarding the keys' arrangement. Detecting the white keys' corner points with respect to their lower part could be done by detecting the borders separating the keys. However, this is a difficult task as the borders are only recognizable by the shadows they cast which can vary strongly in intensity depending on the lighting. Another approach further exploiting the keyboard's composition is also possible and described in more detail here.

After all notes are assigned, the white key corners are calculated starting from the first pair of black keys. Conditioned by the note, the following cases of white keys in between have to be differentiated: 'D', 'E and F', 'G', 'A', 'H and C'. In the case of two white keys in between, the border separating them is set to be exactly at their middle. Otherwise, the left and right borders never align with the left and right borders of the enclosing black keys. Instead, they are set at a fraction of the black keys' width. This fraction is the same for every key of the same note on every piano following the construction guidelines explained above in Section 3.2.1 and can therefore be predetermined. This way, the vertical borders of all white keys can be derived. The leftmost and rightmost key of the piano pose a special case. They are always white and not enclosed by two black keys. Their presence in the frame can be predicated from the lowest and highest black key, respectively, if the piano's range in octaves is known. Otherwise, the keyboard's left and right vertical border could be detected in the

same line scan as the black key intersections, assuming the keyboard's range is known so the border can be distinguished from a key later. The following steps summarize the described techniques and serve as a basis for the technical implementation.

- Calculate an approximate initial orientation for the keyboard from the marker or markers.
- Scan along two lines perpendicular to the keys and detect segments of black pixels which correspond to black keys. Store the length and position of all those segments.
- Try to match each segment from the first line with a segment from the second line so they can form a key.
- Using matched pairs of keys, establish a more accurate orientation of the keyboard.
- Draw pixel wide stripes which cross the keyboard's outline top and bottom in such a way that a well detectable gradient is present. Detect the outline by detecting the edge in each stripe.
- Detect the bottom end of every black key by black-to-white gradient. Fit a line and combine it with the keyboard's top border and matched segments from previous steps for exact calculation of the black keys' corner points.
- Calculate the white keys' outlines by combining the black keys' outlines and the keyboard's outline using predefined ratios.

4 Implementation

In this chapter, the technical details regarding the implementation of the system described in the previous chapter are elaborated. First, the implementation of the tracking algorithm is described. The algorithm is mostly procedural but nonetheless structured into several larger steps, each fit into its own function, for a better overview and separation of concepts used. Every step takes its input data from the previous step, performs its task and prepares its output data most conveniently for the next step. This separation also serves the purpose of making future improvements easier. If a better method for a specific task is found, the corresponding step can be exchanged without effort. Smaller steps, like performing some sort of edge detection on a line, were designed to be as abstracted as possible from the general logic of the program for the same reason. The main steps are called one after the other in the program's *ProcessFrame* function, which will be called on every frame retrieved from the camera stream. They will be especially highlighted in order to expose the program's technological structure.

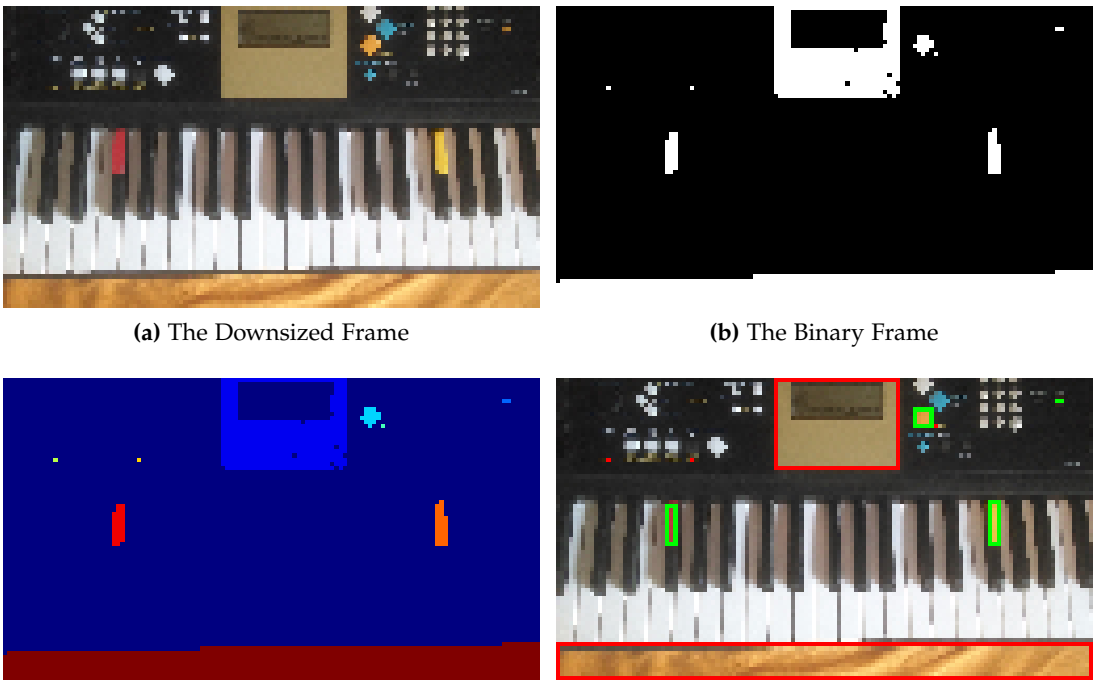
Coordinates in this chapter will be given according to the frame's coordinate system in OpenCV: The origin is at the top left corner, the x-axis corresponds to the frame's width and the y-axis to the frame's height.

4.1 Marker Detection Algorithm

The marker detection as the first part of the application is technically self-sufficient and the necessary starting point, as the key detection relies on a detected marker.

4.1.1 Finding potential Marker Regions

At first, *findMarkerROI* is called, which finds and returns a list of rectangles corresponding to regions in the frame where the presence of a marker is expected. It requires the frame and a list of related lower and upper boundaries for color thresholding, each corresponding to a specific marker, as input. The function before anything creates a strongly downsampled image from the original frame with *cv::resize* and parameter *INTER_NEAREST*. Nearest-neighbor interpolation is chosen for this task because it is one of the fastest interpolation methods and accurate enough in this case. The predefined *resizeFactor* denotes the scaling factor used and should divide both image dimensions as a whole. A smaller value leads to a faster execution of the step but if too small, the marker might not be significant enough in the frame to be detected. A value of *0.125* was small enough yet seemed to give reliable results. All future operations in this function will be performed on the downsized frame. The image is



(a) The Downsized Frame

(b) The Binary Frame

(c) Connected Components

(d) Filtered Connected Components

Figure 4.1: Visualization of Steps during Marker Region Detection
Figure **(b)** shows the frame after color thresholding. The output of a connected components algorithm on the binary frame is displayed in Figure **(c)**. Figure **(d)** visualizes discarded (red) and kept (green) connected components by their bounding boxes. They have been filtered by size. Retained regions will be further adapted before being submitted as potential marker regions.

converted to the CIELAB color space using *cv::cvtColor* in order to improve color thresholding performance. When using the BGR color space, it is very unintuitive to specify a color threshold and any threshold will be very sensitive to changes in lighting (see Section 3.2.3).

A binary image is obtained for every different marker color by applying the thresholding algorithm *cv::inRange* to the frame with the given lower and upper boundaries. Every pixel value outside the boundaries is set to zero. The bitwise OR combination [1] of all binary images obtained will be used in future steps, so only a single small binary image needs to be processed further (see Figure 4.1b). Next, a connected component algorithm is applied to the binary image (*cv::connectedComponentsWithStats*, see Figure 4.1c). The gained components are filtered based on their size according to a minimum and maximum size (*minSize*, *maxSize*). The minimum size filter is needed to eliminate single noisy pixels, whereas any component with $size > maxSize$ can be considered background. The minimum size was obtained by

[1] The bitwise or combination of two binary images can be imagined as overlaying both images with black being transparent and white being opaque.

examining the marker size in a frame where the user is as far away as possible but can still touch the keyboard. The maximum was obtained from a frame showing less than a single octave. Both tresholds include a margin to account for inaccuracies during detection and are scaled according to the *resizeFactor* to account for different frame sizes (see Figure 4.1d).

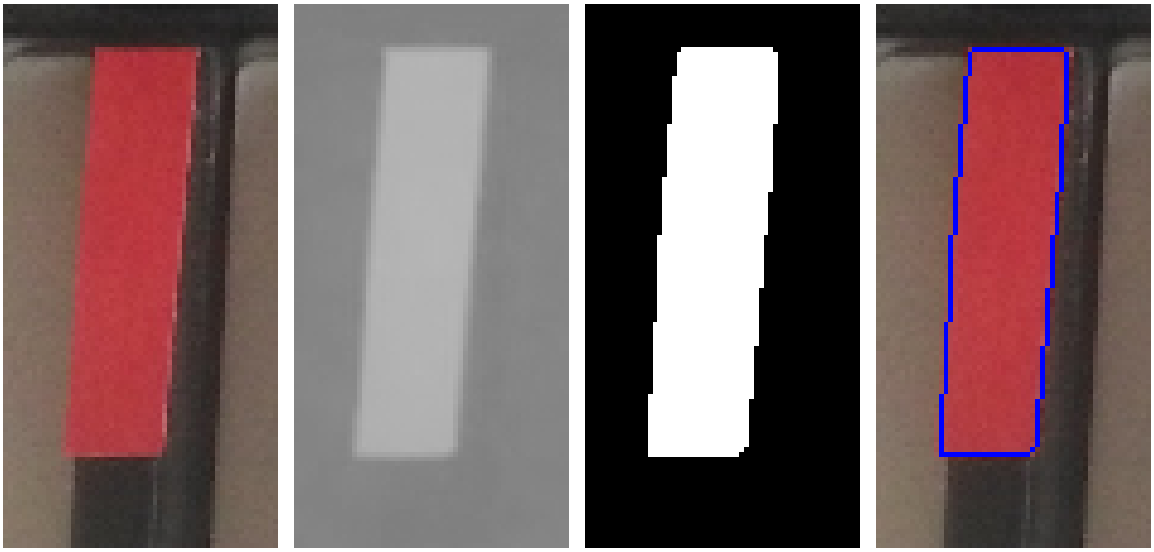
The average color of each of the connected components is calculated and stored. Finally, each components' bounding rectangle is computed, and the rectangles' coordinates are projected back onto the original frame by multiplying with *1/resizeFactor* to obtain a region that contains the detected connected component. To ensure that the connected component, which corresponds to a marker in the original frame, is fully contained within the region, a padding of *markerPadding/resizeFactor* in each direction is applied. The *markerPadding* should have a minimum value of 1 to account for the accuracy loss when downsampling. A higher value is only useful if the thresholding performance was unfavourable. Filtering based on the bounding boxes' size ratio is not possible as this depends strongly on the camera perspective.

After the bounding boxes are known, they are all iterated over in order to detect rectangles which intersect. In this case, the smaller rectangle is discarded. Due to their size and positioning on the keyboard, the marker's regions can never intersect with one another. Regions outside the keyboard so big that their bounding box would intersect with a marker's box should be discarded in previous steps based on their size. Therefore, discarding a marker's region here is very unlikely. This process it targeted at discarding small regions caused by reflections, which could lead to false positives in the next step, and generally reducing the region count. A vector of the obtained regions along with their contained component's average color is returned. If no marker candidate region could be found in the frame, *processFrame* aborts with error code 1.

4.1.2 Detecting accurate Marker Corners

As the second main step in marker detection, the function *findMarkerCorners* is called, which takes the return vector from the previous function and the camera frame as input. For every region separately, it decides if it contains a marker and calculates exact corners or rejects the region. Based on the average color passed, an octave is assigned to the marker. The function returns a vector of *MarkerDetection*s sorted by position from left to right. The custom struct *MarkerDetection* consists of the four marker corners, the assigned octave and a couple of utility functions for further processing. First and foremost, the specified region is extracted from the frame (see Figure 4.2a) and converted to the CIELAB color space. Depending on the expected color of the marker contained in the region, either the alpha or the beta channel is extracted and the other two channels are discarded (see Figure 4.2b). The extracted channel is treated as a grayscale image and blurred using *cv::GaussianBlur* in order to improve thresholding performance in the next step. A kernel size of 3x3 [2] is sufficient and only causes a small accuracy loss.

[2]The kernel's standard deviation sigma is computed with the formula $sigma = 0.3 * ((ksize - 1) * 0.5 - 1) + 0.8$, where *ksize* denotes the kernel size, and will be computed as such in future applications of *cv::GaussianBlur*



(a) Marker Region **(b)** Alpha Channel **(c)** Binary Threshold **(d)** Detected Contour

Figure 4.2: Steps during Processing of a Marker Region
A given marker region **(a)** is converted to the LAB color space and depending on the marker color, the alpha or beta channel is extracted **(b)**. The extracted channel is treated as a grayscale image and thresholded to obtain a binary image **(c)**. Finally, the largest contour in the binary image is selected as the marker's outline **(d)**.

Next, the image is converted to a binary image using *cv::threshold* with flag CV_TRESH_OTSU, which performs a thresholding operation using Otsu's method (see Figure 4.2c). Otsu's method calculates an optimal threshold assuming two classes of pixels in the image by examining the image's histogram. It is a rather costly algorithm but performs very well for varying lighting conditions. Other thresholding algorithms fail more frequently in this case. Dependant on whether the marker's color lies in the high or low range of the extracted color channel, thresholding is performed normally with flag CV_THRESH_BINARY or inverse with flag CV_THRESH_BINARY_INV so that the marker pixels correspond to white foreground pixels. To eliminate small noise, a morphological opening operation is performed with a square kernel of a small size. A kernel size of 3 had the desired effect.

Contours are extracted from the binary frame using *cv::findContours* [3] with flags CV_RETR_EXTERNAL and CV_CHAIN_APPROX_SIMPLE (see Figure 4.2d). The flags cause the algorithm to only return outer contours and perform a basic approximation of the contour chain, reducing the number of returned points without losing accuracy [4]. The largest contour is selected or the region

[3] *cv::findContours* performs a border following algorithm for binary images which was proposed by Suzuki et al. in 1985 [24].

[4] Horizontal, vertical and diagonal line segments are compressed by discarding all points except the segment's end points. Therefore, the simplified polygon is identical.

Figure 4.3: Output of the Marker Detection and Derived Information
The marker candidate regions which served as input for the accurate marker detection are shown as blue boxes. The outlines of accepted marker detections are drawn green, the outlines of discarded contours are drawn red. In cyan, the estimated upper border is drawn. The scan lines, which were derived by vertically shifting the estimated upper border, are drawn purple. They will be used to detect the black keys and should cross as many of them as possible.

is discarded if no contours were found. It is checked if this contour touches a border of the region it was detected in, in which case it is also discarded. The region was defined with a considerable margin in the previous step, so any true marker can be expected to not touch the border. But big surfaces which have a color within one of the marker's thresholds will often be detected more completely in this step due to using Otsu's thresholding method and fill up a larger area. This process proved well at disregarding the contours of those surfaces.

The contour needs to be further approximated by a 4-polygon. For this task, the function *cv::approxPolyDP* can be used, which applies the Ramer-Douglas-Peucker algorithm for decimating the contour's outline. However, the algorithm's parameters have shown to be difficult to tweak for varying conditions. The algorithm often failed to find the precise corners or returned more than four corner points for marker contours. Approximating the contour with a minimum area rectangle using *cv::minAreaRect* instead proved to be much more reliable. This method performs sufficiently well but should be improved in the future. The marker corners are sorted in the following order for convenience: top-left, top-right, bottom-left, bottom-right. To assure that no false positives are returned, the width-to-height ratio of each

detected marker is evaluated against a predefined ratio. In an optimal case, considering the marker is as wide as a key and half as high, a ratio of 1:8 could be present. For filtering, a ratio of 1:3 has proven sufficient to discard false detection successfully, leaving a high margin for imprecision in quadrilaterals corresponding to markers. Since the tracking algorithm relies on detection of a marker, an inaccurate detection is preferred over no detection at all. Besides, the key detection can still be accurate even if the marker was only poorly detected. Marker detection is finished upon returning a vector of structs which each represent a marker. If no marker could be detected, *processFrame* aborts with error code 2.

4.2 Key Detection Algorithm

4.2.1 Establishing Scan Lines

A rough initial orientation of the keyboard can be obtained from the marker detections. The function *getMarkerInformation* takes the vector of *MarkerDetection*s as input and derives two lines to scan along. If two or more markers were detected in the frame, a line is fitted through all markers' upper corner points using *cv::fitLine* with a least-squares distance function. If instead only one marker is detected, a vector is calculated as the mean of the two vectors from the top-left to bottom-left and top-right to bottom-right corners of the marker. The line formed by the perpendicular vector and the mid point of both upper corners is used in this case. The obtained line is treated as the upper keyboard border for now. The two lines to scan along for black keys are obtained by vertically shifting this line by a factor between 0 and 1, where 0 corresponds to no shift and 1 to a shift of double the marker's height, which is the estimated height of the key the marker was placed on.

It has become apparent that a line too close to the keyboards upper border is subject to increased visual noise and reflections when the keyboard has a large flat surface orthogonal to the plane of the keys (for example due the piano's fall board). Furthermore, when the line's direction vector is inaccurate due to marker detection imprecisions or inexact marker placement, strong deviations from the intended height at the image borders are possible. The scan lines should therefore not be too close to the upper or lower border formed by the black keys. Lines going through points at $\frac{1}{4}$ and $\frac{3}{4}$ of the black keys length have shown to perform well and commonly intersect with all keys under all conditions. Lines closer together are not advisable with the current precision of black segment detection, which will be discussed below. Two exemplary lines can be seen in Figure 4.3. If establishing the scan lines fails, an internal error is thrown.

4.2.2 Scanning for Black Key Segments

For a given scan line, along with the frame and the mean marker width, the function *detectBlackLineSegments* finds segments of subsequent black pixels which satisfy given requirements and therefore most likely belong to a black key. First and foremost, the scan line needs to be extracted from the frame and converted into a 1-dimensional matrix to allow for further

processing. This is done using the *cv::LineIterator* class, which implements Bresenham's line algorithm but provides extra functionality for operating on pixels. Bresenham's algorithm chooses optimal pixels for an 8-connected line and is computationally very cheap but does not interpolate between the pixels, meaning that subpixel precision is not achieved. The matrix is initialized to the line's pixel count and values are assigned iteratively. The original coordinates for all pixels are stored in a separate vector as this information cannot be stored in a 1-dimensional matrix.

Different approaches for detecting edges separating black and white segments in the line are available and have been tried. Arguably the most accurate detection results would be generated by an edge detection algorithm which evaluates pixel gradients, allowing for subpixel accuracy. Based on the kernel used for edge detection white-to-black gradients indicating the start of a black line segment and black-to-white gradients indicating the segment's end can be distinguished. However during testing, the parameters for edge detection have been shown to be difficult to tweak for different conditions. The edge detection algorithm tried was sensitive to small reflections which frequently occur on the black keys' surface and produce strong edges. Instead, a thresholding-based approach has turned out to work quite well.

First, the matrix is converted to grayscale using *cv::cvtColor* and a Gaussian blur filter is applied to the line matrix in order to increase thresholding performance. A kernel size of 5 is chosen in order to sufficiently blur small bright spots. The matrix is then converted to a binary matrix by applying a thresholding algorithm using Otsu's method. A global thresholding method is preferred as it discerns bright spots equally in the line and produces far less noise than a local thresholding method. Even if different lighting conditions are present within the line, the strong difference in brightness between black and white keys will lead to a good threshold value in most cases. To deal with small bright spots interrupting black line segments as well as small black segments caused by shadows between the white keys, a morphological opening operation followed by a closing operation is performed. A kernel size of 5 has shown to be successful in removing these artifacts while preserving the key segments.

After preprocessing of the line is finished, the line pixels are iterated over from left to right. Because only black segments fully contained within the line should be detected, constructing black line segments only starts after a white pixel was found. The number of consecutive black pixels up to the next white pixel is counted. If the segment's width is within a range denoting the estimated key width, it is saved, otherwise discarded. The allowed range of widths is based on the marker width but includes a great margin since the keys can have strongly varying widths based on the camera's point of view. A vector of segments denoted by their real coordinates in the frame is returned.

4.2.3 Matching Black Key Segments

For performing the task at hand of matching all segments correctly, various approaches have been considered, implemented and evaluated. The function *findSegmentPair* was defined, which takes the frame, a single line segment from the first line scan along with its index and the full vector of line segments from the other line scan as input. Three different ways of visually determining whether two segments match or not can be specified. The function iteratively matches the single segment with segments from the second line, starting off with the segment of the same index. Matching continues by alternating between left and right segments while moving further away from the single segment's index. The function returns true if a correct match was found, or false if all segments have been tried unsuccessfully or a maximum number of iterations has been reached.

The first method tried for determining a correct match is intensity-based. A line from the middle point of the first segment to the middle point of the second segment is iterated over. All pixel values along the line are added and the average pixel value is calculated and converted to a single grayscale value [5] If the intensity value is lower than a predefined threshold, the match is considered correct. This method did not prove to work well under varying lighting conditions as it does not deal with them.

The second method evaluates edge strength along the same line. The line is extracted and converted to a 1-dimensional matrix. Edge strength at an index is calculated by convolving the line with the kernel $[-1\,0\,1]$. If any edge strength is greater than a predefined value, the function returns false, otherwise true. This method worked very well under certain conditions, but produced a lot of false negatives in other cases. If two segments are a correct match, but the black key contains a small reflection, the line may intersect with it, and a very strong edge will be present.

The third method exploits that even under different lighting conditions, a majority of pixels along the line can be assigned to a white or black key. The line is iterated over and white and black pixels are counted. Pixels are converted to grayscale and discerned based on a predefined threshold [6]. The function returns true for a correct match if the number of black pixels is at least four times greater than the number of white pixels. This will never be the case if both segments belong to different keys. This method handles reflections well as the usual size of a reflection is less than one fourth of the evaluated line. It has been shown to be superior to both previously described approaches in most cases.

Lastly, the function *findSegmentPair_Dir* has been defined, which is constructed the same

[5] A custom function for converting a single scalar to a grayscale value was defined, since *cv::cvtColor* can be called on matrix objects only. It uses the formula $Y = 0.299R + 0.587G + 0.114B$.

[6] A value of 110 has been chosen for this threshold, which is below 127.5, the mid value of black and white. The reason for this is that even in bright light, the black keys will be rather dark, but in low light conditions, the white keys intensity values can be very low.

way as *findSegmentPair* but implements a direction based segment matching. It calculates the angle between the vector formed by the two segment's mid points and a given target direction vector. If the angle is smaller than a predefined value, true is returned. A value of 15 degrees worked well in all conditions. This method is preferred over any visual approaches since it is superior in terms of performance and precision if the target direction vector is precise. However, when using this approach, it is important that the camera was correctly calibrated beforehand. The direction of the keys can vary significantly between the leftmost and rightmost key due to lens distortion otherwise.

4.2.4 Detecting Black Keys

In order to store positions of a black key and provide convenience functions, the struct *PrototypeKey* was defined. To perform the task of forming black keys from segments, a combination of the approaches described above has been chosen. The key direction vector estimated from the marker has been shown to not be accurate enough for direction based segment matching. Instead, the function *calcAvgKeyDir* has been implemented. It chooses a small, given number of segments in the first line, roughly equally distributed along the keyboard, and matches them using the third method described. Next, the mean vector between all matching segments is calculated. To prevent a possible false matching from degrading the accuracy of the estimated vector, all vectors between two matched segments are compared with this mean vector. Only the vectors that do not deviate more than 15 degrees are then used to calculate the final mean vector, which is returned.

If just one marker has been detected previously, the obtained mean vector will most likely estimate the keyboard's orientation better than the marker. Therefore, using the newly obtained direction vector, a new keyboard orientation is estimated and two new scan lines are derived the same way as described above. Segment detection is performed again for both scan lines, and the number of detected segments in both lines is counted. If the minimum number of segments detected in either line is greater than the minimum number of segments in either of the previously scanned lines, the newly detected segments are kept and the whole process of matching is performed again. Otherwise, the algorithm continues using the already matched segments. This way, it is assured that as many keys as possible are detected.

Now *findSegmentPair_Dir* is called for every segment of the first line subsequently with the direction obtained from *calcAvgKeyDir* and correct matches are stored. The average key direction vector is recalculated from the direction vector of every key detected this way. For every matched segment pair, an object of type *PrototypeKey*, which only consist of two sides for now, is created and stored in an array (Figure 4.4). Corner points, octave and note will be calculated later for every key in the array. The count of keys in the array is exactly the number of black keys to be fully detected for this frame.

Detecting the Black Key's Lower Border

In the next step, the line formed by the lower end of all black keys is calculated, which will be used for calculating the black key's lower corners later. For detecting the lower end of a single black key, a line is defined by the bottom segment's mid point and the key's direction vector. The vector is expected to have a length of half of the key's height and the point is expected to be at three quarters of the key's height, therefore the line is expected to cross the black key's lower border (Figure 4.4). Multiple approaches for detecting the start of a white key in the line have been implemented and tested.

A naive approach tried works by finding the first pixel with a high intensity value, expected to belong to a white key, in the line. The same threshold as for matching segments was used. A more complex approach is based on finding a strong intensity gradient. The gradient strength is computed the same way as for matching segments. Now, either the strongest edge in the line or the first intensity gradient stronger than a predefined value is considered the end point. All three methods explored failed regularly, however they mostly failed in the same cases, in particular when reflections or strong shadows were present. Especially when the defined line intersected with the border between two white keys, all methods failed. Nevertheless the clear majority of end points was detected correctly in each frame which has shown to be sufficient. A line will be fitted through the points using *cv::fitLine* with a *RANSAC* [7] algorithm. As falsely detected end points are rarely close to the end points, they are easily discernable as outliers. Therefore the fitted line is as precise as the inliers' precision (see Figure 4.4). Because of the similar detection performance, the intensity based approach was selected for being the simplest method.

Detecting the Keyboard's Upper Border

Finally, the keyboard's upper border is detected, which denotes the upper end of all keys. This task can be approached the same way as detection of the black key's lower border. But since the intensity gradient between a white key and the border is expected to be much higher than the gradient between a black key and the border, scan lines will be defined as ranging from the white keys across the border. The function *findKeyboardUpperBorder* derives scan lines from the known black key borders, detects the likely border crossing in each of them and fits a line with all the points obtained.

The function iterates over all two subsequent black keys and selects a point in the middle of their two upper segments. The scan line is defined by this point and the opposite of the mean key direction vector of both black keys. Due to the keyboard's composition, there are cases when two white keys lie in between the two black keys, therefore the selected point would lie directly on the border between the two white keys and border detection on this line would most likely not produce meaningful results. Therefore, every two subsequent

[7] RANSAC (Random Sample Consensus) is a method capable of detecting and disregarding outliers, e.g. points which do not lie near the line fitting the majority of points well.

black keys with a distance more than 1.6 times greater than the estimated key width are skipped as they are assumed to have more than one white key in between (see Figure 4.4). Skipping these keys instead of recalculating the scan lines in an optimal way has shown to not pose a problem in terms of accuracy. For detecting the border crossing in each scan line, is has emerged that the intensity-based approach described previously performs sufficiently well. The white keys will usually be the objects in the scene with the highest intensity and regarded as such and reflections on the white keys only produce even higher intensity values. An intensity threshold of 120 has proven to produce both few false positives and negatives, even when the border consists of a non-black material [8]. Nevertheless, a RANSAC line fitting algorithm is used for increased stability and accuracy as outliers are still possible.

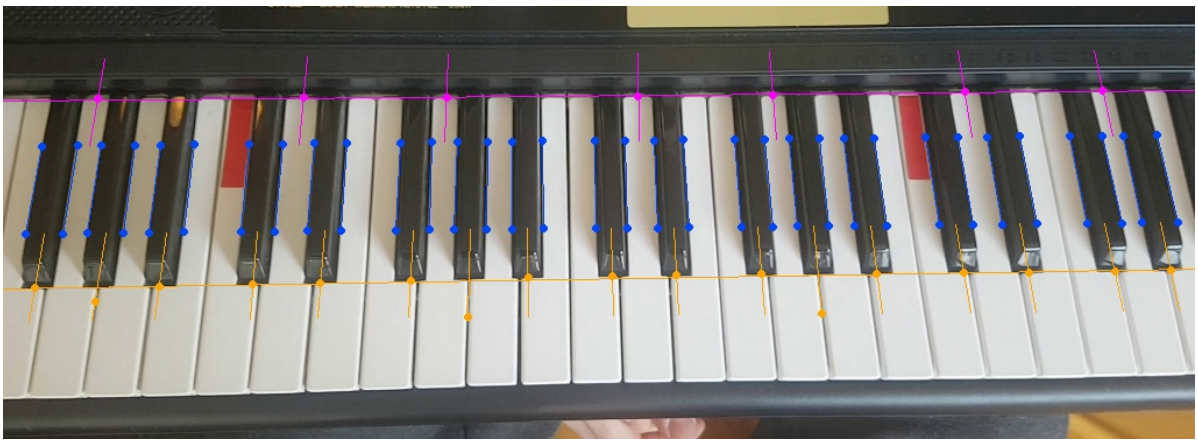


Figure 4.4: Straights required for Calculating the Black Key's Corner Points
All lines used for calculating the black key's corner points can be seen in the Figure. The matched segment points, representing border segments of the keys, are drawn blue. Scan lines and returned border crossings used for getting the black key's lower border are drawn orange. It can be seen very clearly here how outliers are disregarded when fitting a line to the set of points using a RANSAC algorithm. The scan lines and obtained border crossings used for getting the black key's upper border are drawn pink.

Calculating Corner Points

Now, the upper, lower, left and right border of every black key is known and exact corner points can be computed from their intersections. The corner points are stored as floating point values in the *PrototypeKey* array. At this point, detection of all black key's outlines is finished.

[8] Of course, a piano and thereby the border could have any color, including white. As no statistics on piano color could be found, some inexhaustible research has been done. The conclusion has been drawn that most grand pianos are in fact black. Other pianos often have a wooden material, which is dark enough for the proposed detection method. Even colored, gray and white keyboards often still have a small black stripe between the keyboard's body and the keys. However, lightly colored pianos do exist as well and the proposed algorithm will not work with them.

4.2.5 Detecting White Keys

The white keys' outlines can be fully computed from the black key's corner points, which are known, and the keyboard's lower border. In order to do this, a scale must be established on the keys and every black key needs to have a note assigned to them. The white keys will be created as objects first before detecting the keyboard's lower border as this task is more convenient to do when making use of all positions that can be derived so far.

Assigning Note and Octave

First, the key which the marker was placed upon is retrieved as the key with the smallest distance to the marker, computed from center point to center point. In case of multiple marker detections, the first marker is taken. The note and octave which were assigned to the marker beforehand are now assigned to the key as a starting point. As the keyboard's scale is known, the note of the leftmost black key with index zero in the key array can be computed by the index-wise distance to the marker key. A vector holding objects of the type *PianoKey*, which can store and differentiate white keys and black keys, is initialized and the leftmost black key is converted and pushed back as the first object. In a *PianoKey* object, a black key is defined by four points and a white key by eight points, separated into an upper part with the same height as a black key, and a lower part.

Now, the *PrototypeKey* array is iterated over in pairs, starting at index zero. The key at the current index is declared as left, the key at the next index as right. If the left key's note indicates that only one white key is in between the pair [9] , its upper part's positions are set the following way: the white key's left side is the same as the left key's right side and the white key's right side is the same as the right key's left side. Otherwise, two white key objects are created. The left white key's left side and the right white key's right side are assigned as before. The respective other side is the same for both keys and computed as the middle of the left key's right side and the right key's left side. The note and octave are assigned up to the right key by moving the scale's index forward one step at a time.

The white key object or objects are pushed to the key vector. Additionally, the right key is converted to a *PianoKey* object and pushed back as well. If the right key is the last key in the *PrototypeKey* array, this process stops and the array can be deleted.

Detecting the Keyboard's Lower Border

Finding the keyboard's lower border is nearly the same task as finding the keyboard's upper border with the difference that the scan lines' directions are reversed. It is, however, not known exactly how high the lower part of the white keys is. Scan lines can now be derived from the white keys directly instead of using the black keys' positions as before. The function

[9] Two white keys lie between the black keys "D#" and "F#" as well as "A#" and "C#". It is therefore sufficient to check whether the left key's note is either "D#" or "A#".

findKeyboardLowerBorder requires the frame and an array of *PianoKey*s as input. For every white key in the array, a scan line is calculated using the mid point of both lower-upper corners, referred to as $bMid$, and the key's vector v. The scan line is defined by the two points $p_1 = bMid + v \cdot \frac{1}{4}$ and $p_2 = bMid + v \cdot \frac{3}{4}$ in order to assure that all scan lines cross the lower border [10]. To detect the border crossing in each scan line, the same function as for detecting the upper border is used, but a different threshold can be specified. Again, a RANSAC line fitting algorithm is used for the previously mentioned reasons.

Calculating Corner Points

Finally, given the vector of *PianoKey*s and the lower keyboard border, the function *calculateExactWhiteKeyPositions* iterates over all white keys subsequently and calculates their lower parts. For each white key, the next black key to the left and to the right is taken and defined as $bLeft$ and $bRight$, respectively. Their lower corner points are denoted as $cLeft$ and $cRight$. The upper corners of the white key's lower part are referred to as $w.topLeft$ and $w.topRight$. Now, the note of each key is identified. In the case of "C" and "F", the key's left border is a straight line, therefore the upper part's lower corner and the lower part's upper corner are identical and set as such. In the case of "E" and "H", this applies to the right border. All other upper corners are computed as such:

$$w.topLeft = bLeft.cLeft + (bLeft.cLeft - bLeft.cRight) \cdot f$$
$$w.topRight = bRight.cLeft + (bRight.cLeft - bRight.cRight) \cdot f$$

where f denotes a factor which is defined for both sides of a key for every note and describes how much a given white key wraps around the black key on the left or right side as a fraction of the black keys width. For example, the factor $c2d$ describes the positioning of the border separating the keys "C" and "D" and therefore is the same as the factor $d2c$. The following factors have been defined with the following values: $c2d = f2g = 0.7$, $d2e = a2h = 0.3$ and $g2a = 0.5$.

After all upper corner points have been calculated as such, the lowest corner points are calculated by intersecting a line, ranging from the upper corner point downwards in the key's direction, with the keyboard's lower border. The key's direction is hereby calculated separately for every key according to the corner points of the key's upper part. When all corner points have been calculated, *processFrame* terminates with error code 0 to indicate a successful execution. A vector of the detected keys is returned, where each key is described by its corner points in the frame's coordinate system and its note and octave.

[10] The height of a white key's upper part is about 9 cm and the height of the lower part is about 5 cm on a grand piano. The measurements vary between manufacturers and keyboard types, but other keyboards, especially old ones, usually have shorter keys and a smaller ratio between black and white key length.

Figure 4.5: Visualization of Detected Keys

4.3 Piano Tutoring Application

4.3.1 AR Camera

The Android camera can be opened in Unity using a *WebCamTexture*. It enables control over different camera settings such as resolution and can be set to a texture directly. In that case, the texture is updated automatically. Using the Unity UI system, the texture is centered on a *Canvas* set to size of the Android phone's screen. An *AspectRatioFitter* is attached to the texture and controlled with a script to maximize the size of the texture inside the *Canvas*.

4.3.2 User Menu

In the application's current state, input is handled solely via the touch screen. This is sufficient for testing purposes but very self-defeating if the smartphone is stuck inside a HMD and should be changed as part of future work (see Section **??**). The user menu can be opened by tapping anywhere on the screen. It consists of buttons and sliders aligned in a vertical view. One song at a time can be loaded with the Button *Load Song*. When pressed, a file browser is opened and a file from the smartphone's hard drive can be selected. The *Simple File Browser* plugin[11] by Süleyman Yasir Kula has been used for this task. After being loaded, playback of the song can be started with the button *Start Playback*. The playback speed can be set with a slider. The application can be closed by pressing the *Exit* button. It is possible to change settings of the tracking module inside an advanced settings menu. If the tracker fails too frequently in a given environment it is often due to a specific, fixed threshold value. By controlling the parameters within the application, the tracking module can be easily adjusted to the environment. Manually changing the parameters, which would require rebuilding and manually replacing the library in the Unity project, can be avoided.

[11] `https://assetstore.unity.com/packages/tools/gui/runtime-file-browser-113006`

4.3.3 Loading and Playing Midi Files

Midi files do not contain audio information but instead a set of instructions which describe a way to make music. They can contain multiple tracks for different instruments. A single instruction, also called "event", consists of three bytes: a status byte and two data bytes. The status byte indicates the type of instruction, for example a note trigger or a system event. Additionally, each midi file contains a header where overall information about the piece, such as base tempo, is defined. For the prototype, only midi files written for a single piano player, therefore only containing one track, will be considered. Parsing all possible event types is not needed as only key triggers need to be obtained. Key presses and releases are indicated by the status byte. In that case, the first data byte contains the pitch and the second data byte contains the trigger velocity. A special case of "note-off" events is posed by "note-on" events with zero velocity. The pitch is specified in intervals corresponding to a piano tuning and given as a whole number ranging from 0 to 127. Each key on the piano is therefore assigned to a fixed number.

Parsing the Midi File

In order to retrieve which keys are pressed at a point of time in the musical piece, the midi file needs to be parsed. For this task, *smflite* by Keijiro Takahashi was used. It is a minimal class library for handling standard midi files inside Unity and provided free of charge without restrictions. *smflite* is written in C# and does not use any os-specific functions, making it platform independent to a wide extent. Various other libraries with more extended capabilities as *DryWetMidi* and *NAudio* were tried, however none of them were able to run on an Android system.

Playing the Midi File

The midi file is played by a sequencer. In every frame, the sequencer is advanced by the time passed since the last frame. Events that occurred in the meantime are returned and can be applied in the application. Manually adjusting the tempo of the piece is done by adjusting the global speed in Unity. For some augmentation styles, it is necessary to not only know the notes currently played but also the upcoming notes. This is difficult to handle via the sequencer. Instead, events are not applied directly when they occur, but a delay, which can be set, is introduced. When a note-on or note-off event is notified, it is stored in a vector along with its delay time. The delay time is then reduced in each frame until reaching zero. Visual effects, like a note fading in, can be based on the remaining delay time. Note-on events with a delay of zero are active until removed from the vector by the corresponding note off event.

4.3.4 Visualization

A simple visualization approach has been chosen as it is not the goal of the prototype to provide complex augmentations. Keys will be highlighted by a corresponding mesh of the same size and shape. For calculating a mesh, the corner points of the key are taken and

converted to a space ranging from zero to one in both directions. It is not the goal of the prototype itself to display complex augmentations; instead, a system is provided.

4.4 Tracker Integration

As previously elaborated, the tracking module and the Android application need to communicate with each other. Optimally, the tracking functions would be embedded into the Unity project and compiled by Unity for all the different platforms, but this is not possible as Unity can not compile native OpenCV code. Therefore, the tracking module needs to be compiled separately and then loaded inside the project.

4.4.1 Building the Tracking Module

A way of providing an already-compiled program while enabling other programs to call functions is by building a dynamically linked library file. To enable function calling from the outside, the respective functions need to be exposed with a specific syntax. As good practise, every internal function that needs to be called will be wrapped in another function with minimal function arguments and only necessary functions are exposed. This way, the internal structure is invisible and independent. This library file needs to be built separately for every different platform such as Windows or Android. Building for Windows will produce a *.dll* file and building for Android a *.so* file. Compiling code for Windows is expected by OpenCV and straightforward after all of the required libraries have been linked. Compiling native OpenCV code for Android is more convoluted, however, since an Android port wrapping OpenCV in Java has been developed. It is nevertheless possible, but requires a more complex setup. The OpenCV SDK for Android as well as the Android NDK need to be linked and additional settings regarding C++ code compilation are necessary. Additionally, some C++ syntax has to be adjusted as it is not supported by the Android compiler. For this purpose, a second Visual Studio project [12] has been created with the same source files and set up accordingly. The obtained library files can be treated as Unity-Plugins and loaded by Unity if placed in a specific folder. The respective target platform can be set inside Unity, which then handles including the correct library file for the project build.

4.4.2 Communication

Since the tracking module was written in C++ but C# was used as the scripting language inside Unity, not all functions can be called naturally. Instead, means of communication between both systems need to be established. Despite being very similar, C++ and C# still have some different data types. Especially when it comes to OpenCV, which defines a lot of custom data types, it is not always trivial to correctly pass information from one system to the other. As was described previously in Section 3.1, only the video frame needs to be sent to the tracking module and only an array of key detections need to be retrieved. Because this

[12] Visual Studio has been used for compiling the OpenCV application.

needs to be done every frame, efficiency is crucial. Passing the frame for use in OpenCV is especially challenging due to the naturally large size of images. Copying the frame should thereby be avoided at all costs.

Inside Unity, the video stream is contained within a *WebCamTexture* which handles retrieving each frame from the camera. The raw pixel data of the current frame can be acquired by calling *GetPixels32* on the *WebCamTexture*. Is is stored as an array of the data type *Color32*, which consists of four bytes for the red, green, blue and alpha channels. Since the size of the frame should not change while the application is running, it is possible to only allocate the memory for the array once with a fixed size and overwrite the data every frame. OpenCV expects pixel data as a continuous array of bytes, so in theory, this pixel representation can be read by OpenCV aswell. However, parsing the array of *Color32* in OpenCV has shown to be slow and produce memory leaks. No definite reason for this behaviour could be found, but it is suspected that the different block size of the data types might be a reason. Instead, the pixel data is converted to a byte format in Unity using the function *Color32ArrayToByteArray*[13]. The conversion is done using C# marshalling functions[14].

First, the memory for the byte array is allocated at a fixed memory location so that it cannot be moved or collected by the C# garbage collector. Then, each element from the *Color32* array is copied to the byte array. Finally, the pointer to the byte array can be passed to the tracker module. In OpenCV, a *Mat* object wrapping the pixel data for image processing can be created from this pointer without allocating new memory. Now, the internal processing of the frame is triggered and the number of detected keys is returned, which is required in order to obtain the actual key data. Since deleting the *Mat* object does not release the pixel data, the byte array then needs to be freed in Unity. If any keys were detected, the function *RetrieveKeyArray* is called which returns a pointer to an array containing the key positions. A new structure has been defined on both sides of OpenCV and Unity to store corner points and note of a key using only the basic data type *int*, which is defined equivalently in C++ and C#. Using marshalling functions, the array can be reconstructed in Unity from the pointer and the array length, which is equal to the number of detected keys.

For testing purposes, functions for creating and managing a video capture in OpenCV have been created, too. In that case, the frame is copied to a fixed pointer location provided by Unity where the data in memory can assigned to a texture directly. Copying the frame this way has shown to be much faster than the opposite direction, which was described above. However, it was only tested successfully on Windows whereas on Android, the device's camera could not be accessed.

[13] The function was taken from StackOverflow and slightly modified. URL: `https://stackoverflow.com/questions/21512259/fast-copy-of-color32-array-to-byte-array`

[14] C# includes various marshalling function greatly simplify communication with other languages. For example, predefined data structures can be created from raw memory send to a fixed location in memory.

5 Program Evaluation

5.1 Detection Algorithm Performance

All steps in the tracking program have been designed with regard to computational expense. The results are presented here. Since *processFrame* is the only function being executed after initialization of the program and therefore solely determines the continuous runtime, it suffices to critically evaluate this very function's performance. As an effort to compare the algorithm's individual steps and assess their share of the total computational expense, the major step's processing time has been measured separately. For this task, a namespace containing utility functions has been defined. The time consumption of a piece of code can be measured by calling *StartMeasurement(const char* message, int ID)* at the beginning and *TakeMeasurement(int ID)* at the end of the code block with the same arbitrary but unique *ID* and a *message* describing the code block for referencing. The measurements are stored across all runs through *processFrame* grouped by *ID*. By calling *DisplayMeasurements*, each mean of all measurements per *ID* is printed along with the *message*.

The measured values can be seen in table 5.1. They were obtained on a laptop with 8 GB RAM and a i7-2670QM processor (4x2.2GHz) using a video of 1024x576 pixels with 373 frames. The piano keys were successfully detected in all frames of the video. The inherent workings of the color detection may cause the algorithm to perform slower in a more colorful environment as more potential marker candidates regions are detected. In case of a detection failure in the same environment, the algorithm always performs faster. It became evident that after the initial execution of *processFrame* subsequent executions were faster by a factor of up to 100. It is assumed that this effect is brought about by the mechanics of employed optimizations, where memory only needs to be allocated once, and deemed irrelevant due to the high amount of runs that measurements were averaged over. It can be seen that almost three quarters of the detection algorithms processing time is spent during marker detection. In particular it outweighs the actual detection of individual keys as the proposed novelty by far. As insight, it is stated that the marker detection should be optimized further in terms of performance. On the other hand, detection of black segments in the scan lines is a very fast process. Achieving better accuracy in this step by making performance sacrifices would be reasonable. The detection algorithm could successfully process 234 fps on average. This demonstrates the efficiency of the proposed approach.

Program Step	*Execution Time in ms*
Find Marker Regions	1.037
Find Marker Corners	2.245
Estimate Scan Lines	0.0106
Detect Segments in both Scan Lines	0.00034
Calculate the Average Key Direction	0.0214
Single Marker: New Scan and Matching	0.351
Segment Matching, Black Key Creation	0.00988
Detect the Black Key's Lower Border	0.158
Detect the Keyboard's Upper Border	0.0603
Calculate Black Key Corner Points	0.00539
Assign Scale, White Key Creation	0.0235
Detect the Keyboard's Lower Border	0.217
Calculate White Key Corner Points	0.00752
Total Processing Time	4.272

Table 5.1: Execution Time of Steps from *processFrame*
Measurements were performed on a laptop with 8 GB RAM and a i7-2670QM processor (4x2.2GHz) using images of 1024x576 pixels. It can be seen that the marker detection takes significantly more time than the key detection afterwards.

5.2 Detection Algorithm Failing Points

In this section, conditions under which the tracking algorithm failed will be described and examined. Such can be be engendered by certain properties of the piano, the environment or the camera's position.

Piano

Certain assumptions concerning the piano have been made. Were they not met, the tracker would fail. The keyboard's layout must be as described in section 3.2.1 and the keys must be black and white. The keyboard's frame is required to consist of a sufficiently dark material and also surround the keyboard completely.

Environment

The first environmental factor is the lighting of the scene captured by the camera. Because of the fixed thresholds used in some functions, the algorithm will fail in especially dark or bright settings. Detection of the marker can also fail in those settings, as even the chroma channel in the CIELAB color space is somewhat sensitive to lighting. Another difficult condition is

created by shadows that fall onto the keyboard. When performing the segment detection on the scan lines, it is important that the visible area of the keyboard is equally illuminated. Additionally, sharp shadow lines can be mistaken for the edge of a key, leading to false segment and eventually false key detection. This will lead to a heavily distorted calculation of white key positions and a falsely assigned scale. A special case of this problem is posed by light coming from the side, causing the black keys itself to throw shadows onto the keyboard. Strong reflections can cause inaccuracies during some detection steps. Another definitive failing point is posed by objects in the scene that resemble the marker design close enough to be detected as one. Even if another marker is detected correctly in the same scene, the keyboard's orientation estimate will most likely be so bad that no keys can be found.

The second environmental factor is posed by objects occluding the piano. A special case is posed by the piano player's hands as the occlusion caused by them can not be prevented. As the hands mainly occlude the lower part of the piano, it is important that the keyboard's lower border is still visible in about half of the scan lines, otherwise its detection will fail. Another problem arises when black keys are occluded at the location where the scan line is crossing. Since in that case no black segment can be detected, the key can not be created by matching two segments. This issue should be solved as part of future work by deriving the corner points of those keys using positions of other, well detected keys.

Camera Positioning

For describing the camera rotation, the following coordinate system is defined onto the keyboard from the perspective of a piano player. The x axis goes along all keys from left to right, the y axis goes along a key's edge from upper to lower end. The z axis now describes the camera height above the keyboard. The optimal positioning of the camera is above the keyboard facing down along the -z axis. Regarding the image section of the camera, it is important for a successful execution of the tracking algorithm that the white key's upper parts are always visible and sufficiently wide. A low camera angle around the y axis will cause the black keys to cover the white keys' upper parts. A low camera angle along the x axis is less problematic, however following the perspective of an angle lower than 90 degrees, keys and marker appear shorter. This leads to a decrease in detection accuracy and less margin for detection errors in individual steps up to the complete failure of the detection algorithm if the angle is too low. The rotation around the z axis can not be larger than 90 degrees as the marker was not designed with a facing direction. Additionally, the image section's width along the x axis naturally decreases the more the camera is rotated around the z axis. A rotation larger than 45 degrees will cause frequent failures.

6 Future Work

6.1 Detection Algorithm Improvements

A lot of improvements to the detection algorithm have come to mind, but have not been implemented yet due to a lack of time and experience with certain concepts. Some of the proposed enhancements might improve but others might impair performance. An approach on detecting and correcting errors during some detection steps will be presented, too, which will have a performance cost. However as the performance is deemed very high, as presented in section 5.1, improvements entailing a performance penalty are acceptable.

6.1.1 Enhancements

Line Extraction

When extracting the scan lines in all parts of the algorithm, this is done using Bresenham's line algorithm. This algorithm is very simple and fast, but does not interpolate between pixels on the line. By using a different line algorithm which is able to retrieve an interpolated color for points on the line, precision of the extracted line could be increased greatly. Unfortunately, OpenCV only provides Bresenham's algorithm for iterating over pixels in a line, so another algorithm would need to be implemented. It might be possible to adapt Wu's line algorithm, which supports antialiasing, to extract lines.

Scan Line Length

Another improvement regarding the scan lines could be made. In the implemented detection algorithm, the scan lines are often much longer than they would need to be to account for inaccuracies but also varying piano keyboard dimensions. If every user could measure and input his keyboard's exact dimensions, these scan lines could be reduced in length, reducing computation time. Since a shorter line, set the case that it still intersects with the border, contains less total noise, the number of falsely detected border crossings could be reduced too. For example when detecting the keyboard's lower border, the scan lines are defined quite long since the height of the white key's lower part varies between different piano manufacturers.

Edge Detection in Scan Lines

Many tasks in the algorithm come down in their essence to finding a specific edge in a line. They have been solved using approaches of evaluating single pixels instead of looking at pixels with respect to their neighborhood. Fixed thresholds have been used, making the

algorithm prone to lighting changes. Using an edge detection algorithm instead should have several advantages. Edge strength is less sensitive to overall lighting changes in the scene. It is also less sensitive to soft shadows as it deals better with incremental lighting changes. Additionally, edge detection can be performed with subpixel precision. Especially when calculating direction vectors, for example after matching segments obtained in the line segment detection, a higher precision can make a big difference.

Dynamic Threshold Values

Another approach to improving the process of finding edges in scan lines as well as differentiate pixels which belong to black and white better has come to mind. An image histogram could be calculated in each frame and the most probable colors for black and white could be extracted. Then the intensity thresholds can be based on them. This could especially improve the process of initial segment matching as is performed now. To apply as little of a performance penalty as possible, the image histogram could also be calculated using the downsized frame obtained during marker region detection. It would have to be determined if the downsized frame still provides enough accuracy for calculating the histogram.

6.1.2 Extension: Error Correction

In the algorithm, it is known if certain steps fail completely, but at most times, it is not known if the results are meaningful when a step terminates successfully. As all steps are build on top of each other, detecting and correcting errors during crucial steps could greatly reduce the number of detection failures.

Failed Detection of Individual Black Keys

Calculation of the white keys depends on the correct detection of a black keys. But if a single black key does not get detected correctly, the scale will not be assigned correctly to all keys and white keys will be calculated with wrong corner points. The output of the detection algorithm itself can not really be considered meaningful then. However, it would be possible to detect if black keys are missing from the detected keys and even construct them with most likely corner points. The average distance between the black keys can be computed after segment detection and matching. It would be important to only regard similar distances in order to not falsify the average distance calculation. As two separate distances between black keys are present on the keyboard, they would have to be calculated separately. After assigning a note to every key, it could be checked if the distance between two keys matches with their assigned notes. If a key is missing, the note of the keys previous and next to the missing key each determine which one of the two previously calculated distances is expected instead. Using the keys' average direction vector, the four corners points could be estimated.

Keyboard Orientation Correction

In theory, the keyboard's upper and lower border as well as the black key's lower border are all parallel to each other. This could be exploited to check if all three borders got detected correctly. If the direction of one border differs significantly from the direction of the two others, it can be considered as incorrectly detected. In that case, the detection procedure for that border could be performed again with different settings until considered successful. If difficult to detect, all borders could be estimated if a marker was detected and the exact measurements of the keyboard are known. Accuracy would most likely be low if using a single marker detection but much higher if two markers were detected.

6.2 Android Application

The prototype application provides all the expected basic functionality but not in a user-appealing way. Before being released into the market, more work needs to be done regarding the input, visualization and features. A visualization method should be implemented which can resemble any of the approaches presented in Section 2.2.1. Using touch input, as is the case for the prototype, is not advisable since the smartphone is supposed to be stuck inside the HMD. Instead, different input methods should be implemented. Gaze-based interaction could be used. For a specific HMD, it is advised to utilize the provided input options if any. The user menu should then be adapted to function in a 3D environment and also made visually appealing. A menu stabilized at world space coordinates is suggested as it is less disorienting for the user. As few settings as needed should be exposed in the menu to prevent unnecessary complexity. It is suggested to implement an option for setting the application's target frame rate in order to influence cpu load and battery consumption of the smartphone.

6.3 Building for a Smartphone-Powered HMD

The piano tutoring application prototype was designed to be deployed on a smartphone-powered HMD. As it has been successfully deployed on Android already, it only needs to adapted to the respective device. Unity integrates a lot of SDKs required for building the project for a specific AR or VR viewer. However, setting up a VR viewer for visual see-through AR is not trivial as this is usually not intended. The real time camera stream needs to be properly calibrated and positioned in the scene. The Unity UI system, which was used for the prototype, might not be the best tool for this task.

It could be preferable to integrate a ready-made AR camera solution which can naturally handle different HMDs. The only strict requirement for such a toolkit is the ability to retrieve each camera frame by its memory representation for passing to the tracker. *ARCore* seems to meet the requirements and should be tested.

7 Conclusion

During the course of this thesis, a tracker for piano keyboards which utilizes the keyboard's natural features could be developed as a self-sufficient library. This tracking approach enables new possibilities for AR piano teaching systems by lifting some of the current restrictions. It can cope with smaller fields of view of the camera and is believed to be more accurate than previous, purely marker-based tracking solutions. Additionally, the tracking is very fast, more than sufficient to take advantage of a smartphone's native screen refresh rate. This is especially important if an optical see-through HMD is used as it reduces the inevitable rendering delay of augmentations. For use within the keyboard tracking, a colored marker has been designed and a tailored marker detection algorithm has been designed and implemented. It can notably track slim markers, which other popular marker detection algorithms fail to detect. Moreover, it demonstrates how a marker can convey information about the scene without a tag through proper design and placement. Although black and white markers are much more widespread, it is shown that a colored marker detection can be equally efficient.

A prototype for an AR piano tutoring application, which integrates the keyboard tracker, was developed and successfully deployed on Android. For this task, communication between OpenCV and Unity was established under Windows and Android, which involved implementing a means of efficiently transferring non-basic data types from one system to the other. The prototype successfully demonstrates the functionality of the tracker as well as basic components of a piano tutoring system and thus serves well as the basis for a more complex application. The goal of deploying the prototype on a smartphone-powered HMD could not be achieved. Reasons for this lie in the absence of easily usable HMDs of this kind, the nontrivial system architecture of the prototype and a lack of time. At the time of writing, it was difficult to obtain a smartphone-powered OHMD with an openly available SDK. It was discovered that the technology is not yet as market-ready as previously assumed. Most HMDs which can be used or adapted for video see-through AR are designed for VR and do not feature a ready-made AR solution. However, it is expected that the prototype can be deployed on such a platform in the future without major issues. Upcoming devices will likely include improved build support and given some time, a solution for existing HMDs could be found or made. The AR piano tutoring prototype will continue to be worked on in the future with the goal of extending it into a more polished and consumer-friendly application.

List of Figures

List of Tables

Bibliography

[1] *A Summary of Augmented Reality and Virtual Reality Market Size Predictions*. https://medium.com/vr-first/a-summary-of-augmented-reality-and-virtual-reality-market-size-predictions-4b51ea5e2509. [accessed 12-05-2019].

[2] J. Schulkin and G. B. Raglan. "The evolution of music and human social capability". In: *Frontiers in neuroscience* 8 (Nov. 2014). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4166316/.

[3] A. Hille and J. Schupp. "How learning a musical instrument affects the development of skills". In: *Economics of Education Review* 44 (2015), pp. 56–82. ISSN: 0272-7757. DOI: https://doi.org/10.1016/j.econedurev.2014.10.007. URL: http://www.sciencedirect.com/science/article/pii/S0272775714000995.

[4] V. Martins, L. Gomes, and M. Guimaraes. *Challenges and Possibilities of Use of Augmented Reality in Education Case Study in Music Education*. June 2015. DOI: 10.1007/978-3-319-21413-9_16.

[5] M. Arcio Br, G. Wiggins, and H. Pain. *Computers in Music Education*. Nov. 2000.

[6] O. Cakmakci, F. Bérard, and J. Coutaz. *An Augmented Reality Based Learning Assistant for Electric Bass Guitar*. 2003.

[7] J. Chow, H. Feng, R. Amor, and B. Wünsche. "Music Education using Augmented Reality with a Head Mounted Display". In: *AUIC*. 2013.

[8] D. Hackl and C. Anthes. "HoloKeys - An Augmented Reality Application for Learning the Piano". In: *Forum Media Technology*. 2017.

[9] F. Huang, Y. Zhou, Y. Yu, Z. Wang, and S. Du. "Piano AR: A Markerless Augmented Reality Based Piano Teaching System". In: *2011 Third International Conference on Intelligent Human-Machine Systems and Cybernetics*. Vol. 2. 2011, pp. 47–52.

[10] R. Dannenberg, M. Sanchez, A. Joseph, P. Capell, R. Joseph, and R. Saul. "An Expert System for Teaching Piano to Novices". In: *International Computer Music Conference Proceedings* (Sept. 1990), pp. 20–23.

[11] S. Glickman, B. Lee, F. Y. Hsiao, and S. Das. "Music everywhere - augmented reality piano improvisation learning system". In: *NIME*. 2017.

[12] M. Weing, A. Röhlig, K. Rogers, J. Gugenheimer, F. Schaub, B. Könings, E. Rukzio, and M. Weber. *P.I.A.N.O.: Enhancing Instrument Learning via Interactive Projected Augmentation*. 2013.

[13] C. G. Healey and J. T. Enns. "Attention and Visual Memory in Visualization and Computer Graphics". In: *IEEE Transactions on Visualization and Computer Graphics* 18 (2012), pp. 1170–1188.

[14] M. Akbari and H. Cheng. *claVision: Visual Automatic Piano Music Transcription*. May 2015.

[15] J. G. Savard. *The size of the piano keyboard*. `http://www.quadibloc.com/other/cnv05.htm`. [accessed 28-1-2019]. 2011.

[16] P. Suteparuk. *Detection of Piano Keys Pressed in Video*.

[17] J. DeGol, T. Bretl, and D. Hoiem. "ChromaTag: A Colored Marker and Fast Detection Algorithm". In: *ICCV*. 2017.

[18] accessed 06-4-2019. URL: `http://josephdegol.com/pages/ChromaTag_ICCV17.html`.

[19] *Piano Keyboards*. `http://datagenetics.com/blog/may32016/index.html`. [accessed 30-04-2019].

[20] accessed 06-4-2019. URL: `https://commons.wikimedia.org/wiki/File:HSL_color_solid_cylinder.png`.

[21] accessed 06-4-2019. URL: `https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png`.

[22] D. J. Bora, A. K. Gupta, and F. A. Khan. "Comparing the Performance of L*A*B* and HSV Color Spaces with Respect to Color Image Segmentation". In: *CoRR* abs/1506.01472 (2015).

[23] accessed 28-1-2019. 2015. URL: `https://austingwalters.com/wp-content/uploads/2015/02/CIE_Lab.png`.

[24] S. Suzuki and K. Abe. "Topological structural analysis of digitized binary images by border following". In: *Computer Vision, Graphics, and Image Processing* 30 (1985), pp. 32–46.