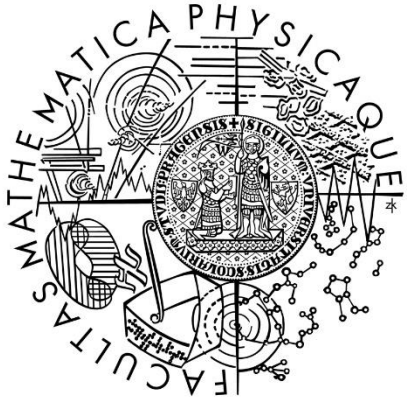
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Zikmund

A Colour Matching Application for Mobile Devices

Department of Software and Computer Science Education

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

Study programme: Computer Science (B1801)

Specialization: Programming and Software Systems (1801R048)

Prague 2015

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

I declare that I carried out this bachelor 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 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date May 22, 2015 signature

Název práce: Mobilní aplikace pro hledání nejbližších barev

Autor: Martin Zikmund

Katedra / Ústav: Katedra softwaru a výuky informatiky

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

Abstrakt: Tato práce popisuje mobilní aplikaci, která umožňuje vyhledávání nejblíže podobných barev v průmyslově užívaných barevných atlasech. Cílem funkcionality aplikace není přesné zobrazení jednotlivých barev v atlasech (pouze v mezích možností zobrazovací technologie mobilních zařízení), ale nabízí pro zadanou barvu konkrétního barevného atlasu jednoduchou a intuitivní cestu pro nalezení nejbližší podobné barvy v jiném barevném systému. Tato funkčnost, která není nikde jinde dostupná, je umožněna spektrálními měřeními tisíci vzorků barevných atlasů. Tento unikátní zdroj dat umožňuje existenci této aplikace, která může najít široké uplatnění pro designéry, kteří používají barevné atlasy při své práci.

Klíčová slova: barvy, barevné atlasy, mobilní aplikace, multiplatformní vývoj

Title: A Colour Matching Application for Mobile Devices

Author: Martin Zikmund

Department / Institute: Department of Software and Computer Science Education

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

Abstract: A mobile application that provides matching functionality between different industrially used colour atlases is presented. The focus of the application is not accurate display of the colours in the atlases, although this is attempted to within the limits of mobile device display technology. Rather, for a given entry into one particular colour atlas, an easy and intuitive way of choosing the closest possible colour from some other colour system is provided. This sort of functionality, which is not available elsewhere, is based on spectral measurements of several thousand colour atlas samples that are available to the app developer. This unique dataset makes such an application possible, which has the potential to be a highly useful niche tool for designers who work with such industrial colour atlas systems.

Keywords: colours, colour atlases, mobile application, cross-platform development

Contents

Introduction

This bachelor thesis aims to provide a useful tool that enables designers and professionals who require the usage of several colour atlases in their line of work to find the closest colours across these atlases in terms of colour difference.

Because the colour atlases are produced by various manufacturers and are usually quite expensive, there is not much cooperation between their makers. Therefore, when the user needs to find similar colours when working with more than one atlas, the search for the matching colour is left to his own abilities and resources.

Finding a good match is not a simple process, however. Because the atlases are designed to be viewed and used under clear daylight, it is necessary to achieve the right light conditions to be able to distinguish the colours. For accurate results, spectral measurements are required.

To address the stated problem, this thesis describes a process of building a tool that will greatly help with this process. This tool will be preloaded with best colour matches across the atlases. Spectral measurements, which are the basis for the dataset used by the application, are provided as one of the content resources of the Advanced Rendering Toolkit.

The tool will be able to offer colour matching across four colour atlases: Munsell, NCS, RAL and RAL Design, as the most universally used ones. The underlying database can be easily expanded however and that means the tool can offer more options in the future if necessary.

The output of this thesis is a mobile application for modern smartphones and tablets. The mobile device market is on the rise and for this reason the application will be easily available to a large number of users. To be able to maximize its potential, the tool is built as a cross-platform application for all three major operating systems on the current market, including Microsoft Windows 8.1 and newer, Google Android 4 and newer and Apple iOS 7 and newer. The thesis will focus on the process of building such cross-platform mobile applications and the challenges associated.

An important goal for this endeavour is to ensure the source code can be easily shared across all the target operating system – specifically so that the main part of the application logic is fully reused, requiring platform-specific implementation of just the

user interface (which is required to ensure it is intuitive to use and fits well with the rest of the target operating system).

The thesis will first describe the theoretical background of colour theory, colour atlases and spectral measurements. Then it will offer some insight into the source of data provided to make the development of this application possible and comment on vital concepts for cross-platform mobile development.

The main part of the thesis will describe in more detail cross-platform mobile development, its specifics, required considerations and everything the decision to support multiple mobile platforms with the same code base will affect in the overall design and architecture of the project. This will include a description of the used Model-View-ViewModel architectural pattern, which will be of great assistance to keep the code clean and succinct, will ensure the optimal separation of application logic and the user interface is achieved and this setup will also be allow future additions and changes to the project.

The last chapter will present the results – mobile application on the three mobile platforms and examples of colour matching that the application provides.

1. Background

This chapter presents basic information from the topics of colour theory and colour atlases. It also lists general rules and terms concerning cross-platform development for mobile devices.

1.1. Colour theory

The colour theory science field covers a wide range of different topics. Because the theoretical background required for this thesis consists mainly of colour measurements and colour reproduction, only these specific topics are be covered.

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." (1) The cone cells in the retina encode light into signals, that are then transmitted to the brain and perception of colour is there created by interpreting the signals. Figure 1 shows the structure of a cone cell. This is the foundation for colour measurement, and for the representation of colour as triple of red, green, and blue values. (2)

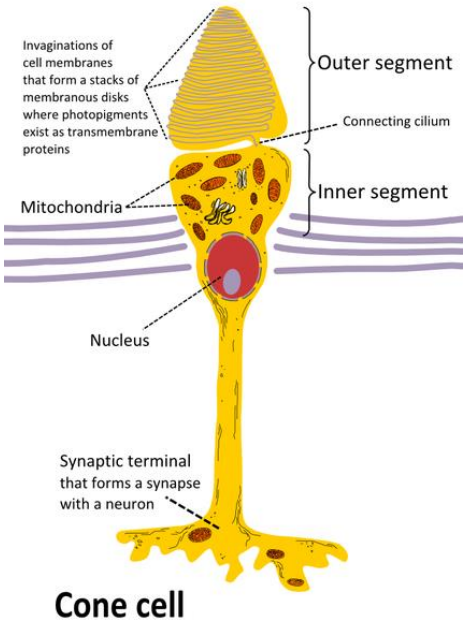


Figure 1: Cone cell structure
Source: (3)

Light can be described as a function of power versus wave length. This is called *a spectral distribution* function or *spectrum*. Coloured light contains wavelengths in the range of 370–730 nanometres. (2)

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.

There are three types of cones that respond to different wavelengths of light – long, medium and short. The information in the spectral distribution is reduced to three values, one for each type of cone and these values then represent the resulting colour.

1.1.3. Digital colour

To represent colour digitally we need to encode colour as RGB pixels which are represented as three numbers. The accuracy of colour reproduced by a digital display is limited by the properties of the display itself. Professionals (for example designers and architects) use colorimeters to measure digital colour produced by their work devices and calibrate their displays to achieve higher colour accuracy.

1.1.4. L*a*b* colour space

Defined by the International Commission on Illumination (CIE from French *Commission internationale de l'éclairage*), the L*a*b* colour space (also known as CIELAB) describes colours in terms of three axes – L*, a*, and b*. (4)

The lightness (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. (4) There are no maximum or minimum values for a* and b*, although they are usually ranging from -128 to +127.

L*a*b* colour space includes all perceivable colours and is device independent.

1.1.5. 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 and is used in most modern electronic devices, printers and operating systems. (5)

sRGB utilizes a simple and robust device independent colour definition. Each of its components (R for red, G for green and B for blue colours) are represented as real numbers in the range from 0 to 1. For digital colour representation purposes this range is then limited usually to 256 different values and the colour components are then ranging from 0 to 255.

Because sRGB operates with a limited number of colours it is a subset of the CIELAB colour space.

1.2. Colour atlases

Colour atlases are usually physical representations of theoretical systems. These usually take a form of a box or book containing a large number of colour samples. These samples offer an intuitive and tangible representation of a given colour.

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

1.2.1. Usage

To make sure that the atlas can be easily navigated and referred to, each colour atlas has a coordinate system that is as intuitive as possible, and which can be easily used to find specific colours within the atlas. Coordinates are usually specific for each atlas, but they generally divide the offered colour space to some specific ranges by some underlying logic, so that similar colours the share similar coordinates.

Browsing through the atlas, the user can search for colours and then use the coordinates to precisely communicate the colour to other users of the atlas.

Colour atlases are used in many industrial design scenarios. Professionals can use them to select accurately select colours and share them with supply chains, clients or colleagues while ensuring high consistency. Colour atlases can also be used as a means for theoretical research and education.

1.2.2. Munsell Color System

Munsell Color was originally created by the artist, researcher and Professor Albert Henry Munsell. Professor Munsell worked in the field of colour theory and wrote several books on the subject. His research was influential in evolving colour science in the 20th century. (6)

Figure 2: Munsell Book of Color
Source: (7)

"Munsell Color Theory is based on a three-dimensional model in which each color is comprised of three attributes of hue (color itself), value (lightness/darkness) and chroma (color saturation or brilliance)." (8) This notation is also referred to as HVC.

The Munsell Book of Color is a colour atlas based on the Munsell Color System. This atlas is offered and sold in several different forms.

1.2.3. Natural Colour System

"The Natural Colour System (NCS) standard colour atlas is based on the Natural Colour System®©, which was developed through decades of research. NCS is used globally for ensuring quality and defining communicating colours cross-industry." (9)

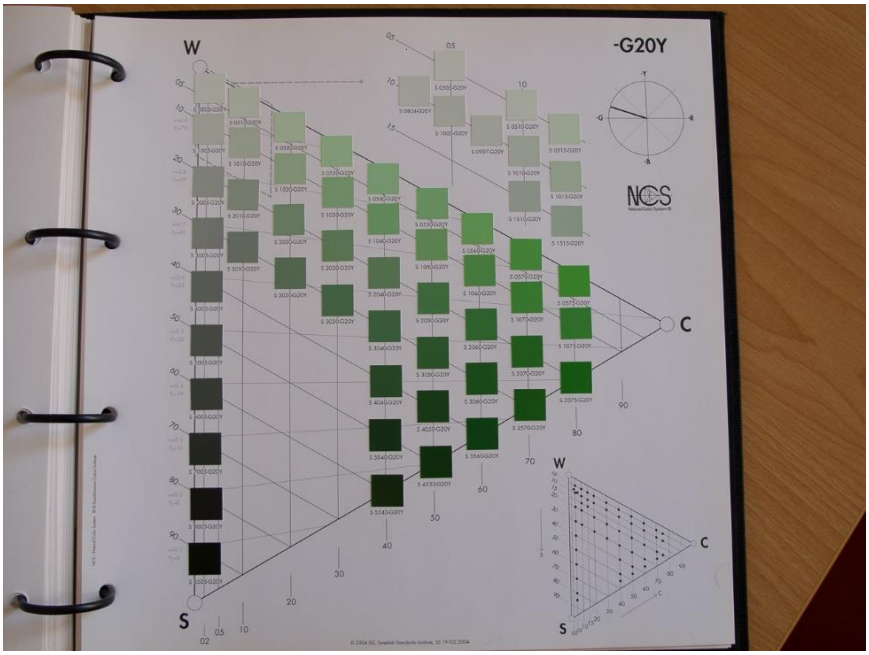


Figure 3: Hue -G20Y in the NCS colour atlas

The three-dimensional model used in NCS is based on elementary colours of yellow, red, blue and green and the nuance of the colour. Nuance is determined by the blackness (darkness) and chromaticness of the colour. (10)

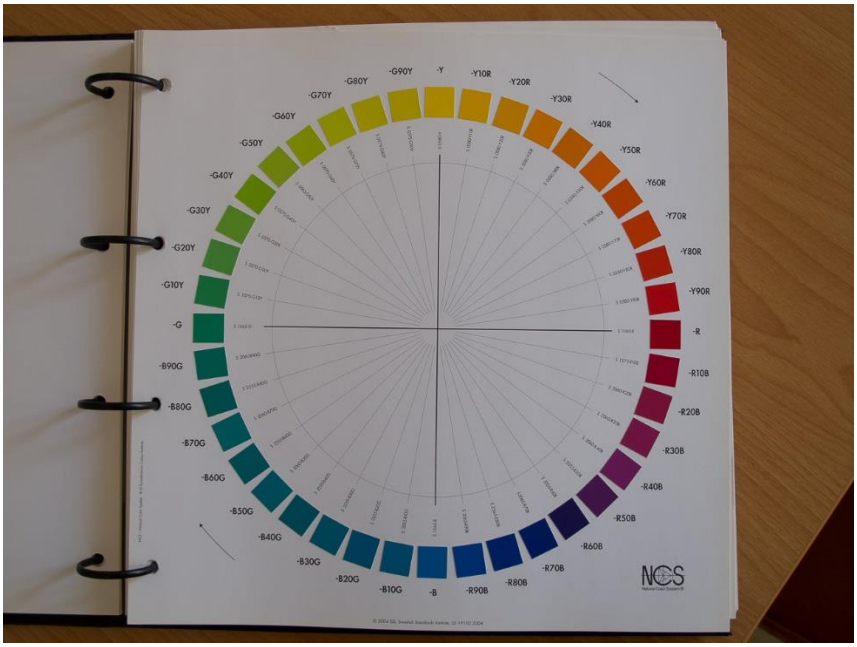


Figure 4: Various hues in the NCS colour atlas

The NCS colour notation starts with one letter. Four our purposes “S” which means the colour is included in the standard colour collection (the one used in this thesis). The following two pairs of numbers describe the percentage of blackness and chromaticness, respectively. The last four characters has two letters on each side, which describe the base natural colours, contained in the colour and the number

between describes the relative closeness to the second one. Individual colour samples are usually laid out on the pages of the atlas in a triangular scheme (Figure 3). The base hues are depicted as a colour circle on the first page of the atlas (Figure 4).

1.2.4. RAL & RAL Design

RAL Classic colour system was invented in 1927 by the Reichs-Ausschuß für Lieferbedingungen und Gütesicherung (State Commission for Delivery Terms and Quality Assurance) and originally consisted of only 40 colours. (11) The palette was later expanded to consist of 210 colours in 1961 (Figure 5). Each colour in RAL Classic has a four digit numeric code and additionally a name.

Figure 5: RAL Classic
Source: (12)

RAL Design (figure 5) is a newer colour matching system from 1993 and was tailored to the needs of architects, designers and advertisers. (11) The colours have no names but are numbered following a CIELAB colour space scheme. Each colour is represented by 7 digits, grouped in a triple and two pairs, representing hue, brightness and saturation.

Figure 6: RAL Design
Source: (13)

1.3. Spectral measurements

An entire scientific field called colorimetry focuses on precise colour measurement. Devices used to measure colour spectrum are called spectrometers.

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

1.4. Advanced Rendering Toolkit

ART is a Predictive Rendering research system under development at the graphics group of Charles University in Prague. ART has not been released to the public yet, but is provided for research purposes in pre-release form.

ART is used as the source of spectral measurement data. The data are 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.

1.5. Cross-platform mobile development

The largest possible audience for the colour atlas tool can be reached only by making it available on as many different mobile devices and platforms as possible. This fits in with the intended usage, as designers might want to look up colour atlas code correspondences while under way, or at a client's site.

1.5.1. Mobile devices and form-factors

The world of mobile devices has been growing extremely fast lately. Since the introduction of first smartphones we have seen growing number of new devices and device types on the worldwide computing device market.

We can usually describe each device in terms of hardware (*form-factor*) and software (*operating system*) aspects.

Mobile devices are usually seen as computing devices that can be easily carried by a human. Because this definition is quite general, it includes a wide variety of devices like smartphones, tablets, notebooks and wearables. Each device type has its specifics – different input methods (touch, digitizer pen, touchpad/keyboard and mouse, or any combination of these), different screen sizes (from small to large scale screens, in many different resolutions) and different use cases (home, business, industrial, all-day usage).

The full potential of the hardware is then exploited by the operating system that is running on the device. There are many different operating systems used in the mobile device world, but the majority of the market consists of device running Google Android, Apple iOS and Microsoft Windows operating systems.

Each operating system spans a different range of devices. Apple iOS targets smartphones and tablets built by Apple. Google Android is an open source operating system, which means it is usually modified by device manufacturers and runs on devices ranging from wearables, through smartphones and tablets to other devices like Chromebooks and TVs. Microsoft Windows platform is currently available on smartphones and tablets and PCs, but is expanding to the Xbox home console market and other targets like holographic computing with Microsoft HoloLens with the upcoming major release.

One of the important consequences of the operating system's ability to run on different form-factors is the fact, that the mobile application can actually be used even on devices that do not fit the definition of a mobile device – desktop PCs for example.

1.5.2. Developing for multiple platforms

Traditional approach for application development is to build an application directly and natively for a concrete platform and operating system. This is of course a

valid solution because each platform has its specifics, requires different development tools, different user interface design and metaphors. It is also the easiest solution that does not directly force any architectural decisions for the applications. However, it poses multiple challenges the development team has to face. First of all, for each of the platforms a separate application is created and this means the investment required for development and maintenance (updating the published applications with new features and bug fixes) is growing proportionally to the number of supported platforms. Even more so – the potential for issues is much larger, because each application has completely separate codebase and therefore presents a room for human-error and unintentional differences in behaviour which can be discovered only later in the development process resulting in higher development costs.

Mitigating the disadvantages of the traditional "one platform – one application" solution requires us to explore the potential of cross-platform development.

Cross-platform development offers an opportunity to develop and build applications for multiple operating systems at the same time while reusing a certain amount of code. This can be achieved in many ways and there is a growing number of cross-platform development tools available for all mainstream operating systems.

The clear advantage of cross-platform development is the fact that we can actually avoid most of the issues of the traditional approach. We can share code and application logic. We can develop all mobile applications at once and in one environment while dropping the time and human investments burdening the single platform development strategy. Errors and bugs in code appear in one place and are more easily discoverable and fixable. Also new changes and features of the application can be added to all supported platforms at once. The choice of programming language is also a factor that can be advantageous with cross-platform development. We are building for all platforms at once and we can use a single programming language to write most of the application logic. It eliminates the need for a larger number of developer teams working separately on an app for each platform and enables easier collaboration.

Cross-platform development has still a few negative consequences. It requires a well thought out project architecture and design that has to be decided on upfront in the project development cycle. It is also bound to the usage of a cross-platform

development tool and is closely relying on its availability, support and capabilities. When a platform introduces new features we want to use, we need to wait for their inclusion in the cross-platform tool of our choice. Also in many cases we cannot directly work with the native code of the platform (some cross-platform tools may not offer this option) and are even further limited by what the tool has to offer.

The most prominent cross-platform development tools used today are Apache Cordova, PhoneGap and Xamarin as well as the usage traditional web apps.

1.5.3. Application architectural patterns

Software architectural pattern help define the basic characteristics and behaviour of an application. (14) Because of the variety of different solutions, architectural patterns are not definite – there are always various patterns that can match a certain problem and the developer needs to decide, which concrete solution meets his needs in the best possible way.

The application programming interface (API), user interface design and programming language supported by an operating system highly influence the architectural patterns that are applicable in application development scenarios.

Basic applications with low demands for maintainability and future extensibility can also be built without any particular architecture in place. This solution is not viable for team-based projects and cross-platform development. It also causes tight dependencies between the user interface and application logic.

Mobile applications usually use Model-view-presenter (MVP), Model-view-controller (MVC) and Model-View-ViewModel patterns to create the presentation layer architecture. Each pattern has its specific behaviour and components.

1.5.4. Portable class library

The portable class library (PCL) is a concept specific to cross-platform development in C# in Visual Studio IDE. PCL is a project type that acts as a common library referenced by mobile application projects written for multiple different platforms.

The developer can set the PCL to be compatible with certain subset of target platforms. Each set of platforms has its specific restrictions so that the code included

in the library is in fact compatible with all selected platforms. The selected subset is then referred to as a “target framework profile”.

1.5.5. Data source

Data represent the core of most applications and systems built today. Mobile applications have multiple different methods for acquiring and storing data. The main differentiation factor is its locality – data can be stored locally on the mobile device itself or it can be stored remotely and accessed via a network.

Remote data are usually located on web servers or cloud. Communication with these endpoints is performed according to different protocols like HTTP/HTTPS, FTP and others. The mobile device sends a request to the remote server and receives a response that contains the data requested. Data can be stored in relational databases, NoSQL databases, as files or can be computed or generated from multiple sources in reaction to a specific request.

Locally stored data are stored in the memory storage of the device. They are accessed directly using the file manipulation interface that is provided by the operating system running on the device. The application developer can store application data as files that are safely isolated from other applications in the private application storage or in public locations (on some platforms). It is possible to include file-based local relational databases for easier data querying and faster access to structured data.

2. Proposed method

2.1. Finding good matches

Colour distance is the main metric that influences the quality of matches we find. The most commonly used colour distance metric created by CIE is Delta E (or ΔE^*_{ab}), where "E" stands for "Empfindung", German for "sensation".

The original mathematical formula (labelled as CIE76) is stated as follows (15):

$$\Delta E^*_{ab} = \sqrt{(L^*_2 - L^*_1)^2 + (a^*_2 - a^*_1)^2 + (b^*_2 - b^*_1)^2}$$

This simple Euclidean distance approach turned out to be insufficient in some scenarios, because it does not factor in perceptual non-uniformities. Therefore the formula for Delta E was evolved and refined twice throughout the years since its inception in 1976. The newest definition was created in 2000 and is called CIEDE2000. (15)

The formula's input are two colours we want to compare in LAB colour coordinates. The output is a non-negative number. The smaller the result, the smaller is the difference between the given colours.

For our purposes, the Advanced Rendering Toolkit provides a function for calculation of Delta E values. We can use this function to calculate distances between each pair of colours across all atlases and take the best matches as part of the dataset used in the mobile applications. Because the atlases can contain several colours that are very close, we decided to always include the three best matches.

2.2. User interface considerations

User interface (UI) of our mobile application is an important part of the user experience. The user needs to be able to take advantage of the offered features comfortably and easily in a way that fits naturally the device he is using, the ecosystem and operating system running on the device.

In this section we describe the decision process that defined the input controls for the colour being matched and show the influence different input methods and screen sizes have on the final user interface design.

2.2.1. Intuitive colour selection

Each colour atlas consist of a large number of colour samples, which can be identified by their unique coordinates. The user knows the coordinates of the colour he wants to work with and needs to be able to quickly select that particular colour in in the user interface of the app.

The easiest option would be to allow just direct text input. This is sadly prone to errors and is very inconvenient on mobile devices with touch based keyboard input.

To make the input more touch-friendly, a selection drop-down list (usually called combo-box or spinner on most platforms) can be used. This method is much more practical, but has a different drawback – for colour atlases with many samples, scrolling through the list can be a very daunting task.

The solution we proposed for the app is an improved version of the combo-box scenario. For a selected input colour atlas, the app will present several combo-boxes, each of them for one of the coordinates. The user can then select each coordinate one-by-one from left to right. To make this even more intuitive, after each selection, the following lists to the right will be automatically filtered to contain only the valid values for the "prefix" the user has already chosen.

2.2.2. Screen sizes

The app is intended to be used on devices with various screen sizes and display resolutions. Therefore it is vital to consider this while designing the user interface.

We decided that the app should support two different main modes of operation – portrait and landscape.

For portrait mode, the input area for colour selection will cover the upper part of the screen and the results area will be displayed at the bottom.

For landscape view, the input area will move to the left hand side of the screen and the results will be shown on the right half of the screen, supporting the full height of the screen.

Of course, to make sure all results can be displayed, the result list is scrollable.

2.2.3. Input methods

The colour matching app has to take into account the forms of input that can be used on each platform we target.

It is easy to approach this problem for tablets and smartphones, which usually feature a touch screen only.

In the Android and Windows device families we can encounter digitizer pen. By using native controls offered by the system, we automatically get the required support for this input out of the box.

Finally, for Windows platform the user can use mouse and keyboard. Mouse works very similarly to the touch input, so there are no special considerations needed in this case. To support keyboard, we have ensure the app responds to the TAB key to jump through active elements.

2.3. Cross-platform considerations

To build a well-designed fully cross-platform mobile application, we have to consider several important points and make some up-front decisions that will influence both the development process and the future extensibility of the application.

Because there are several different techniques for building mobile applications that span more than one operating system, we need to decide on their advantages and disadvantages for the concrete application we are building and the use the acquired knowledge to our advantage throughout the development.

2.3.1. Targeting multiple operating systems

As we mentioned previously, there are multiple different mobile operating systems. Each operating system has its specifics – different use cases, different UI, different programming language and framework support, different application programming interface (API) and different supported hardware capabilities.

Before the development starts, we need to decide which mobile platforms we are going to support. This will in turn affect the choice of programming languages and tools available to us.

The most prominent platforms on today's market are Google Android, Apple iOS and Microsoft Windows. All these systems have different default programming

languages – Java on Android, Objective C/Swift on iOS and C# and Visual Basic on Windows. They also all support HTML5+JavaScript applications and C++ development (which is mainly targeted on games) as secondary options.

Our application will target these three operating systems to cover the largest possible portion of the mobile device market and potential users.

For Android we will support devices with system version 4 and higher. This represents most devices active and in use today. Older versions have large API differences and the recommended practice is to rather target the more recent releases of the operating system.

Apple iOS 7 and newer is supported for Apple mobile devices.

Windows 8.1 and Windows Phone 8.1 are supported for the Windows version of our application. The upcoming release of Windows 10 should be fully compatible after its final release, but there are some issues currently, which we will describe in the 3.4. (Known issues) section of this thesis.

2.3.2. Choosing the programming language and tools

For the three chosen operating systems we need to select a programming language that will be supported and easily integrated into all of them.

First option is to essentially build a HTML5 website offering the intended functionality. This can be done as a completely custom solution or also with tool like jQuery Mobile (16).

This is the simplest solution that can satisfy only the most basic scenarios. It restricts the access to device APIs only to what HTML5 can offer. This surface is even more restricted by the fact that all HTML5 features are not yet fully supported in most mobile web browsers.

Resulting user experience is not native and is radically different from the built-in applications, because it must be run within the boundaries of the web browser (including the fact that the browser controls and “chrome” will be present at all times). All operating systems are usually served the same HTML mark-up, the same styling and the same interactivity. This can be partially improved on server-side through the by analysing the user-agent string sent by the browser with each request and by client-

side JavaScript libraries like Modernizr (17), but compared to direct access to device information are these only partial and less reliable solutions. Most web applications primarily focus on offering a universal touch-friendly experience.

An improvement over the web application approach is a hybrid HTML5 and JavaScript solution through a mobile development framework like PhoneGap (18) or Apache Cordova (19). The application we are building is a local web application that is packaged as a native application for each platform and actually runs in an embedded web browser control.

First of all, UI development is similarly to the web applications option done purely in HTML5 and CSS. For this reason, we can manually imitate a native UI of a target operating system, but are essentially giving up on a fully user-friendly experience the users know. To achieve a native-like experience, the UI would have to be completely re-implemented for each operating system. With access to device information it is much more realistic compared to the classic web application, but still causes unnecessary distraction during development.

We also do not have true direct access to the API provided by the operating system itself, we have to use a subset that the framework offers. This might not be a problem for simpler scenarios, but can be limiting in more advanced cases. It also adds another layer that degrades performance when larger amount of data needs to flow between our application and the system.

Finally – JavaScript is not supported natively on any of the chosen platforms, so its own performance and support will be subject to this fact. This is not a problem for most applications, but cannot be ignored when more involved processing is required (for example video, audio or 3D graphics).

The C++ programming language is supported natively on all platforms, but poses several major disadvantages for the development.

The language itself is much more low-level than the other options we have and hence it is less friendly for programming high-level tasks like UI interaction and would also require considerable investment to simulate platform-specific behaviour that is common in apps built directly for a specific operating system. All the fidelity that C++ offers can be seen as unnecessary burden in light of the tasks we actually require for

building a task-driven mobile application without complex graphical user interface and 3D graphics.

Our choice for the colour matching tool development will be the C# programming language, which is strongly typed, very powerful and has many features that can be an advantage for us. Many third party libraries are supported and there is a large number of community based open-source projects written in this programming language.

The main development tool for C# is the Microsoft Visual Studio IDE. This environment offers extensive capabilities for each phase of the development process.

The only disadvantage of C# in regards to our project is that the language is supported natively only by Windows, but it can be brought to iOS and Android via Xamarin tools.

2.3.3. Xamarin.iOS and Xamarin.Android

The Xamarin platform (20) enables C# language based development on non-Windows platforms including iOS and Android.

The framework itself offers all the APIs and capabilities of both system transformed to a form that is familiar to C# developers. This also includes support for modern language features like asynchronous programming with the async/await keywords and Language Integrated Query (LINQ) expressions. In several cases the offered API is more complete and practical than the original, because of the advanced features offered. This is apparent mainly when comparing Objective C code with its Xamarin.iOS C# based counterpart.

With each new release of the Android and iOS operating systems, Xamarin offers an update on its tools that includes the newly added APIs and features.

“On Android, Xamarin ships a fully functional implementation of the .NET runtime, called Mono.” (21) This runtime covers all core features of C# and .NET and contains a just-in-time (JIT) compiler that transforms IL (.NET immediate language) to the machine byte code. (21)

"On iOS, Xamarin uses Mono, a fully functional implementation of the .NET runtime, to fully compile your app into a native ARM executable ahead of time (AOT)." (21)

Xamarin offers free licenses of the tools for academic purposes. These are required to be able to able to develop C# based applications for Android and iOS.

One of the main advantages of Xamarin is extensive integration into the Visual Studio IDE including features like IntelliSense for code completion and suggestions. Android user interface design is also fully supported and iOS user interface design is included partially, although it is possible to also use the native designer in Xcode IDE on Mac OS X.

Application written in C# for Xamarin.Android can be compiled and debugged directly on a Windows machine with Visual Studio. Xamarin.iOS requires a secondary Mac OS X device with Xamarin tooling and Xcode IDE preinstalled, because of the licensing limitations Apple imposes on iOS application development. The developer establishes a remote network connection via Xamarin Mac Build Host from Visual Studio and can then compile and debug Xamarin.iOS applications by initiating the debugging or building process on the Windows machine. Most debugging features are fully functional.

2.3.4. Overall project structure

The application development will be separated into three distinct projects, each of them serving a concrete purpose.

First project will be used to retrieve the source data from the ART archives. This small tool will be written in Objective C and will contain just the required information to export data to JSON files.

Second project's goal will be to generate a database that will be used as the underlying data store for the client mobile application. This project will take the previously generated JSON files and will create a ready-to-use database as the output. This project will be written in C#.

The main project will be the mobile application itself. This will consist of several subprojects. First will be the "*ColourMatch.Core*". This will contain all business logic that is not platform- or UI-specific. Then will follow projects for each

of the platforms – "*ColourMatch.Droid*" for Android, "*ColourMatch.Touch*" for iOS and three subprojects for the Windows platform ("*ColourMatch.Windows.Windows*" and "*ColourMatch.Windows.WindowsPhone*" for platform project nodes and "*ColourMatch.Windows.Shared*" for code being shared by both Windows releases), because the binary output for Windows and Windows Phone is a package that is tailored for each of the form-factors with specific package manifest file.

2.3.5. Choosing the architectural pattern

We want to build a cross-platform mobile application in a way that will minimize the amount of platform specific code and will decouple the application logic from the user interface.

For Android and iOS, the most used design patterns are MVC (Model-View-Controller) and MVP (Model-View-Presenter). This pattern achieves a good separation between the UI and business logic, but still requires significant investment to display and update the data flowing from the controller to user interface and back.

The Windows platforms offers an alternative in the form of MVVM (model-view-view model). Using this design pattern we can fully decouple the user interface from the underlying application logic.

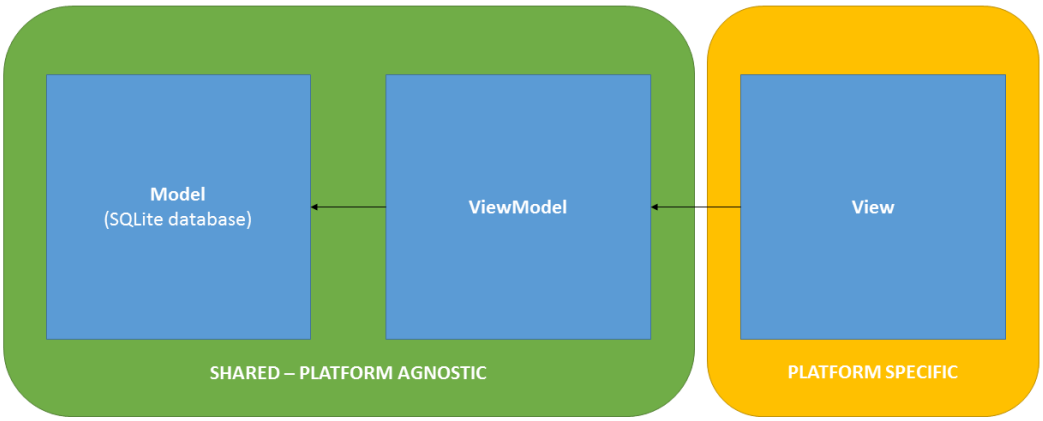


Figure 7: MVVM architecture in cross-platform environment

The main concepts of MVVM are model – data being used, view model – a middle component that offers properties and data to the UI, view – the user interface itself and data binding. An important rule lies in the fact that the view model should not have direct knowledge about the view and the model should not have direct

knowledge about any of the remaining components. View can access only the view model directly.

Data binding is a “connection” between the view and the view model, which can facilitate the flow of data in both directions without additional code required for most actions triggered by the user. The binding is set up in the user interface, usually on one of the properties of a control – *Text* of a label for example. This binding then describes the path of a property exposed by the view model and also defines if it should perform one way or two way binding. One way binding updates the property in the user interface every time the referenced property in the view model changes. Two way binding adds the reverse direction – meaning changes in the user interface update the property in the view model. These changes can be initiated by the user of the application – like input in a text field.

Key component that enables data binding is the *INotifyPropertyChanged* interface. This interface enables types to subscribe to changes of properties. Each time the type signals that a property was updated the subscribers are notified by triggering the *PropertyChanged* event.

Data binding is made even more powerful with value converters. A value converter is a class that acts as a transformation component during data binding. It is possible that the properties the binding connects together are not actually of the exact same type – for example may we have a *Boolean* property on the view model and a *Visibility* enumeration in the user interface, which has also two possible states (this example can be found in the Windows Extensible Application Markup Language (XAML) based user interfaces). Value converter class has two public methods – *Convert* and *ConvertBack*. The *Convert* method is called every time the binding receives a property change notification from the view model. The binding takes the value from the view model, passes is to the *Convert* method (along with possible parameters which can be specified as an extension of the binding) and the method returns a result that is then used in the user interface. The *ConvertBack* method works the same way in the opposite direction.

MVVM pattern also defines a role of Commands. User input in the UI can not only cause changes in properties, but can also trigger actions – a prime example being a button click or tap. “Commands provide a convenient way to represent actions or

operations that can be easily bound to controls in the UI. They encapsulate the actual code that implements the action or operation and help to keep it decoupled from its actual visual representation in the view." (22) The command classes implement the *ICommand* interface, which has two methods – *Execute* and *CanExecute*. *Execute* is called when the command should perform its associated action in response to an event or user interaction in the view. The *CanExecute* method allows or forbids the command to be invoked. Command instances are exposed in the view model as public properties, similarly to the data, although usually in a read-only manner. The user interface once again uses data binding to connect to the command. This means, that the UI controls have to be able to invoke the command. This behavior must be built-into the control or be added to it by the developer.

To enable more complex scenarios, many MVVM-based solutions use an additional component – a messenger. The view can communicate with the view model directly calling its commands or even methods, because the view always can access its associated view model directly. In the other direction this is however not possible. The view model should not have direct access to the view (that would violate the separation principle, view model should not be aware of the view), so the only way for it to communicate is via the property changes that are picked up by data binding in the user interface. When this is not sufficient, the messenger component is helpful. "This class is an implementation of an event bus, a decoupled eventing system in which the sender of the event (or message) and the recipient don't have any knowledge of each other." (23) View model can publish a message that is then received by instances that subscribe to it. This is usually implemented in a weakly referenced way so that the lifecycle of the subscribers does not depend on the lifecycle of the messenger component and the developer does usually not have to manually unsubscribe (although it is still recommended).

Because the MVVM pattern is not directly supported by Android and iOS, we will use a third party library called MvvmCross.

2.3.6. MvvmCross

The MvvmCross library (24) is an open-source project maintained on GitHub. This library is the most comprehensive cross-platform solution for the C# programming language, spanning Windows, Android, iOS and Mac operating systems.

It support reaches even beyond mobile applications to the desktop with Windows Presentation Foundation (WPF) application development or Mac OS X application development.

The library consists of a core library and several plugins and extensions that are optional. Using a high level of abstraction, the library achieved to find a common ground across all supported operating systems and application lifecycles, so that the resulting code can span all of the platforms and their differences efficiently.

MvvmCross uses the MVVM design pattern and bridges it to unsupported platforms. The concept of views is applied to the native UI for the given platform and data binding support is added along with full support of value converters.

In some cases, MvvmCross offers alternatives to native platform controls or components that are extended with MVVM related capabilities. Prime examples of this are list-based controls, which need a way to bind to a data source for their items, or buttons, which typically need to execute a command on tap or click action.

The library also offers an inversion of control (IoC) container. This is deeply integrated into the library itself and allows for constructor injection for views, view models and services. The developer can register singletons and interface implementations in both portable class library and platform projects which is essential for cross-platform development.

In addition to the fact, that MvvmCross is open source, it supports the development of third-party plugins. Adding such a plugin to an existing solution requires just adding appropriate references and a simple bootstrapping code in each of the projects.

Colour Matching applications we are building are using the core part of MvvmCross and its plugins like MvvmCross Visibility plugin (which simplifies binding of visibility-controlling properties to the UI) and Messenger plugin (which provides a way for types to notify other types about certain events in a safe and weakly referenced manner).

2.3.7. SQLite database

The mobile application needs to access the colour atlas data to find matches. The best way to store such structured information is a relational database.

The only database solution that is available for all three operating systems and is well supported is the SQLite database (25), maintained under the public domain license.

This is a light, stable and fast file-based database solution that is well tested for mobile applications.

Our application will come with a pre-filled database with all data required to search for colour matches. We will run SQL queries to retrieve the data we need.

Using the available open-source .NET sqlite-net library (26) and its portable class library extension SQLite.Net-PCL (27) we can perform most of the database facing actions in a fully cross-platform manner, sharing most of the code and even writing familiar and strongly typed LINQ expressions for our queries.

The platform-specific code will configure the target path, where the database is saved and facilitate copying of the source database from the application package to application's private storage location on the device. This has to be handled specifically for each of the operating systems, because file-storage mechanisms are significantly different.

2.4. Application development

In this section we will go through the process of development of the mobile applications and the supporting projects. We will talk about the challenges that are specific for each platform.

2.4.1. Retrieving data from ART

The data used by our application are available in the ART archives. The ART project offers many useful functions and features that make writing an exporting tool very convenient.

ART is best supported on the Mac OS X platform so the exporting tool was also written in Objective C for Mac.

First, for each colour atlas, JSON files which contain information about the colours inside the atlas were generated. For each colour we exported its unique name (with coordinates), the LAB colour value and sRGB colour approximation.

The spectral data in ART were measured using Gretag Macbeth Spectrolino spectrometer. We converted the data to CIELAB. After we have gathered information about all atlases, we created lists of best matches for each pair of atlases (bi-directionally). For each colour in one atlas we searched for three best matches in the other one (using the Delta E values, see section 2.1) and then saved the results to our JSON output file.

This process has a quadratic time complexity, but it doesn't have any impact for us, because the export is not a part of the end-user experience and needs to be performed only once, or when the source data in ART are updated. Also the number of colours in the atlases is not large enough to warrant more complex solution.

To make the usage of the data we generated as streamlined as possible, we also created an additional configuration file which describes all the atlases that were used and contains relative path links to the all JSON files.

2.4.2. Generating the database

The generated JSON data files from ART archives are to be processed using a database generation tool that is written in C# programming language. This project can be found in the main solution as "*ColourMatch.DatabaseGenerator*".

The tool is a very simple Windows Presentation Foundation application that allows the user to select the configuration JSON file and uses the provided information to gather the exported data.

The application first prepares an empty database where the results will be stored and generates the database schema. The schema we used has tables for colour atlases, individual colours and the best matches. The sqlite-net library is able to automatically create the database tables based on the given C# classes and their attribute annotations.

JSON files we use as the data source are already prepared to be used in the database. Apart from simple string-based transformations we don't need many changes to the data.

We note that the database generation project is also built with MVVM-based architecture. In this case we have decided to use another open source solution – MVVM Light Toolkit (28). The advantage is that we avoid unnecessary complexity

bound to the MvvmCross solution, which focuses on MVVM pattern in cross-platform environment. MVVM Light Toolkit is currently available primarily for the Windows platforms (including WPF), although recent development is pointing in the direction of expansion to the Xamarin.Android and Xamarin.iOS platforms.

The generated database file contains all data the app will need for its features to be functional. The solution works well for smaller number of atlases but could be a problem if the number of atlases that are used in the application grows, because the data needs to contain the best matched for each pair of the atlases. The last chapter of this thesis describes possible solutions for this problem in the section discussing possible improvements on the resulting application (3.5. Possible improvements).

2.4.3. Building the shared core project

All the main logic of our application is contained and hidden in the "*ColourMatch.Core*" project. This is where the data-layer, application services and model and view-model components of the presentation layer reside.

The Core project is a portable class library that is targeting Windows 8, Windows Phone Silverlight 8, Windows Phone 8.1, Xamarin.Android, Xamarin.iOS Classic API and Xamarin.iOS Unified API. This particular setup is target framework profile *Profile259*, which is one of the options supported by MvvmCross library.

2.4.4. Colour code data binding

To build out the colour code handling mechanism in our app we will need to find the right solution that will allow us to dynamically add and remove colour code selection lists from the user interface based on the selected colour atlas and will correctly populate the lists with data. We also need to have the data binding set in such a way, which will enable colour selection not only from the user interface, but also from code.

The MVVM pattern usually prepares the data binding along with the view being loaded. Unfortunately this is not an option in our case as the colour code selection lists are non-existent during the loading. To work around this limitation we need to use the MvvmCross capability to create bindings dynamically in code in Android and iOS platforms (for the Windows platform we use the built-in binding capabilities, although the MvvmCross solution is valid as well).

When the user or view model changes the selected input atlas an *InputColourAtlasChangedMessage* message is published by the *ColourMatchViewModel*. The *ColourMatchView* subscribes to this message and is then automatically notified.

The *InputColourAtlasChangedMessage* has a *ColourCodePartCount* property. This property tells the view how many lists need to be created and data bound. The view then creates the appropriate controls and adds them to the user interface.

To initiate the data binding, MvvmCross uses the *CreateBindingSet<View,ViewModel>* extension of the view classes. This creates an instance of *MvxFluentBindingDescriptionSet*, which can be then used to fluently define the data bindings. Both platforms differ slightly in the way the binding creation is handled, but the overall structure is very similar. After the binding is prepared, the *Apply* method is called on the binding description set to make the bindings take effect.

On Windows we directly create instances of the native *Binding* class, set up the required properties and then set the binding using the *FrameworkElement's SetBinding* method. This applies the binding to the passed property.

2.4.5. Colour code list generation

Each supported colour atlas differs in the way it handles the colour codes. The colours have a unique identification name, which has the coordinates embedded. When the application receives the colour codes from the database, it needs to perform their correct partitioning to string arrays so that it can then use the results to filter and display the lists in the input area of the user interface.

The main logic for these actions is located in the classes in the *ColourMatch.Core.Services.ColourCodes* namespace.

The *ColourCodeListsService* service receives requests from the *ColourMatchViewModel*. These requests specify the current setup of the user interface. The service generates the appropriate filtered data and returns them as an instance of the *ColourCodeListsSetup* class. This class is basically just a plain data transfer class.

To facilitate the partitioning scenarios for the colour codes, each supported colour atlas has its own implementation of the *IColourCodePartitioningService* interface. This interface offers methods for partitioning of a code and also for

transforming the selected items in the user interface back to the unique name of the colour.

The Munsell and RAL Design atlases have their names specified in such a way that it is enough to perform splitting of the name string by underscore characters. This basic behaviour is defined in the *BasicColourCodePartitioningService*. The concrete atlases then derive from this implementation specifying the correct atlas names, colour code part counts and prefixes. The NCS and RAL Classic atlases have their own specific implementation that improves usability by manually adding additional partitions that separate the colour codes.

The *ColourCodePatitioningServiceFactory* is a helper class that returns the appropriate partitioning service based on the identification name of the atlas passed to it. It also has a fall back to a generic partitioning service that can handle previously unknown and not yet implemented atlases.

2.4.6. Platform-specific UI code

The main part of platform-specific code that is required for our mobile applications will consist of defining the "*View*" component of the MVVM architectural pattern.

As we mentioned, each platform has its specifics in defining the user interface and we therefore need to use the platform-specific features offered for its design.

As we described in the section on UI considerations, we decided to dynamically display the drop-down lists for input colour code selection. This will require us to create and destroy these visual controls on the fly and also setup the data binding directly in code (which would otherwise not be necessary for the Windows and Android platform).

The lists have to indicate selection changes to the view model and notify it about any modification so that the list of found matches can be refreshed with up-to-date results.

2.4.7. Platform services and value converters

In addition to the user interface that has to be built specifically for each platform we also need to implement specific implementation for services that work

with device or operating system specific features and value converters that use platform dependent types as their results.

As we described in the SQLite database section, each platform has to ensure the right configuration of target path where the local database will be stored and offer a way to copy the database from the installed read-only application package to a private application storage location (the application data folder for Windows, user document directory on iOS and application files directory on Android). The *IPlatformDatabaseManager* interface provides a getter-only property called *DatabasePath* that returns the path where database should be stored and an asynchronous task method *CopyPackageDatabaseAsync* that performs the copying of the database file to the local storage. Each platform implements this interface and then registers its instance as a singleton in the MvvmCross IoC container.

It is worth noting that each of the platform project contains a link to the source database, because this file need to be included directly as a content file in all of the projects so that it is then deployed in the application package.

All three platforms also have different mechanisms for handling application settings. Specifically we can use the built-in *ApplicationData.Current.LocalSettings* container on Windows (which is automatically persisted in the application data location), *NSUserDefaults.StandardUserDefaults* dictionary on iOS and the *PreferenceManager* on Android. The interface each platform has to implement in this case is *ISettingsService*, which has two generic methods *GetSetting<T>* which retrieves a setting by key from the settings storage and *SetSetting<T>* that stores a value to the settings storage. Because the methods are generic, it is also possible to store complex types directly. The implementation used in this application takes advantage of the *Newtonsoft.Json* library to convert the complex types to their JSON string based representation that can be directly persisted and later deserialized.

Each platform also needs to contain definitions for two value converters used with data binding in the user interface.

The first value converter needs to transform the Delta E value of the colour match to a colour used on the rectangular strip indicating the closeness of the match ranging from green to red.

The second converter performs the conversion from sRGB string retrieved from the database to a platform colour type representation.

2.4.8. Building the Microsoft Windows application

The "Windows application" label can be a little misleading, because in fact we have to build two different application packages – one for the Windows 8.1 platform and one for the Windows Phone 8.1 platform. However, apart from separate image assets, package manifests and platform dependent references (for example SQLite), both application share all of their code. This unity will be improved even further in the upcoming release of Windows 10 where the resulting application package will be actually one single unified binary. For our purposes, we can consider Windows as a single platform including both Windows and Windows Phone as all of the code we write is located in the "*ColourMatch.Windows.Shared*" project node. In cases, when the platform has some specifics (the best example being the handling of hardware based back button present only on phones) we use compiler conditional symbols "*WINDOWS_APP*" and "*WINDOWS_PHONE_APP*".

Because the MVVM pattern originates on the Windows platform, it offers the best built-in support for data binding in built in controls. This helps minimize the amount of source code required to set up the connection between the view and the view model, but also has a small disadvantage in terms of code that MvvmCross needs to enable its implementation of value converters to work on Windows.

Windows platform expects the value converters to implement the *IValueConverter* interface. Sadly, this interface is not present in the Portable Class Library. To solve this issue, MvvmCross uses the concept of native value converters on Windows. For each converter we implement deriving from the *MvxValueConverter<TFrom,TTo>* type, Windows project has to include a class deriving from *MvxNativeValueConverter<T>* where *T* is the value converter class we implemented. The *MvxNativeValueConverter<T>* type derives from the *IValueConverter* type and therefore acts as an example of the adapter pattern. The value converters then need to be registered as static resources in application's XAML definition. For this purpose we created a separate resource dictionary in the "*Converters.xaml*" file in the "*Resources*" folder of the "*ColourMatch.Windows.Shared*" project.

To handle the layout changes between portrait and landscape screen orientation we added an *UpdateVisualState* method to the base class for our Windows views. This method is called each time the size of the view changes – which means after screen rotation or after the window the application is running in is resized (this is more prominent in Windows 10 as opposed to Windows 8.1, because in the newer release the applications are allowed to run in arbitrarily sized windows on the desktop and the user can manipulate the size of the view anytime). The method performs a check on the size of the window and sets the appropriate state for the *VisualStateManager*. This class allows for easy design-side declarative manipulation with layout in XAML page definition that does not require any C# code.

Because of some binding inconsistencies we need to reuse combo box controls when transitioning between different atlases. Originally the code in "*ColourMatchView.xaml.cs*" cleared all existing combo boxes and rebuilt them from scratch, but that caused the binding for the first boxes not to be updated. This might be a problem in the underlying API, but it seems to be resolved in Windows 10.

2.4.9. Building the Apple iOS application

Because we wanted to avoid the need to for transitions of the project to the Xcode designer and because the layout of the application is simple enough for manual manipulation all controls are built when the individual views are created in code.

This means we have to first build the controls in response to the *ViewDidLoad* lifecycle event and later also ensure their right positioning in the *UpdateLayout* method. This method is called each time the view is about to appear or is transitioning to a new size.

The iOS operating system does not offer a control similar to the combo boxes on Windows and Spinners on Android. Rather, we have to use a combination of *UITextField* control and a *UIPickerView*. We connect the picker view to the text view via *TextView's InputView* property. This will activate and display the picker view each time the text view receives focus. To bind the data to the picker view, we use the *MvxPickerViewModel* type. This type is defined in the MvvmCross libary adding data binding support to classic model used to display items in iOS pickers. We use the instance of this type to bind items and also to update the user selection and invoke the command that performs selection change check.

Because it not automatically possible to dismiss an opened picker we also add a *UITapGestureRecognizer* on the page. This will go through all text views and try to resign them as first responders, essentially forcing them to lose their focused state.

To demonstrate the designer possibilities, we used Xcode to create the layout for the template of a cell in the found matches list. The list itself can be stretched horizontally based on the resolution of the device it is running on so it was necessary to make it as responsive as possible.

The designer has a feature called constraints for this very purpose. This feature allows the user to define rules, by which should the layout behave. These rules are then checked during runtime and the layout resizes individual controls inside the cell accordingly to the space it is offered.

Xamarin.iOS automatically generates a designer code file where we can expose properties that are connected to controls on the table view cell and we then take advantage of the MvvmCross binding facility to load he data assigned from the table view.

2.4.10. Building the Google Android application

Application for the Android platform required very similar steps to the Windows application, because both offer XML based design support. MvvmCross extends the XML mark-up to add data binding properties to all controls that get automatically wired up and handled.

Important fact about android activities we have to consider is that the activity is always completely recreated when the screen orientation is changed. Therefore, even our input lists are destroyed and we have to recreate them manually each time. To make this possible, we place a fake call of the message handler that responds to input atlas change at the end of the activity's *OnCreate* method.

There is a small issue with the data binding for the *SelectedItem* of individual *Spinner* controls on Android. The default behaviour that is present in the MvvmCross framework causes the binding to be updated *after* the *ItemSelected* event on the spinner is fired. This is a problem, because we need the *SelectedColourCodeParts* array to be in-sync with the user interface when the *ColourCodePartChangedCommand* is executed. We avoided this problem by manually updating the values in *ItemSelected*

event handler right before the command is executed. This means the array is set once before and once after the action, but that does not have any negative implications on the functionality of the application. Note, that the spinner's *SelectedItem* binding is still required, because we need to allow binding for changes in the opposite direction (from view model to view) when the lists are updated with new data after direct colour change or colour atlas selection.

Similarly to the iOS version of the application we use an *UpdateLayout* method in the view to handle responsive behaviour of the page. Android offers different methods for handling this including creating separate layout for different screen sizes and orientations, but in our case manual approach is simpler and more straightforward. In our specific scenario we just need to ensure the main *LinearLayout* container changes its stacking orientation and the inner components are given the correct sizes.

3. Results

This chapter presents the results achieved by this bachelor thesis, explains the user interaction, mentions the known issues and describes the potential for improvements and enhancements of the final project. Finally, we mention the ways the application can be distributed to users.

The source codes of the solution can be found on the compact disk attached to this thesis (see Attachments).

3.1. Applications

Three client mobile platforms were covered following the steps described in this thesis. We will first describe them in general and then focus on specifics of the applications for each of the target platforms.

3.1.1. Main navigation

After the user starts the application a navigational menu is displayed. Here we are presented with two options – "*Find colour matches*" and "*Help*" (see Figure 8).

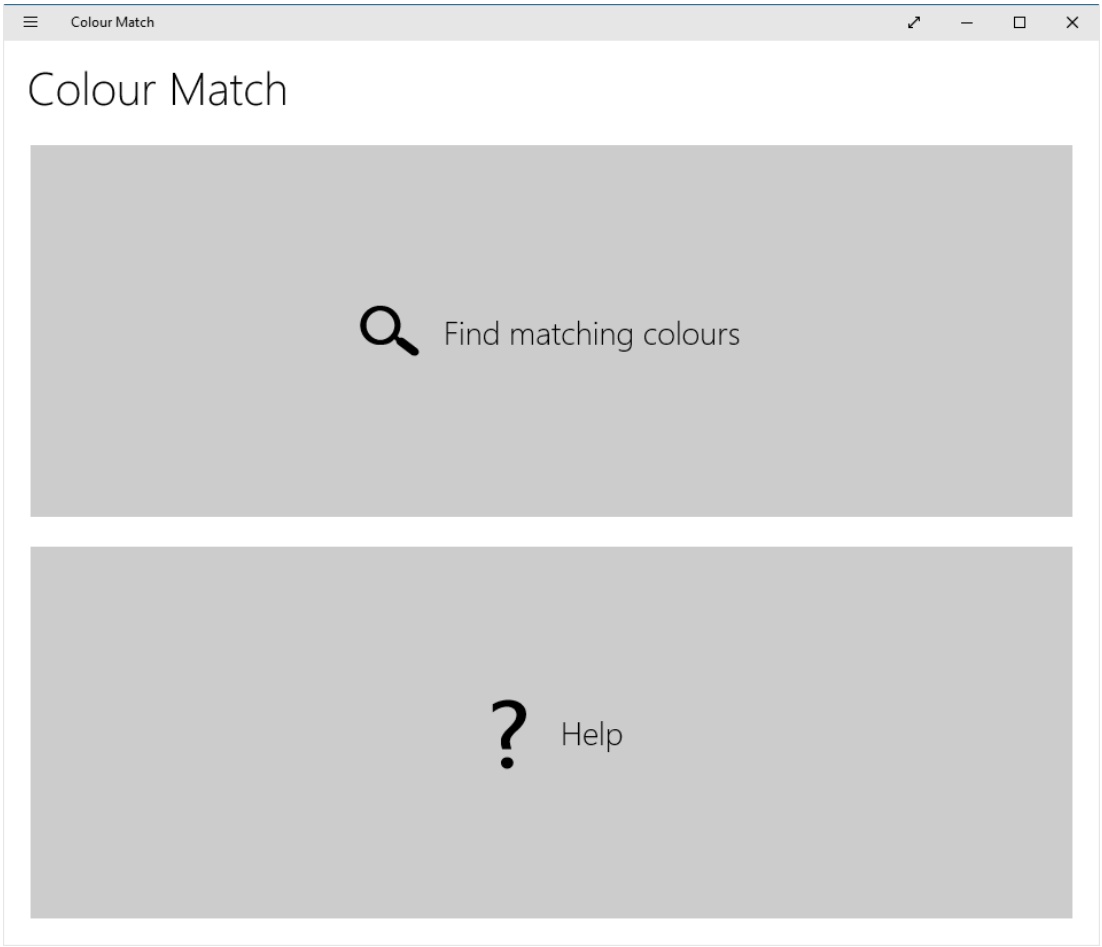


Figure 8: The main menu of the Windows application

The first offered option is actually the entry point to the core experience of the application, the place where we can search for matches and browse the colour atlases.

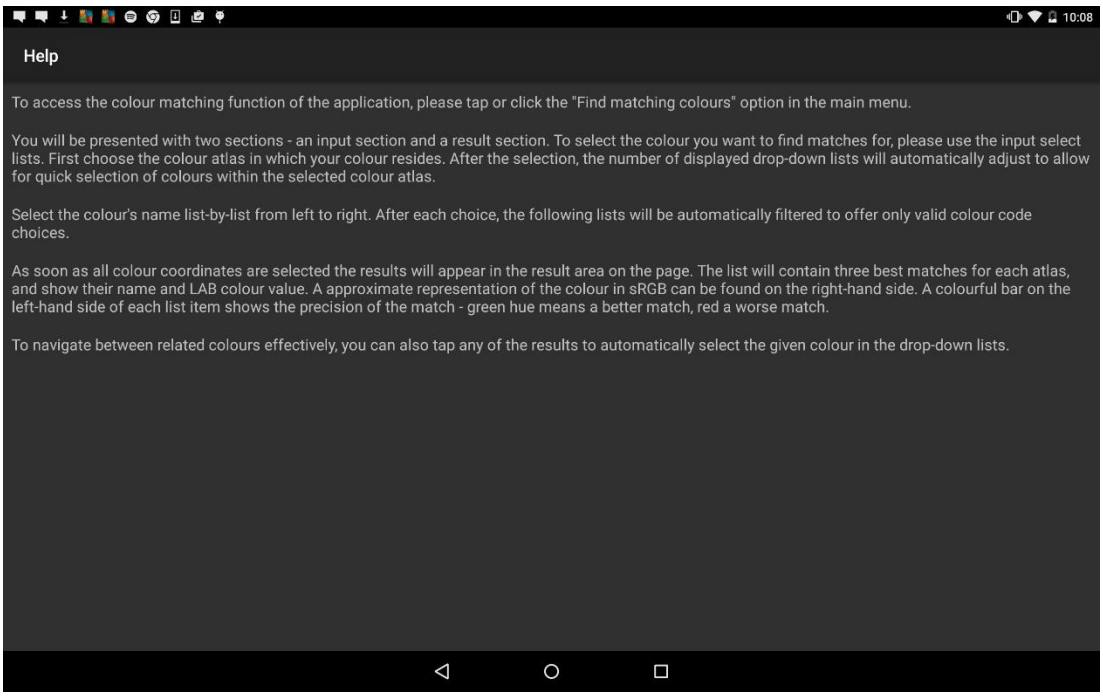


Figure 9: The help interface on an Android tablet

The second option ("*Help*") presents a quick overview and explanation of what the application offers and how the colour input works (see Figure 9). This acts as a user documentation, but it should not be necessary for more experienced users of mobile devices, because we were striving to deliver an intuitive and self-explanatory user interface, that is familiar and similar to other application experiences available on each platform.

3.1.2. Colour matching

After the user selects the "Find colour matches" option in the main menu of the application a new view with colour matching feature appears.

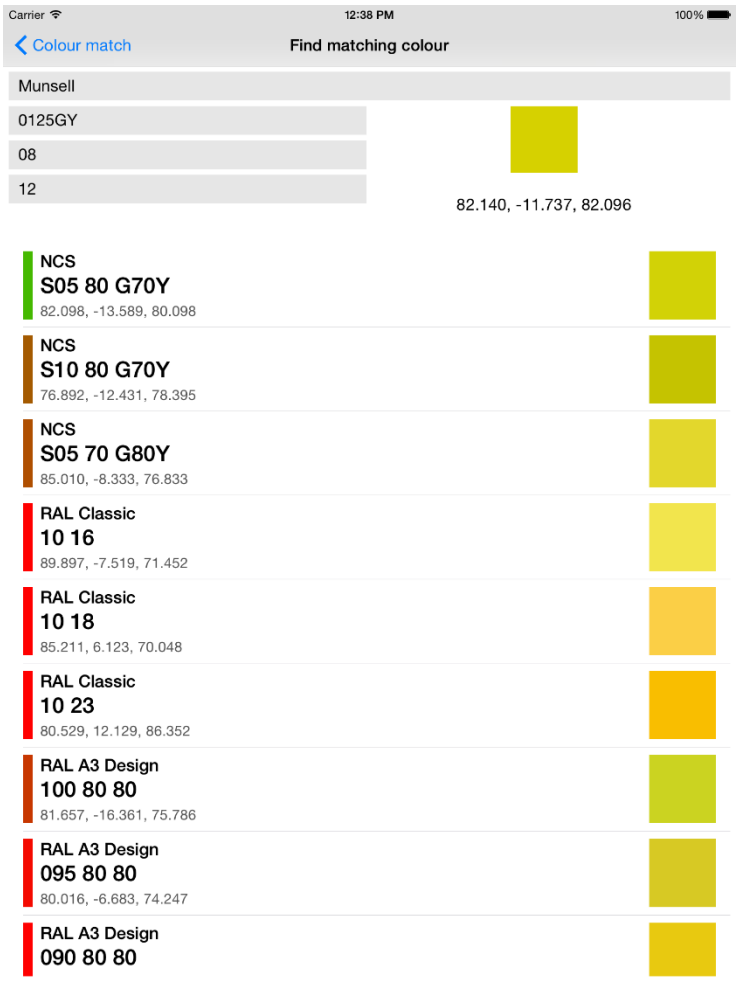


Figure 10: Colour matching screen on iPad (vertical layout)

The layout of this view depends on the size and orientation of the device or window being used. For portrait orientation the input boxes are located at the top of the screen and the list of found matches will appear in the bottom part (see Figure 10). Conversely in landscape orientation, the input is aligned to the left side of the view

and the result list covers full height of available screen on the right-hand side (see Figure 11 and Figure 12). It is worth noting that on devices with smaller screens running iOS and Windows operating system have only the vertically stacked layout enabled.

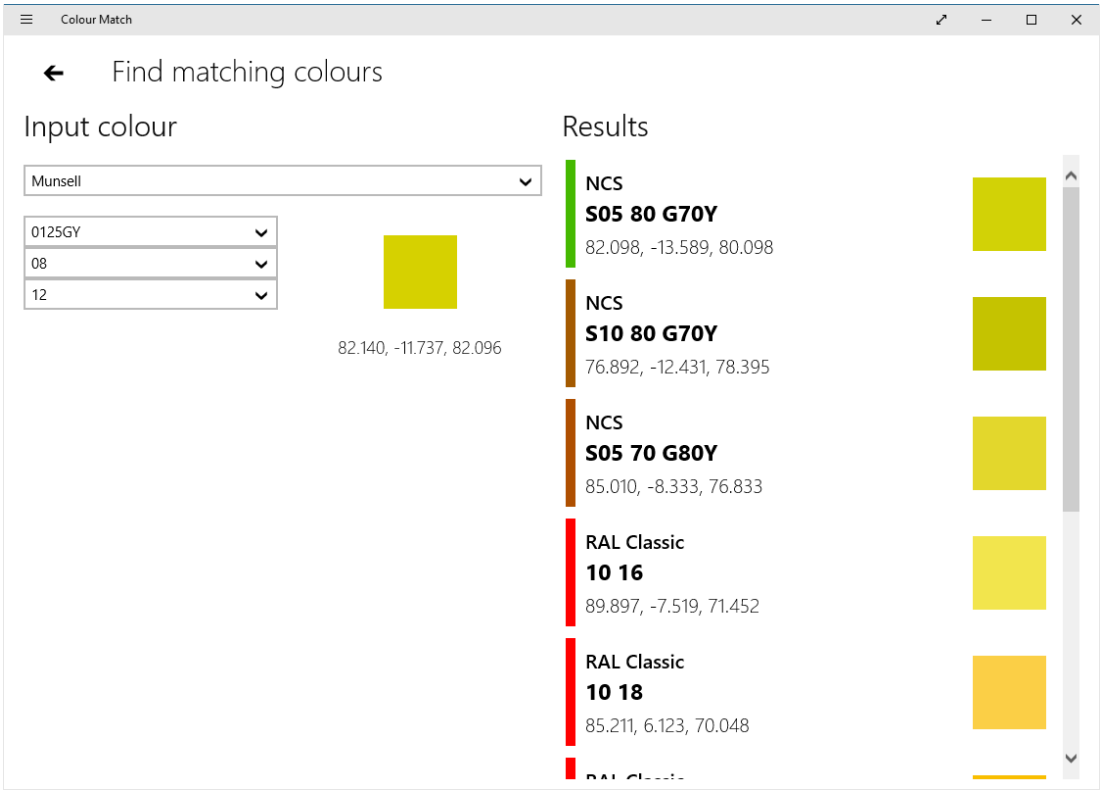


Figure 11: Colour matching screen on Windows (landscape layout)

After the first launch of the application, we have the first options of all lists in the input section preselected – this means Munsell colour atlas colour with coordinates 0124GY 08 12 should be displayed along with its approximate sRGB representation and LAB colour values.

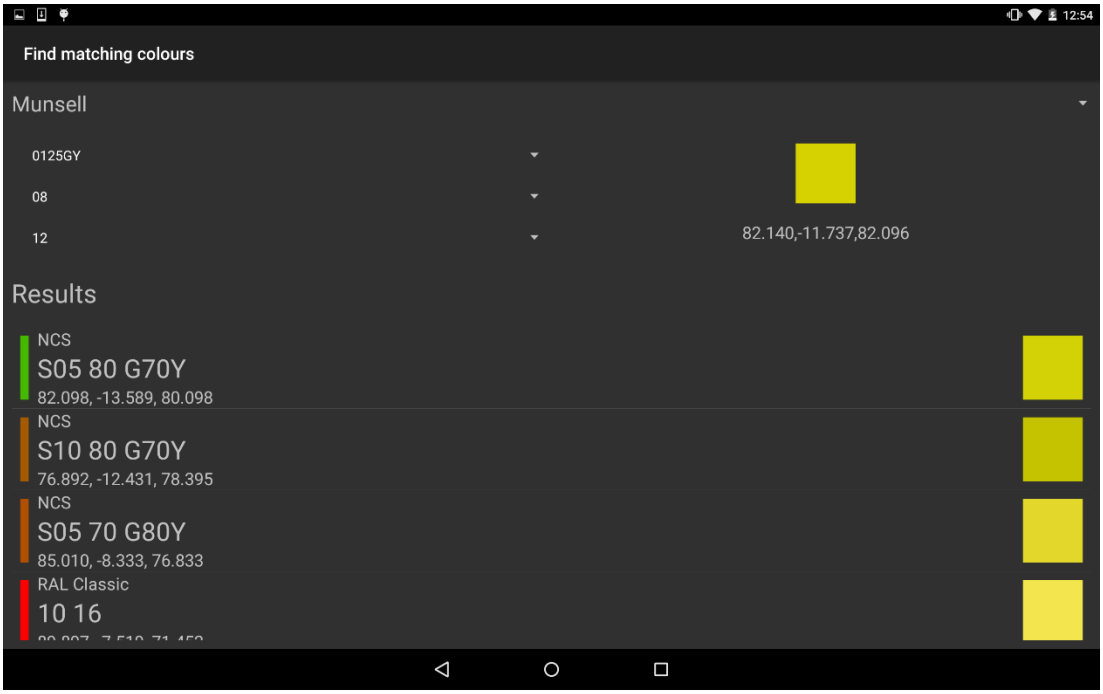


Figure 12: Colour matching screen on Android (landscape layout)

In the results list we now see the matched colours from the other atlases – NCS, RAL Classic and RAL Design (three best matches from each atlas). Each match lists the coordinates of the colour in its atlas, the LAB colour values and sRGB colour approximation. Please note that this colour display is not a good representation of the real-world colour atlas samples and are included mainly for illustration purposes. To see the colours accurately, original (physical) colour atlas samples need to be used. Also on the left-hand side of the match we can see a colourful rectangular strip that identifies the "closeness" of the match. This strip varies in colour from green for the best and closest matches to red for matches that are very imprecise.

There are two different means of selecting an input colour for match searching.

Generally the user has a specific colour she wants to examine. In such scenario the easiest approach is to select the colour using the lists in the input area of the page (see Figure 13). First we need to select the atlas where the colour resides. The atlas selection will automatically influence the number of lists available for selection then. Each list corresponds to one of the colour coordinates in the atlas. For multiple input lists the order in which the user selects the options matters, because the lists are automatically filtered based on the parts that were already selected. This means it is best to select in the lists from the first to the last. Every time the selection changes the

result list is updated in real-time and gives the user immediate feedback on what is happening.

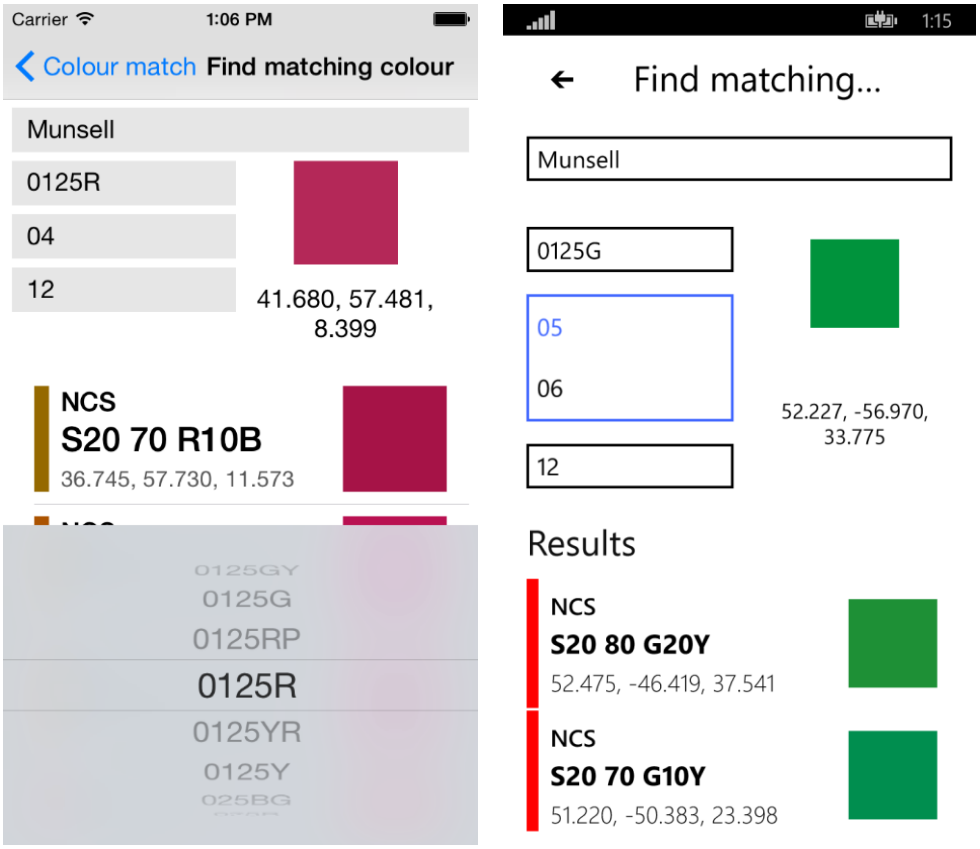


Figure 13: List selection on iPhone (left) and Windows Phone (right)

There is also a second option for input colour selection that is mainly a shortcut when browsing for similar or related colours. When browsing the colours, the found matches in the result list are actionable. That means the user can select any item in the list by tapping or clicking the specific colour (based on his device's input method) and the selected colour will be automatically immediately selected in the input area (setting the appropriate values for its colour code in the lists, including the input atlas list) and its matches will be queried and shown. This allows for fast and direct comparisons between matches across multiple atlases.

3.2. Testing

We have tested out the mobile applications by obtaining all the colour atlases and looking for matches across them.

The tests were successful and the applications were able to find valid matches for each given colour.

To make sure we are not influenced by the match results, we tried to estimate the best matches for several colours and then used the application to confirm our guesses. This has also proven to be successful, although in close match scenarios we sometimes have not chosen the best rated match, but rather the second best match.

3.3. Examples

This section presents some examples of colour matches that illustrate that some colours are represented very precisely in other atlases, but there are also cases where there is no way to find a suitable match.

3.3.1. Munsell 050B 06 08

This input serves one very precise and two good results in the NCS atlas and the acceptable results in RAL A3 Design. In case of the RAL Classic atlas, the matches are very distant (see Figure 14). This is caused mainly by the fact that the RAL Classic atlas covers only a limited range of colours.

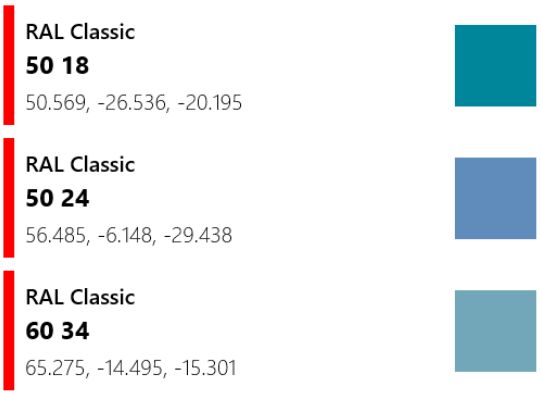


Figure 14: Matches of Munsell 050B 06 08 in RAL Classic

3.3.2. RAL Classic 4009

Selection of RAL Classic 4009 produces almost perfect matches in the NCS and RAL A3 Design atlases. The closest colour in the Munsell colour atlas has coordinates 050RP 06 02. This match is not as precise as the top matches for the other atlases, but it is still within a reasonable range (see Figure 15).

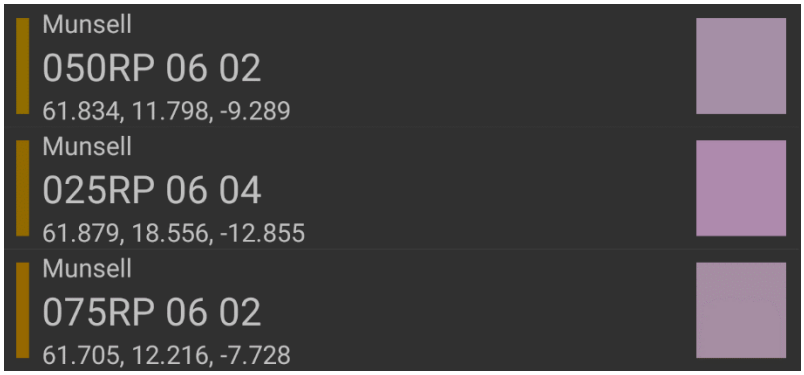


Figure 15: Best Munsell matches for RAL Classic 4009

3.3.3. NCS S05 02 B

This colour has surprisingly good matches in all colour atlases. In Munsell we can find three almost precise matches (see Figure 16), in RAL Classic two very close matches and also three similarly near matches in RAL A3 Design.

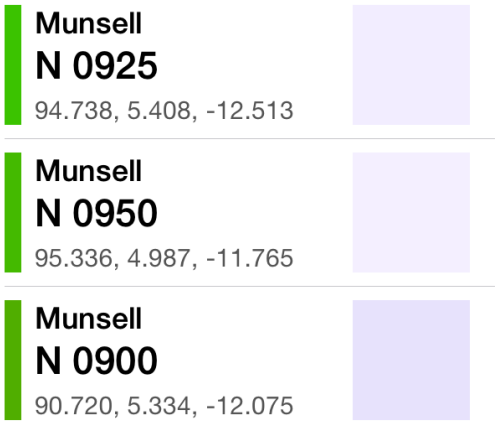


Figure 16: Closest colours to NCS S05 02 B in Munsell

3.3.4. RAL A3 Design 040 70 50

The results for this colour once again show the limitations of the RAL Classic colour range. While Munsell and NCS atlases offer quite similar colours to the input, the nearest match in in RAL Classic is very distant (see Figure 17).

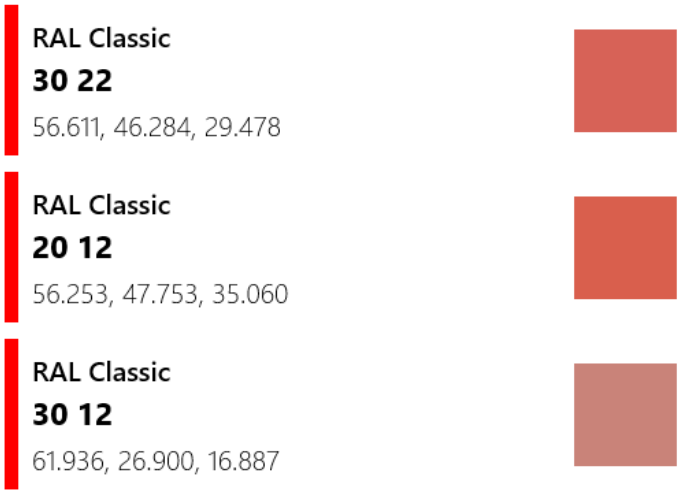


Figure 17: Best matches of RAL A3 Design 040 70 50 in RAL Classic

3.4. Known issues

During testing we discovered an issue with the Windows version of the application. This problem arises when it is run on the Windows 10 Technical Preview operating system with touch input. It is not certain if this problem will actually occur on the final version of Windows 10, but the issue is not caused by the application itself, but by a layout problem in the built-in Windows controls. The exception is caused randomly when the user taps to expand a combo box with larger number of colour codes and it causes the application to crash. Unfortunately we have not found any way to solve or avoid this other than using different input methods on Windows 10 like mouse or pen.

Because the Android device market has very high fragmentation of different device types and vendors, it is very likely the Android version of the application will not be compatible with all devices running the appropriate version of the operating system.

3.5. Possible improvements

There is a potential for extending the colour matching applications to the cloud environment. This could potentially mean offering a cloud-based data source, which would contain a much larger source dataset, able to respond to user queries over the network. This way the device itself would be just a thin client accessing the data, which means it would not have to come with a pre-filled local database.

To preserve the off-line aspects, such application could also offer a caching option or ability to download all data for specific colour atlases at once.

It is worth noting, that thanks to the modular design, we could in this case share code not only across different platforms, but also with the server backend to share common classes for database model and LINQ-based queries.

To expand the range of supported devices, the Windows application will be available for upgrade to the new Windows 10 app model nearing the release of the new version of the operating system. This will make the application run on Universal Windows Platform (UWP), which allows building a single binary package to be deployed to all supported Windows 10 devices including personal computers, tablets, smartphones, Xbox One consoles, Surface Hub, HoloLens and even IoT devices. This will essentially improve the current project setup by avoiding the need for two Windows project nodes and a shared project.

3.6. Application distribution channels

In case we want to publish the application packages to be available to a wide range of users of mobile devices in the world, we can publish our applications to the respective application stores for each of the target platforms.

On Windows we can publish the application on the Windows Store via the Windows Dev Center website (29). The prerequisites for this include creating a Microsoft Account and registering as a Developer.

On iOS, the iTunes store acts as the application distribution platform. After registering as an Apple iOS Developer on the Apple Developer website (30) the application package can be sent for certification and publishing.

On Android there are several store-like options we can offer the application on. The largest is Google Play. We can use the Android Developers website (31) to upload our application to the Google Play store.

An alternative to using application stores is to directly side-load the applications to specific devices. This approach is particularly suited for business-line applications and testing purposes. All operating systems we targeted offer a way to install application packages manually on the device.

Conclusion

This thesis illustrated the process of building a cross-platform mobile application for three different operating systems.

The advantages and disadvantages of cross-platform development were mentioned including the decisions required to successfully prepare for such development process.

We have seen the challenges associated with the diversity of APIs available on each platform and the ways to avoid them and bridge the differences with shared and platform-specific code.

MVVM presentation layer architecture was also part of our discussion as a reliable solution for handling multiple platform projects that share a single core. This helped us fully reuse everything apart from specific user interface and platform-bound services directly.

Mobile applications that are the main result of this thesis can be used as a valuable and reliable tools in real-world industrial and design scenarios and can reach a wide audience with their cross-platform potential.

Bibliography, on-line resources

1. **Choudhury, Asim Kumar Roy.** *Principles of Colour and Appearance Measurement: Volume 2: Visual Measurement of Colour, Colour Comparison and Management.* místo neznámé : Woodhead Publishing, 2014. 9781782423881.

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

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

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

5. **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.

6. **Munsell Color.** About Munsell Color. *Official Site of Munsell Color.* [Online] 2013. [Citace: 18. April 2015.] http://munsell.com/about-munsell-color/.

7. **Fairchild, Mark.** Munsell books image. *Wikipedia.* [Online] 24 January 2005. [Cited: 15 March 2015.] http://commons.wikimedia.org/wiki/File:Munsell_Books.jpg.

8. **Munsell Color.** How Color Notation Works. *Official Site of Munsell Color.* [Online] 2013. [Cited: 15 April 2015.] http://munsell.com/about-munsell-color/how-color-notation-works/.

9. **NCS Colour AB.** About us. *NCS Colour.* [Online] 2015. [Cited: 16 April 2015.] http://www.ncscolour.com/en/about-us/.

10. —. Logic behind the system. *NCS Colour.* [Online] 2015. [Cited: 16 April 2015.] http://www.ncscolour.com/en/natural-colour-system/logic-behind-the-system/.

11. **RAL gemeinnützige GmbH.** RAL Colours history. *RAL Colours.* [Online] 2015. [Citace: 16. April 2015.] http://www.ral-farben.de/content/footer-navigation/footer-ueber-ral-farben/about-ral-colours/history.html.

12. **Colourfeeling.** RAL K5 Fächer RGB. *Wikipedia.* [Online] 11 September 2009. [Cited: 16 April 2015.] http://commons.wikimedia.org/wiki/File:RAL_K5_F%C3%A4cher_RGB.jpg.

13. —. RAL DESIGN System. *Wikipedia.* [Online] 6 February 2008. [Cited: 15 March 2015.] http://commons.wikimedia.org/wiki/File:RAL_DESIGN_System_D2_Farbf%C3%A4cher.jpg.

14. **Richards, Mark.** *Software Architecture Patterns.* : O'Reilly , 2015. 978-1-491-92424-2.

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

16. **The jQuery Foundation.** jQuery Mobile. [Online] 2015. https://jquerymobile.com/.

17. **Modernizr.** Modernizr: the feature detection library for HTML5/CSS3. [Online] 2015. http://modernizr.com/.

18. **Adobe Systems Inc.** PhoneGap. [Online] 2015. http://phonegap.com/.

19. **The Apache Software Foundation.** Apache Cordova. [Online] 2015. http://cordova.apache.org/.

20. **Xamarin Inc.** Mobile App Development & App Creation Software. *Xamarin.* [Online] Xamarin Inc., 2015. http://xamarin.com/.

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

22. **Microsoft.** 5: Implementing the MVVM Pattern Using the Prism Library 5.0 for WPF. *Microsoft Developer Network.* [Online] [Cited: 2 April 2015.] https://msdn.microsoft.com/en-us/library/gg405484(v=pandp.40).aspx.

23. **Bugnion, Laurent.** Messenger and View Services in MVVM. *MSDN Magazine.* [Online] March 2013. [Cited: 2 April 2015.] https://msdn.microsoft.com/en-us/magazine/jj694937.aspx.

24. **Lodge, Stuart.** MvvmCross - GitHub. *GitHub.* [Online] 2015. https://github.com/MvvmCross.

25. **SQLite.** SQLite Home Page. [Online] 2015. http://sqlite.org/.

26. **Krueger, Frank A.** sqlite-net. *Github.* [Online] 2015. https://github.com/praeclarum/sqlite-net.

27. **Krog, Øystein.** SQLite.Net-PCL. *Github.* [Online] 2015. https://github.com/oysteinkrog/SQLite.Net-PCL.

28. **Bugnion, Laurent.** MVVM Light Toolkit. *CodePlex.* [Online] 2015. https://mvvmlight.codeplex.com/.

29. **Microsoft.** *Windows Dev Center.* [Online] Microsoft, 2015. https://dev.windows.com/.

30. **Apple Inc.** *Apple Developer.* [Online] Apple, 2015. https://developer.apple.com/.

31. **Google.** *Android Developers.* [Online] Google, 2015. http://developer.android.com/.

32. **Lodge, Stuart.** Forwards. [Online] 2015. http://slodge.blogspot.com/.

33. —. N+1 days of MvvmCross. [Online] 2014. https://www.youtube.com/playlist?list=PLR6WI6W1JdeYSXLbm58jwAKYT7RQR31-W.

List of Figures

List of Abbreviations

MVVM – Model-View-ViewModel pattern

MVC – Model-View-Controller pattern

MVP – Model-View-Presenter pattern

UWP – Universal Windows Platform

API – Application Programming Interface

PCL – Portable Class Library

IoC – Inversion of Control

SQL – Structured Query Language

LINQ – Language Integrated Query

UI – User Interface

ART – Advanced Rendering Toolkit

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

XAML – Extensible Application Markup Language

W3C – World Wide Web Consortium

HVC – Hue Value Chroma colour notation

RGB – Red Green Blue colour system

NCS – Natural Colour System®©

JIT – Just-in-time compiler

WPF – Windows Presentation Foundation

CIELAB – L*a*b* colour space

Attachments

Compact disc attached to this thesis contains the project files required to successfully build and run the colour matching mobile applications.

Folder "*Code*" contains all project files and Visual Studio 2013 solution file "*ColourMatch.sln*".

The folder "*Documentation*" contains documentation files. "Developer i*nstructions.pdf*" describes the prerequisites, project setup and what is required to successfully build and run the mobile applications. "*User instructions.pdf*" gives simple guidelines on using the applications (more detailed user documentation is available in chapter 3 (Results)). The text of this thesis is also available in this folder as "*Thesis.pdf*"

Finally, the folder "*Exports*" contains a sample of the JSON files exported from ART.