FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Zikmund

A Calibrated Real-World Colour Picker for Augmented Reality Applications

Department of Software and Computer Science Education

Supervisor of the master thesis: doc. Alexander Wilkie, Dr.

Study programme: Computer Science (N1801)

Study branch: Software and Data Engineering (NISDI9M)

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date _____ signature of the author

Special thanks to doc. Alexander Wilkie, Dr. for supervising this thesis.

Název práce: Kalibrovaný výběr skutečných barev pro aplikace v rozšířené realitě

Autor: Martin Zikmund

Katedra: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: doc. Alexander Wilkie, Dr., Katedra softwaru a výuky informatiky

Abstrakt:

Tato diplomová práce popisuje proces tvorby mobilního nástroje pro rozšířenou realitu, který umožní designérům, architektům a výzkumníkům výběr přesné barevné informace z reálného prostředí. Aplikace také umožní nalézt nejbližší barevné vzorky v široké databázi barevných atlasů. Aby bylo možné určit, jak kamerový senzor mobilního zařízení vnímá barvu, je nutné před použitím aplikace provést kalibraci pomocí fyzického kalibračního terče. Na základě získané informace je možné odhadnout souřadnice vybrané barvy v jednom ze standardizovaných barevných prostorů. Hlavní výhodou výsledné aplikace je aspekt mobility. Namísto nutnosti přenášení jednoho či více barevných atlasů může uživatel provádět výběr barev jen s pomocí mobilního zařízení a kapesního kalibračního terče.

Klíčová slova: barvy, barevné atlasy, mobilní aplikace, rozšířená realita

Title: A Calibrated Real-World Colour Picker for Augmented Reality Applications

Author: Martin Zikmund

Department: Department of Software and Computer Science Education

Supervisor: doc. Alexander Wilkie, Dr., Department of Software and Computer Science Education

Abstract:

This thesis describes the process of creating an augmented reality mobile application which allows designers, architects and researchers to retrieve accurate colour information picked from the real-world environment. Specifically, the goal is not only to obtain colour coordinates for the area of interest but to find a set of matching colour samples in an extensive database of colour atlases provided with the application. To properly understand how the camera sensor perceives colour under current lighting conditions, it must be calibrated beforehand using a physical colour chart. Based on this calibration, we can estimate the picked colour coordinates in a standardised colour space. The mobility aspect is the main advantage of the resulting application. Instead of carrying multiple colour atlases, the user can estimate colour matching just using a mobile device and a portable colour chart.

Keywords: colours, colour atlases, mobile application, augmented reality

Contents

Introduction

The goal of this diploma thesis is to create a mobile application allowing designers, architects, researchers and other professionals working with colour to pick accurate colours from the real-world environment. It also enables the users to match the picked colour easily with colour samples from multiple publicly available colour atlases.

Many professionals use colour atlases on a day-to-day basis. Atlases are standardised and serve as a means of communicating and describing colour information during work engagements. There are various colour atlases available on the market. Each provides a unique set of colour samples, usually suited for a concrete scenario. As the atlases target a niche market of users, they are usually quite expensive and may also be hard to acquire. The diversity and cost are the main factors which cause a struggle in the occasion when designers need to collaborate. Without having direct access to the same colour atlas, it is almost impossible to communicate colour reliably without physically being present in the same location.

My bachelor thesis (1) aimed to streamline the process of finding similar colours among two different colour atlases. The output of the thesis is a mobile application available on all major platforms, which allows the user to enter a colour sample from a source atlas and quickly identify the nearest matches from other atlases, along with their relative difference based on their Delta E distance (2) using the CIEDE2000 (CIE Delta E 2000) standard which accounts for perceptual uniformity. The user can also choose from several different light sources, and the search then returns results accordingly to the one selected – as colour may be not perceived the same way when viewed under different lighting conditions.

Although this mobile application provides tremendous value on its own, there is yet another common scenario which it does not cover – finding the closest colour in an atlas based on a real-world colour sample.

Colour atlases, in general, are comprehensive, contain many colour samples, and are built to last. The consequence is that their size and weight make them less suitable for mobile usage. It leads to a challenge when the professional needs to work at a customer site and needs the atlas for reference – carrying it is not comfortable.

This master thesis aims to provide an efficient alternative. It is increasingly common to carry a mobile device – a phone or a tablet – around. Such a device is lightweight and serves many purposes, as opposed to a single-purpose colour atlas. Mobile devices are an ideal target for an application that solves the problem mentioned above.

We intend to create a mobile application, which allows the user to capture a photo in the real world and then tap on it to take a sampling of the colour at selected location. Based on this sample, the application will then search an integrated colour atlas database to find the closest colour samples in each of them. Combination of real-world colour sampling and colour matching across multiple colour atlases is unique a would provide a tremendous value its users. We have not discovered any analogous solution available on the market and our goal is to showcase a prototype of such a new type of application.

Sampling a colour from the output of a digital camera is a complicated task. It is not possible to utilise a processed photo, as that contains only limited colour information and is a result of multiple phases of filtering and processing, which is specific for each camera manufacturer and undocumented. Photo processing alters the colours significantly intending to the photo look "good" in terms of user perception, but not in terms of colour faithfulness, which is critical for our needs. Our application needs to use raw unprocessed camera output instead.

Each mobile device has a different camera sensor, and it is not possible to assume the unprocessed result produced by one camera will match another camera, even if it is the same device model from a different manufacturing batch. Before any operation, we need to calibrate using a set of reference colours. An appropriate device for this task is a pocket "passport" colour chart. The application primarily supports the X-Rite ColorChecker® Passport (3), although it is possible to implement support for additional reference colour charts in the future. Information gathered from the calibration photo of the reference chart is used to understand how the camera colour space behaves. This way, we can project the user-picked colour to a standardised colour space.

The output of this thesis is a mobile application for modern smartphones running the iOS operating system produced by Apple. The thesis text itself focuses on

the process and challenges of building such a mobile application. Note that while we use iOS and iPhone hardware for the application showcased in this thesis, it is possible to port the application to any other platform in a straight-forward manner. As our technique needs low-level access to the camera, which differs considerably in each platform, we chose to concentrate on one particular OS (operating system) and hardware target for our research. However, the core business logic of the application is composed of portable C# and C++ code, which can be aa other platforms, e.g. on Android without extensive effort and could even be integrated into another solution as a set of self-standing utility libraries.

In the first chapter, we introduce theoretical background including colour theory, colour spaces, atlases and ColorChecker® Passport. We cover the technology behind digital cameras and their output formats. Finally, we mention the specifics of mobile application development in general.

The second part of the thesis focuses on our proposed solution. After describing the process as a whole, we delve into specifics of each of the steps. Firstly we demonstrate how we gather the raw camera information from the sensor. Next, we describe the algorithm to identify a ColorChecker® passport device within a photo. Included is a description of all approaches we attempted and reasons that led to the final design. The localised colour chart samples are used for calibration. Afterwards, we introduce colour picker and describe how we can use the results of calibration and apply them to the user-picked colour sample to acquire a reasonable estimate of the actual colour in a standardised colour space. Finally, we use the retrieved colour coordinates to search for the nearest matches in each of the colour atlases included with the application.

The third chapter will present the results – our Colour Picker mobile application with its user interface, and several examples of the colour picking user experience. We note the known issues and include a list of further improvements which could make the product even more useful.

1. Background

This chapter introduces topics and concepts which are fundamental to this thesis, including colour theory, digital cameras and mobile application development.

Some parts of the sections in this chapter were included in my bachelor thesis (1) but have significant relevance for this text as well. Those parts are either cited precisely as in the previous work or edited and expanded as necessary.

1.1. Colour theory

Colour theory is a theoretical and practical field which guides working with colours. The key components are colour perception and colour mixing, although it encompasses a broader range of topics. Earliest mentions of the theory of colour date back to the Greek philosopher Aristotle around 350 BCE. (4)

1.1.1. Colours

"Colour derives from the spectrum of light (distribution of light power versus wavelength) interacting in the eye with the spectral sensitivities of the light receptors." (5) The cone cells in the retina encode light into signals, which transmit to the brain and perception of colour is there created by interpreting the signals. Figure 1 shows the structure of a cone cell.

The way the human brain interprets colour is the foundation for colour measurement, and the representation of colour as a triplet of red, green, and blue values. (6)

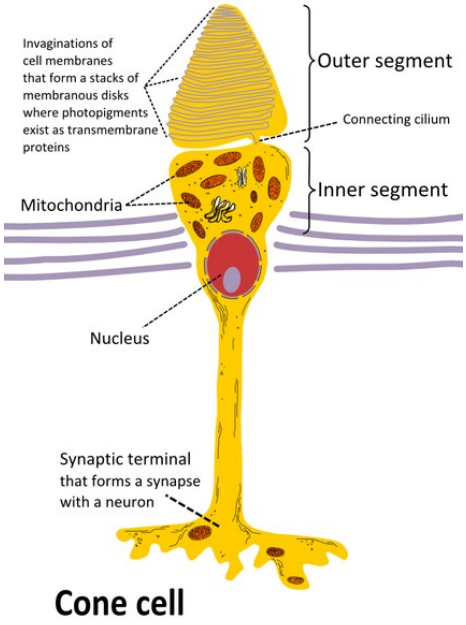

Figure 1: Cone cell structure

Source: (7)

We can describe light as a function of power versus wavelength. We call such function *a spectral distribution* function or *spectrum*. Coloured light contains wavelengths in the range of 370–730 nanometers. (6)

1.1.2. Colour vision

Colour vision is the ability of an organism or machine to distinguish objects based on the wavelengths (or frequencies) of the light they reflect, emit, or transmit. The brain responds to the stimuli produced when incoming light reacts with the cone cells in the eye.

Three types of cones respond to different wavelengths of light – short (blue), medium (green) and long (red). Brain reduces the information in the spectral distribution to three values, one for each type of cone and the mix of these values then represents the resulting colour. Figure 2 shows the sensitivity of each cone type to a given wavelength. The human eye is most sensitive to “green” colour wavelengths around 555 nanometers as it simultaneously stimulates both medium and long cones. (8)

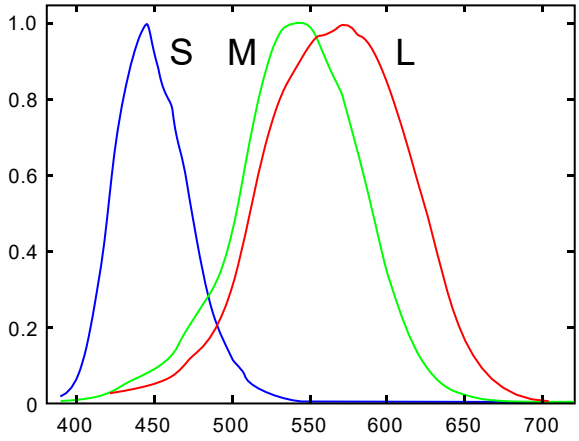

Figure 2: Cone cell sensitivity to wavelengths (9)

1.1.3. Colour mixing

Based on the way the human brain represents colour naturally follows the field of colour mixing. There are two models of colour mixing which are opposite to each other.

Additive colour mixing follows the behaviour of the human eye concerning light. When we combine light of different colours, these colours mix additively. Absence of light produces a black colour. We recognise three base colour channels – red, green and blue. These channels can be combined in varying intensities to form other colours. Reproduction of digital colours in displays of computer monitors and mobile devices uses additive colour mixing as these displays emit light. (10)

Subtractive mixing of colour corresponds to rendering colours on physical substrates and work with reflected light. The base colours, in this case, are cyan, magenta and yellow. “When cyan, magenta, and yellow pigments are laid upon a white, reflective substrate, each completely absorbs – or subtracts – its opposing counterpart from the white light.” (10) Printers include a dedicated black cartridge to produce the colour of black more faithfully.

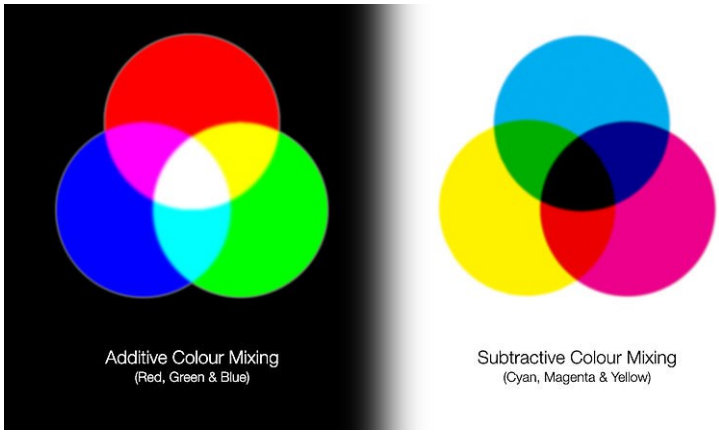

Figure 3: Additive and subtractive colour mixing (11)

1.1.4. Digital colour

As we mentioned in 1.1.3, to represent colour digitally, we need to encode colour into additive channels of red, green and blue. The smallest area of illumination on digital displays is a pixel. We can describe the colour of each pixel by three numbers corresponding to the three base colour channels. The properties of the digital display itself limit the accuracy of colour it is capable of reproducing.

To achieve higher colour faithfulness of digital colour produced by device displays, professionals utilise devices called colourimeters. Such an instrument is designed to "determine or specify colours, as by comparison with spectroscopic or visual standards." (12) Figure 4 shows a colourimeter in the process of calibrating a laptop screen.

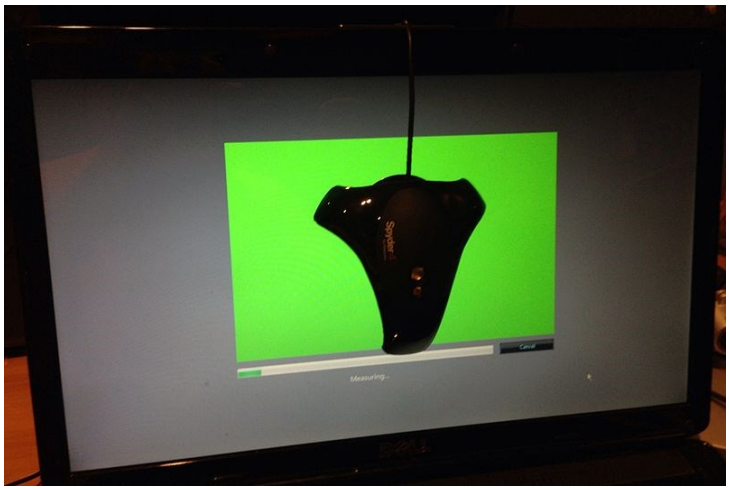

Figure 4: Colour calibrating of a laptop screen using a colourimeter (13)

1.2. Colour spaces

"Colour space, also known as the colour model (or colour system), is an abstract mathematical model which simply describes the range of colours as tuples of numbers, typically as 3 or 4 values or colour components (e.g. RGB)." (14) Each colour we can represent in such colour space is essentially a set of coordinates within such a three- or four-dimensional structure. Colour spaces can be used to describe the colour faithfulness capabilities of a specific digital display.

1.2.1. XYZ colour space

The International Commission on Illumination (CIE from French *Commission Internationale de l'éclairage*) created the XYZ colour space (also referred to as CIE XYZ) in 1931.

CIE XYZ bases its colour representation on the properties of the human eye. The CIE XYZ tristimulus values (X, Y and Z) approximate the behaviour of short, medium and long wavelength cones.

XYZ colour space can represent colours which are impossible to reproduce in the real world and is device independent.

1.2.2. L*a*b* colour space

In 1976 CIE defined a new colour space – L*a*b* (also known as CIE LAB). This colour space became "the reference colour model used by the papermaking and graphic arts industries" (15). It describes colours in terms of three axes – L*, a*, and b*. (16)

The lightness axis (L*) varies in values from 0 for black to 100 for white. The colour axes a* and b* each cover an opposing range of colour hues. The a* axis represents green in negative values, and red in positive values and b* represents blue in negative and yellow in positive. (16) There are no maximum or minimum values for a* and b*, although they are usually ranging from -128 to +127.

Similarly to XYZ, L*a*b* colour space is capable of representing all perceivable colours and is device independent. It is advantageous to use this colour space to measure the perceived colour distance using the Delta E standard (described in more detail in 1.2.3) as the distance between colours in CIE LAB very closely correspond to the colour distance as perceived by the human eye.

1.2.3. sRGB colour space

sRGB is a standard RGB colour space proposed in 1996 by Hewlett-Packard and Microsoft. It was accepted by the World Wide Web Consortium (W3C) as a standard default colour space for the Internet. Besides, most consumer applications, devices, and printers support and default to sRGB. (17)

"sRGB utilises a robust and straightforward device independent colour definition" (18). The value of each colour channel (R for red, G for green and B for blue colours) is a real number in the range from 0 to 1.

It is important to note that sRGB is not an absolute colour space – it cannot express the full gamut of human colour perception as opposed to the theoretical models of CIE XYZ and CIE LAB.

RGB24 is a device-dependent 24-bit digital colour format based on sRGB which represents the colour of each pixel by 3 bytes – each channel has a value ranging from 0 to 255. Because this colour space is minimal and non-linear, we can only use it to approximate a colour, not to represent it faithfully. Even then, thanks to its properties, it is the most widely used colour format for compressed and processed computer graphic formats and in web design.

1.3. Colour distance

The perceived colour difference is an essential topic for design and production. CIE defined Delta E as a measure of colour difference in 1976.

"Delta E is a metric for understanding how the human eye perceives the colour difference." (19) Its values are non-negative numbers where lower value means less perceptible difference.

For Delta E values below 1.0, the colour difference is undetectable by the human eye. Around 10.0, our eyes can recognise the difference quickly. For values above 50.0, the colours differ significantly.

CIE proposed the first Delta E 76 standard as simple Euclidean distance between the two colours in CIE LAB colour space. This standard had some shortcomings (does not correctly address perceptual non-uniformities) and got two revisions – first with Delta E 94 in 1994 and later with Delta E 2000 standard, which is used today as the most accurate colour difference formula.

1.4. Colour atlases

Colour atlases are physical representations of predefined (and standardised) sets of colour samples. These samples offer an intuitive and tangible representation of a given colour. Atlases usually take the form of a box or a book.

The main objective of colour atlases is to enable designers to work with a standardised set of colours in a hands-on fashion.

The most commonly used atlases are Munsell Book of Colours, Natural Colour System, RAL Classic/Design and British Standard Colours. Moreover, there are specialised colour atlases like OBI and Hornbach.

1.4.1. Usage

To make sure that the atlas can be easily navigated and referred to, each colour atlas has a coordinate system. In general, this system partitions the colour space into subsets based on a documented logic. Each colour patch has a unique set of coordinates and a unique name. This way, the colour can be precisely identified and communicated among owners of the same colour atlas.

The producers of colour atlases ensure the colour patches are very accurate and consistent, so professionals can use colour atlases to choose colours and share them with supply chains, clients or colleagues.

1.5. ColorChecker®

ColorChecker Colour Rendition Chart (originally Macbeth ColorChecker) is a physical device used by photographers for colour calibration. “It is designed to deliver true-to-life colour reproduction so photographers and filmmakers can predict and control how colour will look under any illumination.” (20) It was first introduced in 1976 in a paper in the Journal of Applied Photographic Engineering by C. S. McCamy, H. Marcus and J. G. Davidson (3). Originally it was produced by Macbeth but is currently owned and developed by X-Rite, Inc.

X-Rite currently (as of June 2019) produces six different forms of the ColorChecker®.

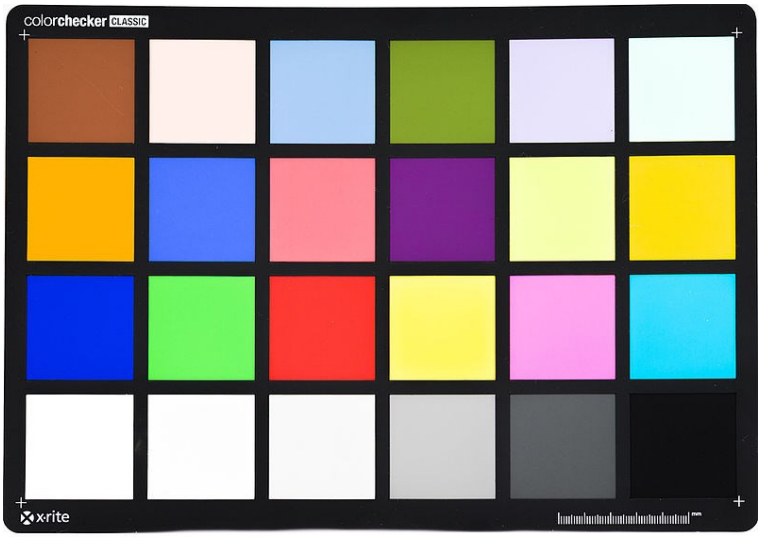

Figure 5: ColorChecker® Classic (21)

The original ColorChecker® Classic (Figure 5) contains an array of 24 colour patches. "Each of the 24 colours found in ColorChecker® Classic represents the actual colour of natural objects, such as human skin, foliage and blue sky, and reflects light just like its real-world counterpart. Since they exemplify the colour of their counterparts and reflect light the same way in all parts of the visible spectrum, the squares will match the colours of representative samples of natural objects under any illumination, and with any colour reproduction process." (22)

Figure 6: ColorChecker® Passport (23)

X-Rite introduced ColorChecker® Passport (Figure 6), and ColorChecker® Passport 2. These passport devices superseded the ColorChecker® Classic by bringing more features in a more portable size. The Passport has multiple “pages”. In addition to the same industry standard 24-patch colour reference target from ColorChecker Classic, it also features a “Creative Enhancement Target”, best suited for shadow details evaluation and highlight clipping. The passports also feature large balance targets for white and grey colours. We have used the ColorChecker® Passport for the development of the Colour Picker mobile application.

We also utilised yet another product from the ColorChecker® range – ColorChecker® Digital SG (Figure 7). This variant is a substantially larger device which includes not only the 24 original colour patches but also 116 additional ones to provide an even fuller colour gamut. (24)

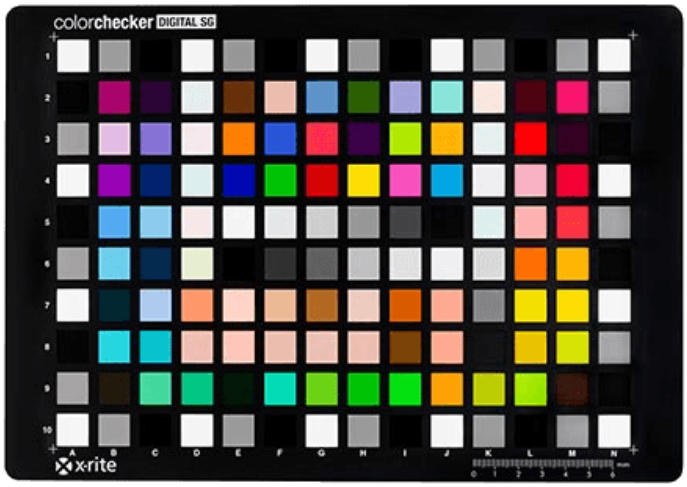

Figure 7: ColorChecker® Digital SG (25)

1.6. Advanced Rendering Toolkit

Advanced Rendering Toolkit (ART) is a Predictive Rendering research system under development at the graphics group of Charles University in Prague. ART is open-source and available to the public.

ART includes many colour- and graphics-related features. It can also serve as a source of spectral measurement data. The data is one of the many resources ART contains, and the toolkit itself does not have any other direct points of contact with the aim of this thesis.

The underlying data provided by the Advanced Rendering Toolkit (ART) was manually measured using a handheld spectro-radiometer of type Spectrolino (Figure 8). The reflectance data has a sample spacing of 10nm.

ART can export the colour atlas data to the JSON format, which is then easy to parse in any programming language.

Figure 8: Spectrolino (26)

1.7. Digital camera

A digital camera is a device which produces photographs (and videos) in digital format. Instead of a physical film, digital cameras store images in digital memory storage. Users can reuse the storage multiple times, and back up their photos on another digital media when necessary. Digital cameras are now widespread, and while there are many dedicated devices available on the market, most digital cameras are nowadays incorporated into multi-functional devices like smartphones, tablets or computers.

A device called image sensor captures light received by the camera lens and converts it to the digital signal. The sensor measures the colour and brightness for each pixel and turns it into a set of numeric values. There are two common types of image sensors.

Semiconductor charge coupled device (CCD) is an image sensor type with high-fidelity and light sensitivity, which can transport the charge created by light across its chip without distortion. Manufacturing this type of sensor is quite expensive, and we can usually find it in professional cameras. CCD can produce high-quality images without noise.

Figure 9: CCD image sensor (27)

Complementary metal-oxide-semiconductor image sensors (CMOS) uses an array of transistors that amplify and move charge via wires. Each pixel can be read separately but has lower light sensitivity (photons hit the transistors instead of the photodiode) and tends to produce noisier images. On the other hand, CMOS sensors are cheaper to manufacture and consume significantly less power. These trade-offs are well suited for smartphone and tablet cameras.

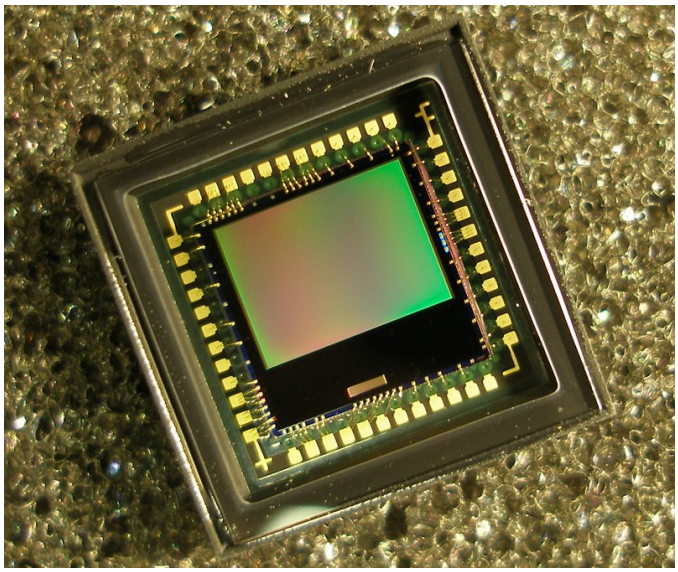

Figure 10: CMOS image sensor (28)

1.7.1. Exposure

In digital cameras, exposure determines the amount of light which reaches the sensor and in turn the overall brightness of the resulting photo. It is a value consisting of three components – shutter speed, ISO and aperture.

Aperture represents the area over which light can enter the camera. In most mobile phones, this setting is not adjustable.

Shutter speed is the duration the sensor is exposed to light while capturing an image. Most commonly we represent this value as a fraction of a second – for example $^{1}/_{100}$ equals 0.01 seconds.

ISO value controls the sensitivity of the camera sensor to a given amount of light. Higher ISO values allow capturing photos in darker conditions without flash but increase the noisiness of the resulting picture.

By combining the three factors, we can optimise the appearance of the captured image. We refer to a photo as *underexposed* when the exposure is too low, and the image is dark. Conversely, we use the term *overexposed* when we talk about a picture which was taken with too high exposure and is therefore too bright.

1.7.2. Raw image format

Camera raw image format is an "uncompressed and unprocessed snapshot of all the detail available to the camera sensor" (29).

The image processing algorithms included in most digital cameras and mobile devices produce a "cleaned up" version of the captured photo – the raw data are processed through several layers of operations before the final processed picture is constructed. These include demosaicing (described in 1.7.3), defective pixel removal, white balancing and noise reduction among others. Especially in case of colours, we cannot depend on the accuracy of the processed photo – the result is optimised to be viewed by the user, not for precise colour reproduction.

Because raw data contain more information than a processed photo, they usually require higher storage size than the processed counterparts. The data is essentially a stream of bytes in a format the camera supports. Specialised tools and

software are needed to display such a raw image. On the other hand, the additional data give an experienced user a high amount of control over the resulting image.

Our application benefits from having access to as high colour accuracy as possible within the limits of the camera sensor a mobile device has. Hence, we need to be able to access the raw image data of the camera sensor programmatically using the operating system's APIs (application programming interfaces).

1.7.3. Demosaicing and the Bayer pattern

Digital camera sensors detect the brightness of light coming into the device. Per-pixel colour filters in front of the sensors are used to capture the red, green, or blue light per pixel and get a coloured image. These filters form an array which is called a colour filter array (CFA), as shown in Figure 11. The arrangement of the filters uses the Bayer pattern invented by Bryce Bayer in 1976. (30)

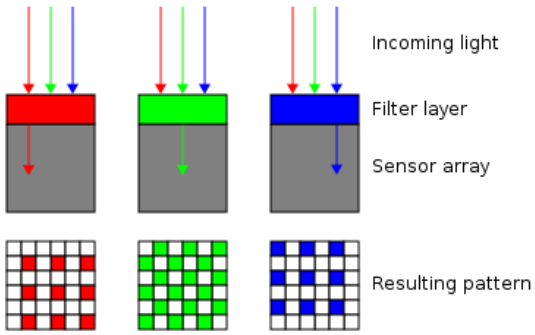

Figure 11: Colour Filter Array (31)

The pattern consists of recurring 2x2 layout of the red, green and blue filter. Because there are four spots in the pattern, it contains the green colour filter twice as the human eye is most sensitive to green. Depending on the actual order of the filters, there are multiple different forms of the Bayer pattern. RGGB depicted in Figure 12 is the most common, but some camera models may output BGGR (as seen in Figure 11) or RGBG.

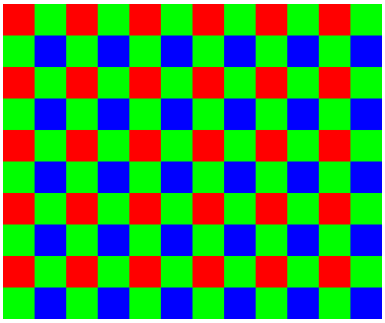

Figure 12: RGGB Bayer pattern (30)

In the process of reconstructing the full-colour image, it is vital to transform the brightness values produced by the colour filters into actual pixels of colour with three colour channels. This process is called *demosaicing* or *debayering*.

There are many different algorithms for this process. The simplest algorithms provide acceptable results but may cause unwanted artefacts and noise.

1.8. kd-Trees

kd-Trees were first invented in 1970 by Jon Bentley. This abstract space partitioning data structure allows efficient searching for closest neighbours in a set of k-dimensional data (32). Each kd-Tree works with a given number of dimensions – for example, for three-dimensional problems we use a "kd-Tree of dimension 3". kd-Trees are a case of binary space partitioning trees.

Data in the tree is stored in the leaves, whereas each non-leaf node is defined by a plane through a dimension that partitions the set points into two halves (33).

kd-Trees offer a fast way to search for the nearest neighbour among the stored nodes. This search has an average $O(\log n)$ time complexity where n is the size of the tree.

1.9. OpenCV

"OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products." (34)

OpenCV was initially developed by Intel Corporation in 2000 and is used extensively by many companies and research groups. The library is licensed under the open-source BSD license and is managed by several community members.

The library is written in C++ to achieve the highest performance. OpenCV provides bindings for other languages including Python, Java and MATLAB. There is also limited support for JavaScript, and there are multiple unofficial bindings for other languages as well.

OpenCV supports many platforms including Microsoft Windows, Linux, macOS, Android and iOS.

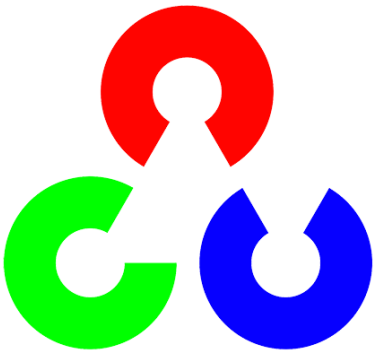

Figure 13: OpenCV logo (34)

1.10. Mobile application development

The field of mobile application development encompasses the process of producing and delivering software for a wide range of portable devices that users carry with them in their daily life. As opposed to traditional desktop applications, these applications often take advantage of special capabilities of the mobile device hardware like a digital camera and a range of sensors.

1.10.1. Mobile device

We define mobile devices as computing devices which can be easily transported and carried by a human. This definition spans a wide range of different forms, including smartphones, tablets, notebooks and wearables.

We can further categorise these devices by their screen size (from small to large, many different resolutions), supported input methods (touch, pen, touchpad/keyboard and mouse, voice, or any combination of these), use cases (home,

business, industrial, all-day usage) and hardware features (digital camera, sensors). The capabilities of the device are also influenced by the operating system it runs on (most commonly Google Android, Apple iOS or Microsoft Windows) and the set of mobile applications the user has downloaded and installed.

1.10.2. Architectural patterns

Software architectural patterns (also design patterns) provide a set of principles on how to build a maintainable and extensible software solution. Each such pattern offers a general solution to a common problem and has an assigned name which makes communicating an architectural decision easier among developers in a team. It is important to note that design patterns offer only guidance and are not imperative – a developer can approach the same problem in many different ways. It is the responsibility of the developer or team to make an educated decision on which pattern (if any) fits the given problem best.

Developers usually base the presentation layer of mobile applications on the model-view-presenter (MVP), model-view-controller (MVC) or model-view-view model (MVVM) architectural patterns. By adequately dividing the presentation layer into its components, it becomes easier to change the UI layer or presentation logic without influencing the remaining parts.

2. Proposed method

2.1. Application workflow

The application functionality, in general consists of three key steps – calibration, colour picking, and finding the best matches.

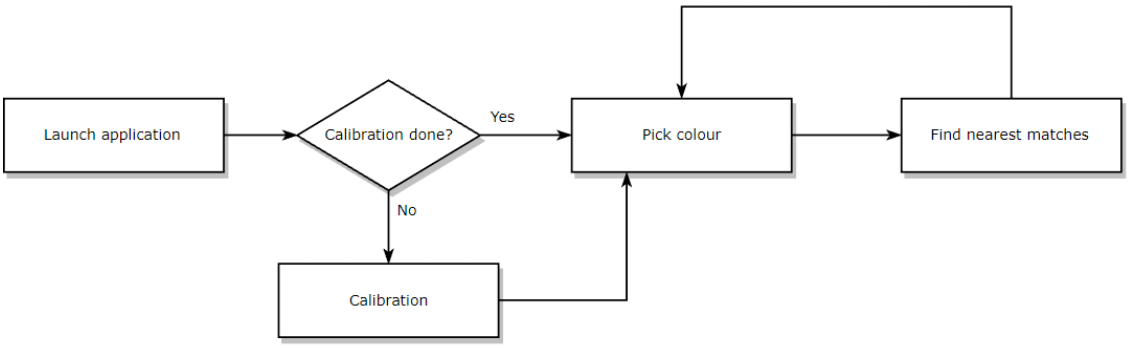

Figure 14: Application workflow

Both calibration and colour picking phases require the ability to capture and process raw image data from the camera sensor, which is covered in Section 2.3.

2.1.1. Calibration

The users must calibrate for the specific camera sensor using the ColorChecker® Passport device prior to using the application for the first time. The calibration allows us to identify the colour space of the camera and understand how it behaves in different areas of the colour spectrum. We calibrate using the 24 standard colour patches on the ColorChecker®.

The calibration needs to be done under daylight as that is the illuminant ColorChecker® is designed for, and other lighting conditions would yield different results.

A vital part of the calibration process is locating the ColorChecker® patches in the captured photo. This part of the problem is discussed separately in Section 2.4. Details about the calibration workflow itself are available in Section 2.5.

2.1.2. Colour picking

With completed calibration, users of the application can capture a photo of the object from which they want to pick a colour by tapping a location on the photo.

To accommodate for noisiness and artefacts in the picture, we pick the colour from an area with a configurable radius. Besides, the users are given visual feedback to see the exact area used and can correct their selection if necessary.

Once we know the red, green and blue camera space values for the picked area, we can use the calibration information to find corresponding coordinates in sRGB colour space.

Section 2.6 contains full details about this phase of the application workflow.

2.1.3. Finding the best matches

Using the pre-calculated sRGB and CIE LAB coordinates of the picked colour, we can search the provided colour atlases for the closest matches and display the results to the user. We describe this in Section 2.7.

2.2. Mobile application development

To design a mobile application, we need to make several vital up-front decisions about the tools and architecture we will use. The choice of a programming language and platform will, in turn, influence the set of devices we can support.

Even if we only want to support a single operating system, we can design the application with cross-platform support in mind. Such mindset promotes a modular architecture and better abstraction of the solution which can hide platform-specific features from the core business logic component of the application and will minimise the amount of code necessary to potentially provide support for a new platform in the future.

2.2.1. Apple iOS

Mobile devices produced by Apple Corporation are running the iOS operating system (with a variant called iPadOS targeting tablet devices).

The primary form factors of these devices are a smartphone and a tablet device. Since the release of iOS 10, devices newer than iPhone 6s support shooting RAW photos with additional APIs (application programming interfaces) included in iOS 11. These functionalities are crucial for the Colour Picker application to function correctly.

iOS is the second most widely used mobile device operating system behind Google Android. We have chosen iOS over Android as it provides a practical API for

working with RAW photos, which can be utilised on all iOS devices, whereas Android APIs are sometimes manufacturer-dependent.

2.2.2. Choosing the programming language and tools

The first-party programming language for iOS application development is called Swift. It superseded the Objective-C language which is however still utilised in some use cases. The integrated development environment (IDE) for Swift and Objective-C development is the Apple Xcode, which is available on Apple macOS. The IDE provides many features including a designer for iOS .xib files (a file format used to declare mobile application user interface) and iOS device deployment tools.

There are two significant disadvantages of choosing Swift or Objective-C for application development. Firstly, both these languages are specifically designed for iOS and are therefore not portable to other operating systems. This limitation negates the potential for porting the application to other platforms in the future. Secondly, the Xcode IDE is available only on macOS and does not include extensibility support, which can be limiting from the developer usability standpoint.

It would also be possible to develop the application using the C++ programming language. Even though this is a very flexible programming language, its level of abstraction is at a lower level than the other available options, which can prove ineffective for higher-level programming tasks like UI interaction. All the fidelity that C++ offers can become unwarranted in light of building a task-driven mobile application without complex graphical user interface and 3D graphics. However, C++ is still required to produce the OpenCV component for ColorChecker photo recognition.

Our choice for the colour matching tool development will be the C# programming language, which is strongly-typed, portable and has many features that can be advantageous for us. The .NET ecosystem on which C# runs has a vibrant marketplace of third-party libraries and there is a large number of community-based open-source projects written in this programming language.

The primary development IDE (integrated development environment) for C# is the Microsoft Visual Studio IDE. This environment offers extensive capabilities for each phase of the development process.

2.2.3. Xamarin.iOS

C# programming language cannot run natively on iOS as it requires .NET runtime support. However, Xamarin.iOS framework (35) includes a port of the .NET runtime for iOS using Mono.

"On iOS, Xamarin uses Mono, fully functional implementation of the .NET runtime, to fully compile your app into a native ARM executable ahead of time (AOT)." (35)

The framework itself exposes C# bindings of the complete iOS SDK (Software Development Kit) for .NET developers - including advanced features like ARKit and CoreML. On top of the native APIs it provides unique language features including asynchronous programming with the async/await keywords and Language Integrated Query (LINQ) expressions. Xamarin updates the API surface with each new release of iOS SDK.

Developers can create Xamarin applications for free as part of the Visual Studio Community offering. A macOS device is required to build, deploy and debug the app on iOS emulator or device.

2.2.4. Xamarin.iOS Binding Libraries

Xamarin.iOS provides full flexibility in terms of code reuse. Existing native libraries built in Objective-C can be integrated with Xamarin environment through the Binding Libraries functionality.

A Xamarin.iOS Binding Library is a particular type of library which contains one or more native iOS libraries with the `.a` extension and a C# interface for their APIs. The developer can reference the resulting library like any other project from the iOS project head. The native library deploys with the application and Xamarin uses the C# interface to communicate with it. Xamarin team uses the same principle is used to implement the entire iOS API surface.

We needed to use Xamarin.iOS Binding Library to integrate OpenCV library into our application, as there is no official build of OpenCV with Xamarin.iOS support yet. Although there is a cross-platform .NET wrapper of OpenCV called EmguCV (36), its stability didn't meet our requirements at the time of development.

2.2.5. .NET Standard

The .NET Framework had initially been available only on Microsoft Windows operating systems. A community-driven effort later brought .NET to Linux in the form of the Mono framework. Nowadays Mono framework runs on Windows, Linux, macOS, iOS and Android. In 2016, Microsoft released a first-party open-source implementation of .NET called .NET Core, which is a portable and modern rewrite of .NET Framework which runs on Microsoft Windows, Apple macOS and Linux.

Each of the listed variants of .NET includes a separate implementation of the base class libraries and targets a different set of platforms and devices. Furthermore, each specific application platform on which .NET runs can declare the available API surface which it supports. The inevitable consequence of this is fragmentation. For example, if a developer has written a library for .NET Framework, it cannot directly compile or run under Mono and vice versa.

The first solution to this problem were Portable Class Libraries (PCLs). PCLs have a predefined set of profiles which are denoted by a number. Each profile allows the library to target a set of platforms. The available API surface is a subset of the intersection of APIs available for all platforms the user selected. The choice is limiting as including a platform with incomplete API surface can severely restrict the capabilities developer can utilise. Besides, the library can reference only libraries targeting the same profile.

.NET Standard was introduced to overcome this problem. It is a specification for implementing the base class libraries on a .NET supported platform. This standard is versioned so a platform can declare compatibility up to .NET Standard of a specific version, meaning the developers can use all APIs introduced up to that .NET Standard release. A newer version provides access to a broader set of APIs but requires targeting newer versions of the target platforms, whereas lower versions of .NET Standard allow targeting more comprehensive set of devices as they have lower requirements.

As of June 2019, the most commonly used version of .NET Standard is 2.0. Usually, the .NET Standard library contains the core business logic of the application which allows for easy portability. Only the user interface and device and platform-specific APIs need to be implemented for each platform separately, and the .NET Standard code is fully reusable across multiple targets. We can use platform-specific

capabilities in the shared code by programming against interfaces and introducing Inversion of Control (IoC). This concept is described in detail in Section 2.2.9.

2.2.6. Overall project structure

Colour Picker project consists of two Visual Studio solutions.

The implementation of ColorChecker® location algorithm is written in C++ and is included in its own separate solution `ColourCheckerFinder.sln` in the `Code/ColourCheckerFinder/ColourCheckerFinder` subfolder. This solution consists of a single Visual Studio 2019 C++ project which references the OpenCV framework. The solution is designed to be runnable as a console application on Windows, which allows for shortened development loop and direct debugging of the checker recognition algorithm.

For deployment into the mobile application, it compiles as an Objective-C Cocoa Library. The project file `ColourCheckerFinder.xcodeproj` can be opened and used in the Xcode IDE on macOS. The iOS version of OpenCV must be provided as a dependency to build this project. Successful compilation of this project produces a native Objective-C library with the `.a` extension. We copy this library manually into the `Code\ColourCheckerFinder.Binding` folder to make it part of the Xamarin.iOS project.

The core solution of the application is `ColourPicker.sln` in the `Code` folder. The solution consists of a set of C# projects. Follows a summary of each:

- `ColourPicker.Core` – .NET Standard library containing business logic.
- `ColourCheckerFinder.Binding` – Xamarin.iOS Binding library project, provides ColorChecker® C# bindings for the native Objective-C library.
- `ColourPicker.iOS` – Xamarin.iOS application project head, references `ColourPicker.Core` and `ColourCheckerFinder.Binding` projects.
- `Tools/ColourPicker.CalibrationToPlyConverter` – console tool that converts an existing colour checker calibration JSON file to a PLY visualisation for use in the MeshLab application (see Section 2.5.5).

- `Tools/ColourPicker.CheckerConfigurationDisplay` – Windows Presentation (WPF) tool, displays the approximate sRGB and RGB24 visualisation of a checker configuration JSON file.
- `Tools/ColourPicker.CheckerDefinitionGenerator` – console which transforms a provided ART-JSON file into application compatible ColorChecker® definition JSON file.
- `Tools/ColourPicker.CppCheckerDefinitionGenerator` – console application, generates a C++ LAB colour array for use in the `ColourCheckerFinder` project.
- `Tools/ColourPicker.RawTester` – WPF application, previews the results of the demosaicing algorithm
- `Tests/ColourPicker.Core.Tests` – a unit test project for parts of the Core project functionality

2.2.7. Model–view–view model pattern

We want to build the application to be easily extensible and maintainable. To achieve this goal, we must choose a suitable presentation layer architectural pattern.

In 2005, Microsoft employees Ken Coopers and Ted Peters designed the model–view–view model (MVVM) presentation layer design pattern to simplify writing event-driven user interfaces. (37) This pattern has three main components – model, view, and view model. Figure 15 shows a diagram of the MVVM pattern.

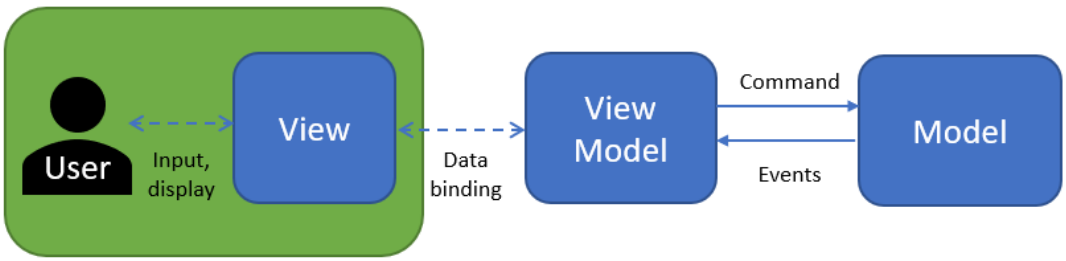

Figure 15: MVVM architecture

The model represents the data our application uses. The objects int his layer are an abstraction of the business logic and data access of the application. It provides

services which can be utilised by the view model layer and declares events when the data changes.

The view component embodies the user interface (UI), which is displayed to the user. The user can communicate with the UI using the device's supported input methods. The View is tailored to optimise the user experience (UX) and format data in a way familiar to the user via internationalisation and accessibility features.

The View Model layer is the middleware layer between View and Model. It should ideally be a thin layer which communicates with the data from the model layer, orchestrates and transforms it into a format which is suitable for display.

Neither the view nor the view model requires direct knowledge of one another, which means both components can be developed independently. The communication between the two layers happens via bi-directional data binding. Data binding is a layer which observes data changes in both UI and View Model and makes sure they are synchronised on both sides. In C#, the data binding facility subscribes to the `PropertyChanged` event in the `INotifyPropertyChanged` interface. Whenever data in View or View Model changes, it must raise the `PropertyChanged` event with the property name or member name, which is relevant for that change.

Besides, user actions can be defined in the view model in terms of Commands, which are object-based representations of actions. A command is a class which implements the `ICommand` interface in C#. The instance is then data-bound in the View to user controls like a button. Commands can even define when the user is allowed to perform a particular action.

Value converters allow the View to implement a further transformation of the View Model properties in case the data types used do not match the format the View expects. The converters are also bi-directional so that the developer can provide a reverse transformation from the View back to View Model.

2.2.8. MvvmCross

The first platform which implemented the MVVM pattern was Microsoft Windows Presentation Foundation (WPF). Later, Silverlight, Universal Windows Platform (UWP) and Xamarin.Forms (XF) platforms implemented it as well. In all

these cases, MVVM depends on a declarative user interface language based on XML called XAML. This markup language has built-in support for data-binding.

On the other hand, Apple iOS does not have built-in support for MVVM pattern. Mobile applications on this platform often use a form of the model–view–controller (MVC) pattern instead. While this pattern is still useful for the structuring of the presentation layer, MVVM allows for more natural decoupling of the views, which is beneficial for quick design iteration.

Figure 16: MvvmCross library logo

The MvvmCross library (38) is an open-source project available on GitHub and maintained by a community of developers, which provides a framework for building MVVM-based Xamarin.iOS and Xamarin.Android applications. MvvmCross provides a full-fledged cross-platform implementation of the pattern, including all its major components. The library also provides a powerful abstraction of the application lifecycle across platforms to simplify integrating all business logic into a shared .NET Standard library instead of writing platform-specific code.

While the framework lacks a declarative user interface language similar to XAML, MvvmCross provides a way to describe data-binding to various built-in user interface components as well as a means to provide custom implementations. The data-binding feature is on-par with XAML data-binding and includes full support for value converters, commands and more. Code sample 1 shows an example of how MvvmCross user can set up data-binding in Xamarin.iOS using a fluent API. It shows both a simple property binding and binding a command to respond to a tap event of a button.

```
var set = this.CreateBindingSet<CalibrationView, CalibrationViewModel>();
set.Bind(this)
    .For(v => v.Title)
    .To(vm => vm.Title);
set.Bind(CalibrateImageButton.Tap())
    .For(t => t.Command)
    .To(vm => vm.CalibrateCommand);
```

Code sample 1: iOS data-binding

Finally, as the project has an active community of developers and projects using it, there are many useful plugins available, which further simplify access to platform-specific functionality from the shared core project.

2.2.9. Inversion of Control

The concept of Inversion of Control (IoC) is a programming principle, which intends to remove hard dependencies within a codebase to make it more maintainable, extensible and testable. It encourages designing code against interfaces instead of specific implementations.

Instead of writing a sequential code where each class is responsible for instantiating the services it depends on, these services are provided to it by the caller (application, or framework). This way, a component may consume the products of other services without having to be aware of their specific implementation and their transitive dependencies.

At the core of IoC lies the concept of Dependency Injection (DI). DI ensures the dependencies are automatically injected into the consuming class upon instantiation. While there are many different DI frameworks available, in C#, there are three most common variants of dependency injection:

- Constructor injection – dependencies are injected as arguments into the constructor during instantiation.
- Property injection – dependencies are injected as properties of the class during instantiation.
- Method injection – the caller injects the required dependency as an argument of a method.

In our application, we are using constructor injection extensively. Each service implements an appropriate interface. The actual classes which implement those interfaces are then set up in the IoC container during application initialisation in the `ColourPicker.Core.App` class for platform-agnostic services and in `ColourPicker.iOS.Setup` class for platform-specific services. MvvmCross has a built-in IoC container which we are using and which takes care of constructor injection during view model and service initialisation.

```
Mvx.IoCProvider.ConstructAndRegisterSingleton<ICheckerFinder, CheckerFinder>();
Mvx.IoCProvider.ConstructAndRegisterSingleton<IImageDrawing, ImageDrawing>();
Mvx.IoCProvider.ConstructAndRegisterSingleton<IToastService, ToastService>();
Mvx.IoCProvider.ConstructAndRegisterSingleton<IImageLoader, ImageLoader>();
Mvx.IoCProvider.ConstructAndRegisterSingleton<IPopupService, PopupService>();
```

Code sample 2: Registering platform-specific services

Code sample 2 shows how we construct and register several platform-specific singleton services in our application with the MvvmCross IoC provider. These are consumed in the view models via constructor injection, as shown in Code sample 3.

```
public PickerViewModel(
    IMvxNavigationService navigationService,
    IColorPicker picker,
    IImageDrawing drawing,
    IPopupService popupService,
    ITestCaseLoader testCaseLoader)
{
    _navigationService = navigationService;
    _popupService = popupService;
    _picker = picker;
    _drawing = drawing;
    _testCaseLoader = testCaseLoader;
}
```

Code sample 3: Consuming services via constructor injection

2.3. Photo capture and processing

Photo capture is a common component of both the Calibration and Colour picking phases of the application workflow.

2.3.1. User experience

Our original approach aimed to perform calibration and colour picking from the real-time camera feed. Through trial and error, we concluded that such an approach has several disadvantages.

When the camera is not still enough, the difference between the intended location, which the user tapped for colour picking and the actual point picked could significantly alter the results. The chances for this problem are pronounced by the fact that when the device screen is physically touched while held in hand, it causes an involuntary but perceptible movement.

Secondly, iOS camera APIs require playing a shutter sound when a photo is being captured to avoid privacy issues. Repeating photo capture in a loop makes the device replay the shutter sound over and over again, which causes a very unpleasant user experience and can be prevented only by muting the phone speaker completely.

For these reasons, we opted for a two-step approach. In case of calibration, the user first taps the capture button to take a photo, and the application then locates the ColorChecker® and displays the identified colour patches. For colour picking, the user first captures a still picture and then can tap different locations on this photo to view the results. Moreover, a single photo can be reused to pick multiple colour samples, and the user interface can display an ellipse to show the source area. This visual confirmation avoids the inherent imprecision of human fingers on a touch screen with visual confirmation.

2.3.2. Camera capture overview

As the process of capturing a RAW and a processed photo is a common functionality for both the calibration and colour picking phase of the application workflow, it is abstracted as a MvvmCross interaction handler `RawCaptureRequestHandler`. When the View Model layer requests camera capture, this handler uses platform-specific APIs to handle the interaction request and produces an instance of `RawPhotoCaptureResult` structure as a result.

This structure contains a Boolean flag to mark if the capture attempt was successful, necessary metadata including the capture timestamp, exposure and ISO values (discussed in more detail in 2.3.3) and reference to both the raw image data

(instance of a custom `RawImage` class) and the processed photo (which encapsulates the native Apple iOS `UIImage` class).

The process of capturing a photo uses the platform-specific `AVCaptureDevice` and `AVCaptureSession` APIs. While the details of this code are not relevant to the goal of this thesis, they include one of the essential challenges of locking exposure and ISO values before the photo capture.

2.3.3. Managing shutter speed and ISO

It is critical to make sure not only to keep the lighting conditions stable but also to maintain the same camera shutter speed and ISO settings used during calibration to achieve consistent results between the camera calibration and colour picking phases. Shutter speed can be retrieved right before capturing using the `AVCaptureDevice.ExposureDuration` property while ISO value is available under the `AVCaptureDevice.ISO` property.

The iOS API allows the camera sensor to lock to a set of pre-defined exposure and ISO values. Unfortunately, from our tests, we concluded that the sensor changes the value too rapidly, even ahead of the capture, for it to be dependable. To avoid this discrepancy, we need to lock the shutter speed and ISO values before capturing manually.

Our code waits for the device to adjust and then locks the exposure duration and ISO values (this uses the built-in `AVDevice.LockExposureAsync` API). Although this method can be awaited and is documented to yield its result only after the exposure is locked (`AVCaptureExposureMode.Custom` mode is set), we found out that this behaviour is not reliable either, so we added another loop to ensure the exposure locking process has properly finalised before we continue capturing the photo. The first iteration of the resulting code is shown in Code sample 4. This code has later been expanded to introduce overexposure handling.

```
if (adjustExposure)
{
    while (_device.AdjustingExposure)
    {
        await Task.Delay(200);
    }

    await SetExposureAsync(
        AVCaptureExposureMode.Custom,
        _device.ExposureDuration,
        _device.ISO);

    while (_device.ExposureMode != AVCaptureExposureMode.Custom)
    {
        await Task.Delay(200);
    }
}
```

Code sample 4: Locking the exposure and ISO values

2.3.4. Avoiding overexposure

The camera on the iPhone model used for testing showed that during daylight, the auto-exposure on the device often resulted in overexposed photos (especially visible on the white patch of the ColorChecker®). With overexposed capture, the values of colour channels get clipped, and colour information is lost. This problem is even more significant during the colour picking phase when we need this information to properly transform the picked colour from the camera colour space to sRGB.

To mitigate this problem, an exposure adjust setting was added in the application configuration, and by default, it is set to 0.8 – meaning the exposure is set to 80% of the value suggested by the camera sensor. The code for exposure and ISO locking takes this setting into account and multiplies the system-suggested exposure duration, as shown in Code sample 5.

```
var exposureDuration = _device.ExposureDuration * Configuration.AdjustExposure;
exposureDuration = exposureDuration.Clamp(
    _device.ActiveFormat.MinExposureDuration,
    _device.ActiveFormat.MaxExposureDuration);
var iso = _device.ISO.Clamp(
    _device.ActiveFormat.MinISO,
    _device.ActiveFormat.MaxISO);
await SetExposureAsync(
    AVCaptureExposureMode.Custom,
    exposureDuration,
    iso);
```

Code sample 5: Adjusting exposure and ISO

2.3.5. Reading camera output

iOS camera can produce both processed and unprocessed image data at the same time. Reading camera output is essential for both calibration and colour picking phases of our application's workflow.

The developer can configure the RAW photo capture using the `AVCapturePhotoSettings` class. It provides a `FromRawPixelFormatType` factory method which allows the user to specify the RAW pixel format type (value from the `CVPixelFormatType` enumeration). The resulting settings instance is then passed to the `AVCapturePhotoOutput.CapturePhoto` method which starts the photo capture process.

When photo capture is finished, the `AVCapturePhotoCaptureDelegate` is triggered. This class provides multiple overridable methods, but in our case, the `DidFinishProcessingPhoto(AvCapturePhotoOutput, AVCapturePhoto, NSError)` overload is used. It is called twice each time we capture a photo – once for the RAW image and once for the processed photo. We can distinguish between these two calls using the boolean `RawPhoto` property of the `AVCapturePhoto` instance, which is passed in as an argument.

The processed image is used to locate the ColorChecker® and is stored as a PNG file.

Raw image data are stored in the `FileDataRepresentation` property of `AVCapturePhoto`. The raw image is essentially a buffer of bytes.

2.3.6. Demosaicing

The raw image data produced by the iOS camera are arranged into a RGGB Bayer pattern of 14-bit channel values. We can choose one of the more straightforward demosaicing approaches as we are not concerned with the display of the image and small artefacts will not matter as for calibration and colour picker we are always averaging the colour from an area with a larger radius.

To get the colour channel values for pixel at coordinates `x` and `y`, we calculate the average brightness values of the red, green and blue filters in a 3 by 3 area around the given coordinates in the raw image (starting at `x-1`, `y-1`, ending at `x+1`, `y+1`). For image's edges, we will ignore the coordinates which are out of bounds and will get the average colour from a the limited area only.

2.4. Locating the ColorChecker® Passport in a photo

To calibrate, we need to locate the ColorChecker® device in the captured photo and find the location of its colour patches. The conditions under which the user will capture the passport device are unknown beforehand, hence need to make sure that the algorithm we choose can accommodate even for cases where the ColorChecker® is not in the exact centre of the captured photo, is slightly rotated or even partially skewed.

2.4.1. ARKit and CoreML

We first attempted to locate the colour checker using the features provided by ARKit and CoreML APIs provided by the iOS SDK. These APIs are especially suited for augmented reality scenarios and initially seemed to be a good fit for our use case.

The first type of API we tested was ARKit plane detection, which can locate surfaces in the captured photo. The limitation of this is the ColorChecker® was usually not discovered when lying flat on the table as it "blended" with its surroundings. This limitation is severe as we expect this is the form in which most users will want to perform the calibration. Moreover, ARKit plane detection currently supports only horizontal or vertical planes, and the user would need to choose beforehand which form of detection is suitable.

CoreML API offers rectangle detection using the `VNDetectRectanglesRequest` class. This API will return the location of rectangular objects detected in the photo. Unfortunately, based on our tests, the results are not always reliable and are further hindered by the fact that the ColorChecker® Passport is not a perfect rectangle – instead, it has rounded corners. The inner rectangle around the colour patches is sharp, but its border is not highlighted so in most cases the rectangle search will not successfully locate it. Furthermore, the ColorChecker® can be standing, or partially open, in which case the device no longer has a rectangular form at all.

2.4.2. Machine learning approach

As iOS devices now support machine learning natively via the CoreML framework, we considered training a custom model that would recognise the colour checker. As this would, however, require a significant number of photo samples and capturing the device under different lighting conditions and we would need to achieve enough precision in locating the colour patches within the checker as they cover only a small area of the overall captured photo, we decided against this approach.

2.4.3. ColorChecker Finder approach

Keigo Hirakawa from the University of Dayton devised a method of finding colour sample locations of a ColorChecker® in a photo in Matlab with an algorithm called CCFind. (39)

As Matlab cannot run on iOS, we tried to port the same logic to the OpenCV framework. In many cases, it was possible to find equivalent or very similar methods as those used in the Matlab solution, but there were points where the APIs diverged significantly, and it was necessary to rewrite those code paths from scratch.

Although the final solution could locate the colour checker and returning results, we quickly realised it has a very significant performance cost. Finding the results on a performant personal computer took tens of seconds, especially for a larger photo. In case of a low-performance smartphone, this would not be acceptable and would make the application less usable.

Another reason for not using the CCFind algorithm in the final solution was the fact that it is copyrighted and allowed for research use only. Hence it would not be

possible to provide the Colour Picker application to end-users without violating this copyright.

2.4.4. A custom approach using OpenCV rectangle search

The method we have chosen as the final solution is a custom solution based on the OpenCV framework. This multi-step process is described in detail in the following sections.

Because in this phase of the ColorChecker® calibration we are concerned with colour patch locations only and do not require perfect colour accuracy, we do not need to work with the full raw image and the processed downscaled PNG photo is sufficient.

The solution will be demonstrated on an example using a ColorChecker® photo in Figure 17.

Figure 17: ColorChecker(R) photo detail (40)

2.4.5. Locating colour patch squares

The OpenCV documentation (41) contains a suitable example for discovering squares in a list of images which we used as a starting point and slightly customised. The code first downscales and upscales the image to filter potential noise and artefacts. For each colour plane of the image, it then searches for contours.

The code then loops over multiple threshold levels and searches for shape contours in the image with the `findContours` function. Each contour is tested to check

if it is similar to a square. We first approximate the contour according to its perimeter, and the resulting shape should have four vertices and be convex (Code sample 6).

```
findContours(gray, contours, RETR_LIST, CHAIN_APPROX_SIMPLE);
vector<Point> approx;
for (const auto& contour : contours)
{
    approxPolyDP(contour, approx, arcLength(contour, true) * 0.02, true);
    if (approx.size() == 4 &&
        fabs(contourArea(approx)) > MIN_CONTOUR_AREA &&
        isContourConvex(approx))
    {
        //...
```

Code sample 6: Finding, approximating and filtering contours

Finally, we need to confirm that the angles between adjacent edges are approximately 90 degrees. For our application, we have added an optimisation to filter away squares with a too large area, as the photo must feature the whole ColorChecker®, so a too large square cannot possibly represent a single colour checker colour patch (see Code sample 7).

```
double maxCosine = 0;
for (int j = 2; j < 5; j++)
{
    // find the maximum cosine of the angle between joint edges
    double cosine = fabs(
        VectorHelper::angle(
            approx[j % 4],
            approx[j - 2],
            approx[j - 1]));
    maxCosine = MAX(maxCosine, cosine);
}

auto bounds = minAreaRect(approx);

if (maxCosine < 0.2 &&
    max(bounds.size.width, bounds.size.height) <= maxSampleSize &&
    RectHelpers::is_almost_square(bounds))
{
    rectangles.push_back(bounds);
}
```

Code sample 7: Checking inner angle sizes and filtering large squares

The contours found by this part of the algorithm are shown in Figure 18.

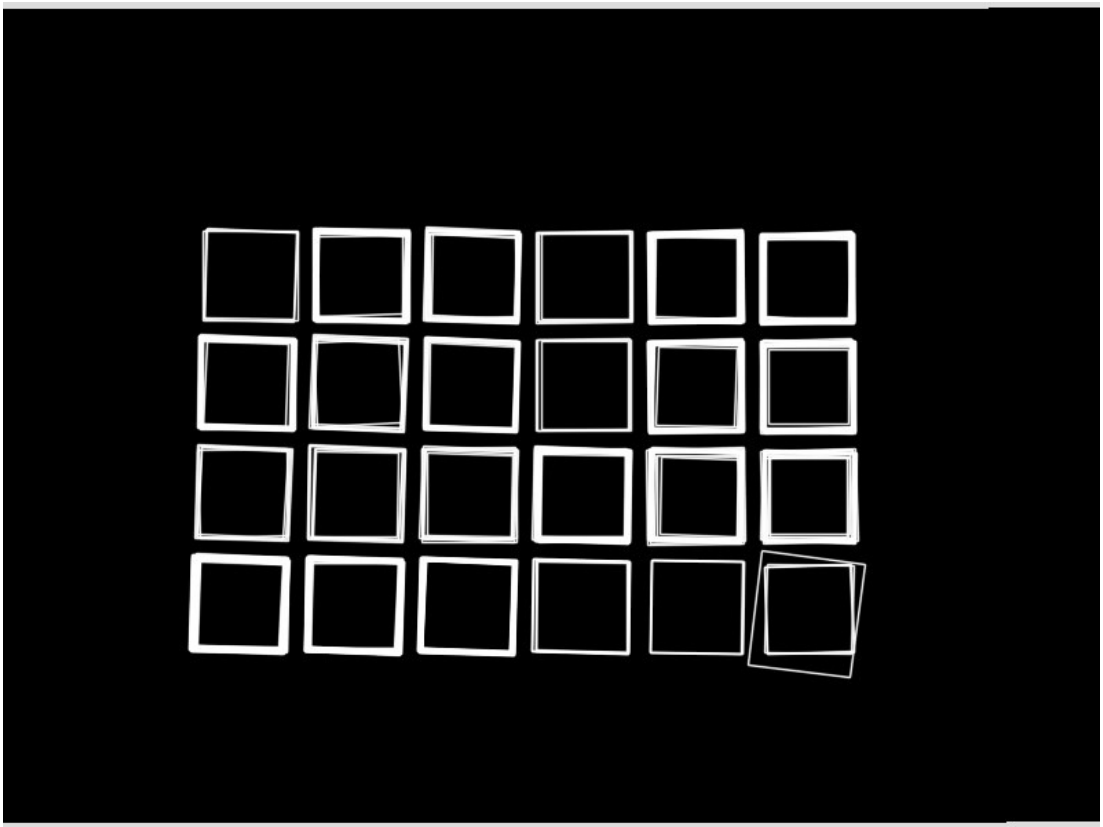

Figure 18: Visualised ColorChecker® patch contours

2.4.6. Finding white patch candidates

We base our search on the white patch of the ColorChecker®. For each contour, we have located in the previous step we calculate the average L*a*b colour around its centre. As shown in Code sample 8, we calculate the colour difference between the white ColorChecker® patch and the resulting colour using CIE Delta E 76 distance and filter out those which differ too significantly.

```
static void find_whites(
    Mat& lab_img,
    vector<RotatedRect>& color_samples,
    vector<RotatedRect>& white_samples)
{
    for (auto color_sample : color_samples)
    {
        auto cieDistance = calculate_sample_color_distance(
            lab_img,
            color_sample.center,
            checkerConfiguration.whitePosition[0],
            checkerConfiguration.whitePosition[1]);

        if (cieDistance <= MAX_WHITE_CIE_DISTANCE)
        {
            white_samples.push_back(color_sample);
        }
    }
}
```

Code sample 8: Finding white patch candidates

2.4.7. Identifying the best candidate

With a set of candidates for the white patch, we need to identify which one can potentially be the ColorChecker® white patch as well as identify the orientation of the checker for this patch.

There are three colour patches above the white patch in ColorChecker® – blue, green and red (these patches are highlighted in Figure 19). This is a unique coloured layout of patches on the ColorChecker®, which is advantageous, as we can use this pattern to quickly rule out invalid matches and orientations.

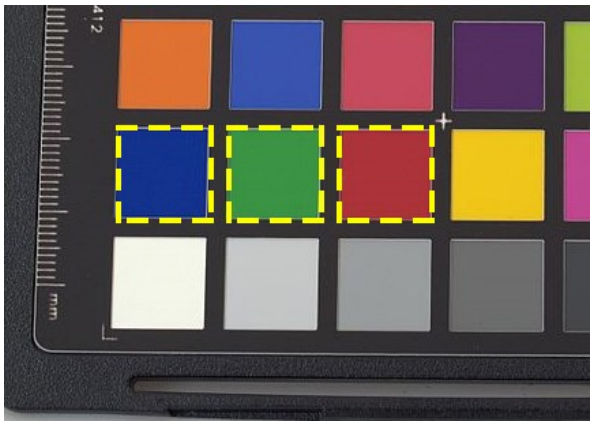

Figure 19: Patches used to determine ColorChecker® orientation

The OpenCV `RotatedRect` class provides an `Angle` property which can be used to retrieve the orientation of the recognised rectangle. We move in a perpendicular direction to each of the four edges of this recognised square. We check for the blue sample in this location and then again perpendicularly to the right to find the green and red samples. We calculate the CIE Delta E 76 distances for the three centre points of potential patches and select the option with the lowest maximum distance. Figure 20 shows the points explored in this part of the algorithm. The blue, red and violet arrows show unsuccessful matches, while the green arrow path matches all three patches.

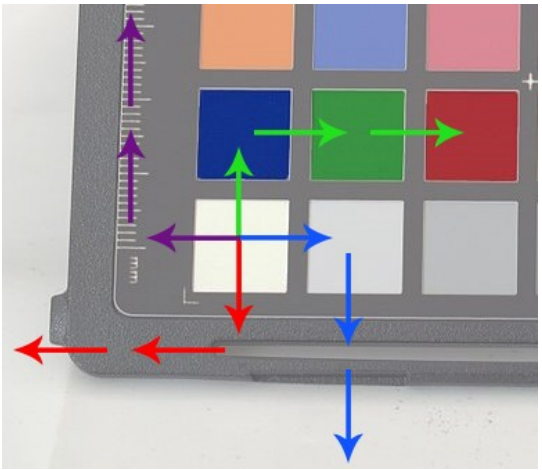

Figure 20: Finding the white patch and orientation

2.4.8. Finding remaining colour patches

If the algorithm identified a white patch which satisfied the required criteria and recognised the right orientation of the ColorChecker®, we conclude that we have found the colour checker's white patch. Further, we can find all its 24 patches starting from this known patch and checker orientation. We do this in a breadth-first fashion. For each unidentified patch, we first calculate its approximate centre using the locations of the known neighbours and their dimensions. We need to accommodate not only for the patch size but also for the margin in between patches on the ColorChecker®. We use the approximate location of the patch centre and search the list of contours to find a square with a centre close to this point. If there is such a contour with matching orientation and dimensions, we mark it as the contour identifying the patch. If there is no such contour, we generate a faux contour with this

approximate centre and same dimensions as the neighbours. We continue this way until all 24 patches have been identified successfully.

Figure 21: Sample output

Finally, the centre locations of all patches are returned to the caller. Figure 21 shows the output locations drawn onto the input photo.

2.5. Calibration

Before the user can use the mobile application, they must calibrate the camera sensor appropriately using the ColorChecker® Passport device.

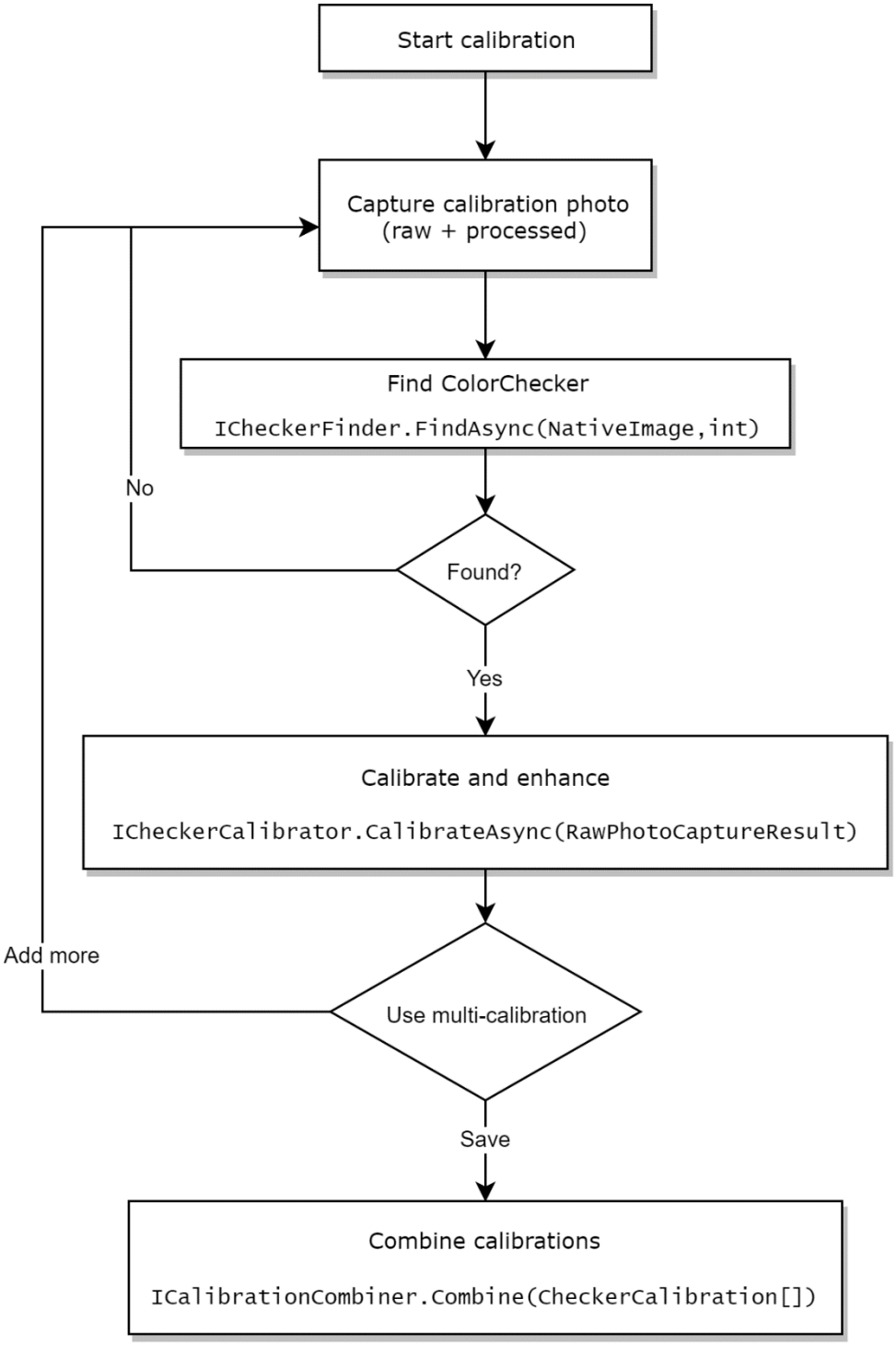

Figure 22: Calibration workflow diagram

2.5.1. Gathering colour values

After we capture the raw and processed photo, we use the algorithm described in Section 2.3 to find the locations of the individual colour patches. We preview the locations to the user to confirm the checker was correctly found in `CalibrationViewModel.CalibrateUsingPhotoAsync`.

If the user is satisfied with the result, we use the calculated locations to gather average colour values in the raw image. For each colour patch, we average the red, green and blue values in an area of a given radius (based on application configuration). Code sample 9 shows how we retrieve the camera space coordinates for every patch.

```
var patches = new List<NormalizedColor>();
for (var patchIndex = 0; patchIndex < checker.Count; patchIndex++)
{
    var position = checker[patchIndex];
    patches.Add( item: rawImage
        .GetAverageColorAroundPoint((int)position.X, (int)position.Y, Configuration.CalibrationRadius)
        .ToNormalizedColor());
}
```

Code sample 9: Averaging colour around the patch centres

2.5.2. Calculating sRGB conversion factors

Because we know the sRGB coordinates, we can calculate the multiplication factor which transforms each of the channel values from the camera space coordinates to sRGB colour space coordinates as seen in Code sample 10.

```
for (var patchIndex = 0; patchIndex < checker.Count; patchIndex++)
{
    var d65Colour = configuration.D65ColorSamples[patchIndex];
    var patchCalibration = new PatchCalibration(
        patchId: patchIndex + 1,
        patches[patchIndex],
        factors: d65Colour / patches[patchIndex]
        );

    patchCalibrations.Add(patchCalibration);
}
```

Code sample 10: Calculating colour patch calibrations

We save the camera space values and factors in the resulting calibration JSON file. This calibration file is persisted in the application data folder so that the calibration is reusable across multiple launches of the application.

2.5.3. Calibration enhancers

We introduced the concept of calibration enhancers when implementing experimental support for the ColorChecker® Digital SG. An enhancer is checker

configuration-specific and implements the `ICalibrationEnhancer` interface which accepts and produces an existing `CheckerCalibration` instance.

One enhancer comes with the application – `SgCornerCalibrationEnhancer`. This is built for ColorChecker® Digital SG and improves the calibration by averaging the camera space coordinates of the white patches in the corners. By extension, the differences between the four corners create a gradient, which is not readily perceptible by the human eye but could influence calibration. The enhancer mitigates this gradient.

2.5.4. Multi-calibration

Because minute changes in lighting could prove problematic for the calibration and the camera sensor may not have uniform sensitivity, we added support for multi-calibration. This way, the user can capture multiple calibrations in the same setting. All captured calibrations are then combined using a class that implements the `ICalibrationCombiner` interface.

The implementation provided with the application performs an averaging of colour coordinates and factors for each of the patches on the ColorChecker® device (see Code sample 11) and returns a final calibration.

```
foreach (var patchGroup in groupedPatches)
{
    var averageColor = patchGroup.Select(p => p.CameraColor).Average();
    var averageFactors = patchGroup.Select(p => p.Factors).Average();
    resultPatches.Add(new PatchCalibration(patchGroup.Key, averageColor, averageFactors));
}
```

Code sample 11: CalibrationCombiner excerpt

2.5.5. Visualisation with .ply files

To understand the way the camera colour space behaves relatively to the sRGB colour space, we needed to be able to surface it visually in three-dimensional space. We utilised an open-source application called MeshLab, which is primarily suited for processing and editing of 3D triangular meshes. This application can read the `.ply` file format (Polygon File Format), which is specifically suited to store three-dimensional graphical data.

The format provides a specification for both an ASCII representation as well as a compact binary format. It is designed to be simple enough so that any

programming language can generate it without requiring a specialised library but also potentially extensible. In its core, a `.ply` file describes a single three-dimensional object and contains a list of vertices, edges and faces that compose it. It can also store colour information for each of the components. (42)

For our purposes, we require a list of vertices to represent the camera colour points and sRGB colour points and a list of edges to connect the related vertices in space. The header of such a `.ply` file will look as shown in Figure 23.

```
1   ply
2   format ascii 1.0
3   comment author: Colour Picker
4   element vertex 288
5   property float x
6   property float y
7   property float z
8   property uchar red
9   property uchar green
10  property uchar blue
11  element edge 152
12  property int vertex1
13  property int vertex2
14  property uchar red
15  property uchar green
16  property uchar blue
17  end_header
```

Figure 23: `.ply` file format header

The file format starts with general metadata about the file itself – its format, and author. The `element` keywords start the definition of types of data stored. The example file in Figure 23 stores vertices as a list of x, y and z coordinates and colour information and a then edges between those vertices, again with colour. The numeric values at the end of `element` declarations provide the number of vertices and edges stored in this file.

The individual elements are then stored in the file line-by-line in groups in the same order as in the header – in our example, we first list all vertices (shown in Figure 24) and then the edges (shown in Figure 25). Note that each line is a list of values in the same order as in the header definition. Also, note the edges reference its start and end vertices by index in the vertex list.

```
0.5354716 0.5343067 0.509857 230 235 230
0.900563 0.919114 0.899896 230 235 230
0.1090008 0.1089561 0.1099595 46 46 46
0.181249 0.182093 0.18155 46 46 46
```

Figure 24: List of vertices in `.ply`

```
38 39 64 2 30
40 41 13 6 20
42 43 148 168 192
44 45 37 12 5
```

Figure 25: List of edges in `.ply`

The implementation of our ColorChecker® calibration to `.ply` conversion is provided in the `ColourPicker.CalibrationToPlyConverter` project, with the main logic within the `ColourPicker.Core.Ply` namespace. Our initial attempt to visualise with PLY directly showcased the camera colour points against sRGB. The result, unfortunately, did not provide enough useful information as the camera data are not evenly distributed in the same (0,1) value interval as sRGB values. An example of such initial .ply file visualised in MeshLab is shown in Figure 26.

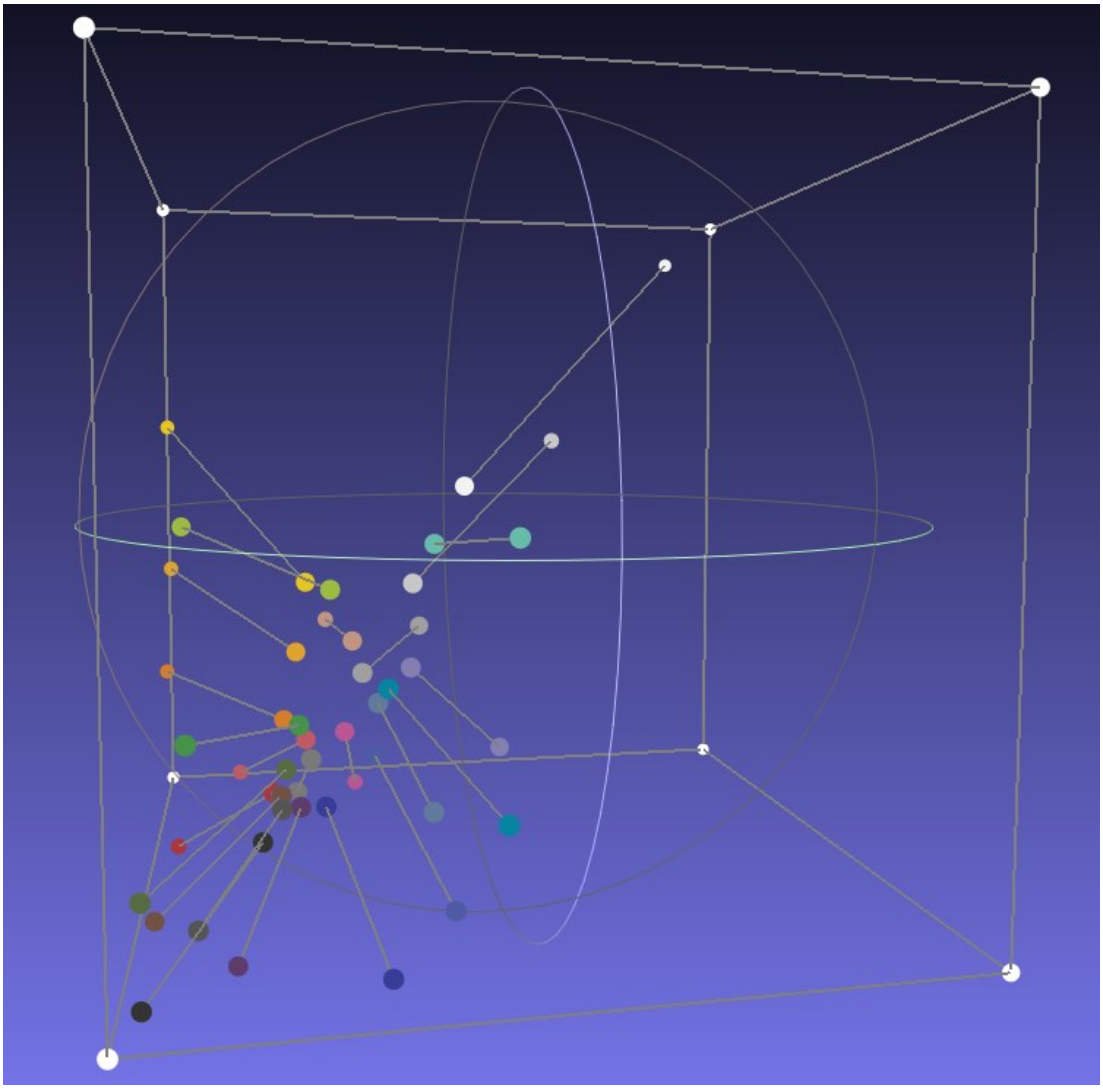

Figure 26: Visualising calibration in MeshLab (no transformation)

It is clear that while the sRGB colour points are well distributed in the cube, the camera space colour points are all concentrated in a small area, which makes it particularly difficult to understand their relationship.

To mitigate this disadvantage, we apply a three-dimensional matrix transformation to the camera colour points to align the camera-black point and camera-white point with each other and therefore scale and align the space properly onto sRGB space. The transformation matrix sequence follows:

1. Compute 3D vector from camera black (CB) to camera white (CW) – produces camera neutral axis vector (CNA)
2. Compute 3D vector from sRGB black (SB) to sRGB white (SW) – produces sRGB neutral axis vector (SNA)

3. Translate CB to the origin
4. Apply rotation that aligns CNA with SNA
5. Scale with a factor that makes the length of the neutral axis equal to SNA (i.e. that scales the length of CNA to SNA)
6. Translate the origin to SB

Implementation of the transformation matrix calculation is in the `ColourSpaceHelpers` class within the `ColourPicker.Core.Helpers` namespace. It uses a `System.Numerics.Matrix4x4` structure and open-source Math.NET Numerics library (43) to produce the rotation matrix.

After applying this transformation to all camera colour points, a significantly more accurate visualisation is produced, as shown in Figure 27.

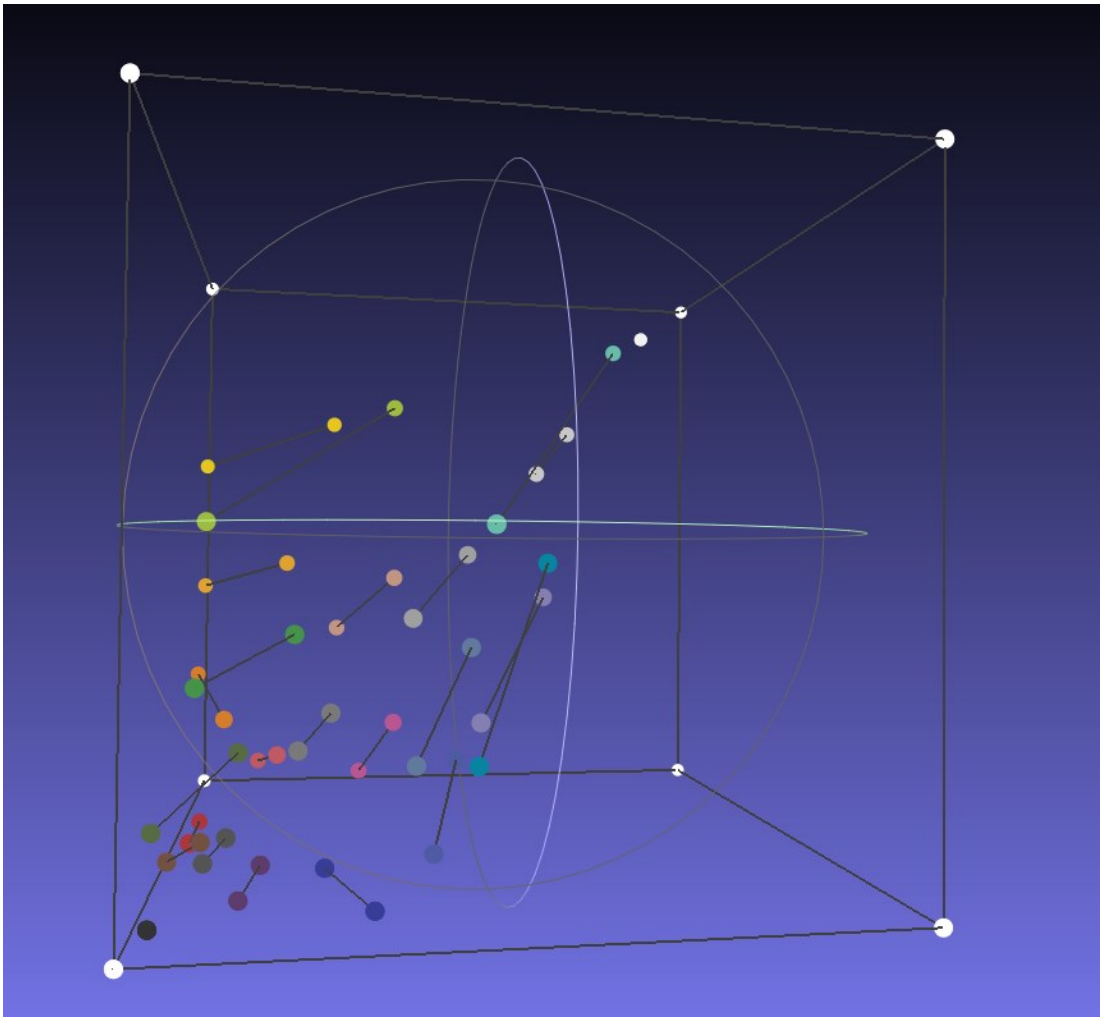

Figure 27: Visualising calibration in MeshLab (matrix transformation)

In this case, the camera black is aligned with sRGB black and camera white with sRGB white, and all other colour points are scaled and distributed in the unit cube appropriately. We will discuss the implications of the camera colour space non-uniformity in Sections 3.2 and 3.4.1.

2.6. Picking a colour

Once the application is calibrated, the user can proceed to pick a colour from his surroundings. The first step is the same as in case of calibration – we capture a raw photo and display a processed PNG image as a preview to the user. The user then taps the photo to select the area from which the colour is picked.

We can get the red, green and blue camera space coordinates of the selected area (with a radius based on configuration) directly from the captured raw image. We need to transform these coordinates into the sRGB colour space coordinates. The approach we have chosen is to utilise the colour patches of the calibrated ColorChecker®, find the closest similar colour patches from the calibrated image and averaging their sRGB conversion factors.

2.6.1. Searching in kd-Trees

To quickly identify the closest ColorChecker® patches, we build a three-dimensional kd-Tree from the calibrated patches. We cannot, however, use the colour channel values themselves as the kd-Tree coordinates, as the sensitivity of the camera sensor differs for each channel. Instead, we must adjust each channel to have approximately the same range so that the distance calculation is fair.

We subtract the maximum and minimum camera space value in the calibration for each channel:

$$red_{\Delta} = max_{red} - min_{red}$$

$$green_{\Delta} = max_{green} - min_{green}$$

$$blue_{\Delta} = max_{blue} - min_{blue}$$

Now we use the red_{Δ} value to calculate a scaling factor for the other channels:

$$scale_{green} = red_{\Delta}/green_{\Delta}$$

$$scale_{blue} = red_{\Delta}/blue_{\Delta}$$

When adding a colour patch into the tree we keep the red channel value intact and multiply the green and blue channels by $scale_{green}$ and $scale_{blue}$ respectively. Searching in this kd-Tree will now assign differences in all channels the same relative weight.

As our algorithm does not require any internal modification of the kd-Tree algorithm, we decided to use an open-source MIT licensed library KdTree available on GitHub (44).

2.6.2. Calculating sRGB conversion factors

Given the user-picked colour sample, we search for a pre-configured number of closest colour patches in the ColorChecker® using the pre-built kd-Tree.

For purposes of further discussion, suppose P represents the camera colour coordinates of the picked colour and C_i are the camera colour coordinates of the i-th closest ColorChecker® patch taken in ascending order of Euclidean distances from P. Euclidean distance of C_i from P will be denoted as d_i. We also denote $S[R]$ as the red channel value, $S[G]$ as the green channel value and $S[B]$ as the blue channel value of camera colour coordinates S. We define the multiplication factor required to transform the red channel of the i-th closest ColorChecker® patch as $f_i[R]$ and analogously $f_i[G]$ for green and $f_i[B]$ for blue channel.

As we have seen in Section 2.5.5, the sRGB transformation factors vary significantly. Therefore it is beneficial to filter away those colour samples which are more than twice as far as the closest match – meaning we take into account only colour patches in the set $C = \{C_1, C_2, \dots, C_m\}$ where m is equal to the lowest value of i for which $d_i > 2d_1$.

Our goal is to calculate the appropriate multiplication factor for each of the three colour channels to transform the picked colour from camera colour space to sRGB colour space.

We assign each of the nearest ColorChecker® patches a weight which is inversely proportionate to the distance of the patch and calculate a weighted-average factor for each channel. We assign each C_i from C a weight w_i which is inversely proportional to their distance from P. Besides, the sum of these weights should be equal to one. In summary, the following must hold:

$$\forall i,j \in [1,m]\ \frac{w_i}{w_j} = \frac{d_j}{d_i}$$

$$\sum_{i=1}^{m} w_i = 1$$

Based on practical testing, we devised an improved solution which uses quadratic inverse interpolation of the weights. In this option, weights are inversely proportional to the square of their distance from P. This way, the weight of colour patches decreases quadratically with increasing distance. The following two formulas must hold:

$$\forall i,j \in [1,m]\ \frac{w_i'}{w_j'} = \frac{d_j^2}{d_i^2}$$

$$\sum_{i=1}^{m} w_i' = 1$$

`InverseDistanceInterpolation` helper class implements both approaches. From our tests, the quadratic interpolation yielded more accurate results as the close matches have a stronger influence on the resulting factors.

With calculated weights $w_1, w_2, \dots, w_m$ we can calculate the sRGB coordinates of P as by multiplying by the weighted sum of the sRGB transformation factors of the selected closest ColorChecker® patches:

$$P_{sRGB}[R] = P[R] \cdot \sum_{i=1}^{m} f_i[R] w_i$$

$$P_{sRGB}[G] = P[G] \cdot \sum_{i=1}^{m} f_i[G] w_i$$

$$P_{sRGB}[B] = P[B] \cdot \sum_{i=1}^{m} f_i[B] w_i$$

2.7. Searching for nearest matches

The final step of our application workflow is to identify the nearest matches in the integrated colour atlases and display the results to the user.

Finding matching colours in colour atlases is handled by implementations of the `IColorMatcher` interface. It surfaces a single method, with two parameters – the colour atlas identifier and information about the picked colour.

Our first attempted approach was to construct a three-dimensional kd-Tree from sRGB coordinates of the atlas colours and then perform a search for the nearest matches against the calculated sRGB values of the picked colour. After thorough testing we discovered the search does not yield reliable results, as sRGB space is not *perceptually uniform*, meaning the Euclidean distance between two colours does not match their perceived distance as viewed by the human eye. The original implementation can be seen in the deprecated `RgbColorMatcher` class.

A more appropriate solution requires first converting the picked colour into the CIE LAB colour space and then use the CIE Delta E 2000 distance algorithm to find the closest colours in each of the atlases. Switching to this approach proved to yield more reliable results at the cost of slightly worse performance, which is however imperceptible to the end-user. The implementation is available in `LabColorMatcher`.

Results are calculated per colour atlas, to avoid calculating results which are not requested by the user. For each atlas, we return a pre-configured number of closest colours (the number is adjustable in the app configuration).

3. Results

In this chapter, we present the product of this diploma thesis – a mobile Colour Picker application. We demonstrate the user experience of working with the application and explore several examples of colour picking and matching process. We mention the known issues and their potential fixes or workarounds. Included is a discussion of further enhancements, which could improve the usability of the mobile application.

The complete source code of the project on a compact disk is attached to this thesis (see Attachments). We also mention how the final product can be distributed to users via the iOS App Store.

3.1. Mobile application

This section will demonstrate the user interface of the application. We focus on what functionality each view of the application offers.

3.1.1. Picker screen

The colour picker screen is the view, which the user sees immediately after launching the application from the app drawer. On the first launch, its controls are disabled as there is no ColorChecker® calibration provided yet. A pop-up alert instructs the user to perform a calibration before using the application – see Figure 28. When the user closes the alert, the application automatically navigates to the Calibration settings screen (described in Section 3.1.2).

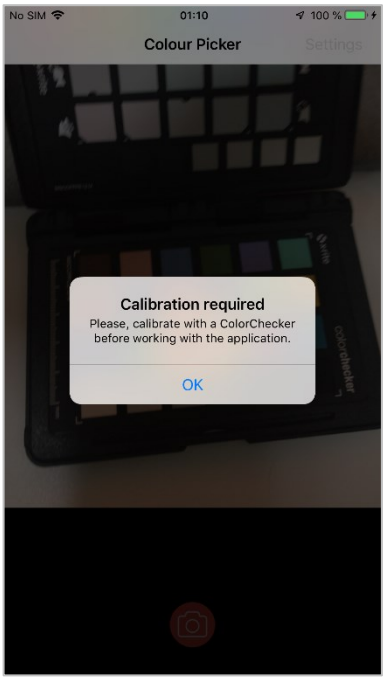

Figure 28: Picker screen (first launch)

Once calibrated, the controls on the Picker screen are enabled. The user can see a real-time preview of the camera in the middle portion of the screen. There are two interactable elements on the screen. The *Settings* button in the upper right corner navigates to the Calibration settings screen (Section 3.1.2). The *Camera capture* button in the bottom part of the view allows the user to capture a photo.

Figure 29: Picker screen ready to capture a photo

The application displays a preview of the captured photo. In case the result is blurry or distorted, the user can tap the *Retry* button in the lower-left corner of the screen – see Figure 30.

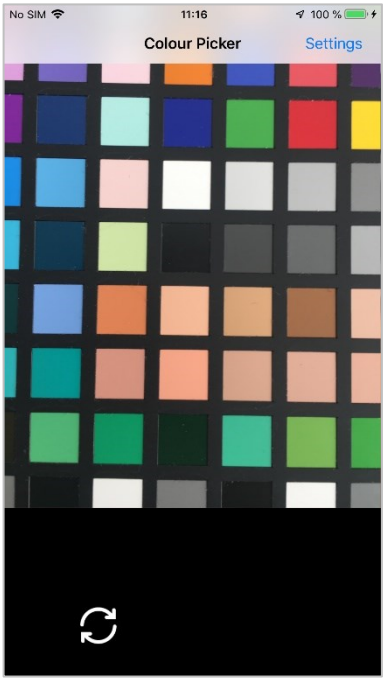

Figure 30: Photo captured

To pick a colour, the user can tap anywhere within the inner area of the image. The application configuration influences the size of the area used for the picking (see Section 3.1.4). In Figure 31, the user tapped within the green sample in the previewed photo.

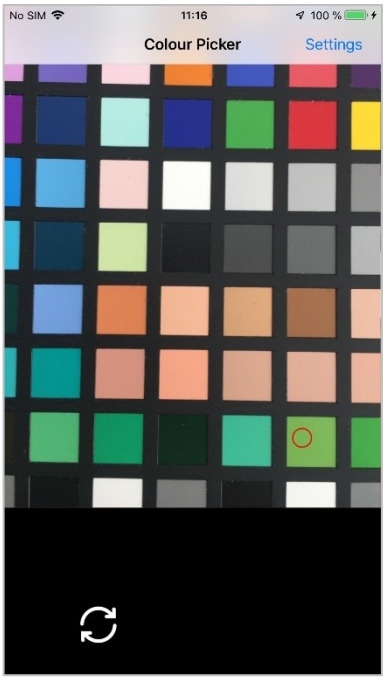

Figure 31: Picking a colour source location in the photo

The application then searches the closest matches in all colour atlases and displays the result in a compact popup list.

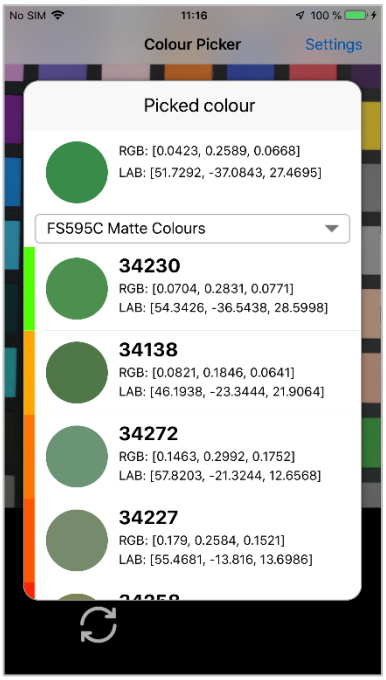

Figure 32: Colour matching results

The pop-up window displays information about the picked colour in the header area. It includes an RGB24 approximation of the colour and the calculated sRGB and LAB coordinates.

To see results from another colour atlas, the user can open the drop-down selector below the header. The application includes complete daylight colour data for the following atlases:

- Hornbach Standard Colours
- Munsell Book of Colours
- OBI Standard Colours
- Natural Colour System
- RAL Classic
- RAL Design
- PANTONE
- British Standard Colours
- Federal Standard 595 Glossy Colours
- Federal Standard 595 Semigloss Colours
- Federal Standard 595 Matte Colours

The matched atlas colours are listed below the atlas drop-down. In addition to the RGB24 approximation of the colour, its standardised name and the sRGB and LAB coordinates, each match has a coloured bar on the left side which corresponds to the accuracy of the match. The Delta E values correspond to the following colour scale:

- 1.0 and lower – Green
- 2.0 – Lime
- 4.0 – Yellow
- 8.0 – Orange
- 16.0 and higher – Red

3.1.2. Calibration settings screen

The user can configure the calibration of ColorChecker® on the Calibration settings screen. As shown in Figure 33, when no calibration has been captured, the user can tap the large area reserved for calibration preview to navigate to the Calibration screen (detailed in Section 3.1.3).

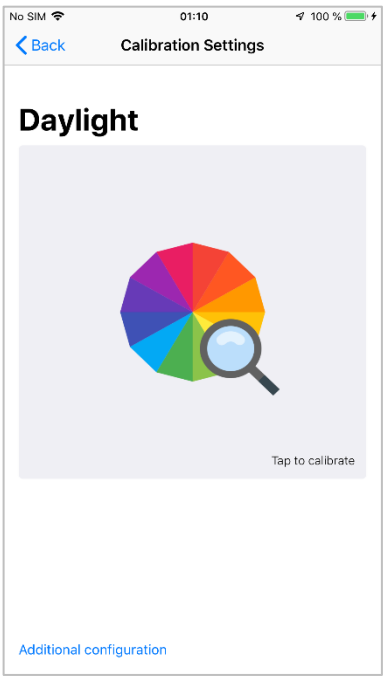

Figure 33: Calibration settings without calibration

When calibration has been set, the Calibration settings screen displays the calibration image, which is currently being used – as shown in Figure 34. The colour

patch locations are also visualised in this preview. At the right side above the image appears an eraser button which allows the user to discard of the calibration image.

Figure 34: Calibration preview

In all cases, the user can navigate back to the Picker screen by tapping the button in the top left corner of the view. Situated at the bottom is a Additional configuration button that allows the user to access to the Configuration screen (Section 3.1.4).

3.1.3. Calibration screen

The calibration screen has a similar layout to the picker screen, as shown in Figure 35. We can see the camera preview in the middle and below we can find the camera button. Additionally, above the button is displayed the name of the currently selected ColorChecker® type used for calibration (adjustable in the Configuration area of the application).

Figure 35: Calibration screen

Tapping the button captures an image and starts the algorithm which locates the ColorChecker® in the photo. If the search is not successful, a temporary toast message is presented to inform the user about this fact (see Figure 36). When the message disappears, the user can retry capturing again.

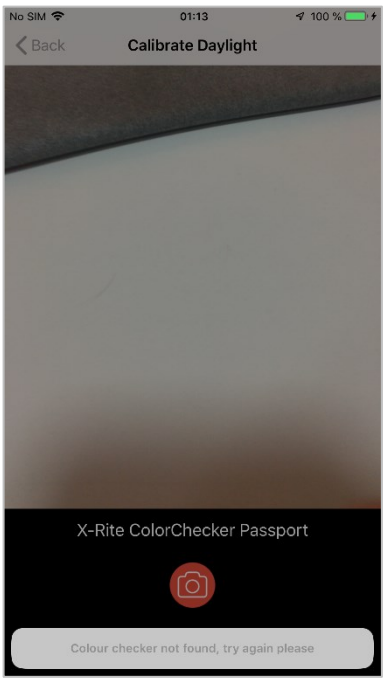

Figure 36: Unsuccessful calibration

Otherwise, the user is presented with a preview of the positions where the individual patches were located in the photo and the respective areas which will be used for the ColorChecker® calibration (see Figure 37). The radius of these areas can be adjusted in the application configuration (see Section 3.1.4).

Figure 37: Preview of localised colour patches

User can retry by tapping the *Retry* button in the lower-left area of the screen in case the patches were located improperly. Tapping the green *Confirm* button in the lower right area will proceed with the calibration.

Once finished, the user is presented with a dialog window shown in Figure 38. To improve the calibration accuracy, the user can tap the *Add more* button to add one or more additional calibration photos. The same dialog will display every time after a successful calibration.

The *Save* button will store and apply the performed calibration (or calibrations) and returns the user to the Calibration settings screen.

To discard of the in-progress calibration and return to Calibration settings screen without making any changes, the user can tap the *Discard* button.

Finally, to close the dialog and return to the *Retry* and *Confirm* button view, the user can press the Cancel button.

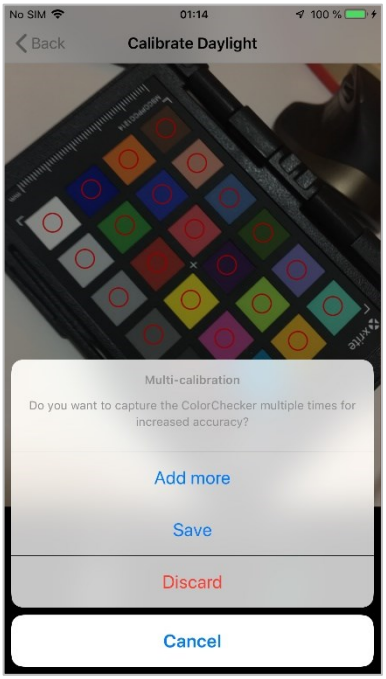

Figure 38: Multi-calibration dialog

3.1.4. Configuration screen

On the Configuration screen, the user can modify several advanced options of the application.

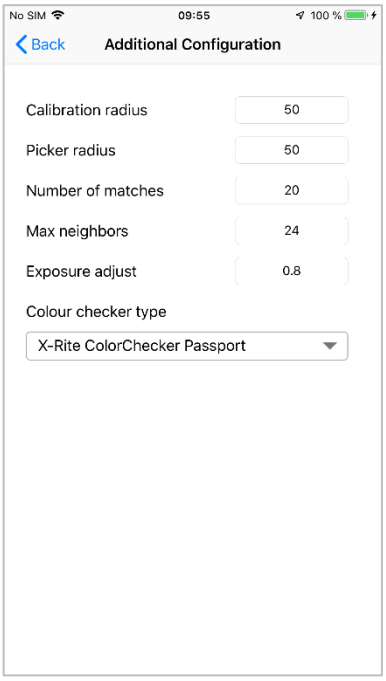

Figure 39: Configuration screen

Calibration radius and *Picker radius* options determine the radius (in pixels) of the circular area from which the colour of ColorChecker® patches and the picked colour will be averaged respectively. The default value for both these options is 50.

Number of matches allows the user to set the maximum number of closest matches for which the application will search in each colour atlas. By default, this number is 20.

Max neighbours setting allows the user to adjust the maximum number of ColorChecker® samples, which are used to calculate the transformation factors when converting the picked colour to sRGB. By default, this number is 24, which equals the number of all colour patches on the classic ColorChecker®. It is essential to note that this number only represents the upper bound. As described in Section 2.6.2, the algorithm ignores colour patches which are too distant from the picked colour.

Exposure adjust is a decimal value between 0.5 and 1.0, which is used to limit the default camera exposure during calibration to avoid overexposure (the reasoning behind this is described in Section 2.3.4).

Finally, the user can use the *Colour checker type* setting to change the colour chart used for calibration. This is an experimental feature, as the application was

developed and tested with the ColorChecker® Passport. The ColorChecker® Digital SG was added for testing purposes only, and its calibration is very unstable. Correction of this problem would require additional changes to the checker recognition algorithm. For more information, see Section 3.4.3.

3.2. Usage tests

We tested the application functionality using the *British Standard Colours* and *Federal Standard 565C Colour* atlases. Also, we verified the quality of recognition of the ColorChecker® colour patches themselves.

We found out the application works very accurately when used against the ColorChecker® colour samples and colours which have close counterparts in the calibration chart. Unfortunately, matching arbitrary colours in general tends to yield less accurate results than expected. This problem is described in Section 3.4.1.

3.2.1. British Colours 0001 Canary

The first example is the recognition of the BS0001 Canary colour.

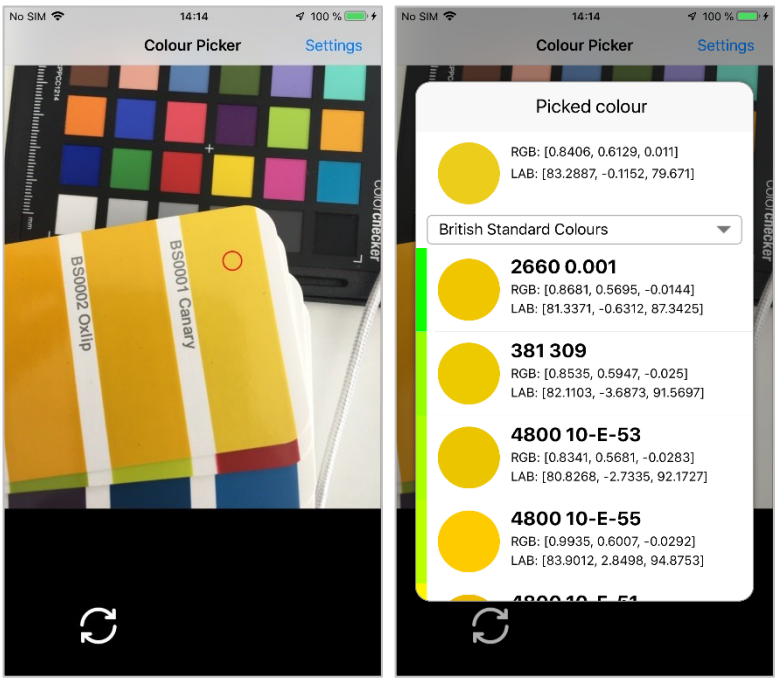

Figure 40: BS0001

The popup window shows the correct match as the closest in the set of British Standard Colours. There are multiple very close matches in the list – first four of them with Delta E distance lower than 4.0.

After switching to the Federal Standard 595C Matte Colours, we can see there is no close matching colour sample. However, in Federal Standard 595C Glossy Colours, we can find a close match (below 4.0 Delta E distance).

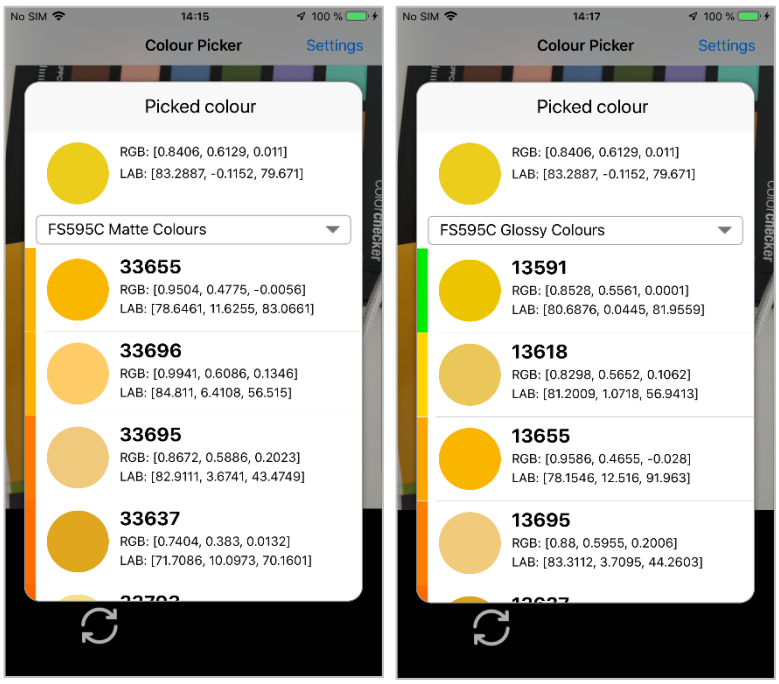

Figure 41: Federal Standard 595C matches

3.2.2. ColorChecker® Yellow Green patch

In this test, we match the ColorChecker® Yellow Green patch. This attempt yields the following set of results in British Standard Colours.

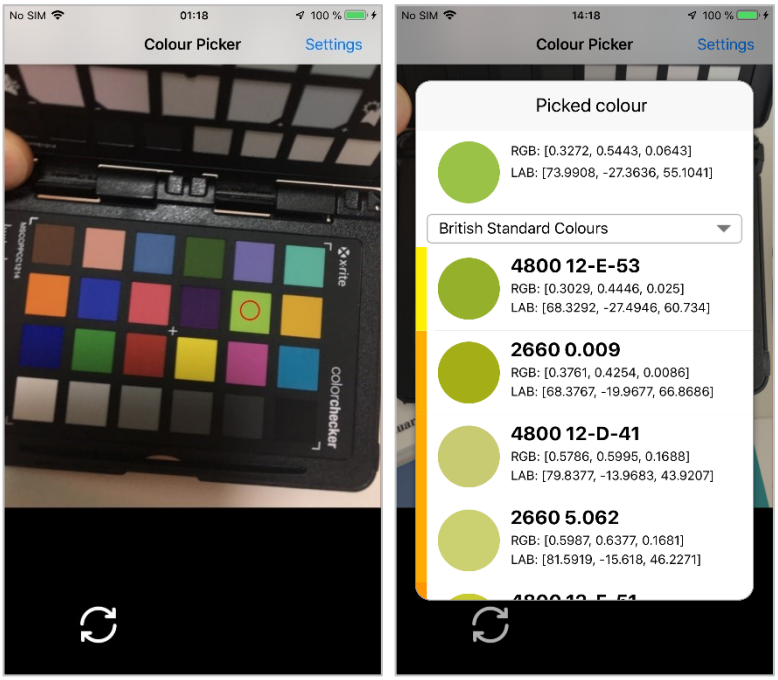

Figure 42: ColorChecker® Yellow Green matches

This is in agreement with both visual confirmation and officially generated values. See figure Figure 43.

Figure 43: Visual confirmation of the BS 12-E-53 match

3.2.3. Federal Standard 595C Matte 35240

In this case, our algorithm fails to match the 35240 colour sample.

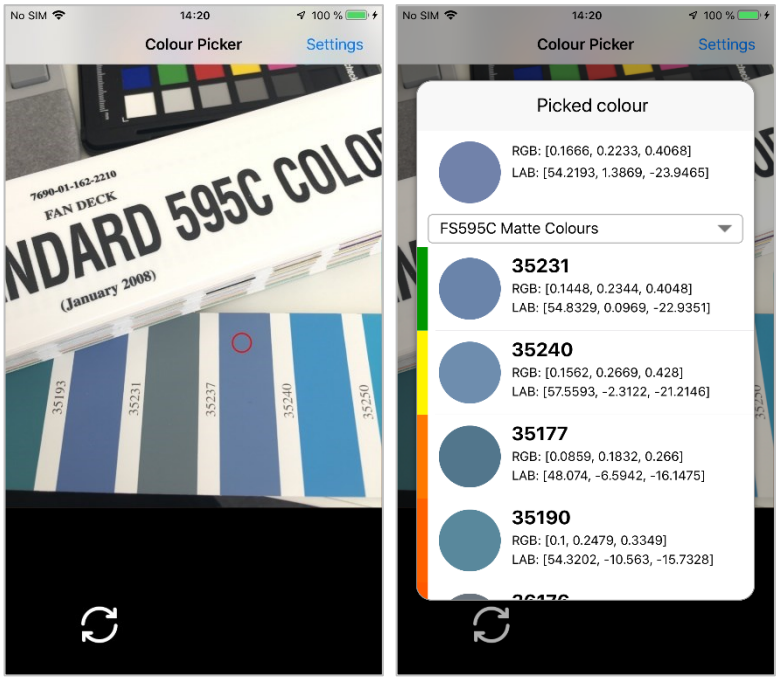

Figure 44: 35231 matched ahead of 35240

Instead of 35240, the 35231 colour is listed as the best match in the result list with 35240 being the second.

The reason for this failure is the fact that both colours are very similar, and the accuracy of our camera space to sRGB colour space transformation is not accurate enough to disambiguate between them.

3.2.4. Smartphone cover colour test

In this example, we demonstrate how picking an arbitrary colour works by matching the colour of a red smartphone cover.

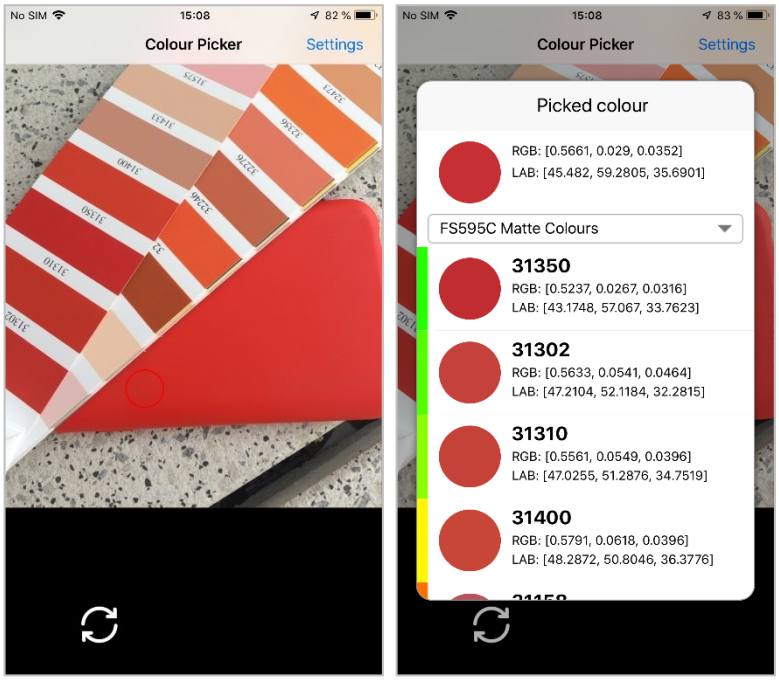

Figure 45: Matching a smartphone cover colour

As seen in the photo, the FS595C Matte Colours contains three very close matches to the picked colour. However, the situation changes remarkably when we tap a different location.

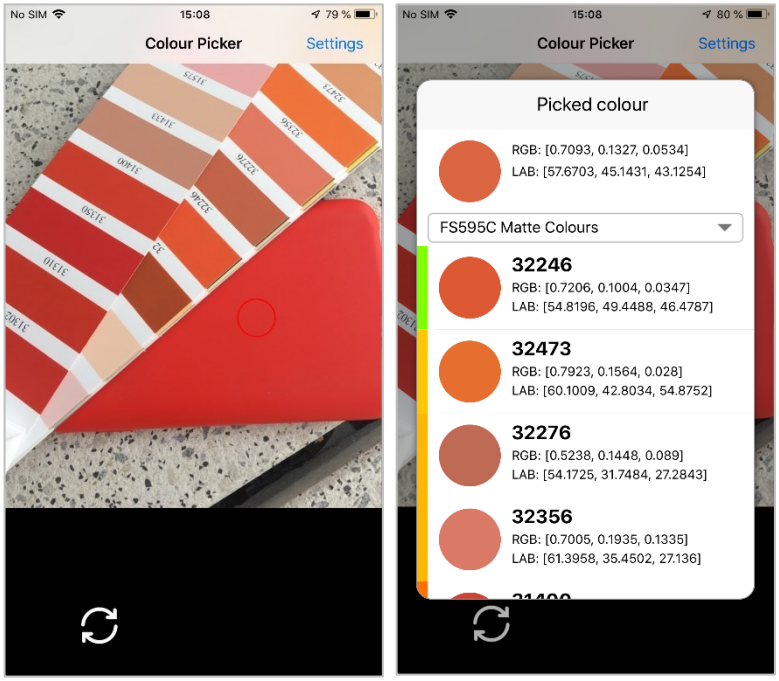

Figure 46: Picking colour from a different area yields different results

Here the picked colour was matched with a different set of colour samples. The reason for this is that the actual unprocessed camera photo captured a slight reflection at this location, which means the colour picking phase of the algorithm evaluated the orange colour patches on ColorChecker® as closer than in the first example, hence their influence on the transformation was also higher. The set of results is unfortunately inaccurate.

3.3. Application distribution

The application can be distributed to users either via the Apple iOS App Store or as a side-loaded application.

To publish the application on the App Store, it must pass a certification process, which includes additional requirements like providing a technical support contact, terms of use, privacy policy and additional image assets including multiple icon sizes and promotional images. List of all requirements and guidelines is available in the App Store Review Guidelines document (45). The application itself needs to be archived for publishing and signed using a certificate. The whole process can be performed on an Apple macOS device via a step-by-step wizard which is integrated into Visual Studio for Mac.

3.4. Known issues

3.4.1. Colour transformation for arbitrary colours

The ColorChecker® device includes a limited set of colour samples, which means the calibration has limited source information it can rely upon. This proved to be a challenge, especially when the user picks an arbitrary colour from the environment and such a colour is too distant from any of the colour samples on the passport device. In such a case, the camera colour space to sRGB colour space transformation factors are calculated with a strong influence of multiple colour samples, which may have profoundly different transformation factors. Even with weighted averaging the resulting values may not correctly reflect the accurate transformation factors.

The `.ply` visualisation of the camera colour space showed its non-linearity and uneven distribution. This is even more profoundly visible when visualising the relationship between camera colour space and sRGB colour space samples in

ColorChecker® Digital SG (see Figure 47). As illustrated in Section 3.2.4, even in a uniformly coloured area, the nearest neighbour search among the ColorChecker® samples in the camera colour space may match the wrong colour patch as the closest one, which influences the factors in an incorrect way compared to its actual expected sRGB colour coordinates.

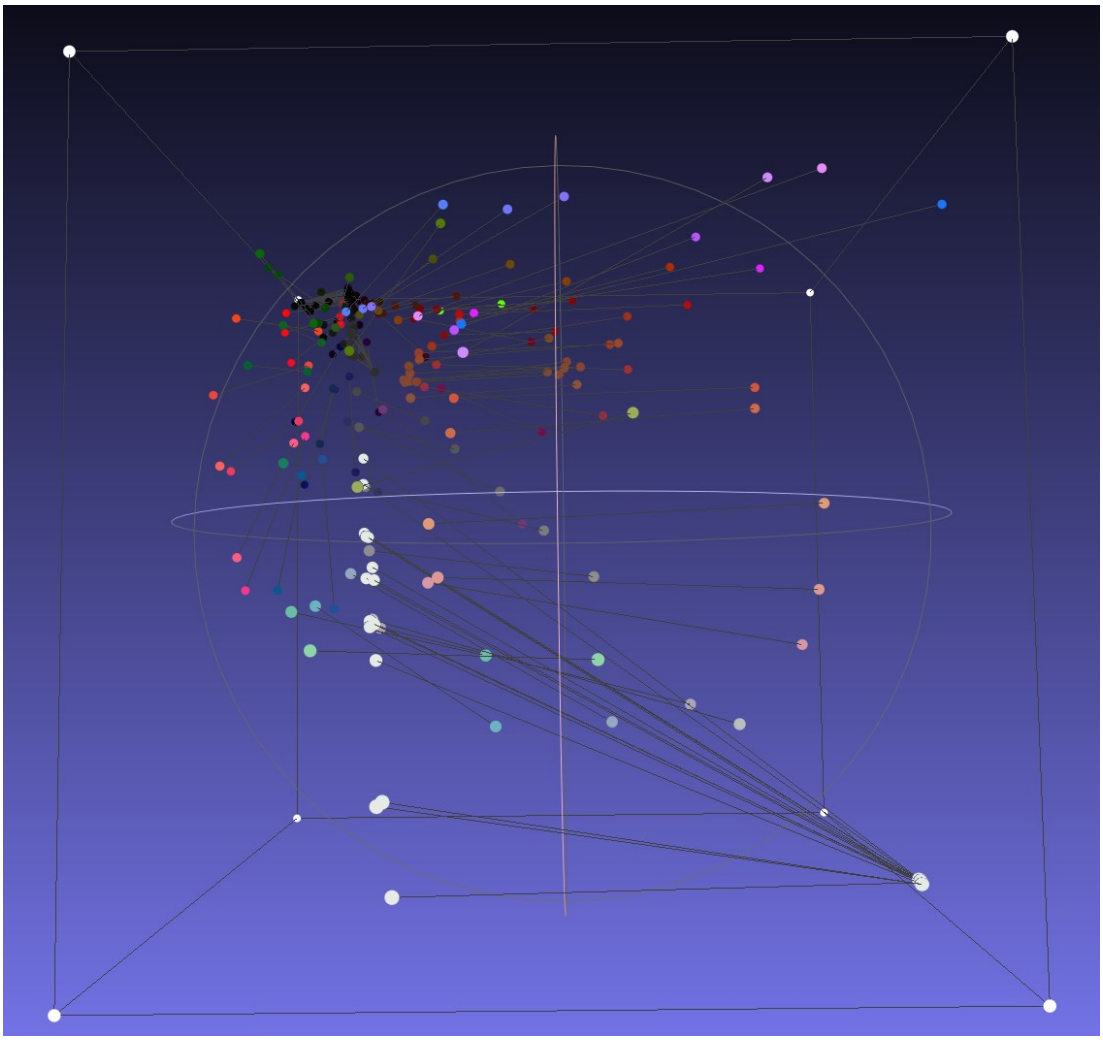

Figure 47: `.ply` visualisation of ColorChecker® Digital SG calibration

An accurate transformation from camera colour space to sRGB would require in-depth knowledge of the camera sensor hardware and proprietary algorithms the manufacturer uses for photo processing. Unfortunately, that would also mean tying the application to a specific camera sensor and would severely limit its universal potential.

During the investigation of this problem, we have found a paper “Strengths and limitations of a uniform 3D-LUT approach for digital camera characterisation”

by research group in Technische Universität Darmstadt (46), which discusses an analogous problem. Even with a more exhaustive set of calibration colour samples, this colour correction problem cannot be resolved easily.

This problem is unexpectedly complex and is the reason this thesis did not manage to achieve its goals in full extent – including the ability to calibrate and use the application under multiple different illuminants. Further research will required to find a more accurate algorithm for camera colour space to sRGB transformation and to make this part of the application reliable and dependable in professional use.

The reason we got caught cold by this is that we assumed normal camera calibration routines to be accurate enough to achieve what we need. Which they are apparently not: what is sufficient to get a reasonable looking photograph turns out to be still considerably below our demands in terms of required colorimetric accuracy. We also under-estimated how non-linear the colour distortions are that have to be compensated during such a calibration. It should definitely be possible to find a high precision calibration technique that accurately deals with these non-linear distortions: but finding such a transform turned out to be beyond the scope of this master's thesis, which instead concentrated on proposing and showcasing an entire novel workflow for mobile applications, and getting the prototype very close to what would be needed for a commercial application in terms of performance. We deem the calibration issues to be the only thing that stands between our prototype and actual commercial usability.

3.4.2. Lighting conditions consistency

We have recognised that the lighting conditions significantly impact the accuracy of the colour matching algorithm. Even slight differences between the calibration and picking phase illuminance may cause significant degradation of the result accuracy.

This is an inherent problem of the fact that the application is meant to be used in an uncontrolled environment. A solution would be to allow providing ColorChecker® calibration and colour picking functionality in the same captured photo. We decided to use separate calibration and colour picking phase to provide more flexibility in use, but a joint solution would be beneficial as an alternative mode of the application.

3.4.3. ColorChecker® Digital SG recognition

The ColorChecker® recognition algorithm was initially built specifically for ColorChecker® Passport. To be able to visualise the camera colour space better, we decided to introduce experimental support for ColorChecker® Digital SG, which has not been extensively tested and tends to misalign the colour patches during calibration if not positioned directly in parallel with the camera sensor and the captured photo is not sharp enough. Due to this limitation, calibration with this kind of checker is not very user-friendly and may result in multiple failed attempts before calibration is successful. Implementing support for multiple colour charts was not in the scope of the thesis, so this issue is included for completeness.

3.5. Potential improvements

3.5.1. Support for multiple illuminants

The application could be improved to allow picking a colour under different illuminants in addition to daylight. Two possible approaches for this problem are available.

An easier solution would require choosing the illuminant type before calibration and including the sRGB coordinates for the ColorChecker® patches and colour atlas samples under this illuminant as the source data for the application. All these data are available in ART.

A more sophisticated solution would be to provide a second calibration for the "scene" in which the user currently is. Then judging by the difference between the daylight and scene calibration, the current illuminant type could be approximated and used to identify and pick colours.

3.5.2. Capturing with flashlight

A way to improve colour matching precision would involve capturing one photo with and one photo without the flashlight integrated into the mobile device. These two photos would allow to judge the reflectiveness of the captured surface and adjust calculations appropriately.

3.5.3. Overexposure guarding

The application could guard against photo overexposure automatically by checking the colour values of the calibrated photo of the ColorChecker®, especially

using the white patch. In case it detects overexposure (a high concentration of raw values near the maximum value of a channel in a single location), the application could discard the calibration, inform the user and adjust exposure automatically for the next calibration attempt.

3.5.4. Raw photo preview

In some cases (as illustrated in Section 3.2.4), the processed photo preview does not adequately show subtle light reflections to the user. These reflections influence the accuracy of the picked colour. It would be beneficial to allow the users to switch between a preview of the processed photo and preview of the raw photo. This way, similar discrepancies could be easier to discover.

3.5.5. Improved ColorChecker® recognition

The ColorChecker® recognition algorithm works well in case the device is well-positioned and visible to the camera, but in case it is too far away or is skewed in space, the colour patches may be mispositioned because of the imprecision. Such a limitation could be overcome by taking the exact shape of the colour patches into account and not depending on it being near-square.

Conclusion

In the course of development of the Colour Picker mobile application, we encountered several interesting problems.

We needed to find a reliable way to locate the ColorChecker® colour patches in a photo to properly calibrate for the camera colour space. Utilising the built-in platform APIs for this was not possible as they did not offer required precision. Using OpenCV based solution was successful, but required creating a layer of indirection in the form of a native Objective-C library which communicates with the C#-based Xamarin.iOS application. The algorithm works reliably for the ColorChecker® Passport device and can be further customised to support more colour charts as required.

Picking a colour from a real-world photo requires a transformation from the camera space to sRGB colour space. Such conversion factors must be based on the information gathered from the calibrated colour patches of ColorChecker® device. The main challenge was the fact that the iPhone camera colour space is non-linear and unevenly distributed. Hence, the factors may vary significantly. For each picked colour, we needed to find a corresponding set of ColorChecker® samples to use when calculating the transformation factors.

We perform the final step of searching for matches in the colour atlases in the CIE LAB colour space, where the distances between colours can be calculated using CIE Delta E measure, which is in line with the perceived colour distance of the human eye.

The algorithm used for camera colour space to sRGB transformation is unfortunately not sufficiently accurate yet for arbitrary colour samples to make the application fully dependable. Further development will be required to make the Colour Picker application a reliable tool for professionals who work with colour atlases and want to enhance the mobility aspect of their daily work.

Bibliography

1. **Zikmund, Martin.** A Colour Matching Application for Mobile Devices. *Thesis repository.* [Online] 15 06 2015. https://is.cuni.cz/webapps/zzp/detail/162870/39052602/?lang=en. 162870.

2. **Wyszecki, Günther and Stiles, W. S.** *Color Science: Concepts and Methods, Quantitative Data and Formulae.* : Wiley-Interscience, 1982. 978-0471399186.

3. **BabelColor.** The ColorChecker pages. *BabelColor.* [Online] BabelColor. http://www.babelcolor.com/colorchecker.htm.

4. **ThoughtCo.** What is the Definition of Color in Art? *ThoughtCo.* [Online] ThoughtCo. https://www.thoughtco.com/definition-of-color-in-art-182429.

5. **Choudhury, Asim Kumar Roy.** *Principles of Colour and Appearance Measurement: Volume 2: Visual Measurement of Colour, Colour Comparison and Management.* s.l. : Woodhead Publishing, 2014. 9781782423881.

6. **Stone, Maureen C.** *A Field Guide to Digital Color.* Canada : A K Peters, Ltd., 2003. 1-56881-161-6.

7. **Kruusamägi, Ivo.** Cone cell image. *Wikipedia.* [Online] March 18, 2010. [Cited: March 12, 2015.] http://commons.wikimedia.org/wiki/File:Cone_cell_en.png.

8. **Zhao, Susan.** Wavelength Of Maximum Human Visual Sensitivity. *The Physics Factbook.* [Online] 2007. https://hypertextbook.com/facts/2007/SusanZhao.shtml.

9. **Vanessaezekowitz.** Simplified human cone response curves. *Wikimedia Commons.* [Online] https://commons.wikimedia.org/wiki/File:Cones_SMJ2_E.svg.

10. **X-Rite.** Additive versus Subtractive Color Models. *X-Rite.* [Online] https://www.xrite.com/blog/additive-subtractive-color-models.

11. **Athabasca University.** File:Subtractive-Additive-Colour-Mixing.jpg. *Wikimedia Commons.* [Online] Athabasca University, 04 02 2013. https://commons.wikimedia.org/wiki/File:Subtractive-Additive-Colour-Mixing.jpg.

12. **The Free Dictionary.** Colourimeter. *The Free Dictionary.* [Online] https://www.thefreedictionary.com/Colourimeter.

13. **Z22.** File:Spyder4Elite calibrating.JPG. *Wikimedia Commons.* [Online] 17 08 2014. https://commons.wikimedia.org/wiki/File:Spyder4Elite_calibrating.JPG.

14. **ArcSoft.** What is Color Space. *ArcSoft.* [Online] ArcSoft. http://www.arcsoft.com/topics/photostudio-darkroom/what-is-color-space.html.

15. **Sappi Fine Paper North America.** Defining and Communicating Color. *Sappi.com.* [Online] 2013. https://cdn-s3.sappi.com/s3fs-public/sappietc/Defining%20and%20Communicating%20Color.pdf.

16. **Adobe Systems Incorporated.** CIELAB - Color Models - Technical Guides. *Adobe Technical Guides.* [Online] 2000. [Cited: 17 April 2015.] http://dba.med.sc.edu/price/irf/Adobe_tg/models/cielab.html.

17. **World Wide Web Consortium, Hewlett-Packard, Microsoft.** A Standard Default Color Space for the Internet - sRGB. *World Wide Web Consortium.* [Online] 5 November 1996. [Cited: 15 April 2015.] http://www.w3.org/Graphics/Color/sRGB.html.

18. **W3C.** A Standard Default Color Space for the Internet - sRGB. *W3C.* [Online] https://www.w3.org/Graphics/Color/sRGB.html.

19. **Schuessler, Zachary.** Delta E 101. *zschuessler.github.io.* [Online] http://zschuessler.github.io/DeltaE/learn/.

20. **X-Rite.** X-Rite Photo. *ColorCheckers for Photo.* [Online] X-Rite. https://xritephoto.com/ph_product_overview.aspx?CatID=154.

21. **Braun, Christoph.** X-rite color checker. *Wikimedia Commons.* [Online] https://commons.wikimedia.org/wiki/File:X-rite_color_checker,_SahiFa_Braunschweig,_AP3Q0026_edit.jpg.

22. **X-Rite.** ColorChecker Classic. *X-Rite Photo & Video.* [Online] X-Rite. https://xritephoto.com/ph_product_overview.aspx?id=1192&catid=154.

23. **Gustafsson, Eje.** X-Rite ColorChecker Passport. *Flickr.* [Online] https://www.flickr.com/photos/macahanc6r/6809577224/.

24. **X-Rite.** ColorChecker Digital SG. *X-Rite Photo & Video.* [Online] X-Rite. https://xritephoto.com/ph_product_overview.aspx?ID=938&catid=154.

25. —. ColorChecker® Digital SG. *X-Rite.* [Online] https://www.xrite.com/service-support/product-support/calibration-solutions/colorchecker-digital-sg.

26. —. Spectrolino product page. *X-Rite.* [Online] https://www.xrite.com/service-support/product-support/portable-spectrophotometers/spectrolino.

27. **Visan, Eugen.** Ccd digital camera. *Pixabay.* [Online] 11 8 2015. https://pixabay.com/photos/ccd-digital-camera-sensor-photo-880363/.

28. **Filya1.** Matrixw.jpg. *Wikimedia Commons.* [Online] https://commons.wikimedia.org/wiki/File:Matrixw.jpg.

29. **Jirsa, Pye.** RAW vs JPEG. *SLR Lounge.* [Online] https://www.slrlounge.com/workshop/dynamic-range-and-raw-vs-jpeg/.

30. **RenderingPipeline.** A look at the Bayer Pattern. *RenderingPipeline.* [Online] http://renderingpipeline.com/2013/04/a-look-at-the-bayer-pattern/.

31. **Cburnett.** Bayer pattern on sensor profile.svg. *Wikimedia Commons.* [Online] https://commons.wikimedia.org/wiki/File:Bayer_pattern_on_sensor_profile.svg.

32. **Carnegie Mellon University.** kd-Trees. *Carnegie Mellon University - School of Computer Science.* [Online] https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf.

33. **Steven Skiena.** Kd-Trees. *Stony Brook Algorithm Repository.* [Online] http://algorist.com/problems/Kd-Trees.html.

34. **OpenCV.** About. *OpenCV.* [Online] https://opencv.org/about/.

35. **Xamarin Inc.** Frequently Asked Questions. *Xamarin.* [Online] 2015. [Cited: 20 April 2015.] http://xamarin.com/faq.

36. **EmguCV.** EmguCv. *EmguCV.* [Online] http://www.emgu.com/wiki/index.php/Main_Page.

37. **jorgemht.** MVVM pattern. *GitHub.* [Online] https://github.com/jorgemht/Xamarin/wiki/MVVM-pattern.

38. **Community project.** MvvmCross - GitHub. *GitHub.* [Online] 2015. https://github.com/MvvmCross.

39. **Hirakawa, Keigho.** CCFind.m. *Intelligent Systems Laboratory - University of Dayton.* [Online] http://issl.udayton.edu/index.php/research/ccfind/.

40. **Lonien, Wolfgang.** X-Rite ColorChecker Passport (detail). *Wikimedia Commons.* [Online] https://commons.wikimedia.org/wiki/File:X-Rite_ColorChecker_Passport_(detail).jpg.

41. **OpenCV.** samples/cpp/squares.cpp. *OpenCV documentation.* [Online] https://docs.opencv.org/trunk/db/d00/samples_2cpp_2squares_8cpp-example.html.

42. **Bourke, Paul.** PLY - Polygon File Format. *PaulBourke.net.* [Online] http://paulbourke.net/dataformats/ply/.

43. **Math.NET.** Math.NET Numerics. *Math.NET Numerics.* [Online] https://numerics.mathdotnet.com/.

44. **Community.** KdTree. *Github.com.* [Online] https://github.com/codeandcats/KdTree.

45. **Apple Corporation.** App Store Review Guidelines. *Apple Developer.* [Online] Apple Corporation. https://developer.apple.com/app-store/review/guidelines/.

46. **Fisher, S., et al.** Strengths and limitations of a uniform 3D-LUT approach for digital camera characterization. [Online] 2016. https://doi.org/10.2352/ISSN.2169-2629.2017.32.315.

List of Figures

List of code samples

List of Abbreviations

API – Application Programming Interface

ART – Advanced Rendering Toolkit

CCD – semiconductor charge coupled device

CFA – colour filter array

CIE – International Commission on Illumination – abbreviation from French from French *Commission internationale de l'éclairage*

CIEDE – CIE Delta E (variants CIEDE76, CIEDE94, CIEDE2000)

CIE LAB – same as LAB

CMOS – complementary metal-oxide-semiconductor

IDE – integrated development environment

IoC – Inversion of Control

LAB – *lightness* and a and *b* colour system

LINQ – Language Integrated Query

MVVM – Model-View-ViewModel pattern

MVC – Model-View-Controller pattern

MVP – Model-View-Presenter pattern

OS – operating system

PCL – Portable Class Library

PLY – Polygon File Format

RGB – Red Green Blue colour system

SDK – software development kit

UI – User Interface

W3C – World Wide Web Consortium

WPF – Windows Presentation Foundation

XAML – Extensible Application Markup Language

Attachments

This thesis includes an accompanying compact disc which contains the project files required to compile and run the Colour Picker mobile application. *Code.zip* archive contains all project files and dependencies.

The folder `Documentation` contains documentation files:

- `Developer instructions.pdf` - describes prerequisites, project setup and hardware and software requirements.
- `User manual.pdf` – provides simple guidelines on using the application.
- `Thesis.pdf` – text of this thesis in PDF format