Optical Microscope Digitalization and Software Implementation

Leemans Jasper

May 6, 2008

Preface

An engineer's main purpose is to use his scientific baggage and ultimately turn it into something useful. In a Master's thesis both the future engineer's theoretical knowledge and his practical side should therefore be present. In my case basic knowledge of physics, in particular optics, was not only brought up again but also deepened in the field of optical microscopy. A more hands-on element was added by combining an internship period with extensive computer programming efforts. If I pass or not is out of my hands now but I can honestly conclude, and many fellow students will agree on this, that the past few months have been hectic but rewarding. Not only did we learn new things about ourselves but foremost about our future profession. Finally I hope you will enjoy reading this thesis as I tried my very best to write it as fluently as possible.

Of course I needed help once in a while and for this I thank my college promotor Toon Goedemé, my supervisors at the Visics Lab[1] Johan Van Rompay and Bert DeKnuydt and their colleague Wim Moreau, for their guidance. My family, friends and fellow students should neither be forgotten for their support throughout my entire academic career.

[1]VISion for Industry Communications and Services, a research group within the K.U.Leuven.

Abstract in English

The aim of this thesis can be divided into two distinguishable parts.

- Digitalization of an optical microscope.
- A computer programming part.

The first part consist mainly out of refurbishing an old optical microscope, playing with optics and attaching different cameras for testing purposes. For this reason a general understanding of optical microscopy was desperately needed. During this study a short manual for the microscope was drawn up because none was available at the lab nor from the manufacturer. Since we are trying to acquire high quality photomicrographs proper illumination becomes a very important factor, especially when reducing dust, scratches and other unwanted image artifacts. Köhler illumination is the most commonly used technique and setting it up correctly is essential in obtaining good photomicrographs.

The second part is the development of a platform independent IIDC compatible GUI application for video capture. Such a program does currently not exist in the wonderful world of open source and free software. The IIDC standard (or DCAM standard) defines the behavior of cameras that output uncompressed image data without audio over a IEEE 1394 FireWire bus. Cameras that support the IIDC standard are usually cameras aimed at industrial or scientific applications such as capturing microscopic images for lab work. The software is written in C/C++ with Qt for the graphical user interface and libdc1394 for the actual image acquisition. A widget which supports video recording is also added, using ffmpeg to encode the raw frames.

A short summary is provided in both English and Dutch. It contains the most important conclusions and findings for both parts of this thesis but is by no means complete.

Abstract in het Nederlands

Het doel van deze thesis kan opgedeeld worden in twee belangrijke onderdelen.

- Het digitaliseren van een optische microscoop.
- Het programmeren van een software applicatie.

Het eerste deel bestaat voornamelijk uit het opknappen van een oude optische microscoop, spelen met de interne optica en het opnemen van beelden door middel van digitale camera's. Om dit goed te kunnen doen was eerst een grondige studie van de theorie achter de optische microscoop op zijn plaats. Tijdens het doornemen van deze theorie werd een korte handleiding voor de microscoop geschreven. Dit was nodig aangezien er geen handleiding meer voor handen was in het laboratorium noch bij de fabrikant. Doordat ons doel het opnemen is van microscopische beelden van hoge kwaliteit is vooral de belichtingstechniek een belangrijke factor. Vooral voor het reduceren van stof, krassen en andere storende elementen is het belangrijk dat de microscoop juist ingesteld staat. De Köhler belichtingstechniek is de meest frequent gebruikte techniek bij optische microscopen welke, indien juist toegepast, microscopische beelden van goede kwaliteit voortbrengt.

Het tweede deel van deze thesis omvat het ontwikkelen van een video capture applicatie, compatibel met de IIDC standaard, platform onafhankelijk en met een degelijke grafische user interface. Zo een programma bestaat op dit moment nog niet in de wonderlijke wereld van open source en free software. De IIDC standaard (ook wel de DCAM standaard genoemd) legt vast op welke manier camera's niet-gecomprimeerde beelden (zonder geluid) doorsturen over een FireWire bus (IEEE 1394). Camera's die voldoen aan deze specificaties zijn meestal camera's gebruikt voor industriële of wetenschappelijke toepassingen, net zoals onze microscoop toepassing. De software is geschreven in C/C++ in combinatie met Qt voor de GUI en libdc1394 voor de eigenlijke communicatie met de camera. Buiten het opslaan van stilstaande beelden is ook het opnemen van video mogelijk gemaakt door het ffmpeg project mee te integreren.

Een korte samenvatting gaat vooraf aan het boek en streeft dus geen volledigheid na. Een Nederlandse vertaling van deze samenvatting is ook aanwezig.

Summary in English

Light paths

A microscope consists out of various optical elements like lenses and diaphragms. It is very important to make a distinction between the *image forming light path* and the *illuminating light path*. As you can see in figure 1 a beam of light is used to illuminate the specimen or sample under the microscope. This beam is not in focus at the level of the specimen but forms a spread out light cone. This ensures an even distribution of the light over the entire surface of the specimen resulting in uniform brightness. On the other hand, the microscope's objective lens is focussed on the specimen and contributes the most to the overall magnification of the image. The objective is of course part of the image forming light path and since the beam of light is clearly not it should now be fairly understandable that dust and scratches in the illuminating light path are not visible in the resulting magnified image.

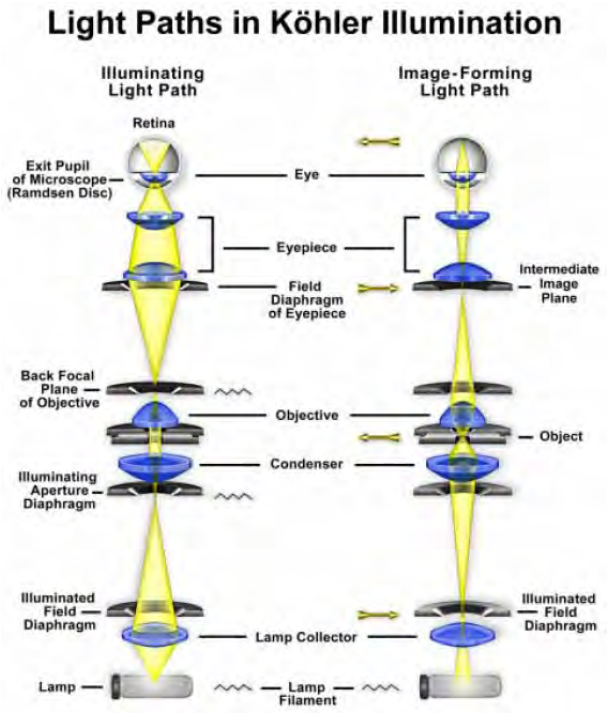


Figure 1: Light paths in optical microscopy (Molecular Expressions ©).

Setting up proper Köhler illumination basically means adjusting the lenses in the illuminating light path to ensure an evenly distributed beam of light and thus no unwanted artifacts (fig. 2). This adjustment should be done every time a different microscope objective is used since every objective has a different lens head. The size and magnification of the lens play an important role here.

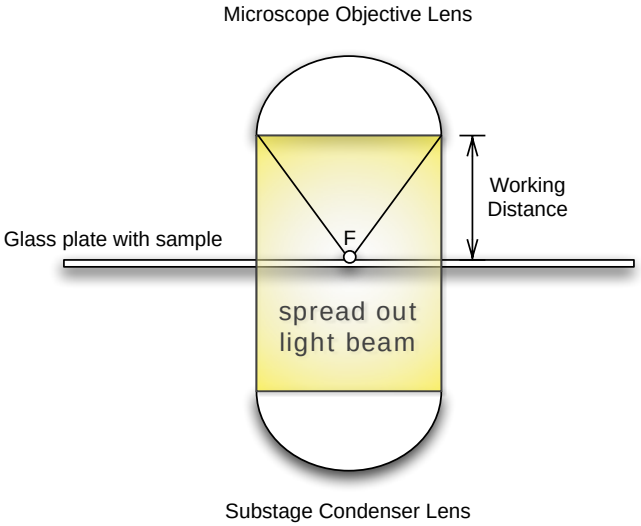


Figure 2: Spread out light beam, focal point F and working distance.

As you can see the *working distance* is the distance where the specimen is into focus and it is the same as the focal distance *f*. In general the higher the magnification of the lens, the smaller the working distance becomes. Another important factor is the angle at which the focal point lies which is the *angular aperture* defined as

$$a = 2 \arctan\left(\frac{D}{2f}\right) \tag{1}$$

With D the diameter of the lens. This factor is a dimensionless number that characterizes the light acceptance ability of the lens. Usually a deviation on this factor is engraved on the barrel of a microscope objective as the *numerical aperture* which defines the immersion medium to be used. In a medium with refraction index $\alpha = 1$ (e.g. plain air) the numerical aperture is defined as

$$NA = \alpha \sin\left(\frac{a}{2}\right) \tag{2}$$

This is all clearly illustrated in figure 3. As you can see in figure 4 a high quality photomicrograph can be obtained when proper illumination is set up, reducing the need for post-processing the image which is always a good thing.

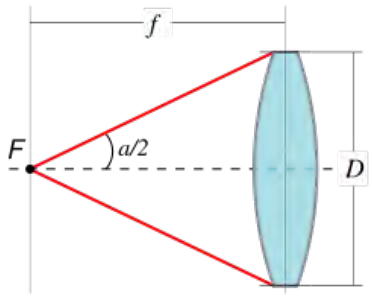


Figure 3: Lens factors.

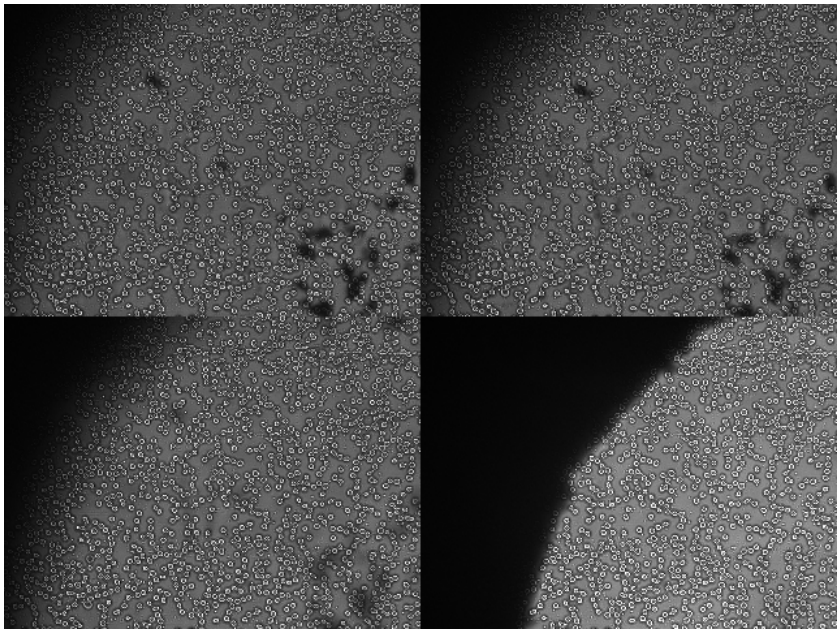


Figure 4: Reducing dust and scratches by means of proper Köhler illumination.

Optical coupling

Another important image artifact we should try to avoid is *lens flare*. The *camera adapter* optically couples the digital FireWire camera to the microscope with additional lenses. The distance between those lenses, and consequently the height of the camera adapter mounted on top of the microscope, should be correct to avoid lens flare. Figure 5 shows a lens flare artifact, figure 6 shows elimination of lens flare by properly adjusting the height as seen in figure 7.

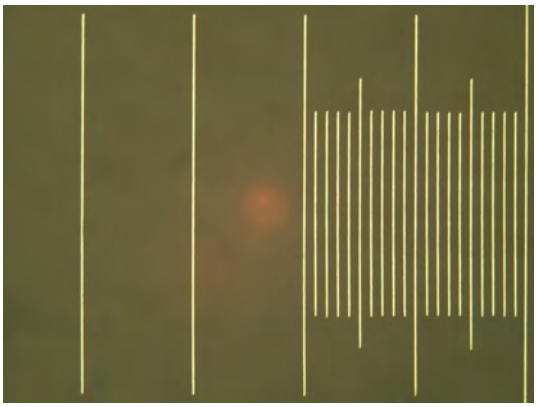


Figure 5: Lens flare.

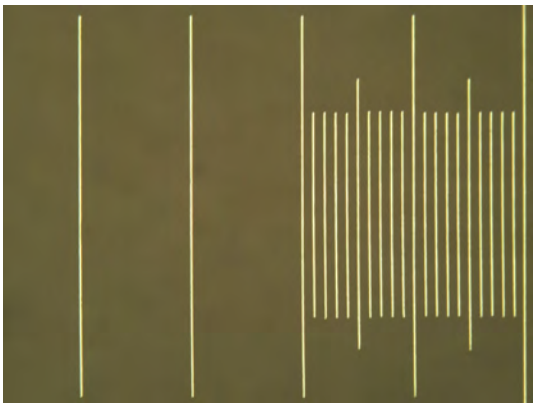


Figure 6: No lens flare.

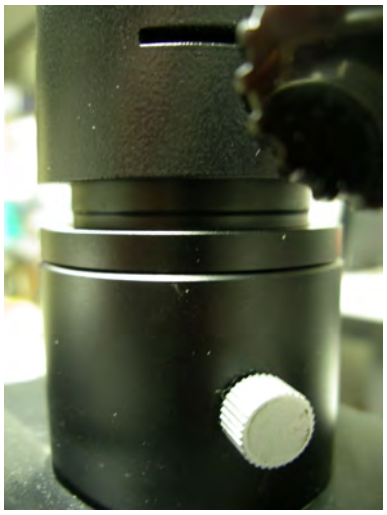


Figure 7: Adjusting eight of the camera adapter.

Filters

Different light attenuating filters can be used to improve image quality. Color filters attenuate certain wavelengths of the spectrum which is useful when a certain color is overly present in an image. Gray filters attenuate all wavelengths (white light) and are used when an image is too bright. Using such a gray filter is better than turning down the light bulb because overdoing this shifts the light more to the red side of the spectrum. A third type of filters are the heat absorbing ones to prevent damage to the specimen. An example of a blue filter, which attenuates red and green in the original image (fig. 8), can be seen in figure 9.

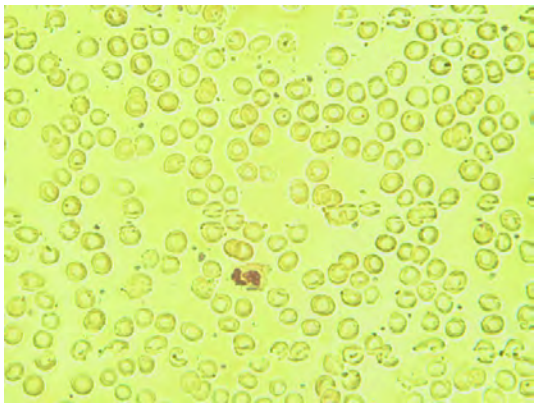


Figure 8: Image of bloodcells with no optical filter in the light path.

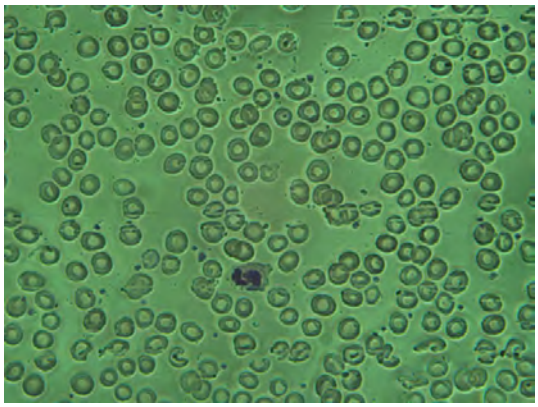


Figure 9: The same image with a blue filter in the light path.

Transmitted and reflected light microscopy

In classic optical microscopy light is transmitted *through* a specimen from underneath, propagates up the microscope's tube to the eyepieces or the camera. This method only works for translucent samples and not for opaque ones like material samples (wood, metals, ...) or integrated circuits. For this reason reflected light microscopy was invented. A beam of light is send down the microscope's tube, reflects on the sample and is consequently captured by the microscope objective. Results as in figure 10 are obtained.

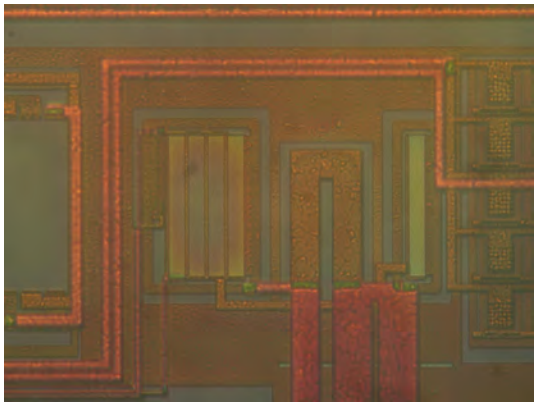


Figure 10: Image of an integrated circuit obtained with reflected light microscopy.

Darkfield microscopy

Regular illumination of a specimen, either in transmitted or reflected light microscopy, is called *brightfield microscopy*. Other techniques exist, one of which is called *darkfield microscopy*. In darkfield microscopy the center of the illuminating rays are blocked and unlike Köhler illumination the light beam is in fact directed at the specimen (fig. 11).

Because no light travels directly through the specimen (or is perpendicularly reflected) the image is generally dark. However because the light scatters, mostly on the edges of a specimen, it still has the ability to enter the objective and produce an image. This yields in a high contrast image, even in unstained (colorless) samples (fig. 12). Darkfield illumination is can be used with both transmitted and reflected light microscopy.

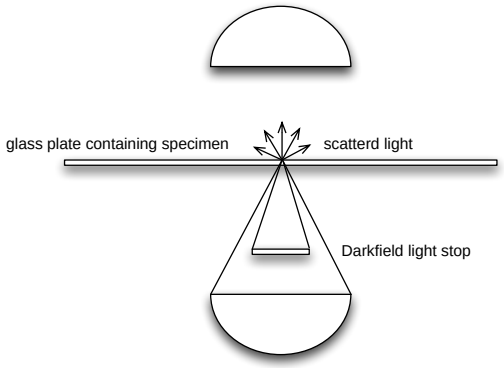


Figure 11: Principle of darkfield microscopy.

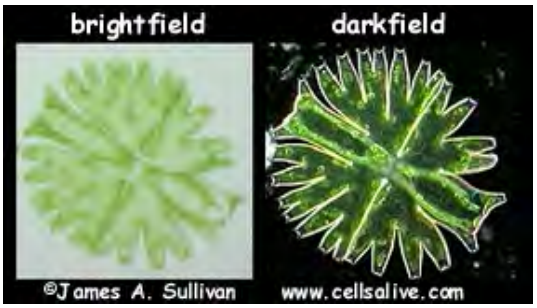


Figure 12: A comparison between brightfield and darkfield illumination.

Software

The software we've created aims to be a platform independent GUI program for IIDC compatible video capture. It can acquire single images, sequential images (with a timed interval) and full motion video. The user has the ability to select all the video resolutions, color modes and framerates his or her camera supports. It should also be fairly easy for any engineer or scientist to add custom image processing algorithms to our code. For microscopy a special measurement calibration feature was added to do fairly accurate measurements on the microscopic scale (fig. 13). Calibration is done with a stage micrometer, a glass plate with predefined engravings in micrometers.

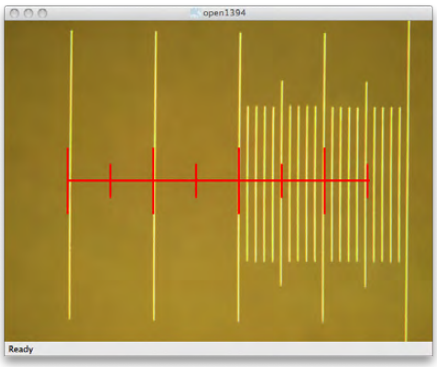


Figure 13: Measurement calibration with a stage micrometer.

GUI programming

C/C++ in combination with the QT toolkit was chosen for this project. Qt is essentially a cross-platform development framework for the creation of graphical user interfaces but also has a wide range of non-GUI related features. It is free and open source unless it is used in commercial applications, in that case a license should be bought. A Qt application is generally created out of a series of existing *widgets* or self-defined extensions to these widgets. Qt has a small drawback in the fact that color images are always internally presented in an rgb32 (0xffRRGGBB) or argb32[2] (0xAARRGGBB) format. Since the cameras do not directly output such format a conversion is always needed from either YUV4xx, regular rgb or mono. In plain C such a conversion function is basically a pixel copying function which is a costly operation and thus slow. Another performance problem in our program arises when drawing frames of high resolution at high framerates to the screen. This is caused by our choice to use Qt's internal drawing engine instead of the native one but this does mean identical results on all platforms. Although the color transformations and drawing functions consume a lot of CPU cycles our program does not lose GUI responsiveness nor stability, even on older machines.

Image acquisition

Actual acquisition of the camera frames is done with an open source C library called libdc1394. It is a pretty low-level library giving us access to all the features described in the IIDC camera standard. This library is currently available for Linux/OS X so our program does not yet run on Windows. However in the near future a Windows port of libdc1394 is planned, making it the best and only choice for cross-platform FireWire video capture. Another GUI for libdc1394 already exists, called Coriander, however it is GTK+ based (not Qt) and only runs on Linux.

MPEG recording

To encode the raw frames coming from the camera into a compressed video stream we needed an encoding library. Libavcodec, which is a part of the popular ffmpeg project, is used. We've chosen MPEG1 as compression format because it generally does not need additional codecs on most systems and is by no means proprietary. Most MPEG encoders use YUV420 frames as input and so does the one we use. YUV420 frames are preferred because of their high compressibility resulting from reduced color information and planar pixel representation. Unfortunately this means another format conversion on top of the already CPU intensive MPEG encoding algorithm. On slow computers or at high framerates this can result in dropped frames as our program takes to much time converting/encoding frames instead of checking for new incoming camera frames.

[2]If the alpha channel is not ignored.

Conclusion

We've succeeded in acquiring high quality photomicrographs and movies by combining optical microscope theory and extensive programming efforts. We also did not forget about the usability of our software. However, performance bottlenecks should be addressed in the future. Especially the color format transformation functions should be optimized and if possible a faster way to draw sequences of images should be considered. In today's multicore world a clean solution would be to move the camera grabbing part of our program to it's own separate thread, ensuring less dropped frames.

Samenvatting in het Nederlands

Beeldvorming & belichting

Een microscoop is een ingewikkeld optisch systeem dat bestaat uit verschillende lenzen en diafragma's. Het is hier erg belangrijk om een onderscheid te maken tussen twee verschillende optische paden, het *beeldvormende lichtpad* en het *belichtingspad*. Zoals te zien is in figuur 14 wordt het specimen onder de microscoop belicht met een breed uitgesmeerde lichtbundel. Hierdoor krijgen we een uniform verdeeld lichtveld over de gehele zichtbare oppervlakte van het specimen. Dit is het belichtingspad en de lichtbundel is dus niet in focus op het niveau van het te onderzoeken specimen. Het microscoop objectief, dat tevens zorgt voor de grootste vergrotingsfactor in het hele optische systeem, is uiteraard wel scherpgesteld op het specimen en bevindt zich dus in het beeldvormende pad. Hieruit valt op te maken dat wanneer er kleine storende elementen zoals stof en krassen zich in het belichtingspad bevinden deze zich niet manifesteren in het uiteindelijk verkregen beeld.

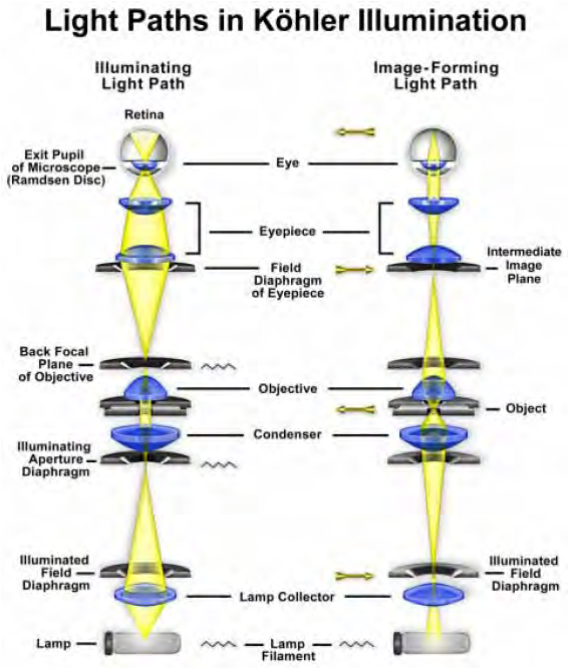


Figure 14: Verschillende optische paden in microscopie (Molecular Expressions ©).

Deze techniek noemt men de Köhler belichtingstechniek en het belangrijkste hierbij is dat de lichtbundel goed is afgesteld tussen 2 lenzen (die in de voet van de microscoop en die van het objectief: fig. 15). Het spreekt voor zich dat wanneer men van objectief verandert er een heraanpassing nodig is aangezien elke lens anders is. Enkele belangrijke eigenschappen van lenzen spelen hierbij een grote rol aangezien deze het licht komende van het specimen moeten opvangen.

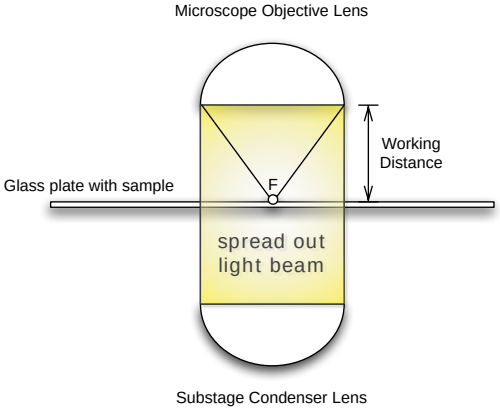


Figure 15: De uitgesmeerde lichtbundel, het focus punt F en de werk afstand.

De *werk afstand* tussen de lens in de tip van het objectief en het specimen is de afstand waarbij het object in focus is, ook wel de focus afstand f genoemd. Hoe hoger de versterkingsfactor van het objectief, hoe kleiner de werk afstand wordt. Ook de openingshoek waarbij dit gebeurt speelt een rol en wordt gedefinieerd als

$$a = 2 \arctan\left(\frac{D}{2f}\right) \tag{3}$$

met D de diameter van lens. Deze factor is een dimentieloos getal dat bepaalt in welke mate de lens in staat is licht op te vangen van het specimen. Afgeleid van deze openingshoek is de *numerieke apertuur* welke altijd gegraveerd is op het objectief. De numerieke apertuur bepaalt meestal het immersie medium dat gebruikt dient te worden (lucht, water, olie, ...). In lucht (brekingsindex $\alpha = 1$) is deze factor gedefinieerd als

$$NA = \alpha \sin\left(\frac{a}{2}\right) \tag{4}$$

Dit alles is grafisch voorgesteld in figuur 16.

Als de belichtingstechniek juist is toegepast en de optica juist is ingesteld kan men microscopische beelden verkrijgen van hoge kwaliteit (fig. 1.29). Zo is er minder nood aan bewerkingen achteraf om een bruikbaar beeld te verkrijgen, wat altijd een goede zaak is.

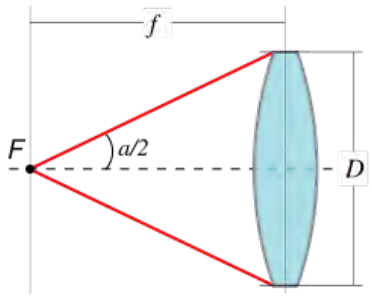


Figure 16: De openingshoek van de lens e.a. lens factoren.

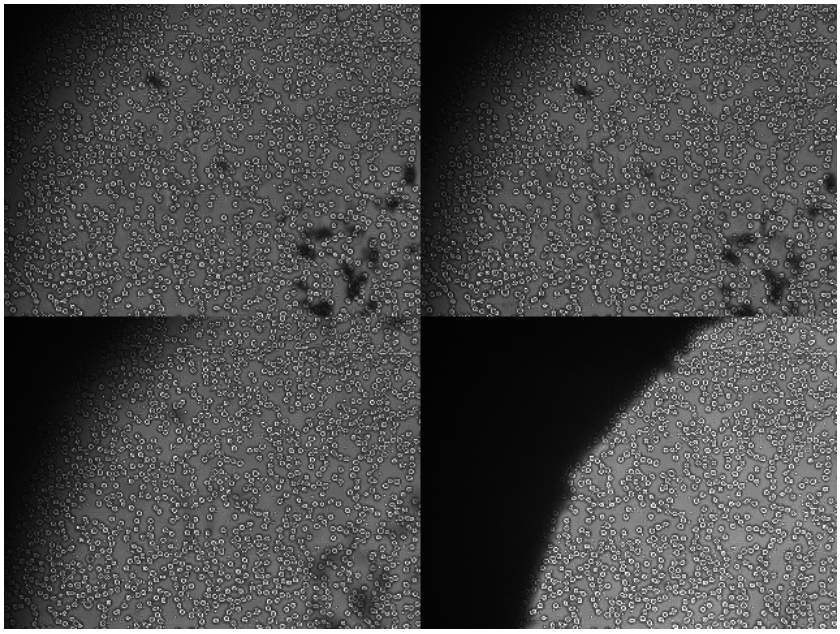


Figure 17: Reductie van stof en krassen door middel van Köhler belichting.

Optische verbinding camera

Een fenomeen waar we bij het koppelen van de camera met de microscoop rekening mee moeten houden is het voorkomen van lichtvlekken. Dit gebeurt door ongewenste reflecties en lichtbreking door het niet goed monteren van de camera adapter. De camera wordt op de adapter gevezen en dan bovenin de microscoop geplaatst. De hoogte is echter regelbaar zodat de juiste afstand tussen de lens van de adapter en die van de microscoop gevonden kan worden. Als deze hoogte juist is afgesteld kan men lichtvlekken zoals in figuur 2.9 vermijden en verkrijgt men een zuiver beeld zoals dat in figuur 19. De afstelling gebeurt zoals in figuur 2.6.

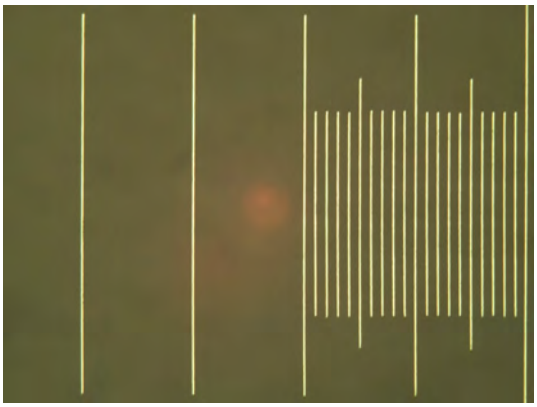


Figure 18: Lichtvlekken.

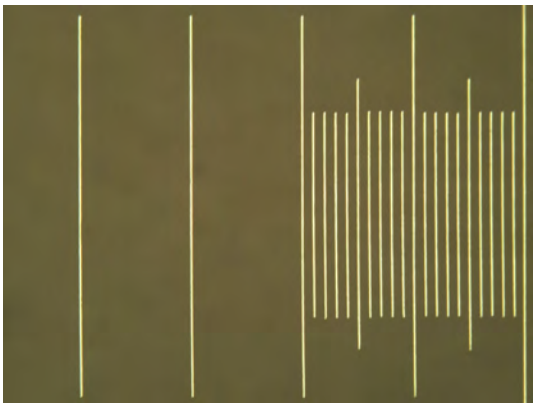


Figure 19: Geen lichtvlekken.

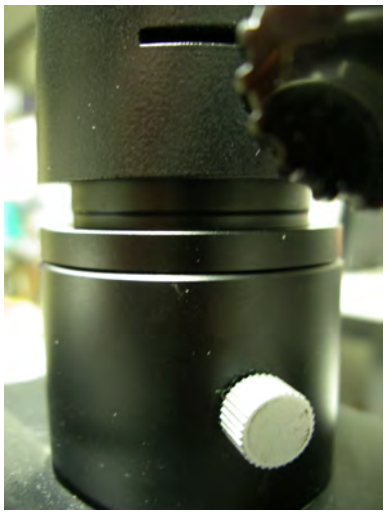


Figure 20: Aanpassen van de optische koppeling tussen camera en microscoop.

Filters

Er bestaan verschillende filters die gebruikt kunnen worden in microscopie om het beeld nog verder te verbeteren. Kleurenfilters attenueren bepaalde golflengtes uit het spectrum wat handig kan zijn als een bepaalde kleur overvloedig aanwezig is in het beeld. In het geval van overbelichting kunnen grijsfilters gebruikt worden, welke alle golflengtes (wit licht) even hard attenueren. Deze methode is beter dan het voltage naar de lamp te verlagen aangezien dit in extremis resulteert in een meer rood gekleurd licht. Een derde soort zijn de filters die warmte kunnen absorberen wat ervoor zorgt dat het specimen onder de microscoop niet te snel beschadigd raakt. Een voorbeeld van een blauwe kleurfilter, welke zowel rood als groen attenueert in het originele beeld (fig. 21), is te zien in figuur 22.

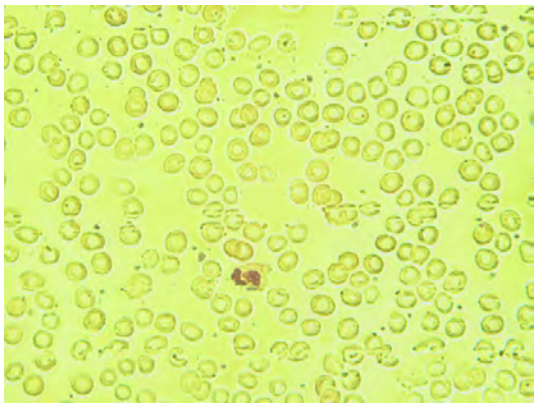


Figure 21: Beeld van bloedcellen zonder kleurenfilter.

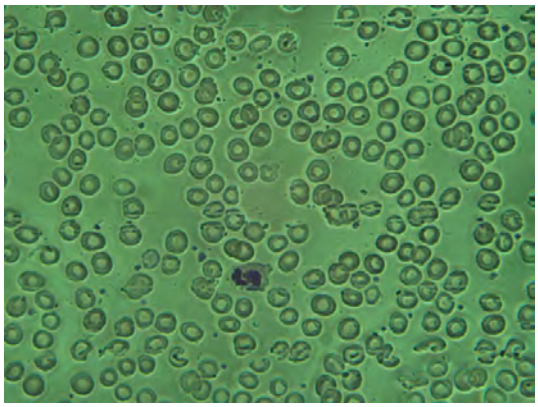


Figure 22: Hetzelfde als figuur 21 maar dan met blauwe kleurenfilter.

Klassieke en reflectie microscopie

In klassieke optische microscopie wordt het licht van onderuit *door* het specimen gestuurd waarbij de bundel verder propageert naar boven toe, door het objectief e.a. optica tot aan de oculairen of de camera. Deze methode werkt uiteraard enkel voor specimens die doorschijnend zijn en dus niet voor bijvoorbeeld materiaalkundige samples zoals hout, metaal of IC's. Om zulke samples toch te kunnen bekijken heeft men reflectie microscopie uitgevonden. Licht wordt langs bovenaf op het specimen gericht en de reflecties worden vervolgens opgevangen door het objectief, wat resultaten geeft zoals in figuur 23.

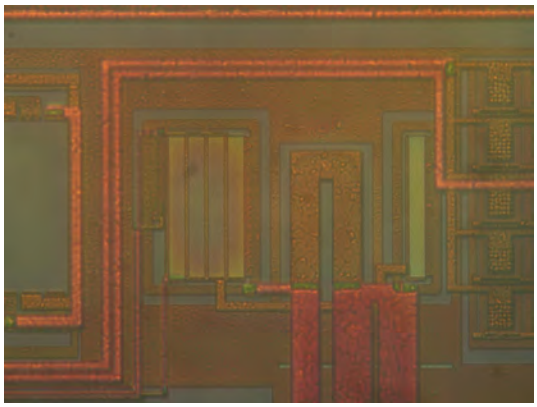


Figure 23: Foto van een IC genomen via reflectie microscopie.

Donkerveldmicroscopie

Wanneer een specimen op een normale manier belicht wordt, hetzij in klassieke of in reflectie microscopie, spreekt men ook soms van *lichtveldmicroscopie*. Andere technieken bestaan ook, één daarvan is *donkerveldmicroscopie*. In deze belichtingstechniek worden de centrale stralen van de lichtbundel geblokkeerd door middel van een lichtstop (fig. 24).

Verder is de lichtbundel ook niet meer uitgesmeerd zoals dit bij de Köhler methode het geval was. Doordat er dus geen lichtstralen rechtstreeks door het specimen gaan krijgen we in eerste instantie een donker beeld. Daar er toch enkele lichtstralen schuin op het specimen invallen krijgen we diffractie, refractie en reflectie fenomenen die zich vooral manifesteren aan de randen en de uitgesproken silhouetten van een specimen. Dit resulteert in een beeld met een hoog contrast, zeker tegenover de donkere achtergrond, zelfs bij specimens die van zichzelf kleurloos zijn. Donkerveldmicroscopie kan toegepast worden bij klassieke en reflectie microscopie. Een vergelijkend voorbeeld is te zien in figuur 25.

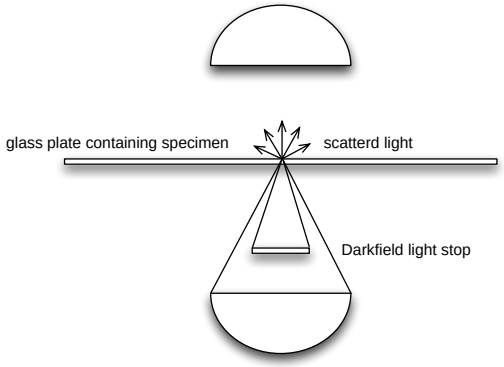


Figure 24: Principe van donkerveldmicroscopie.

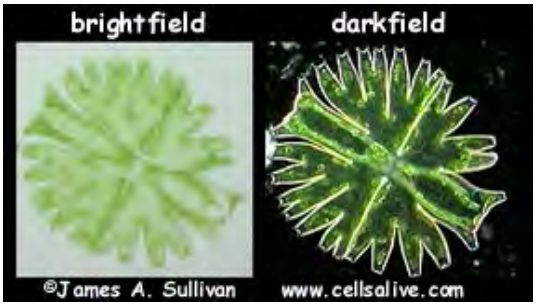


Figure 25: Een vergelijking tussen lichtveld - en donkerveldmicroscopie.

Software

Zoals eerder vermeld is ons doel het creëren van een platform-onafhankelijke toepassing voor het vastleggen van beelden komende van IIDC compatibele camera's. Het resulterende programma kan stilstaande beelden opslaan, sequentiële beelden (op het ritme van een timer) en volwaardige video beelden. De gebruiker kan alle beeldresoluties, kleurenmodes en beelden per seconden instellen die de geselecteerde camera aankan. Verder zou het voor elke ingenieur of wetenschapper relatief eenvoudig moeten zijn om onze code uit te breiden met extra functionaliteit zoals zelfgeschreven algoritmen. Speciaal voor onze microscoop toepassing hebben we een calibratie functie ingebouwd waarbij de gebruiker een dradenkruis kan afmeten tegenover een micrometer plaatje onder de microscoop. Zo'n plaatje heeft graveringen op gekende afstanden, meestal enkele micrometers van elkaar (fig. 26).

De grafische user interface

We kozen C/C++ in combinatie met Qt voor dit project. Qt is in essentie een cross-platform framework voor het creëren van grafische gebruikersinterfaces maar het heeft ook vele andere voordelen. Het is een open source project tenzij men het gebruikt voor een commerciële applicatie, in dat geval moet een licentie gekocht worden. Een op Qt

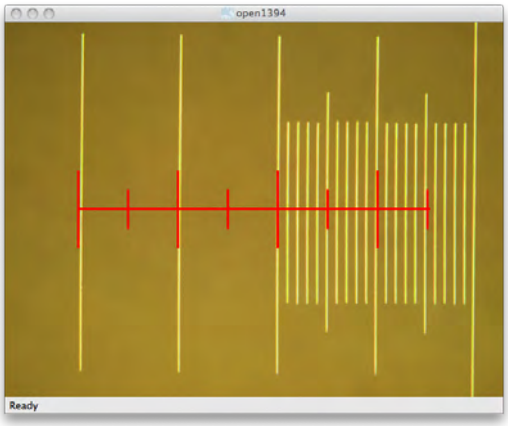


Figure 26: Calibratie voor metingen op microscopische schaal.

gebaseerde applicatie bestaat over het algemeen uit een verzameling van bestaande *widgets* of uitgebreide varianten van bestaande widgets. Een klein probleem doet zich voor bij het gebruik van de camera beelden in Qt. Intern wordt een kleurenbeeld in Qt voorgesteld als een matrix van vier niveaus (rgb32 - 0xffRRGGBB of argb32 - 0xAARRGGBB) wat niet hetzelfde formaat is waarmee de camera beelden uitstuurt. We zullen dus telkens een conversie moeten doen vanuit YUV4xx, mono of standaard rgb. Wanneer zo een conversie functie wordt geschreven als een gewone programmatorische lus wordt dit in essentie een operatie waarbij pixels gekopiëerd worden. Dit is een trage operatie, zeker voor hoge resoluties. Een tweede performantie probleem doet zich voor bij het tekenen van de beelden op het scherm. Doordat we de interne Qt engine gebruiken voor het tekenen verkrijgen we identieke resultaten op alle platformen maar dit is ook iets trager dan de platform-afhankelijke manier. Dit wordt echter pas een probleem bij hoge resoluties of veel beelden per seconde. Over het algemeen is dit niet zo'n groot probleem aangezien de camera's op hoge resoluties zelf niet veel beelden per seconden uitsturen. Ondanks het feit dat ons programma redelijk CPU intensief is blijft de grafische user interface toch heel responsief, zelfs op oudere machines wat niet onbelangrijk is.

Figure 27: Qt logo.

Verkrijgen van beelden

De communicatie met de camera en dus het verkrijgen van de beelden doen we via een open source C bibliotheek genaamd libdc1394. Het is een bibliotheek op een redelijk laag abstractie niveau maar geeft ons zo een flexibele toegang tot alle mogelijkheden die de IIDC standaard biedt. De bibliotheek bestaat voorlopig enkel voor Linux/OS X dus ons programma werkt op dit moment nog niet op Windows. In de nabije toekomst gaat dit echter veranderen want een Windows versie is gepland wat libdc1394 de beste en tevens enigste mogelijkheid maakt om aan cross-platform FireWire video capture te doen. Net zoals ons programma bestaat er reeds een GUI voor libdc1394 genaamd Coriander maar deze werkt enkel op Linux en is niet gebaseerd op Qt.

Opnemen van video

Om de beelden van de camera om te zetten naar video moeten we deze comprimeren en hiervoor is dus een encodeer bibliotheek nodig. We kozen hiervoor *libavcodec* wat een onderdeel is van het populair open source ffmpeg project (tevens multiplaform). We encoderen de beelden naar het MPEG1 formaat omdat elk systeem dit zonder problemen kan afspelen (extra codecs etc.) en dit ook het eenvoudigste te implementeren was. Nu is het zo dat MPEG encoders meestal als invoer YUV420 beelden eisen dus hier is weer een conversie nodig en dit bovenop de reeds rekenintensieve mpeg compressie. In sommige gevallen kan dit leiden tot framedrops (het niet tijdig een nieuw beeld opvragen aan de camera) wat lastig kan zijn. YUV420 wordt geprefereerd omdat het beter te comprimeren is doordat de pixels per kleurvlak samenzitten en een deel van de kleur informatie wordt genegeerd. Er bestaat ook een mogelijkheid om sequentieel opgenomen stilstaande beelden via ffmpeg om te zetten naar een filmpje, wat vooral bij traag evoluerende processen een betere oplossing is dan full-motion video.

Conclusie

We zijn erin geslaagd om beelden van de microscoop op te nemen in hoge kwaliteit door theorie over microscopie te combineren met programmeer vaardigheden. We vergaten ook de gebruiksvriendelijkheid van onze software niet uit het oog. Er zijn echter nog verbeteringen mogelijk, vooral op het gebied van performantie. Om alles vlotter te doen werken zouden de functies die de kleuren formaat transformaties doen moeten geoptimaliseerd worden. Indien mogelijk kan een betere manier om de beelden op het scherm te tonen gezocht worden. Ook kan het gebruik van multithreading in de software verbeteringen opleveren door bijvoorbeeld een thread volledig te wijden aan de interactie met de camera zodat het verliezen van frames geminimaliseerd wordt.

Contents

List of Figures

Chapter 1

Introduction to optical microscopy

The first chapter of this book enlightens the reader about the basics of optical microscopy, ranging from the interaction with the human eye to more advanced subjects like types of illumination, digital photomicrography, color filters, etc. Our goal here is to gain some basic knowledge of microscopic principles that are mentioned in other parts of this book and therefore this chapter is more like a summary. If you are interested in learning more about these subjects Michael W. Davidson [1] and Mortimer Abramowitz [2] have some great books online in cooperation with microscope manufacturer Olympus. Michael W. Davidson also maintains the Molecular Expressions website at Florida State University [3]. Nikon Inc., another well known microscope manufacturer, also has a fine website [4] in cooperation with Molecular Expressions.

1.1 Human eye perception

We remember from basic physics or biology that the human eye can perceive wavelengths between roughly 400 and 750 nm in the electromagnetic spectrum (fig. 1.1). These wavelengths represent the colors violet and red respectively and any color in between. Therefore the microscope must emit light in this visible region or make wavelengths outside this region visible with techniques like fluorescence microscopy. Furthermore the eye can also sense differences in brightness ranging from black to white and all grays in between.

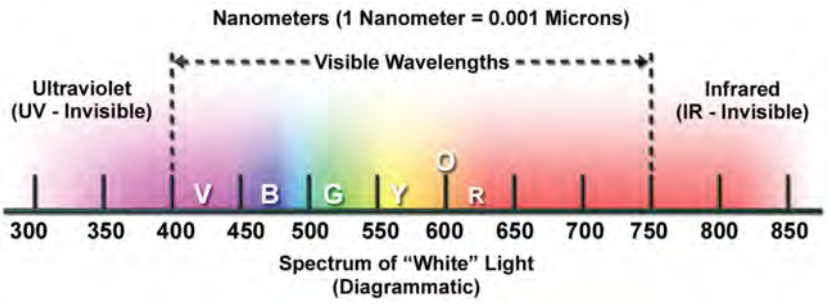


Figure 1.1: The electromagnetic spectrum (Molecular Expressions ©).

The part of the eye responsible for detecting color and various levels of brightness is the retina (fig. 1.2 and fig. 1.3). The retina (opposite to the lens) is composed out of *rod cells* that sense brightness and *cone cells* which are sensitive to colors. The cones can be divided into three types where about 64% of the cones are red-sensitive, 32% are green sensitive and 2% are blue sensitive. Their relative numbers really don't say much as the cones have different grades of sensitivity to bring the colors back into proportion. The blue cones are the most sensitive and this partially makes up for their disadvantage in numbers, still yellow is the color we see the best [6]. A good website about this subject is the one from Georgia State University [5].

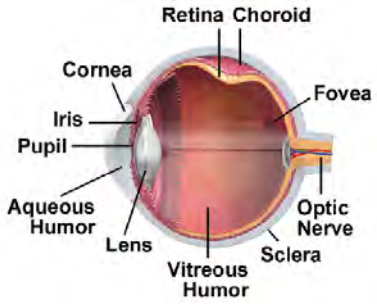


Figure 1.2: Intersection of the human eye (Molecular Expressions ©).

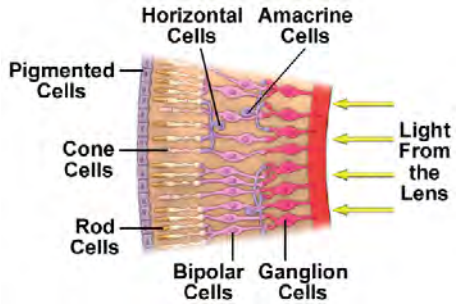


Figure 1.3: The retina in detail (Molecular Expressions ©).

Now why is this important to us? First lets have a look at the distribution of rods and cones on the retina. At the center of the retina, a couple of degrees of the optical axis, we find the central fovea (fig. 1.4). Also known as "the yellow spot" the central fovea is the area of sharpest vision and lies on the visual axis (0° in figure 1.5). This is because it has the highest density of rods and cones on the entire retina, resulting in a maximum resolution of space (spatial resolution[1] [8]), contrast, and color. Outside the center of the fovea cone density decreases very steep. Rods peak at the periphery of the fovea and decrease gradually with the angle.

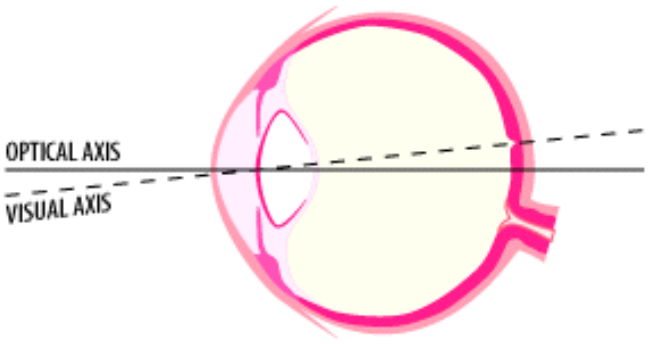


Figure 1.4: Optical vs. visual axis (Adobe ©[6]).

[1]Spatial resolution is a measure of how closely lines can be distinguished in an image, generally expressed as line pairs per millimeter [lp/mm].

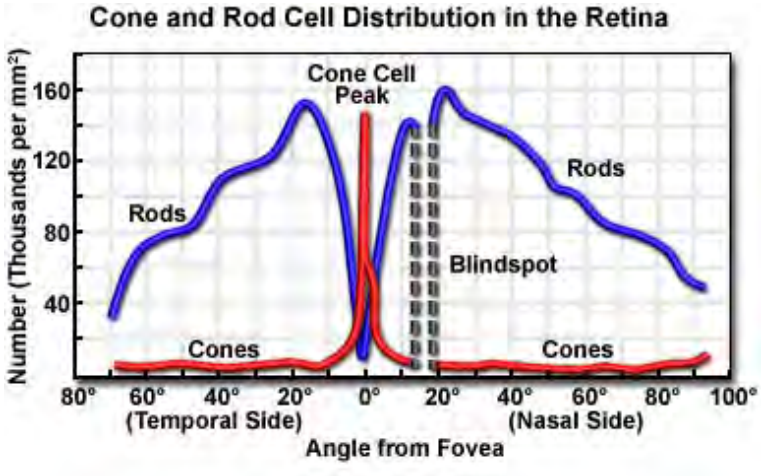


Figure 1.5: Distribution of receptor cells on the retina (Molecular Expressions ©).

The distance between an object and the eye influences the angle at which the bundle of reflected light coming from the object is "spread" onto the retina. Due to the lens of the eye the angle is opposite proportional to the distance. Hence an object too close to the eye will lose it's detail as a bigger part of the reflected light that enters the eye is captured by the less populated part of the retina. This is illustrated in figure 1.6. The eye's lens can compensate for this effect by changing it's shape, but only to a limited degree. How microscopes solve this problem is explained in the next part of this chapter.

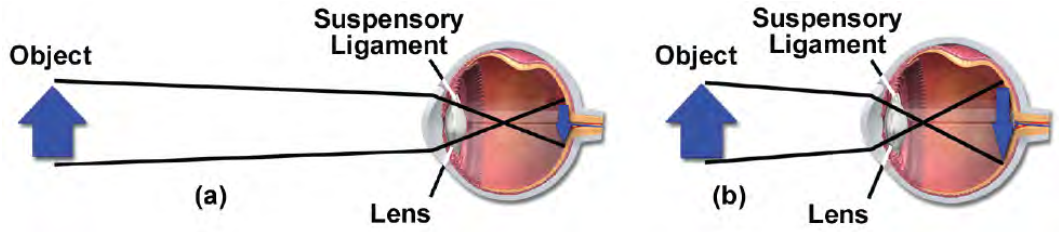


Figure 1.6: Relationship between object distance and visual angle on the retina, (a) object is far away and (b) object is close (Molecular Expressions ©).

1.2 Principles of magnification

As the conventional viewing distance of the human eye is limited by the effect described above, a solution for viewing small objects up close is needed. Let us first have a look at how a simple magnifying glass works (fig. 1.8). By placing a convex lens[2] between the eye and the object the reflected light coming from the object falls onto the retina at a

[2]A convex lens is a lens which is spherical at both sides or in other words thicker in the center than at the periphery.

sufficiently small angle. This does not only mean preservation of detail but it also means the brain is being "fooled" in thinking that the object is actually larger and farther away then it really is. To get this virtual image into focus the lens has to be positioned correctly between the eye and the object, centered and at the right focal distance f (fig. 1.7).

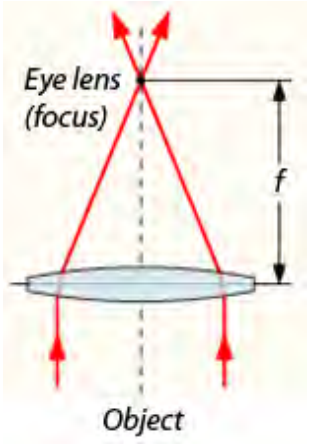


Figure 1.7: Focal length between the eye's lens and the convex lens.

Now combining the principle of a magnification glass with more lenses (for higher magnification) and a light source, we can construct a simple compound microscope. In figure 1.9 a basic version of such a microscope is shown. There's lots of different ways manufacturers constructed microscopes in the past and therefore variations of this layout exist. We can however divide the optical pathway into four important parts out of which every modern optical microscope is composed of. Generally a two stage magnification is used by combining the magnification of the *objective* with the one from the *eyepiece*[3]. The objective, which sits very close to the specimen, allows for very high magnification. It's magnification can range from 2X to +100X but depends on the used immersion medium. For the highest magnifications water or oil immersion is used but for lower values air is sufficient. More on this, including a summary of different types of objectives, can be found in section 1.4. The next step in magnification is the eyepiece which usually has a magnification lower or equal to 10X. Combining a 100X high magnification objective with a 10X eyepiece results in a maximum magnification of 1000X. Modern optical microscopes can magnify up 1500X, for higher magnifications an electron microscope is used instead. A third important part of the optical pathway is the microscopes *body tube* which connects the beam coming from the objective to the eyepiece(s) by the use of prisms and/or mirrors. Finally the *condenser* shines the light onto the specimen. The person looking trough the microscope isn't actually looking at the specimen itself but rather at an image of the specimen that is being projected inside the eyepieces, right in front of the eye. However, it's safe to say that the image will be an accurate representation of the specimen when the equipment is used correctly. Next we will discuss these parts in more detail.

[3]An eyepiece is also sometimes referred to as an ocular, we will use the former designation in this book.

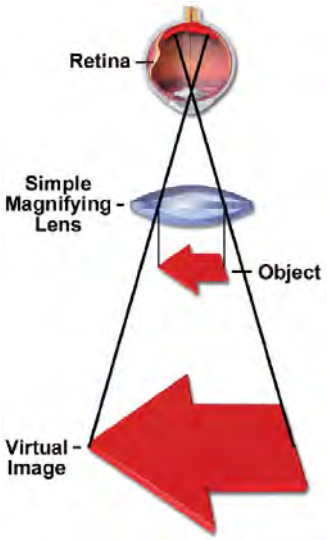


Figure 1.8: Working principle of a magnifying glass (Molecular Expressions ©).

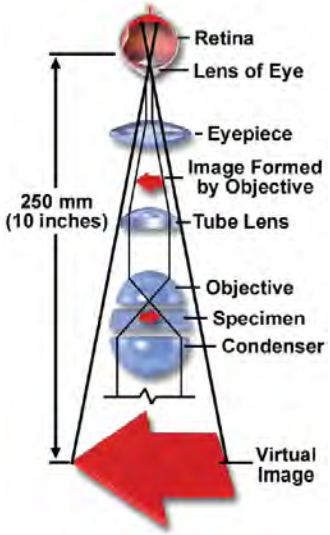


Figure 1.9: Layout of a basic compound microscope (Molecular Expressions ©).

1.3 Image Formation

After reviewing the basic principles of magnification we can now investigate other aspects in microscopic image formation. We will only introduce those properties that are needed to properly use and understand the equipment. The theoretical background will help us understand how image properties (contrast, detail, brightness,...) can be altered by different components of the microscope. Correctly using all of the microscope's features is essential, especially when trying to take photomicrographs of high quality (which is the aim of this thesis).

1.3.1 Aperture, Airy discs and resolution

When light passes through small spaces of a specimen or is reflected by it (see: section 1.5) and enters the microscope objective it scatters, yielding a diffraction pattern known as *airy discs* (fig. 1.10). This means that the resulting image at the end of the optical light path[4] is not constructed out of perfect points but rather out of a series of airy discs. Airy discs are points surrounded by concentric circles of different intensities. The smaller the angular aperture (see next paragraph), the bigger the working distance but the more closely the discs cling together having a negative impact on image quality. The ability to distinguish separate details in the image is called the *resolving power* and depends on whether or not these airy disc overlap (fig. 1.11). The resolving power can be enhanced by increasing the light-catching ability of an objective by enlarging it's angular aperture. For this reason objectives with large magnification factors (a high resolving power) are always meant to be used with specialized mediums because they have such high angular and numerical apertures.

[4]The eye of an observer, photographic film or a digital camera

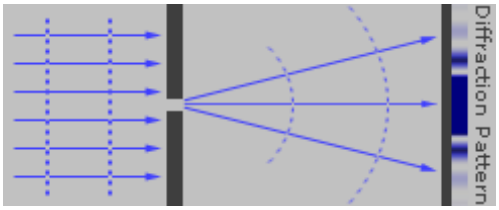


Figure 1.10: Diffraction occurrence when light passes through a specimen and enters the small opening of the objective.

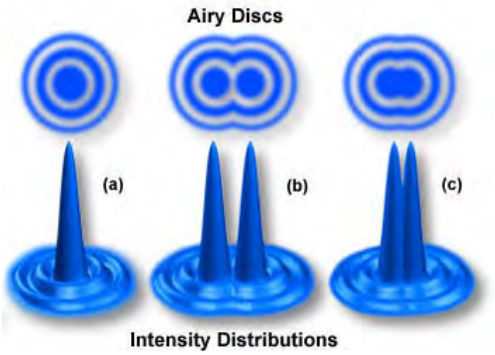


Figure 1.11: Airy disc diffraction pattern visualizing the resolving power (Molecular Expressions ©).

As the light coming from the specimen is scattered around in all directions the resolving power of the lens is directly subject to the amount of light it can capture. The angle at which the lens should captures the light is called the *angular aperture* and lies at the distance where the lens yields a sharp image of the specimen. The angular aperture depends on the convex of the lens and it's size. If we increase the curvature of the lens we simultaneously increase the angular aperture but we also decrease the working distance[5] (fig. 1.12). The reason for this is a change in focal distance which depends on the convex of the lens. Furthermore, if we make the lens bigger it can catch more scattered light and this automatically makes for a higher angular aperture (fig. 1.13).

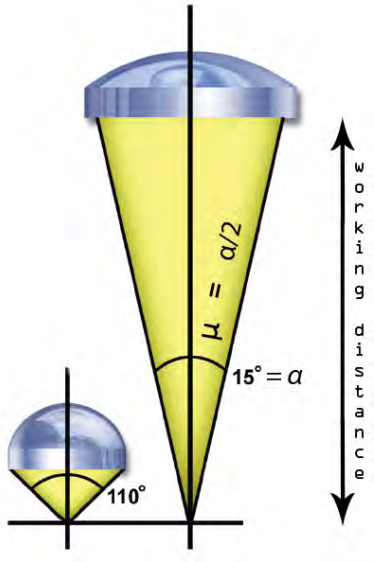


Figure 1.12: Angular aperture, working distance and lens curvature (Molecular Expressions ©).

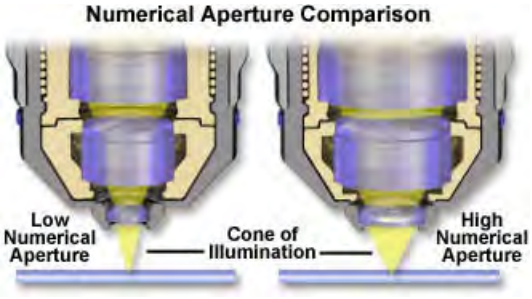


Figure 1.13: Angular aperture and the size of the lens (Molecular Expressions ©).

[5]The working distance being the distance from the objective front lens to the specimen surface.

Now we can determine the *numerical aperture* (N.A.) of a microscope objective which is expressed by the following formula:

$$NA = n \sin \mu \tag{1.1}$$

where n is the refraction index of the material between the specimen's cover glass and the lens and μ is half the angular aperture α of the first lens. The numerical aperture is always engraved onto the objective as it is relevant to what immersion medium should be used.

Let's look at two examples. Suppose we use an objective to be used with plain air as immersion medium (a "dry" objective) and a thin cover glass is used to protect the specimen, then we have a refraction index close to 1 (n = 1 for vacuum). With α's theoretical limit at 180 °, μ can never exceed 90 ° and $\sin \mu = 1$. This means the theoretical N.A. of a objective used in air is limited to 1. Of course α can never be 180 ° and air is not vacuum so generally it's less than 1. When using a specialized immersion oil as transfer medium with a refraction index of n = 1.515 a theoretical limit of N.A. = 1.5 can be obtained. Notice in figure 1.14 refraction when using immersion oil is negligible due to the fact that n = 1.5 for glass and therefore no noticeable transition is present between the two mediums.

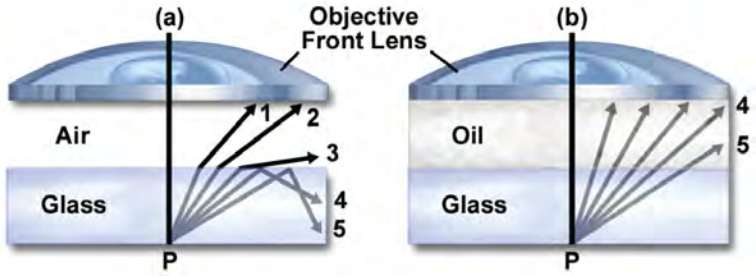


Figure 1.14: Refraction in different transfer mediums, (a) using air and (b) using a specialized immersion oil (Molecular Expressions ©).

Now that we've studied the numerical aperture of a lens we can link this to the previously discussed property *resolving power*. This ability to *distinguish separate details in the image* can be expressed by *resolution*. Resolution is defined as the actual distance between two distinguishable points and can be expressed as

$$r = \frac{\lambda}{2NA} \tag{1.2}$$

for non-luminous objects (Abbe) and for self-luminous objects (Raleigh) as

$$r = \frac{1.22\lambda}{2NA}. \tag{1.3}$$

These two formulas are easy to comprehend. A higher N.A. means a smaller resolvable distance r, a shorter wavelength λ also means a smaller r. Hence, a smaller r means a better resolution and thus strongly depends on the light used and the N.A. of the objective.

As we will see later in this chapter the total magnification of a microscope objective is rather complicated and uses many lenses. However the *useful magnification* is limited due to the diffraction problem discussed above. In general the limit is 1000X the numerical aperture of the objective as higher magnification will not yield more detail (e.g. for N.A. = 0.25 a good magnification of 250X can be obtained). Therefore high magnification objectives usually use a high angular aperture lens. In general, the objective working distance decreases as the magnification and numerical aperture both increase. It is however good practice to combine eyepieces to further magnify the image as otherwise the working distance and size of the lens would be too excessive. Modern microscopes can obtain a total magnification of 1500X (objective x eyepiece(s)) with a theoretical resolution of around 0.2 micrometers (diffraction limit).

1.3.2 Eyepieces and Camera Adapters

Eyepieces are used to further magnify the image coming from the objective and to make it visible to the human eye. Eyepieces are sometimes called oculars and different optical designs exist. They can generally be divided into two main categories, the negative and positive eyepieces. Negative eyepieces are the most uncomplicated and are also called Huygenian eyepieces. When an eyepiece only has it's magnification inscribed on the housing it's most likely an Huygenian eyepiece. In their simplest form they are not corrected for optical aberrations and are intended to be used with the cheaper achromat objectives (see: 1.4). The other main eyepiece design is the Ramsden design, also known as positive eyepieces (fig. 1.15).

Figure 1.15: Two main eyepiece designs, negative (Huygenian) and positive (Ramsden) (Molecular Expressions ©).

Both basic Huygenian and Ramsden eyepieces have two lenses and a built-in diaphragm but variations on these designs exist when more lenses are added for extra optical correction. Examples of optically corrected eyepieces are the *Plan* or the even better *Periplan* designs. Just like *plan* objectives they are designed to enhance the size and flatness of the viewfield. In Huygenian eyepieces the curved side of the field lens is pointed downwards and the diaphragm sits in the middle. For the Ramsden design the diaphragm sits at the bottom and the curved side of the field lens is pointed upwards.

For measurements on the microscopic scale reticles can be mounted on top of the eyepieces (fig. 1.16). In order to measure correctly you will need a reference, generally a stage micrometer. A stage micrometer is a microscope slide with lines engraved at very small but accurate distances (fig. 1.17). It might be a good idea to implement reticles in our imaging software simply as an overlay on the digitalized photomicrograph (for applications were measurement is important).

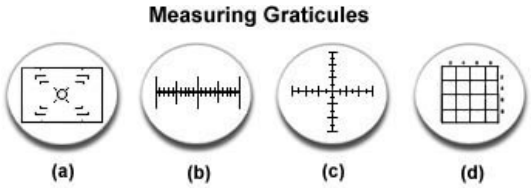


Figure 1.16: Different kinds of reticles for measurement (Molecular Expressions ©).

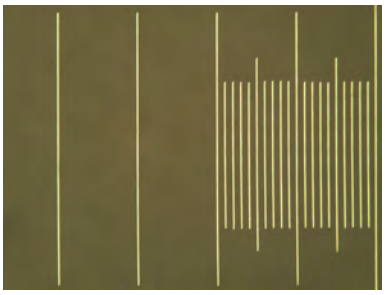


Figure 1.17: Stage micrometer that came with a Leitz Ergolux microscope, engravings respectively 10 and 100 μm apart.

A specialized form of eyepieces are the photo eyepieces, also called projection lenses (fig. 1.18). Since they are used to *project* an image onto a CMOS sensor, CCD or plain photographic film they must produce perfectly flat-field images. They can also be chromatically corrected for color microphotography but this will only give good results when they are combined with a color corrected objective. Just like normal eyepieces they also have a magnification factor but do not yield an image that is clearly visible to the human eye. They are used together with a *camera adapter* (fig. 1.19) to mount the camera on top of the microscope (fig. 1.20). It is also possible to couple a camera to an eyepiece using special adapters but since the eyepieces are designed for the human eye this does not produce optimal results (fig. 1.21). Therefore this is only done when no base to mount the camera adapter is present on top of the microscope.

1.3.3 Conjugate planes

In order to understand the workings of the condenser in the next section and to comprehend Köhler illumination (see: section 1.5.1) later on we will now take a short look at *conjugate planes* in the optical microscope. Conjugate planes are a set of focal planes in an optical system that are simultaneously in focus (fig. 1.22). We can theoretically define two optical paths in a microscope, the illumination path and the image-forming path. For example the specimen plane is in focus at the image-forming path, this is logical because you want the image of the specimen to be in focus. The light used to illuminate the specimen may not be in focus at this plane as this will introduce image artifacts ascribed to dust and scratches in the illumination path. Being that the illuminating rays are not in focus in the same plane as the specimen results in a evenly distributed beam of light yielding proper illumination over the entire viewfield of the specimen. All this will become clearer as we will look at some examples in the next section.

Figure 1.18: Photo eyepieces also known as projection lenses (Molecular Expressions ©).

Figure 1.19: A camera adapter to be mounted on top of the microscope (Molecular Expressions ©).

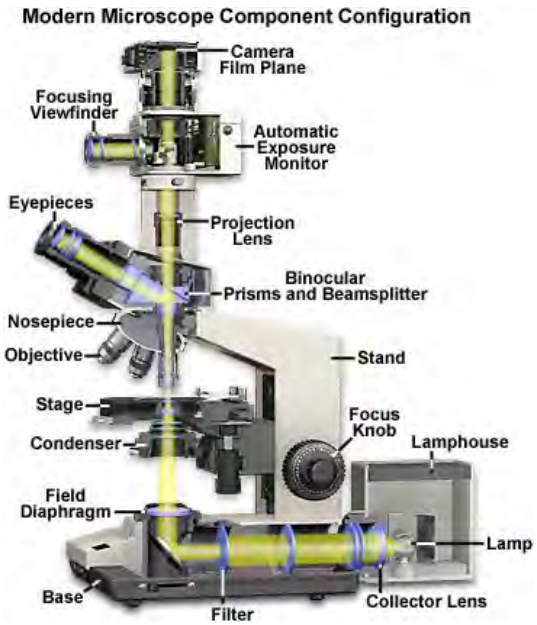


Figure 1.20: Camera mounted on top with an adapter (incl. the projection lens) (Molecular Expressions ©).

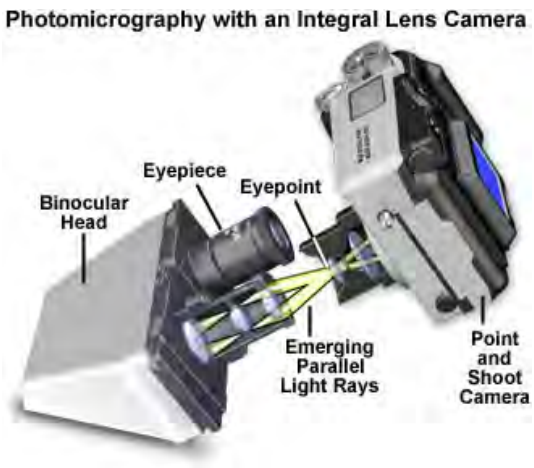


Figure 1.21: Camera coupled to the eyepieces, a simple solution (Molecular Expressions ©).

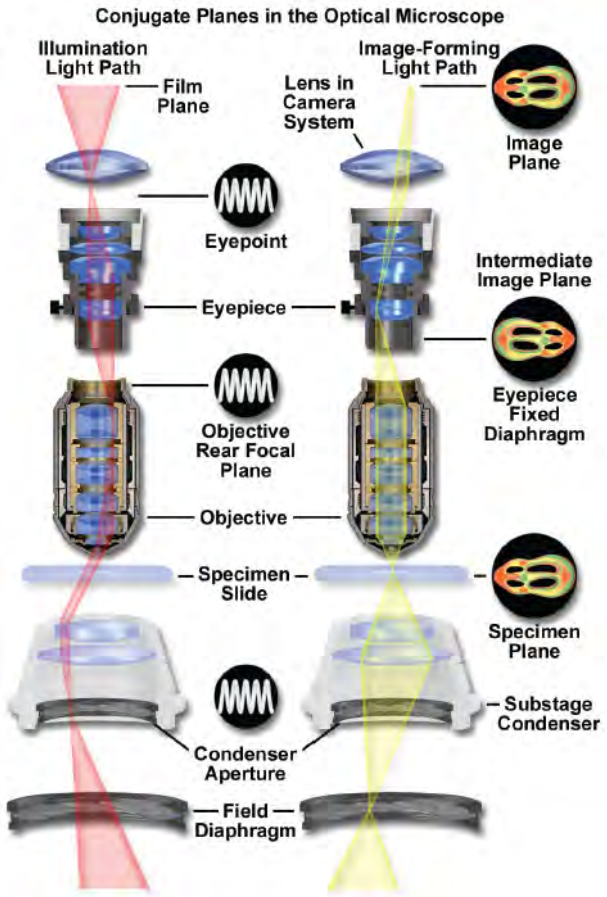


Figure 1.22: Conjugate focal planes in an optical microscope (Olympus ©).

1.3.4 Substage Condenser and Diaphragms

Correctly working with the substage condenser (especially in transmitted light) is essential for obtaining good photomicrographs. The substage condenser gathers light from the microscope's light source and concentrates it into a cone of light that illuminates the specimen with uniform intensity over the entire viewfield. Two main designs exist (fig. 1.23). The least corrected substage condenser is the Abbe condenser. More expensive is the Aplanatic-Achromatic condenser well corrected for chromatic and spherical aberration. As we've seen in the theory about *numerical aperture* different objectives require different light cone setups because of their distinctive light-catching abilities. The design of the substage condenser can also play a part in this as seen in figure 1.24.

Every time the microscope objective is changed some adjustments to the condenser will have to be made. This is because the condenser directly influences the *numerical aperture* of the system (see: section 1.3.1). For full usage of the objective the N.A. of the condenser should match that of the objective. This can be shown when looking at the formula for the total N.A of the system

$$NA_{system} = \frac{NA_{objective} + NA_{condenser}}{2}. \tag{1.4}$$

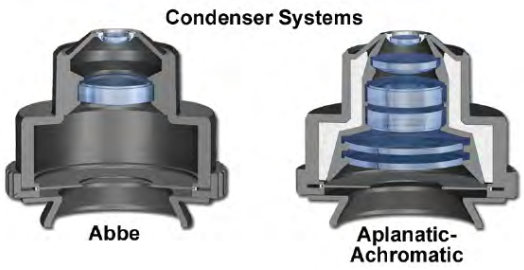


Figure 1.23: Two main substage condenser designs (Molecular Expressions ©).

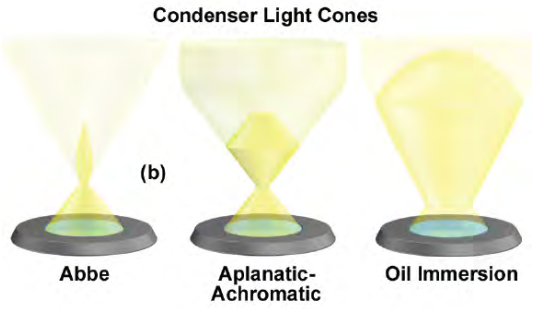


Figure 1.24: Resulting light cones of the different designs (Molecular Expressions ©).

The useful N.A. of the system is however limited to the N.A. of the objective so there is no use in making the $NA_{condenser}$ bigger than $NA_{objective}$. Generally the higher the N.A. of the objective the wider the light cone. The numerical aperture of the substage condenser can be altered by opening and closing the aperture diaphragm located at the base of the condenser (fig. 1.25). Note that in figure 1.25 the image-forming light path is shown and not the illumination light path

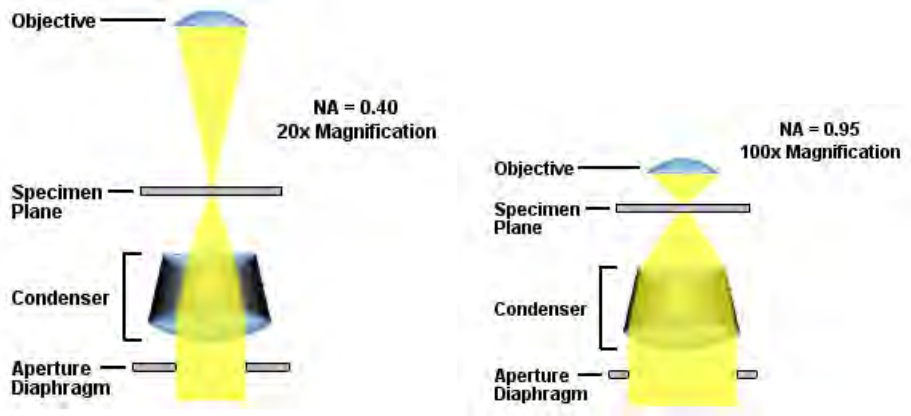


Figure 1.25: The substage condenser influences the numerical aperture of the system (Molecular Expressions ©).

Let's go over the steps to obtain a decent photomicrograph using the Köhler illumination technique[6] with the Leitz Ergolux microscope we use at the lab.

- Fine focus the microscope on the specimen.
- Close the *field diaphragm* in the base of the microscope completely until it is visible, most likely as a blurry circle (fig. 1.26(a)).
- By adjusting the *height* of the condenser you can fine-focus this silhouette (y-direction) (b).

[6] Another small tutorial is found on: http://microscopy.berkeley.edu/courses/TLM/condenser/kohler.html.

- With the two screws (or knobs) you can center it (x/z-direction) (c).
- Open the field diaphragm (fig. 1.27) just enough so that its edges are just beyond the field of view (d).
- Finally you can adjust the *aperture diaphragm* in the condenser (fig. 1.28) to alter contrast.

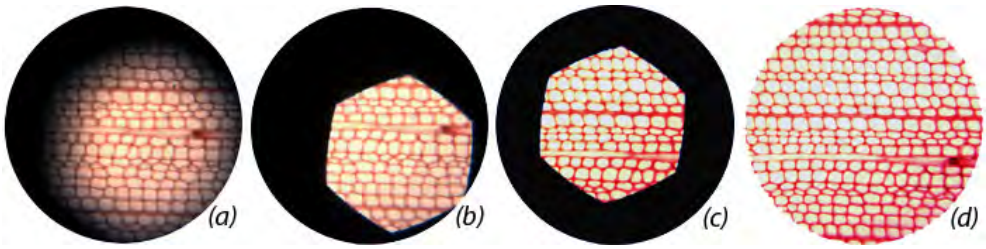


Figure 1.26: Proper adjustment of the substage condenser for Köhler illumination.

Figure 1.27: Field diaphragm in the base of a Leitz Ergolux microscope.

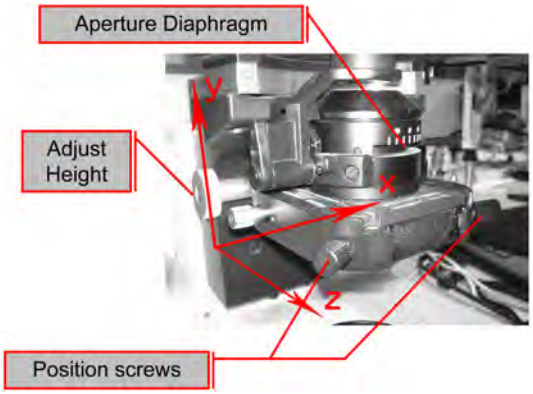


Figure 1.28: The Leitz Ergolux' substage condenser in detail.

Bringing the field diaphragm's silhouette into fine-focus has an apparent reason. In figure 1.22 you can see that both the field diaphragm and the specimen belong to the *image-forming path*. This is the working principle of Köler illumination (see: 1.5.1). It results in a light beam that's not in focus at the same plane as the specimen, clearly reducing artifacts attributed to dust and scratches in the light path. This can be seen in the following set of images (fig. 1.29). When a swing-lens is used as the top lens of the condenser, removing it will yield a change in the light's focal plane and readjustment will be needed (fig. 1.30 and fig. 1.31).

Top lenses that can swing out are used for objectives with low magnification (5X or lower) since Köhler illumination does not apply at such low magnifications (see: 1.5.1). Therefore some substage condensers are equipped with such a swing lens to remove the top lens from the optical pathway (fig. 1.32). In this case the aperture diaphragm no longer controls the numerical aperture and should be set to it's maximum. Instead the field diaphragm will be used to control the illuminating rays and is therefore responsible for brightness and contrast.

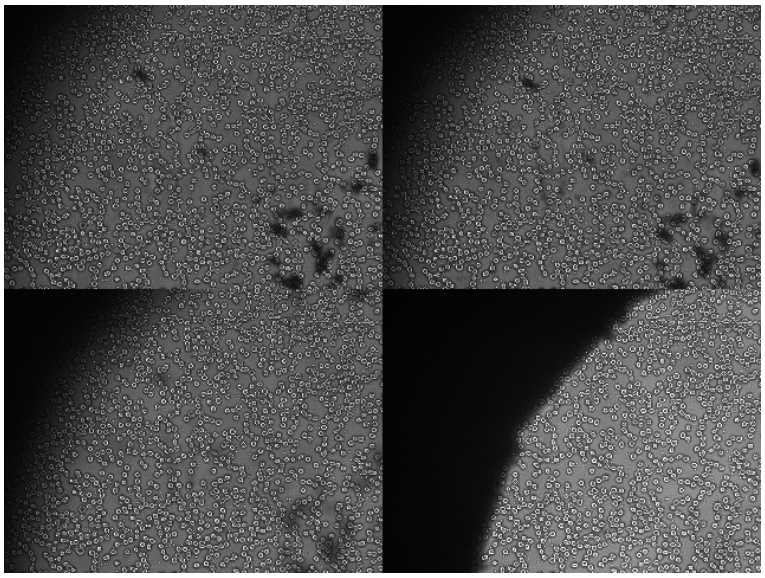


Figure 1.29: Reducing occurrences of dust and scratches with Köler illumination.

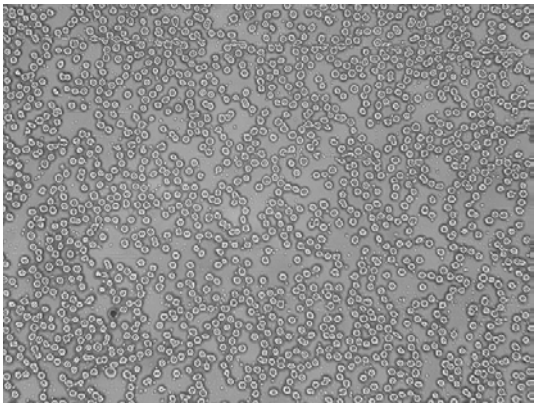


Figure 1.30: With the top swing-lens, illumination path not in focus with the specimen.

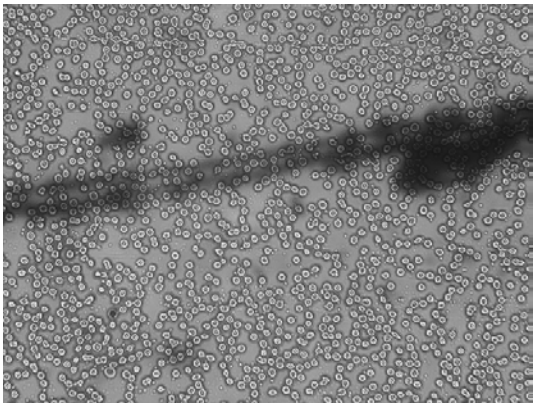


Figure 1.31: Without the top swing-lens, artifacts visible and readjustment needed.

Figure 1.32: Swing-lens condenser for high and low magnifications (Molecular Expressions ©).

1.3.5 Depth of Field

Just like in classic photography we can define a *depth of field* for optical microscopes. The depth of field is defined as the distance in where a certain object remains in focus (fig. 1.33). Generally the higher the magnification of the objective, the smaller the depth of field will be. Because this depth of field is very small in microscopy we have trouble getting the entire specimen into focus when working with high magnifications.

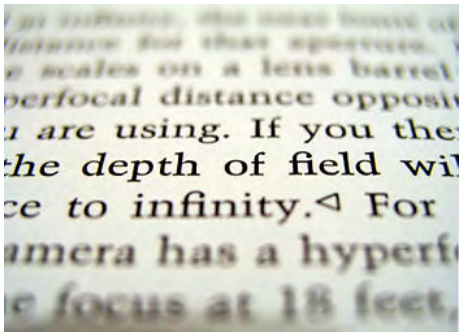


Figure 1.33: Depth of field in classic photography.

In microscopy, especially transmitted light microscopy, a lot of specimens are translucent. This allows us to focus onto different depths of the specimen as if it were slices. It's up to the researcher/scientist/engineer to make the right decision for the depth of field and this decision depends on what part of the specimen he or she is interested in. Let's look at an example of red blood cells. All images are taken with a 50x/0.85 NPL Leitz Fluotar $\infty/0$ dry objective in combination with an 8X Periplan photo eyepiece and coupled to an AVT industry-grade FireWire camera. Furthermore a blue color filter was used to enhance image quality and contrast (see: 1.3.6).

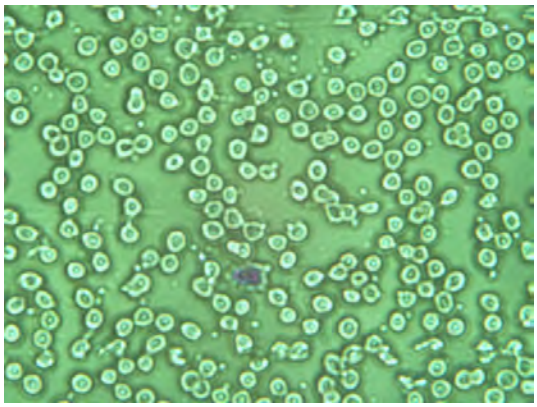


Figure 1.34: Cells completely out of focus.

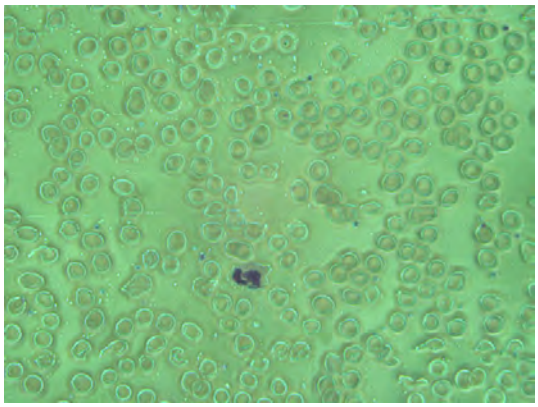


Figure 1.35: top of the cells into focus.

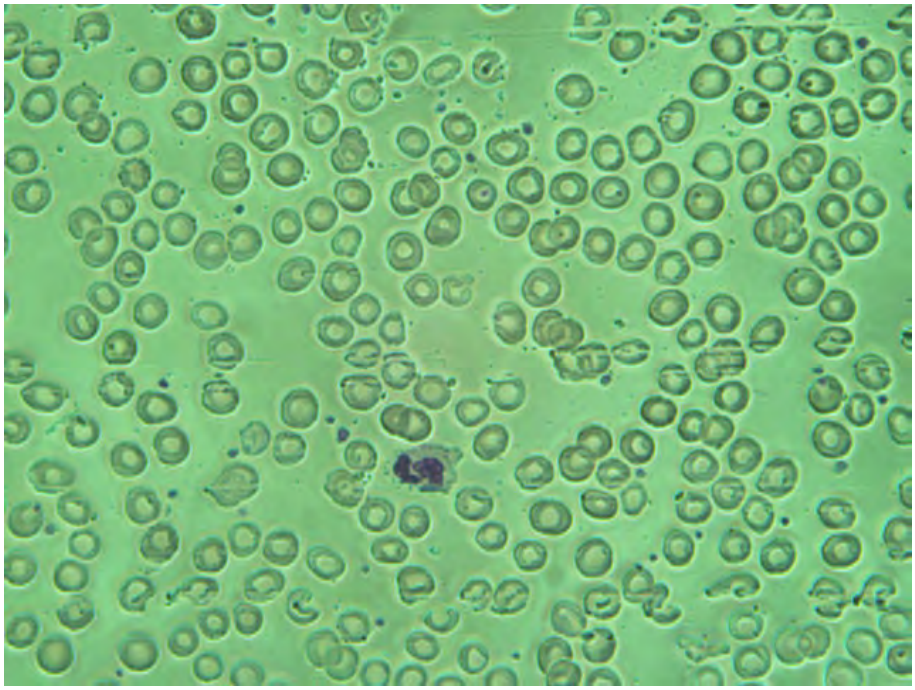


Figure 1.36: Center of the blood cells in focus, their defining donut shape can now be clearly distinguished.

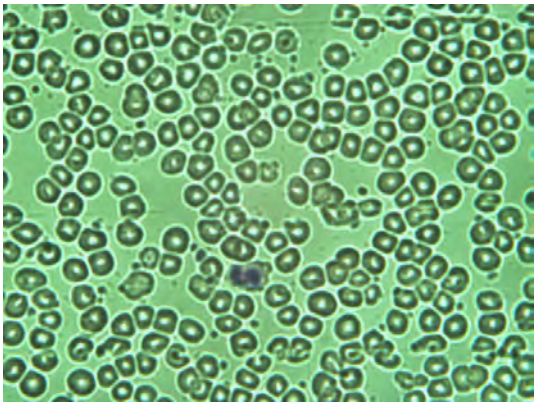


Figure 1.37: Bottom of the cells in focus, you can now see the silhouette of the cells clearly and how they are connected to their neighboring cells.

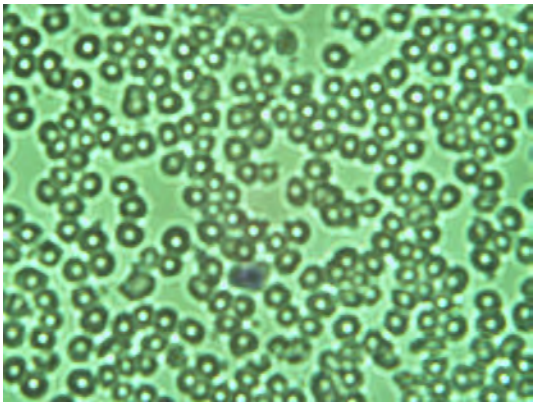


Figure 1.38: Cells again completely out of focus.

1.3.6 Filters

Photomicrography is susceptible to unexpected color shifts attributed to a wide range of complicated causes ranging from *chromatic aberration* (see: 1.4.3) to light source voltage fluctuations. Manufacturers therefore provide a whole range of different filters for example color compensation filters, neutral density filters, heat-absorbing filters, etc. Color compensation filters attenuate a certain color in the spectrum which can be used to filter a color that's excessively present or to enhance contrast. Neutral density filters are used in situations where the illumination appears to bright for comfortable observation. These gray filters reduce transmitted light evenly over all wavelengths. A special type of filter is the heat-absorbing filter. As 90% of radiation emitted by a tungsten light source is in the infrared region this manifests itself as heat which can damage the specimen. Heat-absorbing filters can therefore be placed in the light path, generally close to the light source. Now let's look at a color compensating filter example.

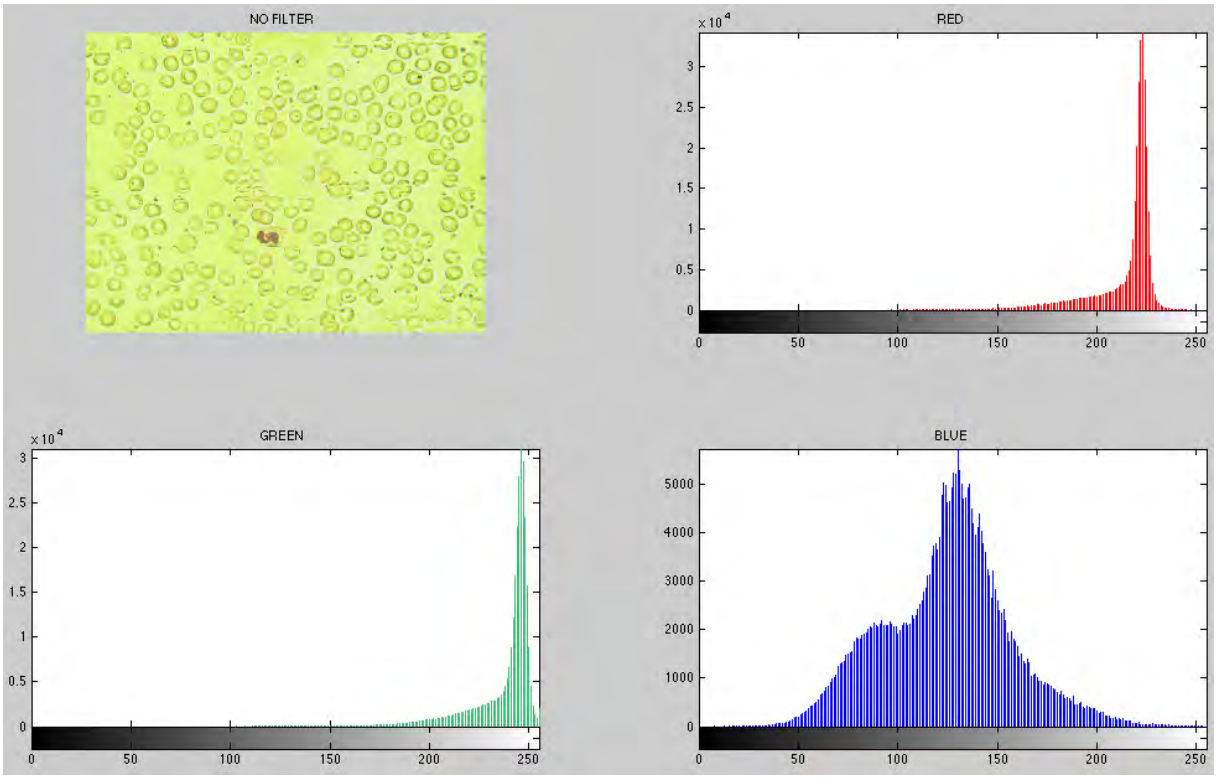


Figure 1.39: Bloodcells in Köhler illumination without a color filter.

Without any color filter applied it's obvious that red and green present a far larger portion of the pixels in the image than blue. You can see this clearly in the color histograms (fig. 1.39). To balance this out we can use a *blue filter* which will attenuate red and green but will pass (or even slightly amplify) the blue part of the spectrum. This will result in an even distribution of red, green and blue pixels an thus resulting in a good photomicrograph with proper brightness and good contrast (fig. 1.40).

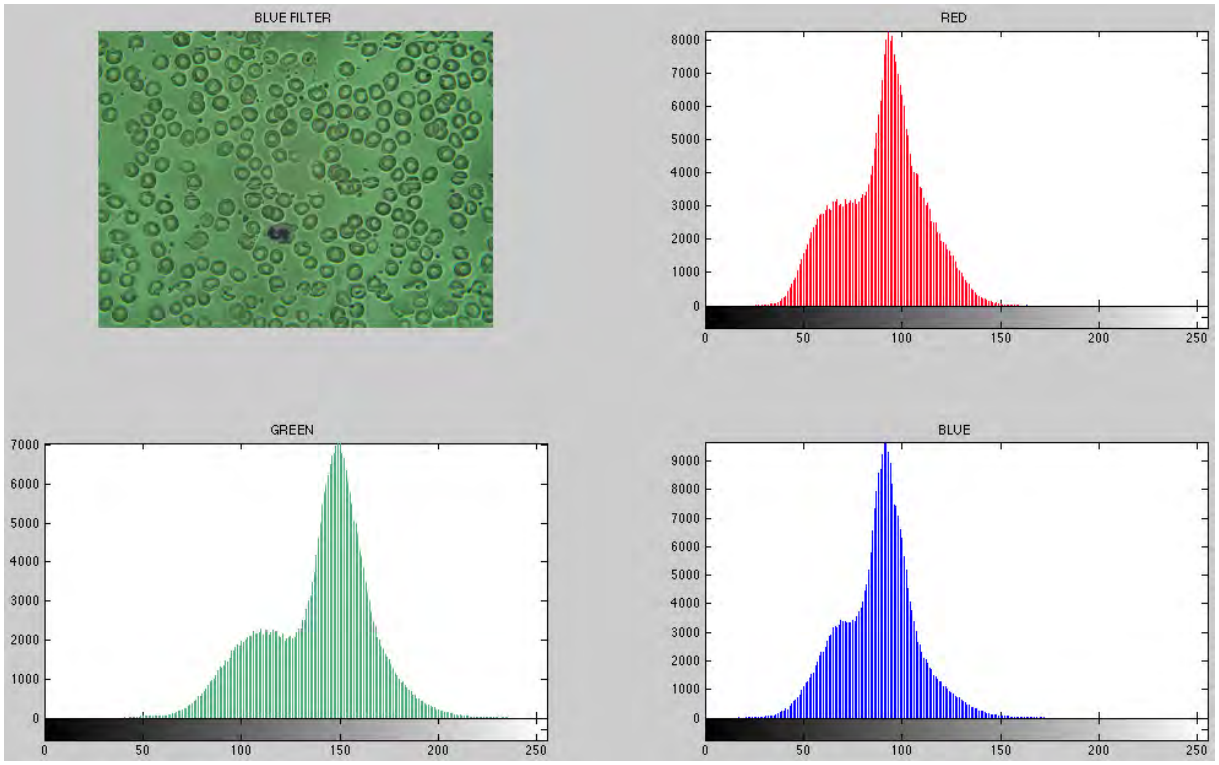


Figure 1.40: Bloodcells in Köhler illumination with a blue color filter.

1.4 Microscope objectives

Microscope objectives are the most important part of the optical microscope as they are responsible for the first magnification step, which is the biggest one. They are very complex to construct and therefore quite expensive. Many different types of objectives exist and in this chapter we will cover their basic properties. Naming conventions for microscope objectives are based on their ability to correct certain optical abberations so we will also discuss the origins of such anomalies.

1.4.1 Infinity corrected optics

Every lens has a fixed point of focus but in order to transfer the light from the objective to the eyepieces a long way has to be covered[7]. A finite focal distance is therefore a problem and is solved by the use of infinity corrected optics (fig. 1.41). An infinity corrected lens projects an image that is collimated to infinity[8], and the image is brought into focus somewhere else (e.g. in the eyepieces or the camera lens). Most modern microscopes use infinity corrected objectives and can be identified as they are marked with an infinity symbol ∞.

[7]Usually between 160mm and 250mm depending manufacturer.
[8]In other words, perfectly parallel beams of light.

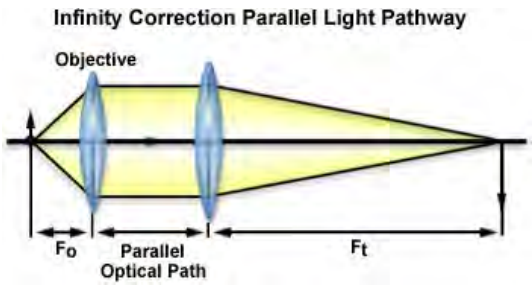


Figure 1.41: Magnification with infinity corrected lenses (Molecular Expressions ©).

The total amount of magnification of an infinity corrected optical system can easily be calculated with the following formula: $M_o = \frac{F_t}{F_o}$. Where F_o is the objective's focal length, F_t is the focal length of another lens used to bring the image back into focus and M_o is the system's total amount of magnification.

1.4.2 Spherical Abberation

When light of one frequency (monochromatic light) passes through the center of a lens it is not refracted. However, the more we deviate from the center the more the rays will bent, yielding in different focal planes (fig. 1.42). This anomaly is called *spherical abberation* and gets even worse when using light that contains different frequencies[9] (see: 1.4.3). As the lens catches rays from all the illuminated points of a specimen this effect will have to be compensated if we want a clear image. Lens manufacturers will take this into account when designing microscope objectives and make the necessary corrections (fig. 1.43).

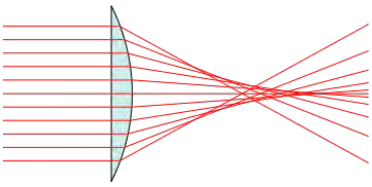


Figure 1.42: Spherical abberation in uncorrected lenses.

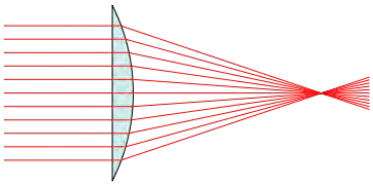


Figure 1.43: A perfectly spherical corrected lens.

1.4.3 Chromatic Aberration

Due to the different wavelengths of colors another anomaly occurs where each color frequency has a different focal plane. This effect is called *chromatic aberration*. It is caused by an effect called *dispersion*. Dispersion occurs when light passing from air into the glass lens is slowed down, each color at a different rate as a result of their wavelength. As the light then exits the lens back into air refraction occurs[10] and not all colors bend the same way (fig. 1.47). The result of chromatic aberration are overall soft images if the

[9] For example white light which contains all colors.
[10] As seen in spherical abberation, due to the shape of the lens.

different focus planes are close together or complete lack of a certain color if the planes lie far apart. Again objective manufacturers will correct the lenses to bring the main colors (RGB) to a common focus.

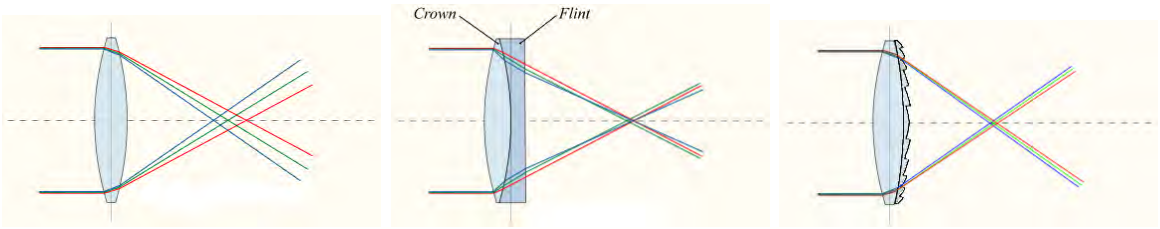


Figure 1.44: Chromatic aberration in uncorrected lenses and possible solutions.

1.4.4 Other types of aberrations

Other types of optical aberrations can occur in optical microscopy. For example *coma* and *astigmatism* are aberrations ascribed to misalignment of the substage condenser. Misalignment of the condenser can also attribute to the presence of lens flare. Because these kinds of image anomalies are not related to the objectives but rather to incorrect usage of the equipment we will not discuss them any further. When our short tutorial on Köhler illumination is correctly carried out (see: 1.3.4) these aberrations will not manifest themselves. If you want to learn more about these aberrations visit http://micro.magnet.fsu.edu/primer/lightandcolor/opticalaberrations.html.

The last optical aberration is *geometrical distortion* which is not that common as microscope manufacturers try to avoid it during their design phase. It is hard to detect but is most severe when the examined object under the microscope contains straight lines. We define two kinds of geometrical distortion namely *pincushion* and *barrel* (fig. 1.45). More information about this distortion can also be found on the web site mentioned above.

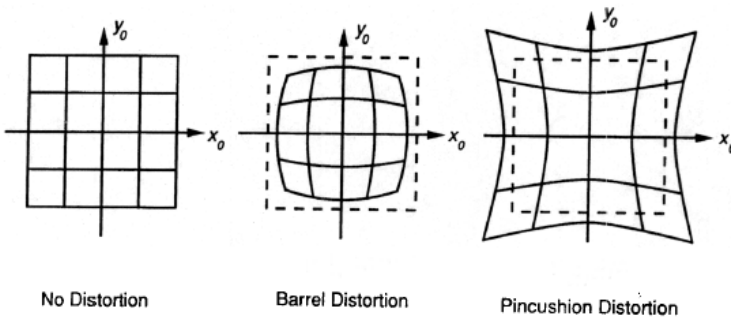


Figure 1.45: Types of geometrical distortions (Molecular Expressions ©).

1.4.5 Types of objectives

All of the major microscope manufacturers prove three important types of objectives (fig. 1.46). The least expensive type being the *achromatic* objectives. They are corrected for chromatic abberations as colors blue and red share the same focal plane. This means green is not in focus at the same focal length as red and blue which makes achromatic objectives not suitable for *color photomicrography*. They are however corrected for spherical aberration at the green wavelength. A green filter can be used when focussed in the red-blue region to get rid of typical green halos resulting from the difference in focal length. Objectives not specifically labeled are achromats.

Next we have the *fluorite* objectives[11]. They are also chromatically corrected for red and blue but usually closer to the green focus plane. Hence they are quite suitable for color photomicrography. Furthermore they are spherically corrected for two wavelengths, blue and green and therefore more expensive. They are constructed from advanced glass formations containing fluorite and this is how they got their name.

The third and most expensive objectives are the *apochromats*. Red, green and blue are brought into focus in the same plane thus apochromats are the best objectives for color photomicrography. Sometimes even a fourth color (deep blue) is added and they are also spherically corrected for up to four colors. Their higher price tag generally also results in a higher numerical aperture[12] and therefore providing brighter images with more contrast.

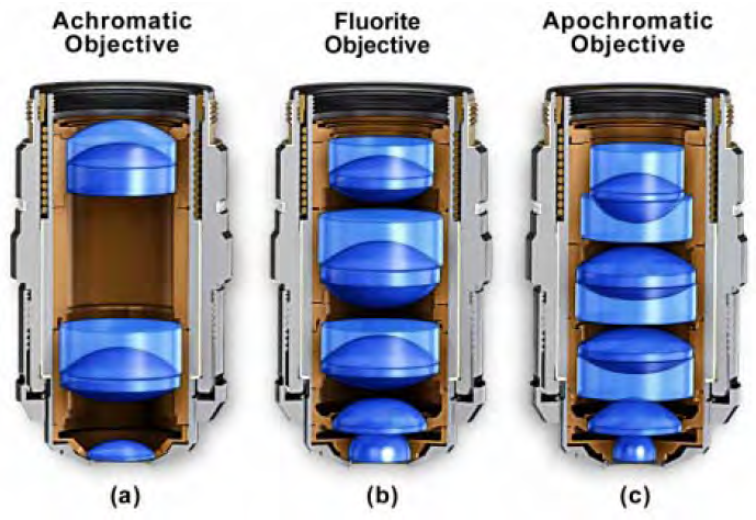


Figure 1.46: Three commonly known types of optical microscope objectives, (a) achrotmats (b) fluorites and (c) aphochromats (Molecular Expressions ©).

[11]Sometimes also called semi-apochromats.
[12]The N.A. indicates the light acceptance angle and therefore the resolving power, see: 1.3.1

1.4.6 Field Curvature

Due to the fact that curved lenses are used, the flat virtual image of the specimen will become curved as well after it passes through a series of lenses. The resulting image will therefore not be sharp in the entire field of view but only at the center or the edges. In other words, the obtained image will have a similar spherical shape as the lenses. This aberration can be very annoying in photomicrography as the image has to be entirely into focus if we want to capture it on film or on the surface of a CCD or CMOS image sensor. Luckily lens designers have produced flat-field corrected objectives. They are designated with the "plan" prefix, Plan Achromat, Plan Fluorite and Plan Apochromat. To obtain the best results in photomicrography they should be combined with flat photo eyepieces (Plan or Periplan).

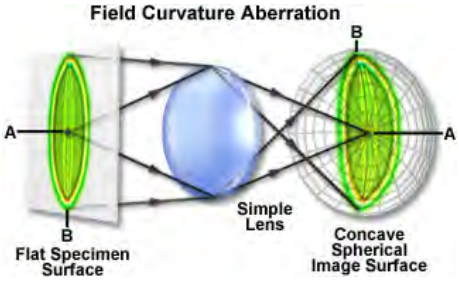


Figure 1.47: Field curvature as a result of the curved surface of a lens (Molecular Expressions ©).

1.4.7 Example and Color Codes

All the specifications we talked about are normally clearly engraved on the barrel of the objective. An example is given in figure 1.48. Color codes are also used which can be practical (fig. 1.49). If two color rings are present the one closer to the specimen is the *immersion color code* and the other one the *magnification color code*.

Figure 1.48: Engraved objective specifications (Molecular Expressions ©).

Immersion color code	Immersion type
Black	Oil immersion
Orange	Glycerol immersion
White	Water immersion
Red	Special
Magnification color code	**Magnification**
Black	1x, 1.25x
Brown	2x, 2.5x
Red	4x, 5x
Yellow	10x
Green	16x, 20x
Turquoise blue	25x, 32x
Light blue	40x, 50x
Cobalt (dark) blue	60x, 63x
White (cream)	100x

Figure 1.49: List of color codes.

1.5 Illumination techniques

Specimens under the microscope normally do not emit light on their own even when special techniques like fluorescence microscopy[13] are used. Therefore the specimen is illuminated with an artificial light source. Usually tungsten-halogen lamps are used but LED's are gaining popularity due to their excellent radiated spectrum[14] (fig. 1.50). White LED's also emit less infra-red radiation. This results in less dissipation of heat and can be important as a hot light source can degrade the specimen quickly.

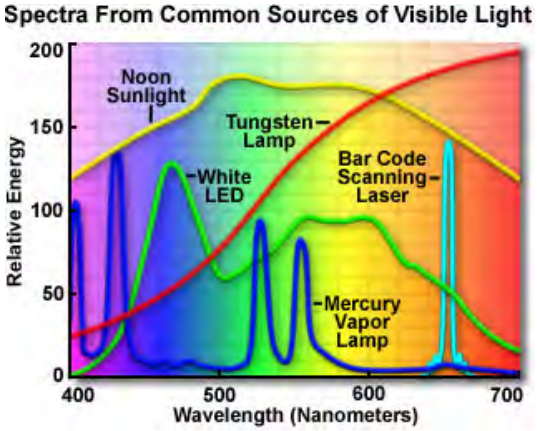


Figure 1.50: Comparison of different light sources (Molecular Expressions ©).

[13] In fluorescence microscopy the specimen becomes luminescent through excitation of molecules. To invoke this state of excitement ultraviolet or visible light photons can be used. Hence a light source is still necessary.

[14] A uniform distribution of emitted wavelengths represents white light of high quality which is ideal for microscopy.

The specimen can be illuminated in two directions by techniques called *transmitted light microscopy* and *reflected light microscopy*. When using transmitted light microscopy the light passes through the specimen from underneath and into the microscope's objective. In reflected light microscopy a beam splitter in the form of a partially silvered mirror or prism is used (fig. 1.51), sending the light down the objective and directing the reflected light back up towards the microscope's body tube. Reflected light microscopy is usually used when a specimen, even when sliced very thin, still isn't translucent enough. Examples of such specimens are metallurgical samples (metals, wood, polymers, plastics, semiconductors, ...) and complete integrated circuits. Most modern optical microscopes can be used in both configurations as seen in figure 1.52. While transmitted and reflected light microscopy are two ways of sending the light trough the microscope we can also define a variety of different *illumination techniques*. Proper illumination is crucial in achieving high-quality images in both microscopy and photomicrography. The type of illumination used defines the color correctness and contrast of the image, the uniformity of brightness and whether or not lens flare is introduced. We will only discuss *Köhler illumination* and *darkfield illumination* in detail as they are frequently used in this book and the only ones our Leitz Ergolux microscope supports. Other techniques like phase contrast illumination, fluorescence microscopy, Rheinberg illumination, etc. are not discussed as they are specialized techniques and well outside the scope of this book.

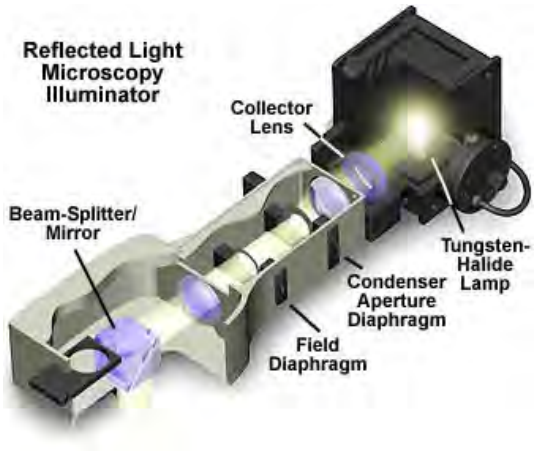


Figure 1.51: Reflected light optical pathway (Molecular Expressions ©).

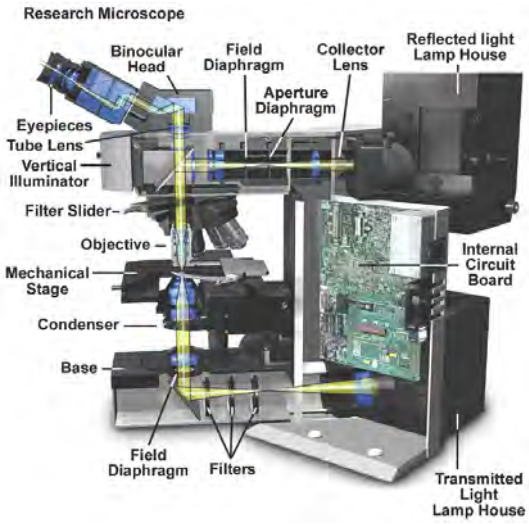


Figure 1.52: Modern microscope with both transmitted and reflected light capabilities (Molecular Expressions ©).

1.5.1 Köhler illumination

Köhler illumination, first introduced in 1893 by August Köhler of the Carl Zeiss corporation, is recommended by all manufacturers of modern microscopes. This technique produces specimen illumination that's uniformly bright and free from lens flare making it perfectly suitable for both microscopy and photomicrography. Köhler illumination is both elegant and simple (fig. 1.53). A *lamp collector* lens is placed in front of the light source

and focusses the light cone at the level of the *aperture diaphragm* of the *substage condenser*. Closing or opening the aperture diaphragm then controls the angle of the light, making sure the lenses inside the condenser produce parallel rays yielding a uniformly bright illumination of the specimen. Changing this angle changes the size and shape of the illumination cone coming from the condenser and thus changing the *numerical aperture* of the optical system and the contrast of the image. Because the light is not focused at the level of the specimen (hence a different *conjugate plane*) it is essentially grainless and does not suffer deterioration from dust and imperfections on the glass surfaces of the condenser and lamp collector lenses. The diameter (not angle) of the light bundle can be adjusted with the *field diaphragm*. Meaning that the field diaphragm does affect numerical aperture and should not be used to influence illumination intensity. The field diaphragm should be opened just a bit over the entire viewfield, overdoing it can lead to lens flare which in his turn leads to loss of contrast. There is however an exception when objectives with very low magnification are being used. They generally also have a very low numerical aperture. By using a swing-out top lens the condenser's aperture can still be matched to this low value. When this is the case the aperture diaphragm no longer influences the light cone and the field diaphragm should be used instead. Finally the image is focussed at the intermediate image plane inside the eyepieces where it can be seen by the observer's eye.

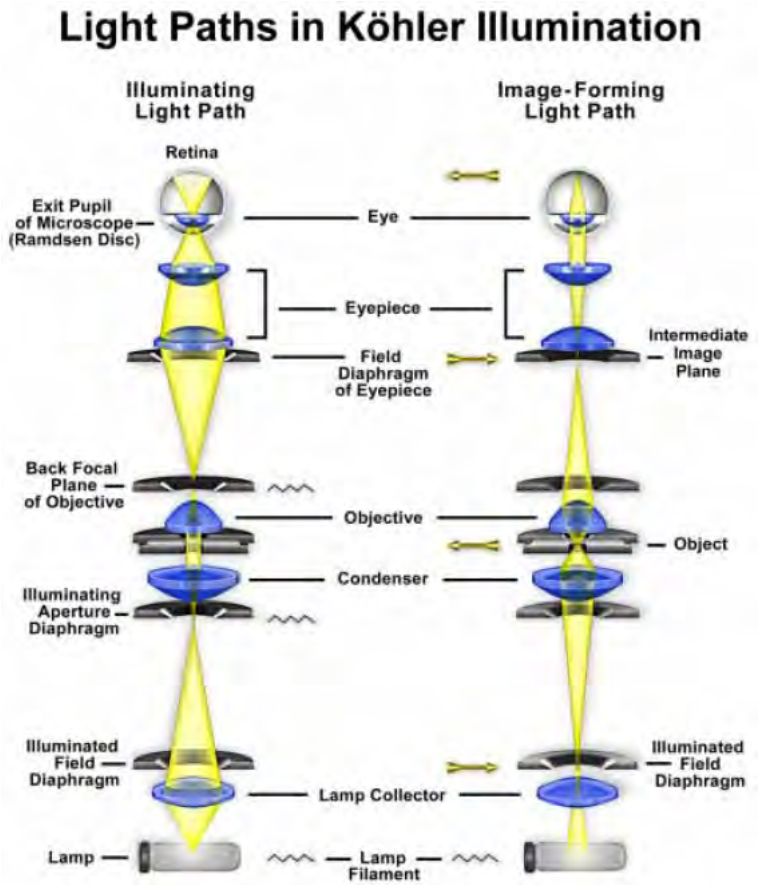


Figure 1.53: Light paths in Köhler illumination (Molecular Expressions ©).

Illumination intensity should be controlled by the use of neutral density filters or by reducing the voltage to the lamp. The latter is not usually recommended as turning down the voltage to much results in a warmer color temperature. The spectrum of the light shifts more to the red which is not good for microscopy and certainly not for photomicrography. The light source should be centered to the optical axis of the microscope and a frosted glass filter can be used on the lamp collector lens to further ensure the evenness of the light. Centering the lamp is usually done with screws on the lamp housing (fig. 1.54).

Figure 1.54: Screws to center the light source of a Leitz Ergolux microscope.

Some of the rules for Köhler illumination described above also apply when using *reflected light* instead of transmitted light. A reflected light substage condenser is easier to use as it is always at the correct height, already centered reasonably good and doesn't need a swing-out top lens. There's no need to alter the factory settings and therefore you normally can't. A field and aperture diaphragm are present just like in transmitted light microscopy and they can be used to control both brightness and contrast. They are however switched compared to a transmitted light condenser, first the aperture then the field diaphragm. The field diaphragm now controls the contrast and the aperture diaphragm the brightness (fig. 1.55 and fig. 1.56). Notice the pre-focused field iris due to the fixed height of the condenser.

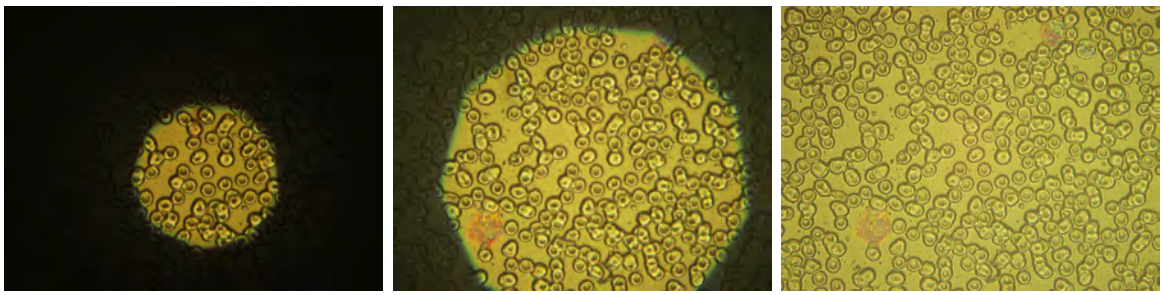


Figure 1.55: Changing the field diaphragm in reflected light microscopy to alter the contrast.

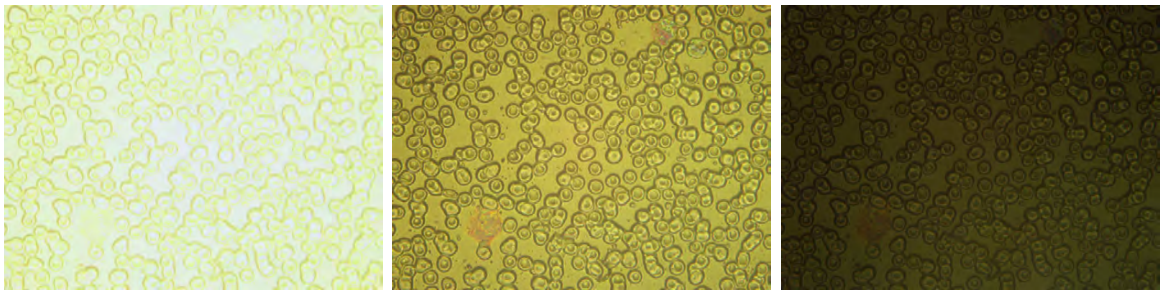


Figure 1.56: Adjusting the brightness in reflected light microscopy using the aperture diaphragm.

1.5.2 Darkfield illumination

Darkfield illumination is a simple and popular technique to make unstained transparent specimens clearly visible. It is obtained by blocking the central light rays which normally pass trough the specimen (fig. 1.57). This results in a dark background, hence the name. Only rays from the sides will hit the specimen and won't directly go into the microscope's objective. The light hitting the specimen from the sides will diffract, reflect and/or refract and these faint rays will enter the objective.

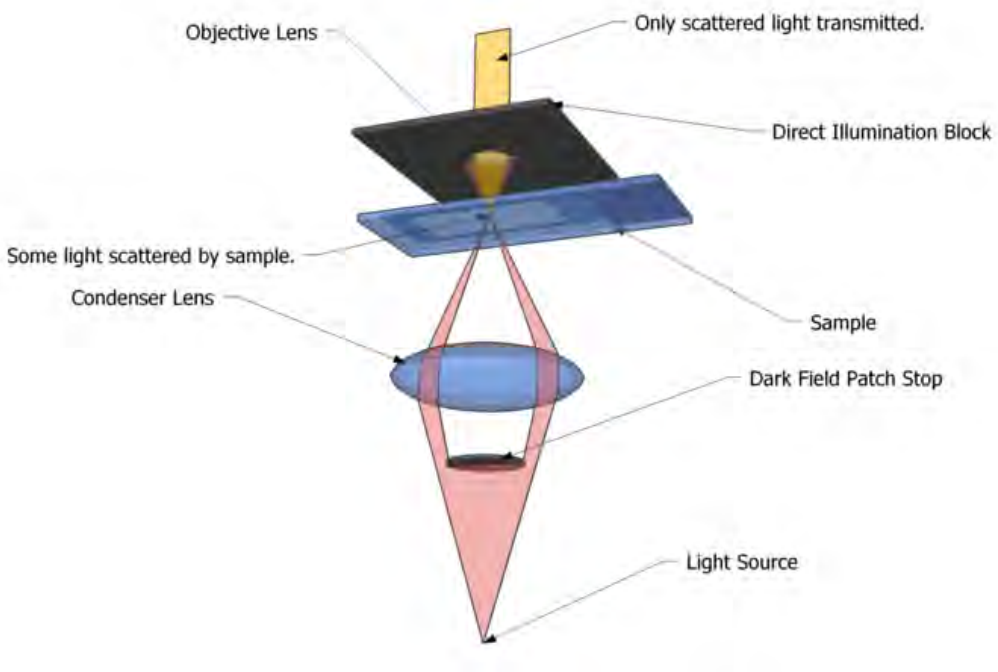


Figure 1.57: Principle of the darkfield illumination technique.

Darkfield illumination is also very popular in reflected light microscopy to enhance contrast. A darkfield mirror block with a light stop to block the central rays is then used (fig. 1.58 and fig. 2.17).

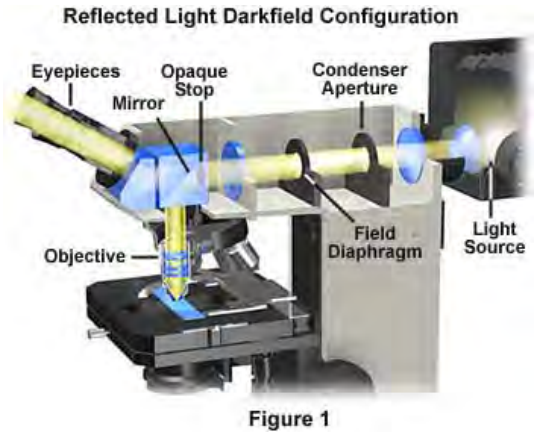


Figure 1.58: Reflected light darkfield illumination setup (Molecular Expressions ©).

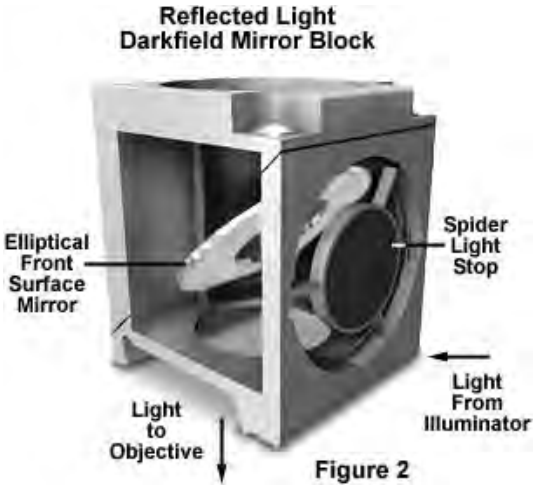


Figure 1.59: Mirror block for reflected light darkfield microscopy (Molecular Expressions ©).

The Leitz Ergolux microscope only has reflected light darkfield capabilities, nevertheless let's compare some examples.

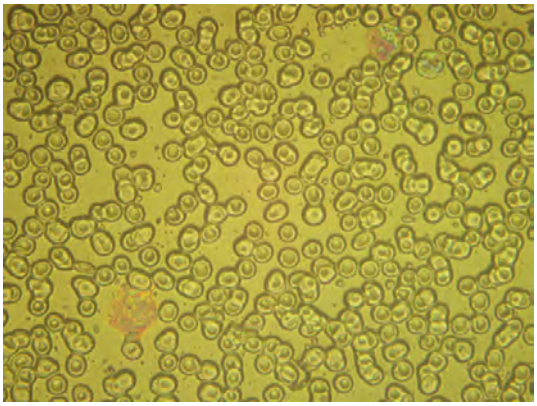


Figure 1.60: Blood cells in normal brightfield reflected light.

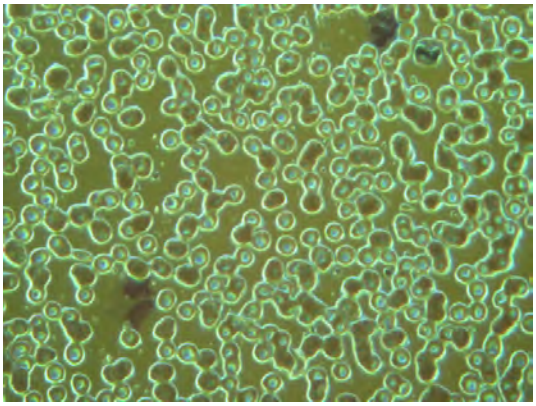


Figure 1.61: The same blood cells under darkfield reflected light.

Reflected light microscopy is mostly used for metallurgical purposes and inspection of integrated circuits, hence this CMOS sensor example.

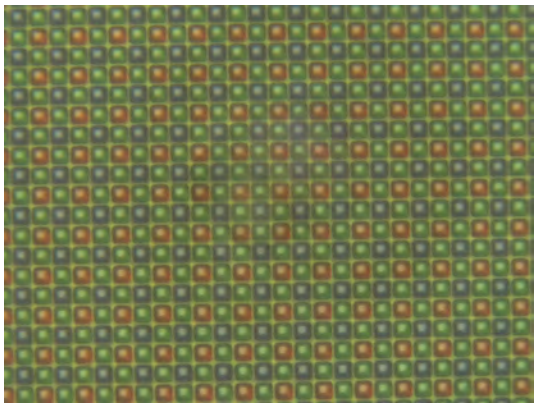


Figure 1.62: CMOS image sensor under normal brightfield reflected light showing the typical Bayer pattern.

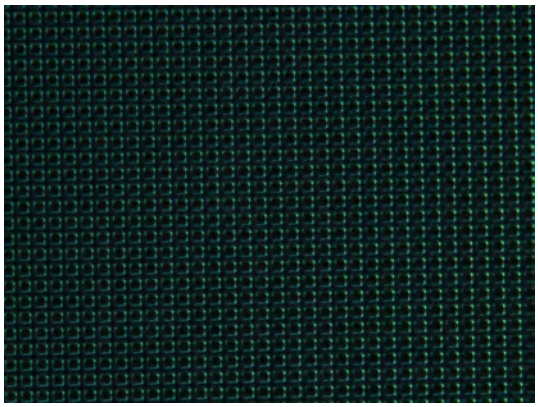


Figure 1.63: CMOS sensor under darkfield illumination.

Chapter 2

Leitz Ergolux: a short manual

This short manual is mainly aimed at the VISICS staff having to work with this specific microscope, the Leitz Ergolux. The original company, Ernst Leitz GmbH, has been split up and reformed in 1983 creating Leica Camera AG, Leica Geosystems AG and Leica Microsystems GmbH. This gives us a fair idea of how old the microscope is but this also meant having no luck at getting our hands on a manual. So we have to write our own.

2.1 Lab Setup

Below you can see an overview of the microscope setup at the laboratory. The most important parts are highlighted and we will explain them individually.

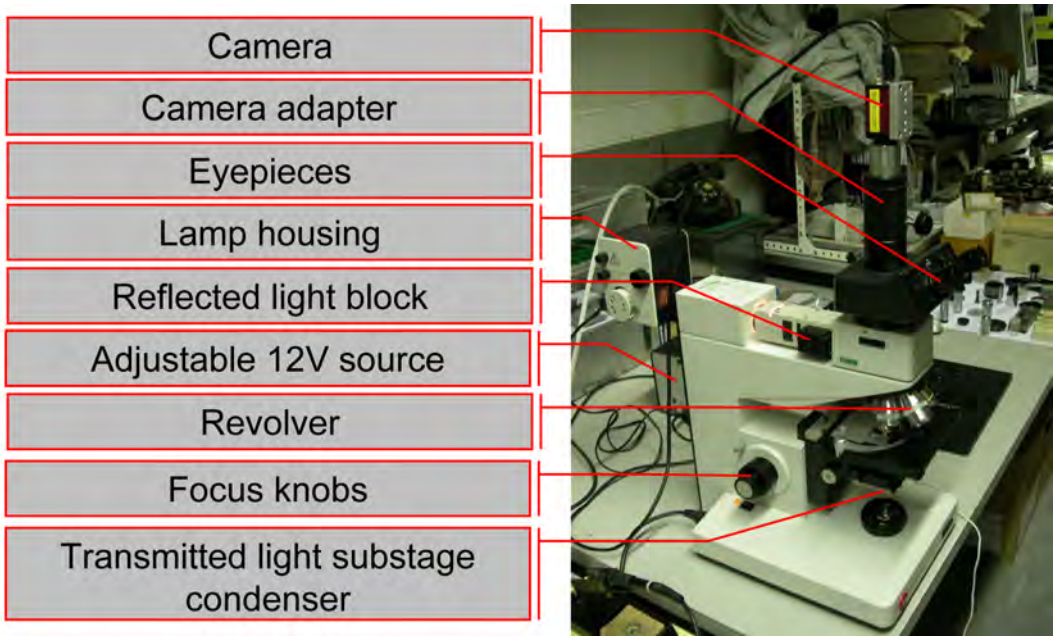


Figure 2.1: Overview of the lab setup.

2.1.1 Cameras

In the lab we tested two AVT cameras which could easily be mounted on the camera adapter (Marlin and Dolphin camera families fig. 2.2 and fig. 2.3). Allied Vision Technologies offers high grade cameras for industrial and scientific image processing (http://www.alliedvisiontec.com/). Since microscopic photography involves a high level of detail resolution is an extra important factor for us.

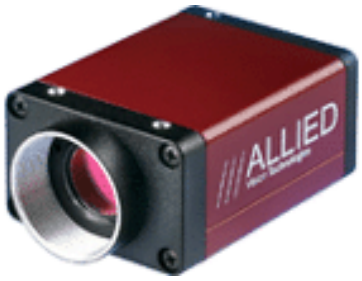


Figure 2.2: AVT Marlin.

Figure 2.3: AVT Dolphin.

2.1.2 Lamp and lamp power source

Originally the Ergolux came with spare light bulbs to fit into the lamp housing at the back of the microscope. However, we can assume lamp quality is strongly risen over the years. Therefore we've chosen a new Philips 100W/12V lamp suited especially for microscopes (fig. 2.4 and appendix B).

Figure 2.4: Philips microscope lamp.

The lamp's voltage, thus brightness, can be controlled with an external power source which can be important in Köhler illumination (fig. 2.5). The power source is connected to a power outlet on one side and to the lamp housing on the other. The lamp housing has screws to position the lamp and it's filament, which we already discussed in section 1.5.1. The housing can be placed on the bottom or the top of the microscope, respectively for normal and reflected microscopy.

Figure 2.5: Leitz external 12V lamp source.

2.1.3 Camera adapter and mounting the camera

The ergolux microscope was equipped with a camera adapter for the original analog film camera. Luckily for us the mechanical connection has not changed over the years so our digital AVT cameras can be easily mounted on top of this adapter. The adapter consists of 4 pieces (fig. 2.6) plus a fifth photo eyepiece which should be inserted to make the optical connection.

Figure 2.6: Ergolux camera adapter.

Three photo eyepieces are available in the lab but only the 3.2X and especially the 8X piece give good results with our cameras (fig. 2.7). Normal eyepieces can also be fitted inside the camera adapter and although they are not designed for this they sometimes yield good results.

When the adapter, with the camera screwed on, is fitted on top op de microscope the focal distance should be taken into account. This can be visually checked by adjusting the height until all lens flare is removed from the image (fig. 2.8). A thumbscrew to lock the adapter in it's final position is available.

A good way to check for lens flare is to focus the microscope on the *stage micrometer* that was supplied. Subsequently one can calibrate their microscopic measurements as the engravings in the micrometer are 10, 50, 100 and 500 micrometers apart (fig. 2.9).

Figure 2.7: Photopieces to be fitted inside the camera adapter.

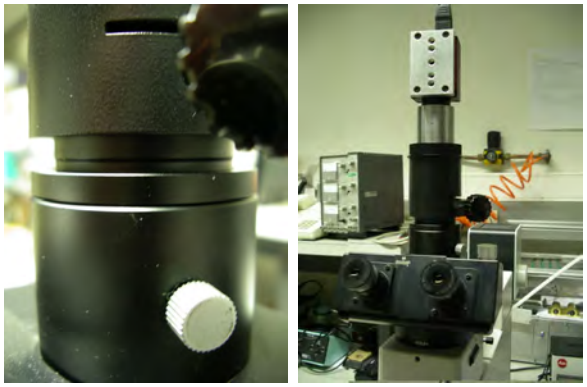


Figure 2.8: Properly mounting the camera and the adapter.

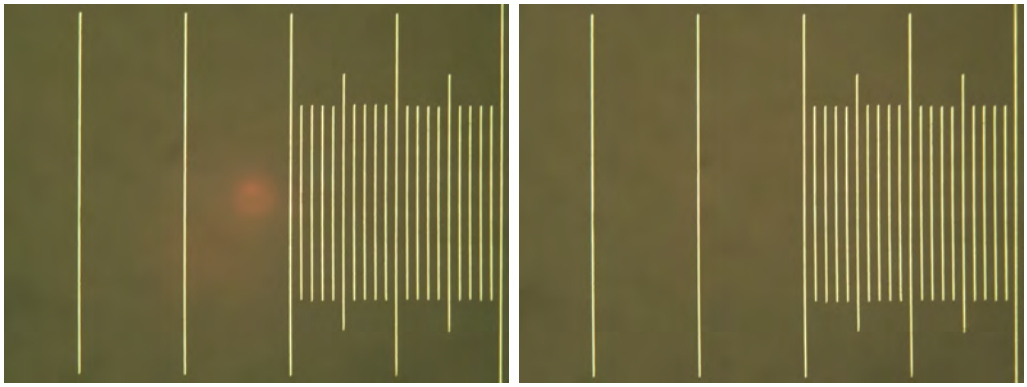


Figure 2.9: Removing lens flare by adjusting the adapter height.

2.2 Focussing on a sample

The Ergolux has two focussing knobs on each side, the small one is for fine tuning (fig. 2.10). A good working method is to put the glass with the specimen in place, put the objective to the side of the specimen glass (just touching!), raise the objective slightly with the one of the turning knobs, position the specimen under the objective and fine tune with the small knob until the image is in focus. This is safer because at high magnifications the working distances become so small one can easily go to far and damage the specimen or the glass.

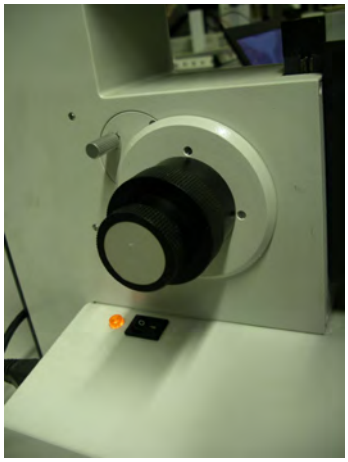


Figure 2.10: Focus turning knobs on the Ergolux.

2.3 Revolver and objectives

The revolver is mechanically powered when the microscope is turned on (ON/OFF button on the left side, underneath the focus knobs). It can house up to five different objectives which can be turned both ways with a button (fig. 2.11 and fig. 2.12). Always put the revolver in a high position when switching objectives because not all objectives are the same size and this could terrible damage their lens heads. In the lab the following objectives are available:

- R Pl 2x/0.04 infinity/-
- NPL 20x/0.35 DF infinity/0
- NPL FLUOTAR 16x/0.45 160/0.17
- NPL 5x/0.09 DF infinity/-
- NPL 10x/0.20 DF infinity/0
- NPL FLUOTAR 50x/0.85 DF infinity/0 [1]

[1] Even though this part was badly damaged by a previous user it still produces very good images.

Figure 2.11: The revolver's turning button, on the right side, in the foot of the microscope.

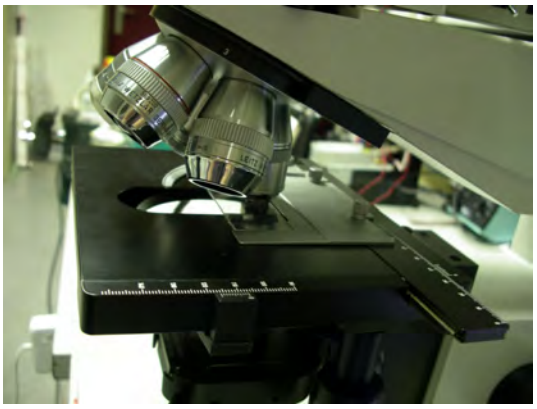


Figure 2.12: The revolver with 5 objectives mounted.

2.4 Filters

Different color and light attenuating filters can be used with the Ergolux microscope (fig. 2.13). The large ones can be fitted in the base of the microscope, in the field diaphragm (fig. 2.14), for transmitted light microscopy. The small ones go in the reflected light block for reflected light microscopy (fig. 2.15). An example of their usage has already been described in section 1.3.6.

Figure 2.13: Light attenuating filters.

2.5 Illumination

Transmitted light microscopy can be controlled through the substage condenser above the foot stand of the microscope (fig. 2.16). We've already discussed setting up proper Köhler illumination in section 1.3.4. A nice java applet for Köhler illumination can be found at http://www.microscopyu.com/tutorials/java/kohler/. For Reflected light microscopy this was discussed in section 1.5. The Erglux' darkfield illumination is easily enabled by sliding the black darkfield mirror block into position, located in the reflected light block

Figure 2.14: The field diaphragm with a yellow color filter fitted.

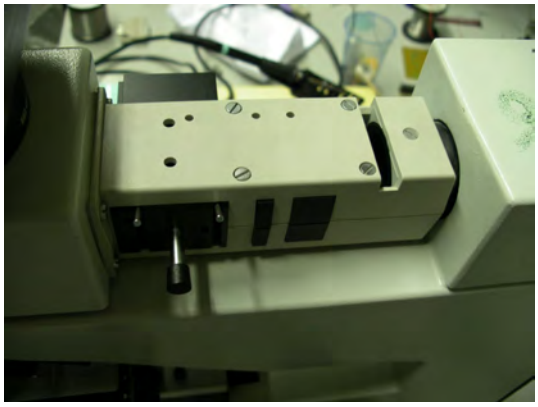


Figure 2.15: The Ergolux' reflected light block with filter slots.

as highlighted in figure 2.17.

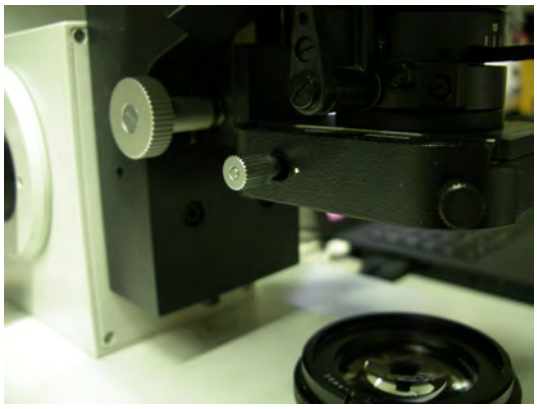


Figure 2.16: Ergolux substage condenser.

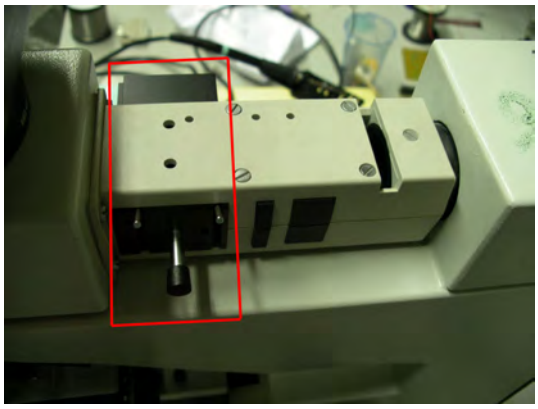


Figure 2.17: Mirror block slides in for darkfield illumination.

Chapter 3

Image Acquisition

In this chapter we will explain in short the IIDC standard, different video formats and how we actually capture images with libdc1394. Libdc1394 is the C/C++ capturing library we've chosen for our software. After also considering other alternatives (you can read about them in the literature study) libdc1394 seemed to have to best future road map for full triple platform compatibility. Linux and Windows compatibility is important, OS X compliance is a nice extra. Currently libdc1394 works perfectly on the first and the latter but the Windows port has not yet been released. Recently the project (version 2.0.x) got out of release candidate so no doubt it will attract more developers and a Windows version will see daylight pretty soon.

3.1 IIDC

IIDC (Instrumentation & Industrial Digital Camera) is the FireWire data format standard for live video and was previously known as the DCAM standard [9]. Cameras that follow the IIDC or DCAM specifications are FireWire based cameras (ieee1394 or ieee1394b) which are preferred over USB based cameras in most scientific midst. It is special in a way that is transmits *uncompressed* video over the FireWire bus unlike the ordinary DV (Digital Video) camcorder protocol. The standard is aimed at cameras for industrial applications like machine vision systems and other computer vision applications. The fact that incoming images are uncompressed reduces the load on the system processing them which is useful when embedded systems, generally equipped with less CPU power, are used. IIDC allows camera manufacturers to implement popular imaging algorithms inside the camera itself. The user then has the ability to let the camera do the image processing before sending the result over the bus. This is an advantage in both processing time, as the controller on the camera is optimized for such algorithms, and application programming effort. Documentation can be found on the official 1394 Trace Association website http://www.1394ta.org/Technology/Specifications/.

3.2 Video Formats

IIDC cameras support different video formats and each format supports different resolution and color modes (see: Appendix A). One special mode called *Format 7* allows the user to set a custom frame size (width, height) and position (top, left) to only view a part of actual image. Format 7 also allows for a custom *byte per packet* value to define a maximum transfer speed over the bus (e.g. when the FireWire cable is too long and 400 Mbps is causing problems). Because this value defines the frame transfer speed it is also related to the actual framerate. A range of color modes is also available depending on the chosen format 7 mode. Our application has basic format 7 support.

3.3 Image Processing Libraries

Although the software designed is mainly aimed at capturing images rather than processing them it should be fairly easy to enhance it's functionality by using image processing libraries. Two well known free libraries are the Integrating Vision Toolkit http://ivt.sourceforge.net/ and Intel's Open Source Computer Vision Library http://opencvlibrary.sourceforge.net/. We mention them here for any developer that wants to customize our software for specific needs.

3.4 Libdc1394

We quote from the libdc1394 website:

> Libdc1394 is a library that provides a complete high level application programming interface (API) for developers who wish to control IEEE 1394 based cameras that conform to the 1394-based Digital Camera Specifications (also known as the IIDC or DCAM Specifications). The library currently works on Linux, Mac OSX and (soon) Windows.

This means that the same code base currently compiles and runs flawlessly on both Linux and OS X and this should also be true for Windows in the future. Documentation is provided on the projects homepage http://damien.douxchamps.net/ieee1394/libdc1394/ and is sufficient and clear but not really up to date as the project is very active and undergoes regular changes. However the source code is well documented and clean and because the project is not that big you catch up rather quick. It has support for format 7 and special AVT[1] functions are available if needed in the future. The API works pretty low level which results in a dozen lines of code before we can begin camera capture but it also means flexibility (we can access all camera settings provided by the IIDC standard separately). The API can be downloaded at the projects SourceForge page http://sourceforge.net/projects/libdc1394/ and comes with some small sample programs to get you started. Some Linux distributions provide a binary package, otherwise you'll have to compile and install from source. Furthermore there is no need for a specific vendor

[1] Allied Vision Technologies cameras are the ones we use at VISICS.

driver[2] which is another advantage of using IIDC to control the camera. Our software therefore should work with all IIDC compatible cameras, directly through the FireWire bus. In the end the software was tested with a webcam and two AVT industry grade cameras. In the rest of this section we will discuss the inner workings of this library and how to use it.

3.4.1 Capture setup

Setting up libdc1394 from within a C/C++ project is fairly straightforward. First of all we have to include the library header file which should be available to us if we installed the library correctly.

```
#include <dc1394/dc1394.h>
```

Next we should define some elements that are needed to set up basic capture.

```
//the camera
dc1394camera_t *camera;
//frame width and height
unsigned int width, height;
//a video frame
dc1394video_frame_t *frame=NULL;
//this is a context in which cameras can be searched and used
dc1394_t * d;
//list of all available cameras
dc1394camera_list_t * list;
//a variable to store libdc1394 error codes
dc1394error_t err;
```

Now we can get a list of all cameras attached to the system.

```
//create a new libdc1394 context
d = dc1394_new();
//generate the list of all available cameras
err = dc1394_camera_enumerate(d, &list);
```

Libdc1394 uses a range of internal error codes and provides functions to catch them and display them to the user via the console. Almost all functions within libdc1394 return these error codes. In the following snippet of code we will initialize the first camera on the bus, camera 0. After we have selected a camera we can free the list if we don't need it anymore.

```
DC1394_ERR_RTN(err,"Failed to enumerate cameras");

if (list->num == 0) {
    dc1394_log_error("No cameras found");
    return 1;
}
```

[2]It does need some other FireWire libraries installed on Linux like raw1394 and others.

```
//initialize the first camera on the bus
camera = dc1394_camera_new (d, list->ids[0].guid);

if (!camera) {
    dc1394_log_error("Failed to initialize camera with guid %llx",
    list->ids[0].guid);
    return 1;
}

dc1394_camera_free_list(list);
```

Since we are developing a graphical application the error catching macro DC1394_ERR_RTN and function dc1394_log_error will be replaced by our own. We will ignore the libdc1394 errors in the rest of this example[3]. Now before we can start the camera we must first set up the transmission.

```
dc1394_video_set_iso_speed(camera, DC1394_ISO_SPEED_400);
dc1394_video_set_mode(camera, DC1394_VIDEO_MODE_640x480_RGB8);
dc1394_video_set_framerate(camera, DC1394_FRAMERATE_7_5);
dc1394_capture_setup(camera, 4, DC1394_CAPTURE_FLAGS_DEFAULT);
```

Of course different settings can be used with these functions. ISO speeds 100, 200, 400 and 800 are possible. Capture resolutions from 160x120 to 1600x1200 in various color modes (RGB, MONO, YUV) and framerates up to 240fps can be set. The last function sets up the ring buffer (more on this later) where 4 is the default number of frames in the buffer. Finally we can have the camera start sending us data.

```
dc1394_video_set_transmission(camera, DC1394_ON);
```

The camera is now transmitting frames and consequently our frame buffer is filling up at the desired framerate.

3.4.2 Ring buffer

To store the frames coming from the camera libdc1394 uses a circular buffer (fig. 3.1). Such a structure is often used for buffering data streams. It has a fixed size of memory mapped frame buffers that can be set with the dc1394_capture_setup() function. We must manually grab the frames from the buffer and free their space otherwise the buffer will fill up and we will lose frames. The following function returns a pointer to the first frame structure in the ring.

```
dc1394_capture_dequeue(camera, DC1394_CAPTURE_POLICY_WAIT, &frame);
```

Now we can do stuff with the actual image which is stored as a uint_8 matrix inside the frame structure. When we are done with the frame we must free it to make room for a new one.

```
dc1394_capture_enqueue(camera, frame);
```

[3]This example is based on the grab_color_image.c example that came with the library.

When we don't enqueue the buffer in time it will fill up completely and capturing will stop. Consequently we must dequeue and enqueue at the camera's framerate if we don't want any dropped frames. The buffer can consist out of only one frame but we recommend more to counter frame loss due to timing variations. One can also call multiple dequeue functions in a row to get pointers to successive frames in the buffer, for example to compare them. In memory the buffer is presented as a circularly-linked list (fig. 3.2).

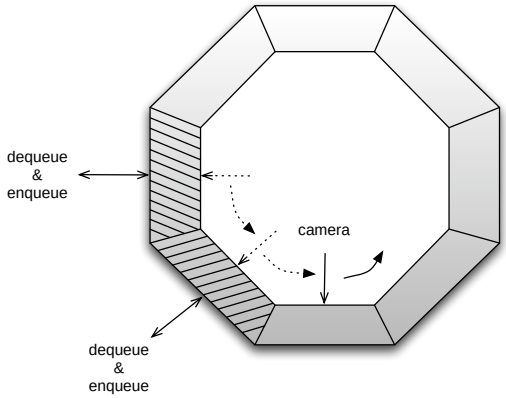


Figure 3.1: Ring buffer structure containing the frames.

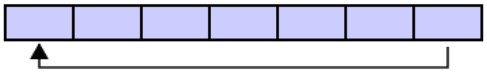


Figure 3.2: Ring buffer structure in memory.

The first frames coming from the camera are usually useless because the camera needs some time to adjust, mostly due to automatic features. The number of corrupted frames depends on the camera.

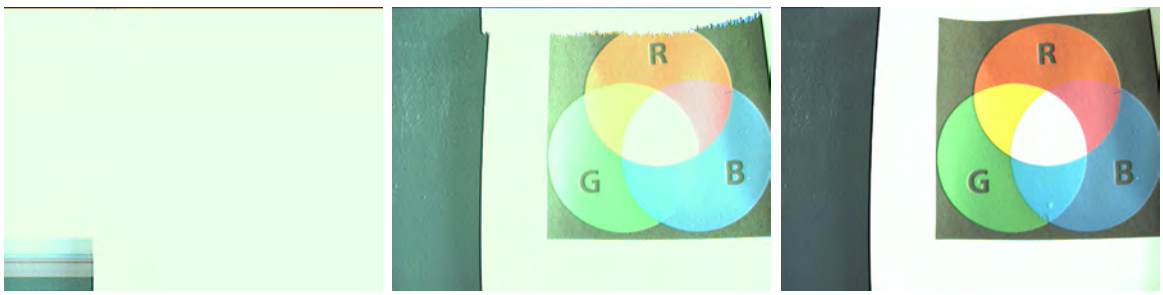


Figure 3.3: The first, third and sixth frame.

Fetching frames from the ring buffer can be done in two ways, either by waiting on a frame or polling for a frame (DC1394_CAPTURE_POLICY_WAIT or DC1394_CAPTURE_POLICY_POLL). It is generally smarter to poll for a frame as otherwise when no frame is present you're program will wait for a new one and consequently hangs at this point. When using the polling method a NULL pointer is returned when no new frame is ready in the ring buffer.

3.4.3 Cleaning up

If we want to stop transmission or close our program we must clean up properly. First stop the transmission then free the structures properly.

```
dc1394_video_set_transmission(camera, DC1394_OFF);
dc1394_capture_stop(camera);
dc1394_camera_free(camera);
dc1394_free(d);
```

If for some reason this should not happen nicely (program crashes, users plugs out camera,...) it could be necessary to reset the bus.

```
dc1394_reset_bus(camera);
```

3.4.4 libdc1394 2.0.1 functions

Here we explain (in short) some other libdc1394 functions one can encounter when reading the source code of our application. This will come in handy because documentation on the official site is sparse and outdated. For more details we refer to libdc1394's source code.

dc1394_video_get_supported_modes() gets all the supported video modes for a given camera.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_get_supported_modes(dc1394camera_t *camera,
dc1394video_modes_t *video_modes);
```

dc1394_video_get_supported_framerates() gets all the supported framerates for a given camera and video mode.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_get_supported_framerates(dc1394camera_t *camera,
dc1394video_mode_t video_mode, dc1394framerates_t *framerates);
```

dc1394_video_get_mode() gets the current video mode from the camera.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_get_mode(dc1394camera_t *camera,
dc1394video_mode_t *video_mode);
```

dc1394_video_set_mode() sets the camera to the specified video mode.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_set_mode(dc1394camera_t *camera,
dc1394video_mode_t video_mode);
```

dc1394_video_get_framerate() gets the current framerate from the camera, the returned value is only meaningful when not in Format 7 mode.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_get_framerate(dc1394camera_t *camera,
dc1394framerate_t *framerate);
```

dc1394_video_set_framerate() sets the camera to the specified framerate, this function returns an error when the camera is in Format 7 mode.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_set_framerate(dc1394camera_t *camera,
dc1394framerate_t framerate);
```

dc1394_video_get_iso_speed() gets the current iso speed.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_get_iso_speed(dc1394camera_t *camera,
dc1394speed_t *speed);
```

dc1394_video_set_iso_speed() sets the current iso speed.

Defined in: video.h

Usage:

```
dc1394error_t dc1394_video_set_iso_speed(dc1394camera_t *camera,
dc1394speed_t speed);
```

dc1394_format7_get_color_codings() gets all the available color codings for a given format 7 mode and camera.

Defined in: format7.h

Usage:

```
dc1394error_t dc1394_format7_get_color_codings(dc1394camera_t *camera,
dc1394video_mode_t video_mode, dc1394color_codings_t *codings);
```

dc1394_format7_set_color_coding() sets the selected color coding for a given format 7 mode and camera.

Defined in: format7.h

Usage:

```
dc1394error_t dc1394_format7_set_color_coding(dc1394camera_t *camera,
dc1394video_mode_t video_mode, dc1394color_coding_t color_coding);
```

dc1394_format7_get_max_image_size() gets the maximal image size for a given format 7 mode, we use this to constraint user input.

Defined in: format7.h

Usage:

```
dc1394error_t dc1394_format7_get_max_image_size(dc1394camera_t *camera,
dc1394video_mode_t video_mode, uint32_t *h_size,uint32_t *v_size);
```

dc1394_feature_get_all() gets a list of features and whether or not they are available for the given camera.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_get_all(dc1394camera_t *camera,
dc1394featureset_t *features);
```

dc1394_feature_get_value() gets the current value of a given feature.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_get_value(dc1394camera_t *camera,
dc1394feature_t feature, uint32_t *value);
```

dc1394_feature_set_value() sets a feature to the specified (valid) value.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_set_value(dc1394camera_t *camera,
dc1394feature_t feature, uint32_t value);
```

dc1394_feature_get_mode() gets the current mode for a given feature, values can be DC1394_FEATURE_MODE_AUTO or DC1394_FEATURE_MODE_MANUAL.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_get_mode(dc1394camera_t *camera,
dc1394feature_t feature, dc1394feature_mode_t *mode);
```

dc1394_feature_set_mode() set the given feature to manual or automatic.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_set_mode(dc1394camera_t *camera,
dc1394feature_t feature, dc1394feature_mode_t mode);
```

dc1394_feature_whitebalance_get_value() gets the current values for the white balance, a U and V value is returned.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_whitebalance_get_value(dc1394camera_t *camera,
uint32_t *u_b_value, uint32_t *v_r_value);
```

dc1394_feature_whitebalance_set_value() sets the white balance by means of a U and V channel value.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_whitebalance_set_value(
dc1394camera_t *camera, uint32_t u_b_value, uint32_t v_r_value);
```

dc1394_feature_is_present() checks whether or not a feature is supported by a camera.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_is_present(dc1394camera_t *camera,
dc1394feature_t feature, dc1394bool_t *value);
```

dc1394_feature_get_boundaries() gets the min/max values for a given feature and camera, only useful when the feature is supported by the camera.

Defined in: control.h

Usage:

```
dc1394error_t dc1394_feature_get_boundaries(dc1394camera_t *camera,
dc1394feature_t feature, uint32_t *min, uint32_t *max);
```

Chapter 4

Software

The following chapter talks about the design of our software application. The most important requirement was using Trolltech's Qt[1] as a base for the GUI. Unlike the already existing Coriander application http://damien.douxchamps.net/ieee1394/coriander/ which is Linux only, we aim to create a platform independent graphical user interface for libdc1394. No such program exists at this time of writing and although ours will not be as full-featured as Coriander, all the basic functionality will be there. The most important but also most troublesome task is to get the raw images from the capture library into our application and displaying them fluently. You will read about our findings here but we also leave some room for improvement.

4.1 About Qt

Qt (pronounced "cute") is a platform independent framework for the development of GUI programs. Some modules included in Qt can be used separately from the graphical user interface to create command-line driven tools. Qt itself is based on C++ and open source but requires a license when used for commercial applications. When creating GUI's the developer combines different elements or so-called "widgets". These widgets can be predefined (window, input box, button,...) or subclassed from these to add more functionality. Generally a Qt/C++ projects contains a bunch of source files, header files, a resources file and a special .pro file to combine everything into a project. Out of this .pro file we can automatically generate a make file by running the following command from the console.

```
qmake foobar.pro
```

Compiling and linking of the project is then easily initiated by running the make command.

```
make
```

Our project should compile flawlessly, if you have libdc1394 2.0.1 and Qt 4.3 or higher installed. For more information about libdc1394 read chapter 3. Qt can be obtained (open

[1] Trolltech was recently acquired by Nokia.

source or commercial) from their website http://trolltech.com/products/qt. If you are starting Qt development the reference pages are a good place to start http://doc.trolltech.com/ or the mailing lists and their archives if you run into trouble http://lists.trolltech.com/.

4.2 Application outline

As a design approach we've chosen a Single Document Interface (SDI) for our program. Multiple instances of the main window can be started, independently of each other, so multiple cameras can be addressed at the same time[2]. A monitor widget inside the main window shows the camera stream and extra functionality is added by different dialogs. The user can access these dialogs via the menu bar. Screenshots of our application after startup are shown in figure 4.1 for both the Linux and the OS X version.

Figure 4.1: Linux version (left) and OS X version (right).

4.3 Loading an image

A video frame captured by libdc1394 (or any other capturing library for that matter) is usually in the form of an unsigned integer matrix. The size of this matrix depends on the color depth of the image and the resolution. Usually the size is width x height x 8bit for a monochrome image with 8 bits/pixel, multiply this by 3 for an image with color information (RGB, YUV,...). 16 bits/pixel formats also exists for very sophisticated cameras.

We must load this image data into a format we can use inside Qt, for example to draw onto the screen or to save the image easily to disk. Two internal formats for image presentation are present, QImage and QPixmap (a QPicture class also exist, but this is

[2]Although opening two completely independent versions is faster because our program does not support multithreading yet.

something completely different). Supposedly using a QPixmap to draw onto a widget is faster (or a sequence of images), because it uses the native drawing engine of the OS (e.g. X11 on Linux/Unix). A QImage on the other hand is optimized for I/O and pixel operations, which makes it more flexible to use. It does not use the native drawing engine but Qt's own and therefore gives the same exact results on all platforms. Qt documentation [14] states the use of both as:

> Typically, the QImage class is used to load an image file, optionally manipulating the image data, before the QImage object is converted into a QPixmap to be shown on screen.

To our experience however, converting the QImage to a QPixmap before drawing does not yield a noticeable performance increase. QImage independence of the native drawing engine also allows for special effects like anti-aliasing, different composition modes, alpha blending, ... to be identical on all platforms. Most importantly saving images to disk is very easy with QImage due to it's optimized I/O functions. Numerous formats like .bmp, .tiff, .jpg, .png, etc. are supported. A QImage object to store a color image is always in a three layer RGB format plus a fourth layer which is either ignored (0xffRRGGBB) or used for the alpha channel (0xAARRGGBB). The alpha channel defines the opacity relative to the background. Now before we can load the QImage with our camera frame we must first convert the 3 layer presentation (24bit) to this 4 layer version by artificially adding an alpha channel (32bit format). For this purpose we've written conversion functions, for example the next function does an RGB24 to RGB32 conversion by stuffing each fourth byte with alpha channel information.

```
void RGB24_to_RGB32(uint8_t *src, uint8_t *dest, uint32_t width,
 uint32_t height){

register int i = (width*height*3)-1;
register int j = (width*height*4)-1;
register int r, g, b, alpha = 255; //opacity 100%

while (j >= 0) {

        b = src[i--];
        g = src[i--];
        r = src[i--];

        dest[j--] = alpha;
        dest[j--] = r;
        dest[j--] = g;
        dest[j--] = b;
        }
}
```

Other conversion functions (MONO8 to RGB32, YUV4xx to RGB32) have been written and are partially based on existing libdc1394 functions and macros (myconv.h). You can review them in Appendix C. To test the cost of such conversions we developed a small benchmark application. This application calculates the average conversion time out

of 1000 conversions based on a random dummy image. This test is then run several times for common resolutions and different color mode conversions. To get an even better idea we've put an old an a new system head to head in the test. The results can be seen in figure 4.2 for an old Pentium 3 - 733Mhz and figure 4.3 shows the same test for a new Core Duo - 2Ghz[3].

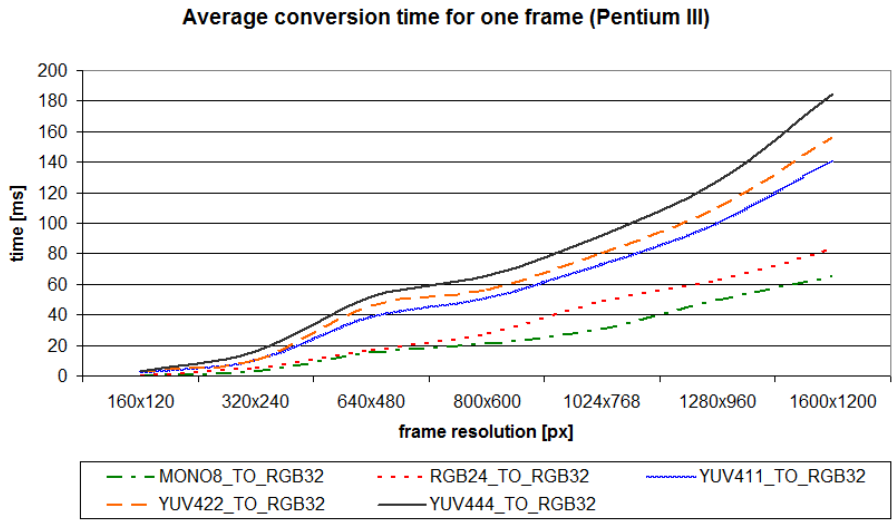


Figure 4.2: Pentium III benchmark results.

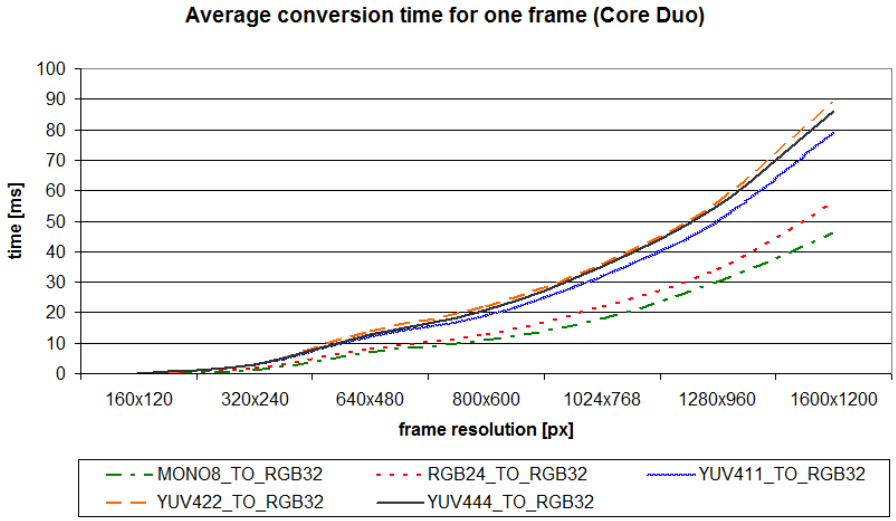


Figure 4.3: Core Duo benchmark results.

[3]Only one out of two cores was used because our test program does not support multithreading.

The tests show a significant increase in performance on the newer system, which was expected. The gap between the various YUV4xx conversions becomes smaller but the speed difference with the functions that have no color transformation, only byte stuffing, is still noticeable. Nevertheless copying pixels is still a costly operation, even on a fast machine. These conversion times become problematic at high resolutions or high framerates. For example when capturing at 30 fps the maximum time it may take for one frame to be converted is 33 ms. If higher we will experience delays and most likely dropped frames.

Supposedly we could speed the functions up by using bit shifts instead of multiplications and pointers instead of temporary variables. The same function as we've seen before would then look like this.

```
void RGB24_to_RGB32_0(uint8_t *src, uint8_t *dest, uint32_t width,
 uint32_t height){

register int j = ((width*height) << 2) -1;

while (j >= 0) {

        *dest++ = *(src+2);
        *dest++ = *(src+1);
        *dest++ = *(src);
        *dest++ = 255;
        src = src + 3;
        j = j - 4;
        }
}
```

However this has absolutely no noticeable result so we stick to the original functions for convenience.

4.4 Signals between Widgets

As seen in the introduction about Qt a general Qt application consists out of various *widgets*. These can be seen as independent objects, not knowing about the existence of one another. Of course a mechanism is needed to set up communication between these widgets, to signal events and pass information between them. This is accomplished with *signals and slots*. Signals are emitted by a widget and are connected to a slot belonging to another widget. The slot is actually just like a function and is executed immediately after the corresponding signal is received.

```
connect(sender object, SIGNAL, receiver object, SLOT)
```

The signals themselves can contain data which gives the same results as evoking the slot function with certain parameters. Consequently the signal and slot definitions should contain the same parameters and are always return type *void*.

```
//in the sender's class definition
signals:
    void sendCameraPointer(dc1394camera_t *camera);

//in the receiver's class definition
public slots:
    void acceptCameraPointer(dc1394camera_t *camera);

//the connection
connect(sender object, SIGNAL(sendCameraPointer(dc1394camera_t)),
receiver object, SLOT(acceptCameraPointer(dc1394camera_t)));
```

Signals can also be relayed through another widget as can be seen on figure 4.4. The signals between the Monitor widget and the Dialog widgets are relayed through their parent widget MainWindow as only MainWindow has knowledge of all the other (child)widgets.

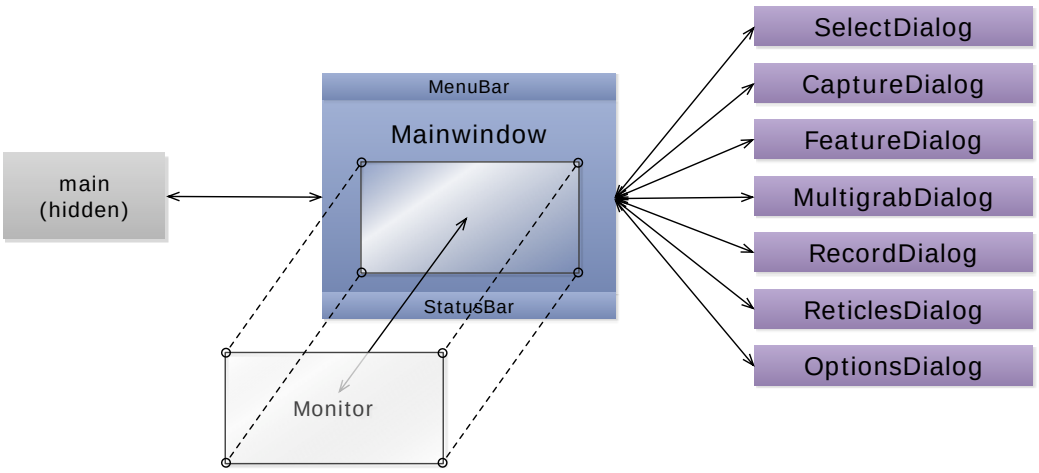


Figure 4.4: Schematic presentation of signals between our different Qt widgets.

4.5 Classes

In this section we will discuss the various classes we've created. Almost all of them (except for the Monitor class) are subclassed from the existing QDialog class. These dialogs are the main interface to the user and thus add functionality to our program. The monitor class is subclassed from a QGLWidget for hardware accelerated drawing onto the screen.

4.5.1 SelectDialog

The SelectDialog class provides a dialog for the user to select a camera out of all the available cameras detected on the FireWire bus (fig. 4.5). It uses libdc1394 functions to

get a list of all the available cameras but does not initiate capture itself. This is left to the Monitor class which catches the signal emitted by the SelectDialog.

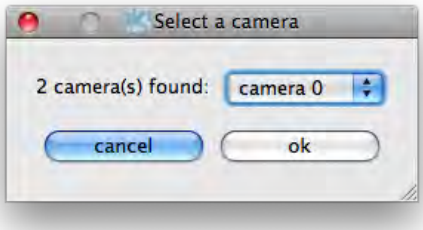


Figure 4.5: The camera selection dialog.

When the list of cameras is created (see: example chapter 3) we can enumerate them inside a QComboBox. The corresponding camera number is then signaled to the Monitor, through their parent widget MainWindow, when the user clicks *ok*. Capture is immediately started at the lowest supported resolution and the highest framerate. When *cancel* is clicked we do nothing but call the close() function on the dialog.

```
for(unsigned int i = 0; i <= (list->num - 1); i++){
    cameraComboBox->addItem((tr("camera %1").arg(i)),i);
}
```

When a libdc1394 error occurs or no camera is found a nice messagebox informs the user of the problem. When running our program on a Linux machine a common libdc1394 error occurs when the user has no read/write permissions on the FireWire device. To fix this login as root (or use sudo) and run the following command.

```
(sudo) chmod 777 /dev/raw1394
```

This issue does not occur when running our application on OS X unix.

4.5.2 CaptureDialog

The CaptureDialog let's the user choose a number of capture settings and displays some basic info about the selected camera (fig. 4.6). Capture settings include the image resolution, the color mode, the framerate and the FireWire bandwidth usage. Only the video modes supported by the active camera can be chosen and their corresponding framerates[4]. Generally one should always keep the ISO speed at 400Mbps (default) or 800Mbps when the camera supports it (experimental - not tested). 200Mbps and 100Mbps should only be used when having problems communication at high speeds over very long FireWire cables. Functions to get the current iso speed and set a new one are reviewed in section 3.4.4. We can enumerate the supported modes like this.

[4]E.g. our test camera only supports 30 fps in YUV411/MONO8 mode.

```
//list of available modes
dc1394video_modes_t video_modes;

//load the modes for this camera
dc1394_video_get_supported_modes(camera, &video_modes);

//enumerate them inside a QComboBox
for (unsigned int i = 0; i <= (video_modes.num - 1); i++){
    generalTab->modeComboBox->addItem(enumToString(video_modes.modes[i]),
    (video_modes.modes[i])); }
```

The same can be done for the framerate. The video mode is used as an extra parameter because the supported framerates depend on the selected video mode.

```
//list of supported framerates in a mode
dc1394framerates_t framerates;

//load the framerates for this camera and mode
dc1394_video_get_supported_framerates(camera, videoMode, &framerates);

//enumerate them inside a QComboBox
for(unsigned int i = 0; i <= (framerates.num - 1); i++){
    generalTab->framerateComboBox->addItem(enumToString(
    framerates.framerates[i]), (framerates.framerates[i]));
}
```

Now the user can select the videomode and corresponding framerate. When *ok* is clicked a signal is sent to the Monitor widget to restart capturing with the new settings. Take note that the enumToString() function is one of our own functions that translates libdc1394 numerical codes into their corresponding QStrings. Is it defined in mytrans.h.

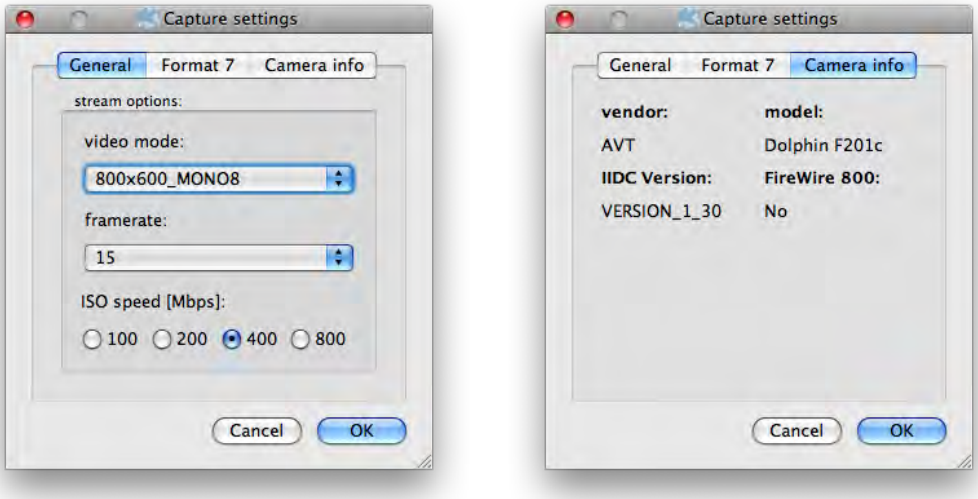


Figure 4.6: The capture settings dialog.

When the selected camera also supports Format 7 mode one can also set the frame position and size. In this case no framerate can be set as this depends on the camera's ability to capture a given frame size at a given color mode. If the format 7 ROI[5] is small, a high framerate will be achieved and vice versa. The framerate is partially related to the *bytes per packet* value which is needed to set up Format 7 transfer mode with the camera. When the ROI is set we can ask the camera for a *recommended byte per packet* value for the given frame size. Using a value lower than the recommended one results in a less than ideal framerate as the camera will not be able to transfer the images as fast at it can capture them. A higher bbp value has no use and will only result in more bandwidth usage than necessary. Since the framerate cannot exactly be derived from the bbp value we must check with regular intervals to see if a new frame was received. Setting this polling interval to 30 times per second should be sufficient, even for very small ROI's. The corresponding code snippet for this can be seen in section 4.5.8.

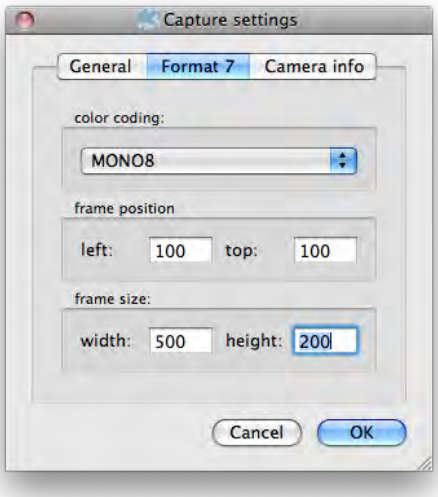


Figure 4.7: Setting the Format 7 options.

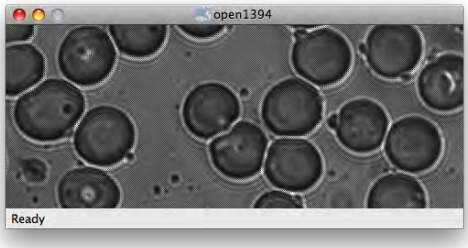


Figure 4.8: Resulting ROI camera stream.

To validate the user input for the position and size of the frame we can get the maximum image size for a given Format 7 mode and camera.

```
dc1394_format7_get_max_image_size(camera, currentMode,
&maxWidth, &maxHeight);
```

As long as top, left, width and height are positive integers and

$$\text{left} + \text{width} < \text{maxWidth}$$
$$\text{top} + \text{height} < \text{maxHeight}$$

the ROI input is valid. A schematic view is shown in figure 4.9.

[5]Region Of Interest

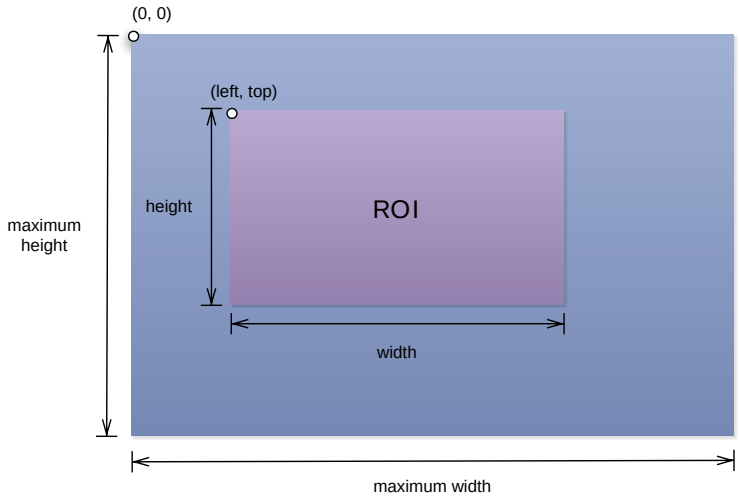


Figure 4.9: Format 7 ROI within the image size constraint.

Each format 7 mode supports different color codings. We can get them using the dc1394_format7_get_color_codings() function and set the selected one with dc1394_format7_set_color_coding() (see: section 3.4.4).

4.5.3 FeatureDialog

The FeatureDialog class enables the user to change certain camera features like brightness, hue, saturation, etc.

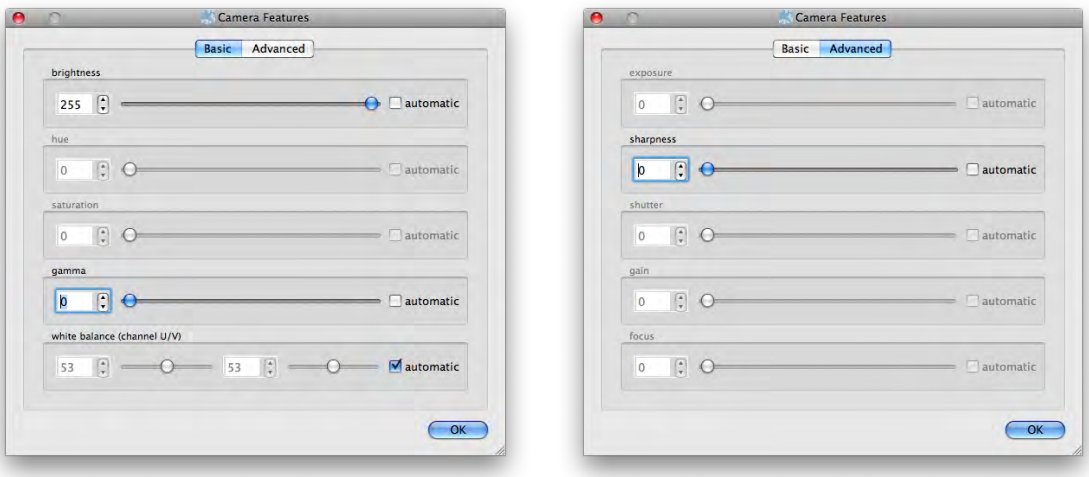


Figure 4.10: The features settings dialog.

When the dialog is evoked we first determine whether or not a certain feature is supported by the connected camera, if not we disable user input for that feature. For the supported ones we first load in the current values and their working mode (auto/manual). If setting a feature to automatic has no effect, the camera probably does not support auto mode on that feature. The IIDC standaard and libdc1394 support numerous camera features. In our program only the important ones are currently implemented as can be seen in figure 4.10. We support: brightness, hue, saturation, gamma, white balance, exposure, sharpness, shutter, gain, focus.

The following function can load any camera feature we use into our Qt dialog widget (except for whitebalance which has more parameters).

```
bool FeatureDialog::initFeature(dc1394feature_t cameraFeature,
QSlider *slider, QSpinBox *spinbox, QCheckBox *checkbox){

dc1394error_t err = DC1394_SUCCESS;
dc1394feature_mode_t mode;
uint32_t currentFeatureValue = 0;
uint32_t min = 0, max = 0;
dc1394bool_t isPresent;

//first check if the feature is available
err = dc1394_feature_is_present(camera, cameraFeature, &isPresent);
if(err){
    libdcErrorMessage(err,"Could not get feature!");
    return false;
}

if(isPresent){
    //first get the current value
    err = dc1394_feature_get_value(camera, cameraFeature,
    &currentFeatureValue);

    if(err){
        libdcErrorMessage(err,"Could not get feature value!");
        return false;
    }

    //get the min and max values for this feature
    err = dc1394_feature_get_boundaries(camera, cameraFeature,
    &min, &max);

    if(err){
        libdcErrorMessage(err,"Could not get feature min/max!");
        return false;
    }

    //set the slider and spinbox to this range
    slider->setValue((int)currentFeatureValue);
    slider->setRange((int)min, (int)max);
```

```
    spinbox->setValue((int)currentFeatureValue);
    spinbox->setRange((int)min, (int)max);

    //check the mode here (auto/manual)
    err = dc1394_feature_get_mode(camera, cameraFeature, &mode);
    if(err){
        libdcErrorMessage(err,"Could not get feature mode status!");
        return false;
    }

    if(mode == DC1394_FEATURE_MODE_AUTO){
        checkbox->setChecked(true);
    } else {
        checkbox->setChecked(false);
    }

    //return true or false to enable the groupbox
    return true;
    } else {
    return false;
} }
```

When the user changes the value, either by means of the QSlider or the QSpinBox, a slot corresponding to that feature is called and sets the value.

```
void FeatureDialog::brightnessChanged(int value){

    dc1394error_t err = DC1394_SUCCESS;

    err = dc1394_feature_set_value(camera, DC1394_FEATURE_BRIGHTNESS,
    (uint32_t)value);

    if(err){
        libdcErrorMessage(err,"Failed to set the new brightness!");
        return;
    }
}
```

A similar slot is called when the user checks or unchecks the checkbox for auto/manual mode.

```
void FeatureDialog::brightnessModus(bool checked){

    dc1394error_t err = DC1394_SUCCESS;

    //set the feature to manual or auto
    if(checked){
        err = dc1394_feature_set_mode(camera, DC1394_FEATURE_BRIGHTNESS,
        DC1394_FEATURE_MODE_AUTO);
        basicTab->brightnessSlider->setEnabled(false);
        basicTab->brightnessSpinBox->setEnabled(false);
```

```
        } else {
        err = dc1394_feature_set_mode(camera, DC1394_FEATURE_BRIGHTNESS,
        DC1394_FEATURE_MODE_MANUAL);
        basicTab->brightnessSlider->setEnabled(true);
        basicTab->brightnessSpinBox->setEnabled(true);
        }

    if(err){
        libdcErrorMessage(err,"Failed to set the brightness to auto/manual!");
        return;
        }
}
```

A slightly different variations on these functions exist for the white balance feature which uses extra parameters (U/V channel). As you can see we try to catch as many errors as we can and display them in a messagebox using our own libdcErrorMessage() function. This function basically shows a libdc1394 error code and our own message. In the example code above errors normally never occur but the libdcErrorMessage() function is also used in other parts of our program to catch more critical errors.

4.5.4 MultigrabDialog

Unlike the *grab single image* function in the Tools menu, which only captures one image, this dialog has the ability to grab multiple images within a timed interval. The user can set a timer interval in milliseconds[6], seconds or minutes and the total number of images he or she wants grabbed. The total acquisition time will be displayed. A folder must also be selected (browse button) and a partial filename and file format. The filename is concatenated automatically with the image number and additional zeros for alphabetical correctness. For example when grabbing 20 images in .png format they will be saved to disk as dummy000.png, dummy001.png, dummy002.png, ..., dummy018.png, dummy019.png.

When the *start* button is pushed we first validate the user input and then initiate a QTimer which sends out a signal at the specified interval. During grabbing all user input will be disabled except for the *stop* and *close* button. The *stop* button immediately stops the grabbing process, the close button only closes the dialog but grabbing will continue.

```
int timeScaleFactor = 1;

if(unitComboBox->currentText() == "sec") timeScaleFactor = 1000;
if(unitComboBox->currentText() == "min") timeScaleFactor = 60*1000;

//set the intervaltimer (in milliseconds)
intervalTimer->setInterval((intervalLine->text().toUInt())*timeScaleFactor);

//and start the timer
intervalTimer->start();
```

[6]Intervals lower than roughly 100ms are not realistic due to the framerate of the camera and the grabbing performance of our application.

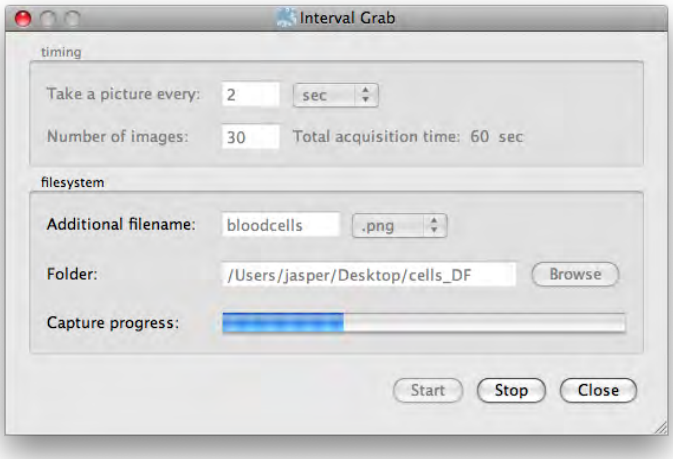


Figure 4.11: Grab images with regular intervals with the MultigrabDialog.

When all pictures are taken the timer is stopped with *intervalTimer->stop()*. Saving the images to disk it not done by the MultigrabDialog class itself. It just sends a signal to the Monitor class with the specified filename. The Monitor class will save the current QImage in memory (the most recently grabbed camera frame) to disk via a simple Qt save function.

```
void Monitor::saveImage(QString filename){

//if format = 0 QImage will guess according to the filename
//100 = max quality!

q_image->save(filename, 0, 100);
}
```

In the next section we will discuss the recording of a *full-motion* movie by encoding the camera stream directly to a movie file. However, when we are observing very slow evolving processes this is generally not a good idea as most of the time nothing interesting will happen. Also file size and consequently the hard drive space should be taken into account. As we will use parts of ffmpeg in our C/C++ project (see: section 4.5.5) we can also use ffmpeg as a stand-alone terminal application to combine separate images into a movie file. These separate images can easily be acquired with our MultigrabDialog as seen above.

```
ffmpeg -r [framerate] -b [bitrate] -i [input] outputfile.mp4
```

When the framerate and the bitrate are left out the defaults are taken, respectively 25fps and 200kbits/s. The extension of the ouput file defines the codec used. In the above example mpeg4 is used but ffmpeg supports a wide range of other codecs too. For example when a series of images is taken and named sequentially as rgb000.png, rgb001.png, rgb002.png, ... we can combine them into a movie file with the following command.

```
ffmpeg -r 10 -b 2000 -i rgb%03d.png rgb2000.mp4
```

In which case the bitrate is the most prominent factor for quality after compression. The width and height of the resulting movie will be the same as the input images unless specified otherwise by an extra *-s parameter*[7].

4.5.5 RecordDialog

The RecordDialog class uses ffmpeg's *libavcodec* to encode raw frames from the camera to an mpeg1 stream. Ffmpeg is actually a platform independent command line tool for audio/video conversion, libavcodec is a part of this project but can be used separately. Together with libavformat (an audio/video mux/demux library) both are used in a variety of open source multiplatform projects (e.g. ffdshow, MPlayer, Handbrake, ...). More info can be found on http://ffmpeg.mplayerhq.hu/ including the source code you will need to compile our application.

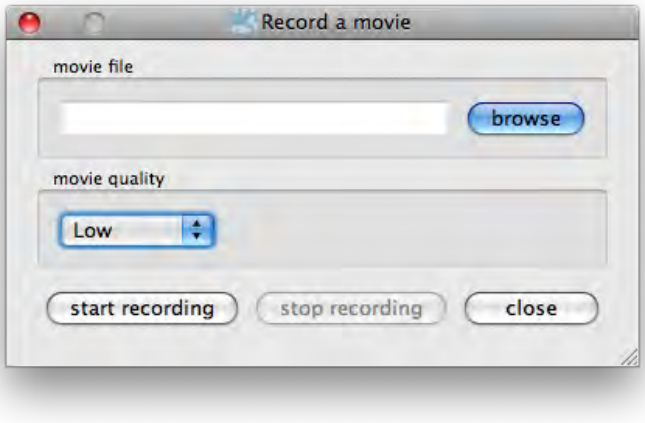


Figure 4.12: Record an MPEG video with the RecordDialog class.

The ffmpeg mpeg encoder only takes YUV420 frames as valid input, so again we'll have to do some sort of conversion. The YUV420 format is a planar format which is different from the regular image presentation of alternating (A)RGB or YUV pixel values in one matrix. The image data is stored in three separate matrices but only one U and one V pixel is stored for every four Y pixels. This means color information is reduced to only 1/4th of a regular YUV or RGB image presentation, as can be seen in figure 4.13 . YUV420 is commonly used in video compression because the reduced color information and the separate planes contribute to better compression techniques. The slight loss of color information is not really noticeable as the human eye is more sensitive to luminance than to chrominance.

[7]See the ffmpeg online documentation for the different size options.

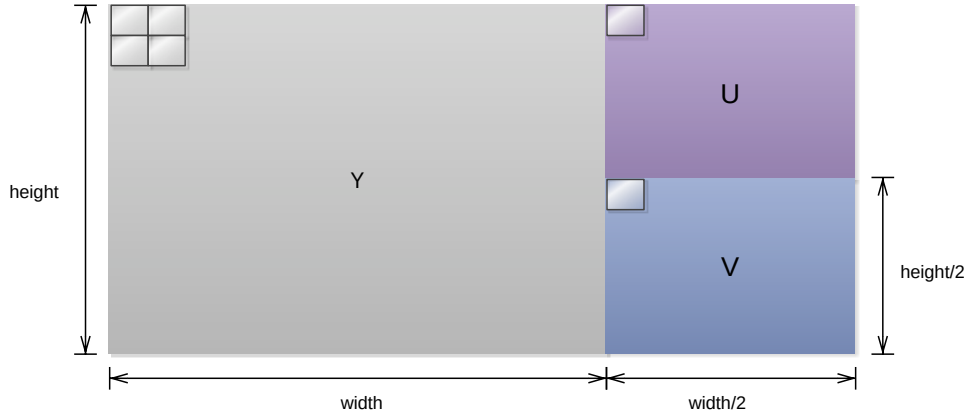


Figure 4.13: Graphical presentation of a YUV420 planar frame.

There's no real use in writing multiple conversion functions for YUV420 as we will always have to do two conversions anyway[8]. For this reason one function for ARGB32_to_YUV420 transformations should suffice. Every Y pixel is copied but only the U and V pixels for every *even* row and *even* column are copied. Although taking an average of the U/V values for every four pixels would be better than always copying the information of the fourth pixel, this would be computationally more expensive and does not yield noticeable image quality gain.

```
void RecordDialog::rgb32_to_yuv420frame(uint8_t *src, AVFrame *dest,
int width, int height)
{

register int i = ((width*height) << 2) - 1; //4 channels
register int j = (width*height) - 1; //1 channel
register int k = ((width/2)*(height/2)) - 1; //half a channel
register int y, u, v;
register int r, g, b, a;

for(int m=0;m<height;m++) {
    for(int n=0;n<width;n++) {

        a = (uint8_t) src[i--]; //ignore alpha channel
        r = (uint8_t) src[i--];
        g = (uint8_t) src[i--];
        b = (uint8_t) src[i--];

        //exec libdc1394 conversion macro
        RGB2YUV(r, g, b, y, u, v);

        // Y - grayscale plane - always copy this
        dest->data[0][j--] = y;
```

[8]One conversion to ARGB32 for Qt and another to YUV420 when actually recording.

```
            // one Cr/Cb value per block of four pixels
            if(((m%2)==0)&&((n%2)==0)){
                dest->data[1][k] = u;
                dest->data[2][k--] = v;
            }
        }
}
}
```

The *AVFrame* in which we save the resulting YUV420 image is a structure defined by libavcodec, we allocate memory for it and define pointers to the different planes.

```
uint8_t *yuv420_buf;
AVFrame *yuv420_frame;

//allocate ffmpeg frame
yuv420_frame = avcodec_alloc_frame();

//make room for the yuv420 image
yuv420_buf = (uint8_t*)malloc((c->width * c->height * 3) / 2);

//initialize the planes in the YUV420 frame
yuv420_frame->data[0] = yuv420_buf;
yuv420_frame->data[1] = yuv420_frame->data[0] + (c->width * c->height);
yuv420_frame->data[2] = yuv420_frame->data[1] + (c->width * c->height) / 4;
yuv420_frame->linesize[0] = c->width;
yuv420_frame->linesize[1] = c->width / 2;
yuv420_frame->linesize[2] = c->width / 2;
```

The codec itself is set up in the following manner. We always record at a steady 30fps as the mpeg1 standard does not support the same framerates as the camera. The quality of the recorded movie clip is defined by the bitrate, which can be set by the user. We allow three settings: *low* (1.25Mbit/s - VCD quality), *medium* (5Mbit/s - DVD quality) and *high* (15Mbit/s - HDTV quality). If the chosen quality (bitrate) is too low to encode a given frame size at it's lowest quality *avcodec* will ignore the bitrate we set and pick a new minimum one. The same counts for when a bitrate is chosen that's way to large for the selected frame size. So the bitrate value we set is more like a *hint*. Generally *medium* quality is the best option without the file size getting to big.

```
AVCodec *codec;

AVCodecContext *c;

//allocate codec context
c = avcodec_alloc_context();

//setup the codec properties
/*average bitrate, in bps*/
c->bit_rate = bitrate;
```

```
//resolution must be a multiple of two
c->width = width;
c->height = height;
//fps, mpeg1 supports 23.976, 24, 25, 29.97, 30, 50, 59.94, and 60
c->time_base= (AVRational){1,30};
//emit one intra frame every ten frames
//decreasing gop -> creates more of a MJPEG encoder than an MPEG
c->gop_size = 10;
c->max_b_frames=1;
//mpeg1 only takes yuv420 frames
c->pix_fmt = PIX_FMT_YUV420P;

//open the codec
if (avcodec_open(c, codec) < 0) {
    errorMessage("Could not open codec!");
    return;
}
```

Finally to actually encode the image and write it to a movie file we need an output buffer for the compressed image. We know it will generally not be bigger than the uncompressed YUV420 frame so it's sufficient to make the buffer just as big. When the user has selected a framerate lower than 30 for the camera we will need to write extra intermediate frames if we want the timebase to be correct. We do not allow recording of a movie when the camera's framerate is higher than 30fps. Performance wise, writing to file is generally not a problem but encoding can be a real bottleneck.

```
outbuf_size = (c->width * c->height * 3) / 2; //is sufficiently large
outbuf = (uint8_t*)malloc(outbuf_size);

FILE *f;
f = fopen(filenameString, "wb");
if (!f) {
    errorMessage("Could not open file!");
    return;
}

for(int i = 0; i < (30/realFramerate); i++){

    //flush the buffer
    fflush(stdout);

    //encode the image,
    //unfortunately this has to be done in the loop to minimalize
    //mpeg artifacts, try it outside the loop and see for yourself :-)
    out_size = avcodec_encode_video(c, outbuf, outbuf_size, yuv420_frame);
    fwrite(outbuf, 1, out_size, f);
}
```

As long as the out_size value is not 0 we still have data coming from the MPEG encoder. Therefore when recording is stopped we must continue to write these delayed frames. To make it a valid MPEG file we also add a certain byte sequence to the end of the file.

```
//write the delayed frames
while(out_size){
    fflush(stdout);
    out_size = avcodec_encode_video(c, outbuf, outbuf_size, NULL);
    fwrite(outbuf, 1, out_size, f);
}

//add sequence end code to have a valid mpeg file
outbuf[0] = 0x00;
outbuf[1] = 0x00;
outbuf[2] = 0x01;
outbuf[3] = 0xb7;
fwrite(outbuf, 1, 4, f);
fclose(f);      //close the file
```

The above ffmpeg implementation is based on the apiexample.c example of libavcodec. It is a pretty basic example of how to use ffmpeg encoding and decoding functions. If you want to learn more about ffmpeg or want to implement more sophisticated codecs (mpeg4, xvid, ...) you'll better have a look at the ffmpeg output_example.c file. Both can be found in the ffmpeg source package or on the official website (mentioned a few pages back).

4.5.6 ReticlesDialog

For microscopic imaging applications we added the ReticlesDialog class (fig. 4.14). Reticles or crosshairs can be used to align or center the image, but more importantly to do measurements on a microscopic scale. This method is also commonly used on non-digital microscope systems where the reticles are inlaid in the eyepieces.

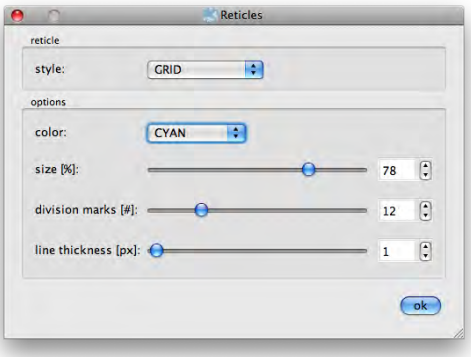


Figure 4.14: The ReticlesDialog class.

Three different kinds of reticles were implemented and 9 different colors are available (important to stand out from the background) (fig. 4.15). The reticles are calibrated by means of a stage micrometer (a glass plate with predefined engravings) and therefore we must fit the reticle to the micrometer. For this reason the user can alter the size, the number of divisions and the line thickness.

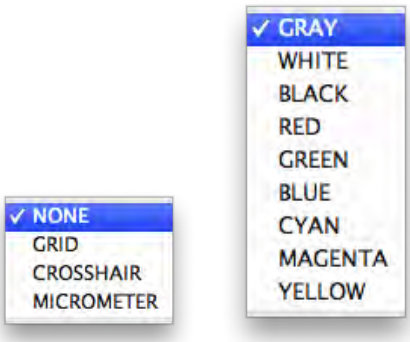


Figure 4.15: Reticle types and colors.

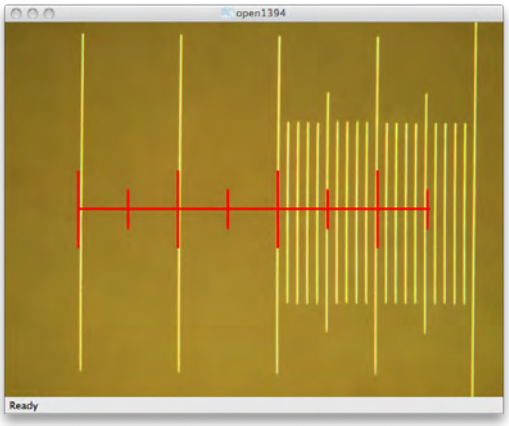


Figure 4.16: Calibrating the micrometer reticle.

An example of fitting the *micrometer* reticle to the real micrometer is shown in figure 4.16. When calibrated correctly we can roughly say that the red separators are 100μm and 50μm apart. Of course these settings (and therefore measurements) are only valid when the user uses the same objective for both the calibration and the inspection of the specimen.

Another example, of a grid this time, is given in figure 4.17. We first calibrate the grid on the smallest engravings in the stage micrometer (10μm apart) and then replace the micrometer glass with a human blood sample. We can now estimate the average size of a human bloodcell to be just under 10μm^2 (fig. 4.18).

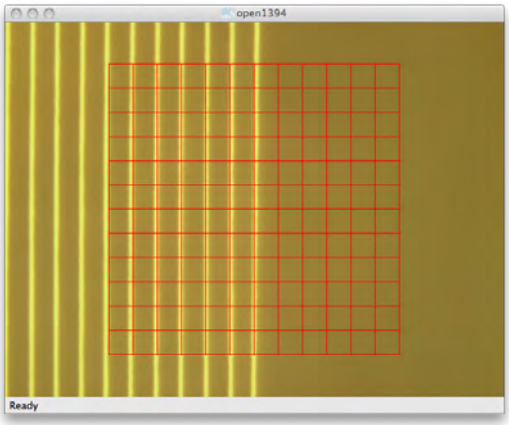


Figure 4.17: Calibrating the grid reticle, lines 10μm apart.

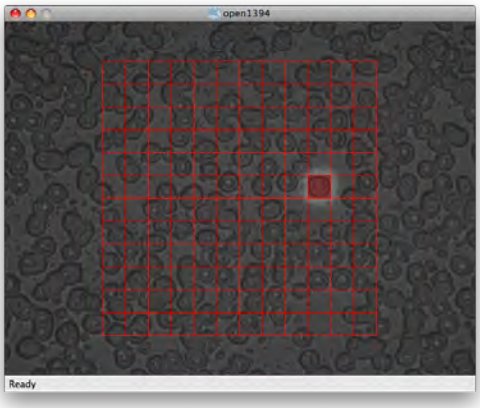


Figure 4.18: Estimate the size of human bloodcells.

The reticles are not drawn on the image itself but rather as an overlay. This is done inside the same paint event as the drawing of the image and is very fast (no noticeable delays with the reticles enabled). We will discuss the drawing functions we wrote to create the dynamic reticles. First, by means of the image width and height (which is the same size as the window viewfield) we can easily find the center of our drawing surface (X, Y). Z is defined as a percentage of the width or height, depending on the type, and equals the reticle's size. We can then constraint the region in which we have to draw by four coordinates.

(X - Z/2, Y)
(X + Z/2, Y)
(X, Y - Z/2)
(X, Y + Z/2)

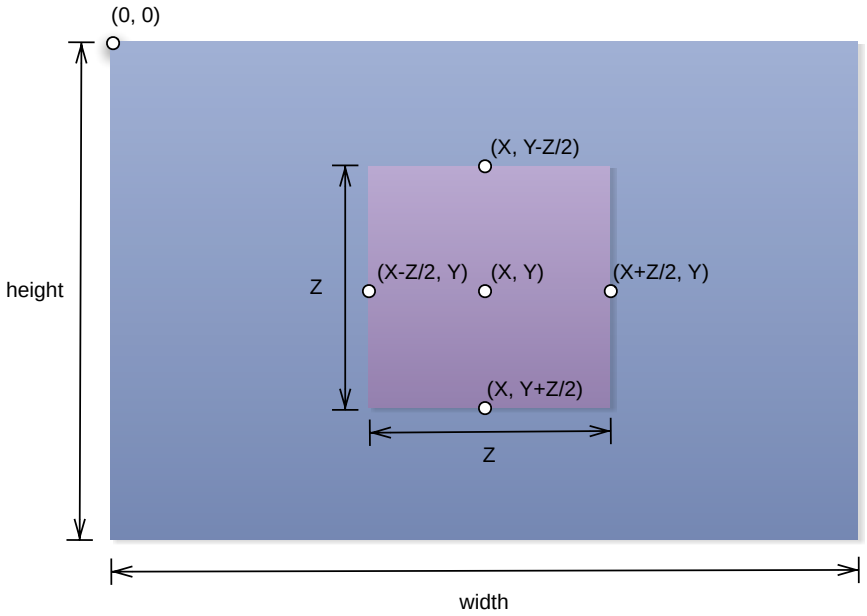


Figure 4.19: Drawing region for the reticles within the window viewfield.

Qt has a bunch of build-in drawing functions for its QPainter class. This class allows us to paint inside any Qt widget. We use drawLine(), which draws a line between two points in the 2D space, and drawEllipse() for an elliptic curve.

```
void drawLine ( int x1, int y1, int x2, int y2 )
void drawEllipse ( int x, int y, int width, int height )
```

With all this information the following drawing functions should be fairly comprehensible.

```
//draws a grid in both x & y directions
void drawGrid(QPainter *painter ,float size, int div, QColor color,
int thickness, int width, int height){
```

```
    //define the drawing pen
    QPen myPen(color);
    myPen.setWidth(thickness);
    painter->setPen(myPen);

    int X = width/2;
    int Y = height/2;
    //the grid is a square so limit it at the viewfield height
    int Z = (int)(height*(size/100));

    //Horizontal lines
    for(int i = 0; i <= div; i++){
        painter->drawLine(X-(Z/2), Y-(Z/2)+(i*(Z/div)), X+(Z/2),
        Y-(Z/2)+(i*(Z/div)));
    }

    //Vertical lines
    for(int i = 0; i <= div; i++){
        painter->drawLine(X-(Z/2)+(i*(Z/div)), Y-(Z/2),
        X-(Z/2)+(i*(Z/div)), Y+(Z/2));
    }
}

//draws a crosshair disc
void drawCrossHair(QPainter *painter ,float size , int div, QColor color,
int thickness, int width, int height){

    //define the drawing pen
    QPen myPen(color);
    myPen.setWidth(thickness);
    painter->setPen(myPen);

    int X = width/2;
    int Y = height/2;
    int Z = (int)(height*(size/100));
    int D = Z/div;

    //vertical line
    painter->drawLine(X, Y-(Z/2), X, Y+(Z/2));
    //horizontal line
    painter->drawLine(X-(Z/2), Y, X+(Z/2), Y);

    //height < width so limit the circle there
    for(int i = 0; i <= div; i++){
        painter->drawEllipse(X-(Z/2), Y-(Z/2), Z, Z);
        Z = Z-D;
    }
}
```

```
//draws a micrometer
void drawMicrometer(QPainter *painter ,float size, int div, QColor color,
int thickness, int width, int height){

    //define the drawing pen
    QPen myPen(color);
    myPen.setWidth(thickness);
    painter->setPen(myPen);

    int X = width/2;
    int Y = height/2;
    //the max size of the micrometer reticle is a percentage
    //of the width because it only expands horizontally
    int Z = (int)(width*(size/100));
    int D = height/10;
    bool toggle = true;

    //horizontal line
    painter->drawLine(X-(Z/2), Y, X+(Z/2), Y);

    //Vertical lines, alternation long/short
    for(int i = 0; i <= div; i++){
        if(toggle){
            painter->drawLine(X-(Z/2)+(i*(Z/div)), Y-D,
            X-(Z/2)+(i*(Z/div)), Y+D);
            toggle = !toggle;
        } else {
            painter->drawLine(X-(Z/2)+(i*(Z/div)), Y-(D/2),
            X-(Z/2)+(i*(Z/div)), Y+(D/2));
            toggle = !toggle;
        }
    }
}
```

4.5.7 OptionsDialog

The program options dialog is a very simple dialog and only has four features (fig. 4.20).

The first feature is the *negative* feature which basically just turns the camera stream into it's photo negative equivalent. This is a very easy and rather fast operation in Qt.

```
if(negative) q_image->invertPixels();
```

As you can see this operation is performed on the QImage and not on the actual raw image data associated with it (see: section 4.5.8). This is a small limitation as you will not get the inverted image when recording a movie since the recorder uses the original ARGB32 raw image as input. On the other hand, saving an image in negative is possible because the save() function is called on the QImage.

The second feature is the *frame limiter* feature. As we explained before one of the bottlenecks in our program can be drawing the fast sequence of images to the screen. If you were to expand our program with some image processing functionality you will not have a lot of CPU cycles left, especially at the higher resolutions. For this we provide functionality to only draw one in every x frames. This features is also useful when recording movies as the mpeg encoding is quite a costly operation and more free cpu time means less missed frames. A small improvement can be seen when comparing the CPU loads for 640x480 in YUV411 @ 30fps in figures 4.21 and 4.22 (only one in every 100 frames is actually drawn to the screen). These graphs also show us that the main bottleneck is situated in the video capture and conversion functions.

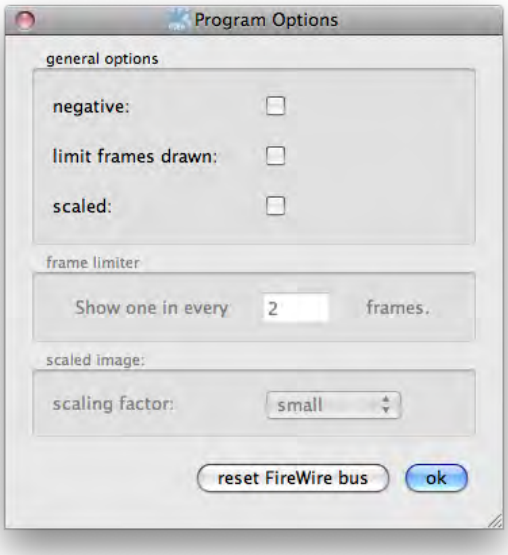


Figure 4.20: The OptionDialog class provides general program options.

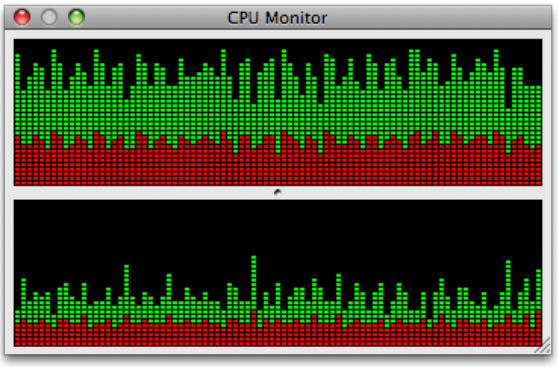


Figure 4.21: Duo Core CPU load with frame limiter disabled.

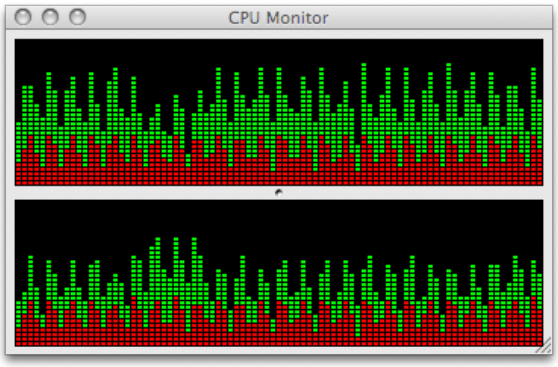


Figure 4.22: Duo Core CPU load with frame limiter enabled.

The third option is a scaling option. Initially this feature was added for the case when we capture resolutions that are too big to show on a small screen (e.g. 1600x1200). The feature only scales the image shown on screen and not the ones being grabbed or recorded. Four scaling factors can be chosen: *small* (320x240), *medium* (640x480), *large* (800x600) and *huge* (1024x768). Scaling down a big resolution shows a similar CPU graph improvement as seen with the *frame limiter* option, scaling up has of course a negative effect.

As explained in chapter 3, in exceptional cases, it can be necessary to reset the FireWire bus. The *reset FireWire bus* button also resets the whole program (closes the current instance and opens a new one). There's absolutely no harm in clicking this button, even when reset is not needed.

4.5.8 Monitor

Finally we get to the most important widget of our program. The Monitor widget is the central widget in the MainWindow. It receives signals from all the dialogs and reacts on them. It's main task is to initiate capture and poll the ring buffer for frames at regular intervals to ensure no missed frames. It also evokes the conversions to RGB32, loads the resulting data into a QImage and draws it to the screen. When selected by the user it will also call the reticle drawing functions, count the frames for the frame limiter option or convert the images to their negative. When all the capture information is received from the user dialogs the startCamera() function is called to initiate capture.

```
bool Monitor::startCamera(){

    dc1394error_t err = DC1394_SUCCESS;

    //set iso speed
    err = dc1394_video_set_iso_speed(camera, currentISOSpeed);

    if(err){
        libdcErrorMessage(err,"Could not set iso speed!");
        cleanup_and_exit();
        return false;
    }

    //set the camera mode
    err = dc1394_video_set_mode(camera, currentMode);

    if(err){
        libdcErrorMessage(err,"Could not set video mode!");
        cleanup_and_exit();
        return false;
    }

    /*format 7 specific settings HERE!*/
    if(format7Mode){
```

```
//set the format 7 image position
err = dc1394_format7_set_image_position(camera, currentMode,
currentFormat7Mode.pos_x, currentFormat7Mode.pos_y);

if (err){
    libdcErrorMessage(err,"Could not set format 7 image position!");
    cleanup_and_exit();
    return false;
}

//set the format 7 frame size
err = dc1394_format7_set_image_size(camera, currentMode,
currentFormat7Mode.size_x, currentFormat7Mode.size_y);

if (err){
    libdcErrorMessage(err,"Could not set format 7 image size!");
    cleanup_and_exit();
    return false;
}

//set the selected format 7 color coding
err = dc1394_format7_set_color_coding(camera, currentMode,
currentFormat7Mode.color_coding);

if (err){
    libdcErrorMessage(err,"Could not set format 7 color coding!");
    cleanup_and_exit();
    return false;
}

//Get the recommended byte per packet
uint32_t packet_bytes;
dc1394_format7_get_recommended_packet_size(camera, currentMode,
&packet_bytes);

//set this packet size
err = dc1394_format7_set_packet_size(camera, currentMode,
packet_bytes);

if (err){
    libdcErrorMessage(err,"Could not set format 7 packet size!");
    cleanup_and_exit();
    return false;
}

/*end of format7 settings*/

} else {
    //if not format 7 a framerate should be set!
```

```
        err = dc1394_video_set_framerate(camera, currentFramerate);

        if(err){
            libdcErrorMessage(err,"Could not set framerate!");
            cleanup_and_exit();
            return false;
        }
    }
    //capture setup and size of ringbuffer
    err = dc1394_capture_setup(camera, 4, DC1394_CAPTURE_FLAGS_DEFAULT);

    if(err){
        libdcErrorMessage(err, "Could not setup camera!.");
        cleanup_and_exit();
        return false;
    }

    //start the transmission
    err = dc1394_video_set_transmission(camera, DC1394_ON);

    if(err){
        libdcErrorMessage(err,"Could not start camera iso transmission!");
        cleanup_and_exit();
        return false;
    }
    return true;
}
```

The camera is now filling up the ring buffer with frames. Because we know the exact framerate[9] we can calculate the time in which a new frame is expected. We must also provide a buffer for the converted image. This must be done before every new capture as the frame size could have been changed by the settings.

```
timerInterval = (int)(1000/framerate);  //in milliseconds

//get the width and height of the used video mode
dc1394_get_image_size_from_video_mode(camera, currentMode,
&width, &height);

//allocate 4 channel image buffer for the future conversion here
//and only once -> prevent memory leaks
rgb32_image = (uint8_t *)malloc(4*width*height);

fps_timer->start(timerInterval);
```

Every time the interval is reached the timer sends a signal. This signal is connected to the grabFrame() slot which is then evoked. The function dequeues a frame and checks whether or not a NULL pointer was returned (because we POLL, see: chapter 3). If the frame is valid it is ours to use until we enqueue it again. We load the QImage as a

[9]Except when format 7 is used, in that case the framerate is set to 30.

QImage::Format_RGB32 format, other possibilities are the QImage::Format_ARGB32 and QImage::Format_ARGB32_Premultiplied formats. We use the regular one as we don't use the alpha layer and this is supposed to be faster. Take note that the QImage is associated with the rgb32_image buffer. This means that the QImage data is actually the same data as in the buffer, so the buffer cannot be freed before the QImage is destroyed. If the frame limiter option is not set then the frameLimit is set to 0. This ensures that the repaint event is called every time.

```
void Monitor::grabFrame(){

    dc1394error_t err = DC1394_SUCCESS;

    //dequeue the ring buffer
    err = dc1394_capture_dequeue(camera, DC1394_CAPTURE_POLICY_POLL,
    &frame);

    if(err){
        libdcErrorMessage(err,"Could not capture a frame!");
        cleanup_and_exit();
        return;
    }

    //check if a valid frame was polled
    if(!frame){
        //no need to redraw, no new frame anyways
        return;
    } else {
        //converts frame->image to rgb32_image
        if (frameToImage32(frame, rgb32_image) != DC1394_SUCCESS){
            qDebug() << "error with conversion";
            return;
        }

        //if we are recording emit a signal to the RecordDialog
        if(isRecording) emit writeFrame(rgb32_image);

        //destroy the previous q_image -> prevent memory leaks
        q_image->~QImage();
        //reload a new QImage with the new data
        q_image = new QImage(rgb32_image, width, height,
        QImage::Format_RGB32);

        if(q_image->isNull()){
            qDebug() << "loading QImage failed";
            return;
        };

        //next free the frame to make room for a new one
        dc1394_capture_enqueue(camera, frame);
```

```
        //if the user wants a negative image do the inversion
        if(negative) q_image->invertPixels();

        //frame limiter if enabled in the OptionsDialog
        if(frameCounter == frameLimit){
            frameCounter = 0;
            //manually envoke the paint event
            this->repaint();
        } else {
            frameCounter++;
        }
    }
}
```

The function *frameToImage32()* gets the color coding from the frame itself and then does the proper conversion by calling the specific conversion functions. These functions are defined in myconv.h. As you can see below the color modes RGB16/MONO16 are not tested but they should work. RAW16 is not supported yet.

```
dc1394error_t Monitor::frameToImage32(dc1394video_frame_t *frame,
uint8_t *rgb32_image){

    switch(frame->color_coding){
        case DC1394_COLOR_CODING_RGB16:
            //WARNING: RGB16 = EXPERIMENTAL = NOT TESTED!
            RGB16_to_RGB32(frame->image, rgb32_image, width, height,
            frame->data_depth);
            break;
        case DC1394_COLOR_CODING_YUV444:
            YUV444_to_RGB32(frame->image, rgb32_image, width, height);
            break;
        case DC1394_COLOR_CODING_YUV422:
            YUV422_to_RGB32(frame->image, rgb32_image, width, height,
            frame->yuv_byte_order);
            break;
        case DC1394_COLOR_CODING_YUV411:
            YUV411_to_RGB32(frame->image, rgb32_image, width, height);
            break;
        case DC1394_COLOR_CODING_MONO8:
        case DC1394_COLOR_CODING_RAW8:
            MONO8_to_RGB32(frame->image, rgb32_image, width, height);
            break;
        case DC1394_COLOR_CODING_MONO16:
            //WARNING: MONO16 = EXPERIMENTAL = NOT TESTED!
            MONO16_to_RGB32(frame->image, rgb32_image, width, height,
            frame->data_depth);
            break;
        case DC1394_COLOR_CODING_RAW16:
            qDebug() << "RAW16 not supported yet";
            return DC1394_FAILURE;
```

```
            break;
        case DC1394_COLOR_CODING_RGB8:
            RGB24_to_RGB32(frame->image, rgb32_image, width, height);
            break;
        default:
            qDebug() << "This color coding is not supported (yet)";
            return DC1394_FAILURE;
    }

    return DC1394_SUCCESS;
}
```

Next we'll take a look at the paintEvent() where the actual drawing happens. Except from drawing the image (scaled or not) and the reticles we also notify the user if we are currently recording and/or scaling. This is done by putting some text in the bottom left corner of the Monitor.

```
void Monitor::paintEvent(QPaintEvent *){          //here we draw!

    //painters should be local to the paintEvent!
    QPainter painter(this);

    int w = 0, h = 0;

    //set the proper size
    if(isScaled){
        w = scaledWidth;
        h = scaledHeight;
    } else {
        w = width;
        h = height;
    }

    //define the viewfield
    QRectF target(0.0, 0.0, w, h);

    //finally, draw the image onto the widget
    if(isScaled){
        QImage scaledImage;
        scaledImage = q_image->scaled(scaledWidth, scaledHeight,
        Qt::IgnoreAspectRatio, Qt::FastTransformation);
        painter.drawImage(target, scaledImage);
    } else {
        painter.drawImage(target, *q_image);
    }

    //determine if we should overlay a reticle
    //defined in reticles.h and reticles.cpp
    switch(currentStyle){
        case NONE:
```

```
            break;
        case GRID:
            drawGrid(&painter, currentSize, currentDiv,
            currentColor, currentThickness, w, h);
            break;
        case CROSSHAIR:
            drawCrossHair(&painter, currentSize, currentDiv,
            currentColor, currentThickness, w, h);
            break;
        case MICROMETER:
            drawMicrometer(&painter, currentSize, currentDiv,
            currentColor, currentThickness, w, h);
            break;
        default:
            break;
    }

    //if we are recording or scaling, notify the user
    if(isRecording && !isScaled){
        painter.setPen(Qt::red);
        painter.drawText(10, h-10, "recording...");
    }

    if(isScaled && !isRecording){
        painter.setPen(Qt::red);
        painter.drawText(10, h-10, "scaled");
    }

    if(isScaled && isRecording){
        painter.setPen(Qt::red);
        painter.drawText(10, h-10, "scaled & recording...");
    }
}
```

You might have noticed the cleanup_and_exit() function in the error handling parts of the code. This is an important function as it must stop capture in a clean way. It determines in what state the program is in and depending on this state it frees certain buffers and pointers in a particular order.

```
void Monitor::cleanup_and_exit(){

    if(isCapturing){
        fps_timer->stop();
        dc1394_video_set_transmission(camera, DC1394_OFF);
        dc1394_capture_stop(camera);
        //free the q_image before we free the corresponding rgb32_image
        q_image->~QImage();
        q_image = new QImage(":/images/nocamera.png");
        free(rgb32_image);
        isCapturing = false;
```

```
    }

    if(isInitialized){
        if(reset){
            qDebug() << "Resetting bus...";
            dc1394_reset_bus(camera);
        }
        dc1394_camera_free(camera);
        dc1394_free(d);
        isInitialized = false;
    }

    qDebug() << "camera successfully freed";
}
```

In the *tools* menu you can pick the *grab single image* function. This opens a standard Qt file dialog and is therefore not one of our own self-created widgets. The standard pad is the location from where our program was started. By altering the file extension you can determine the file format. When the *save* button is clicked a signal is sent to the Monitor to save the most recently grabbed image to disk (see: section 4.5.4). There's a slight difference between the file dialog on Linux/Mac because it takes over the OS's native layout and looks (fig. 4.24 and fig. 4.23).

```
void MainWindow::grabImage(){

    QString fileName = QFileDialog::getSaveFileName(this,
    tr("Save image to disk"), "untitled.png",
    tr("Images (*.png *.jpg *.jpeg *.bmp *.tiff)"));

    if(!fileName.isEmpty()) emit signalSaveImage(fileName);
}
```

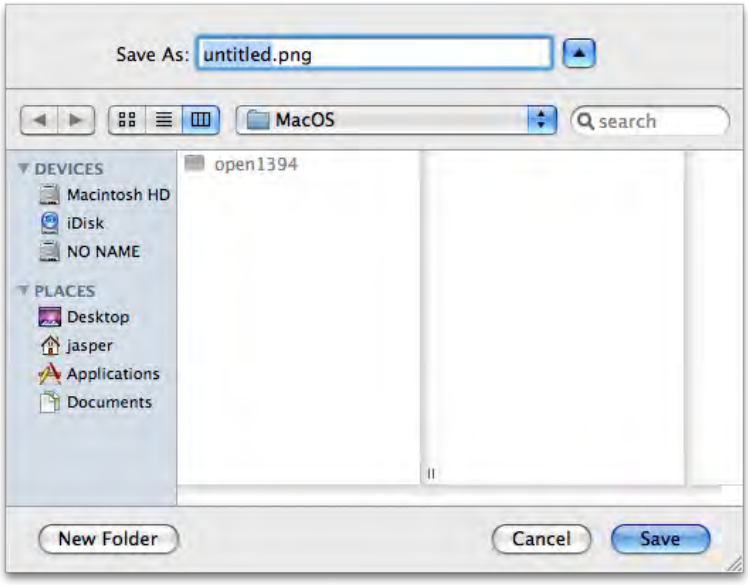


Figure 4.23: The file dialog on Mac os X.

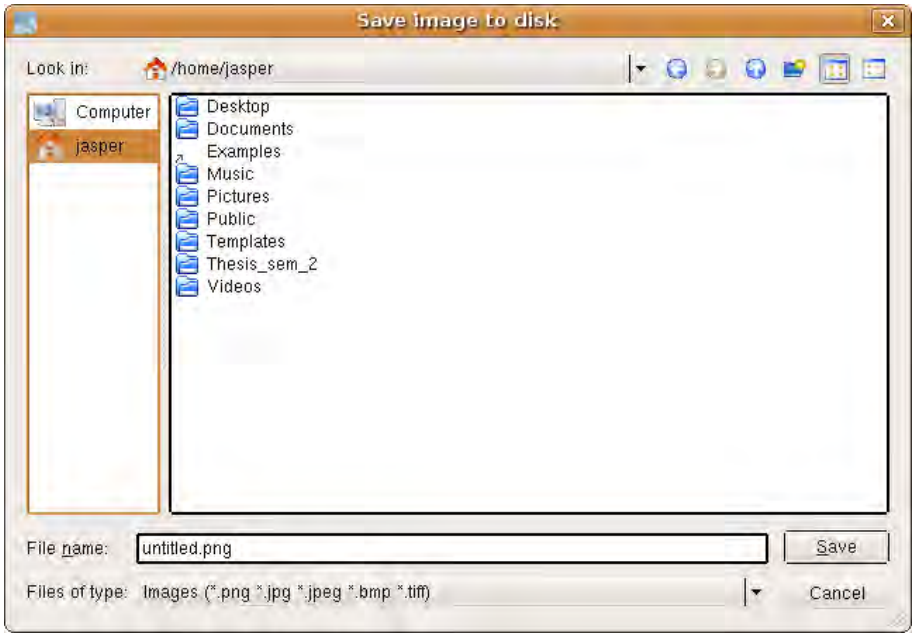


Figure 4.24: The file dialog on Linux.

Chapter 5

Conclusion

Although optical microscope theory is not an easy subject we've succeeded in acquiring high quality photomicrographs with a fairly old microscope. This also tells a lot about the build quality of such an expensive optical system, which is still worth a considerable amount of money today. We reinstate it by modernizing it with digital cameras and accompanying software so it can be used for future computer vision projects on the microscopic scale. As for the software itself, it aims to be as platform-independent as possible and is compatible with all IIDC FireWire cameras making it useful for other industrial/scientific projects as well.

This all sounds very nice but of course there is always some room left for improvement. First of all the color format transformation functions should be optimized as they consume a lot of CPU time especially at higher resolutions and framerates. This can probably be accomplished by using SSE instructions or even GPU specific features but my knowledge concerning these subjects is limited. Furthermore a faster way to draw the sequences of images to the screen in Qt could also be considered. Although we already have some hardware acceleration by drawing in a QGLWidget instead of a regular QWidget there might be better alternatives. Qt supports openGL on all platforms so a direct openGL implementation should be possible. We should note that drawing the images is only a substantial bottleneck on slower machines. On newer machines it can be neglected so improvement here might not be needed. The conversion functions remain the most noticeable bottleneck even on fast machines. Luckily GUI responsiveness is never really a problem as Qt seems to handle this nicely. As for the MPEG1 encoder taken from the ffmpeg project we can assume this to be reasonably optimized as ffmpeg is a mature project used in many open source video applications. The software functionality itself can of course also be extended as the source code is available on the CD accompanying this book. To keep the software up-to-date changes in the libdc1394 project should be followed up regularly as this is a highly active library. Full compliance with the latest version should also ensure an easy transaction when the Windows port is finally released.

I'm pretty sure my software will be put to good use and if you are using it or are altering the source code you can always contact me for questions via email

j_leemans@hotmail.com

I'll do my very best to answer your questions.

Bibliography

[1] MICHAEL W. DAVIDSON, MORTIMER ABRAMOWITZ, *Optical Microscopy*, http://micro.magnet.fsu.edu/primer/pdfs/microscopy.pdf, 1999, 41 pages.

[2] MORTIMER ABRAMOWITZ, *Mircoscope Basics and Beyond*, http://micro.magnet.fsu.edu/primer/pdfs/basicsandbeyond.pdf, 2003, 50 pages.

[3] MICHAEL W. DAVIDSON, *Molecular Expressions*, http://microscopy.fsu.edu/, Florida State University Research Foundation.

[4] VARIOUS AUTHORS, *MicroscopyU*, http://www.microscopyu.com/, Nikon Inc., Florida State University and Molecular Expressions.

[5] VARIOUS AUTHORS, http://hyperphysics.phy-astr.gsu.edu/hbase/vision/, Georgia State University.

[6] ADOBE, *The Physiology of Human Vision*, http://dba.med.sc.edu/price/irf/Adobe_tg/color/vision.html.

[7] A. E. CONRADY, *Applied Optics and Optical Design*, part two, Dover Publications, 1960, 841 pages.

[8] PAUL SUETENS, *Fundamentals of Medical Imaging*, 4th printing, Cambridge University Press, 2002, 280 pages.

[9] VARIOUS AUTHORS, *IIDC 1394-based Digital Camera Specification*, version 1.31, 1394 Trade Association, 2004, 85 pages.

[10] VARIOUS AUTHORS, *GenICam Standard: Generic Interface for Cameras*, Version 1.0, European Machine Vision Association, 2007, 46 pages.

[11] BRIAN W. KERNIGHAN, DENNIS M. RITCHIE, *The C Programming Language*, second edition, 35th printing, Prentice Hall PTR, 1988, 272 pages.

[12] HERBERT SCHILDT, *C++: The Complete Reference*, fourth edition, McGraw-Hill/Osborne, 2003, 1023 pages.

[13] JASMIN BLANCHETTE, MARK SUMMERFIELD, *C++ GUI Programming with Qt 4*, 3rd printing, Prentice Hall/Trolltech Press, 2006, 537 pages.

[14] TROLLTECH, *Qt developer documentation*, http://doc.trolltech.com/.

Appendix A

IIDC Video Formats & Modes

4.2.2 Inquiry register for video mode

Offset	Name	Field	Bit	Description
180h	V_MODE_INQ_0 (Format_0)	Mode_0	[0]	160 X 120 YUV(4:4:4) Mode (24bit/pixel)
		Mode_1	[1]	320 X 240 YUV(4:2:2) Mode (16bit/pixel)
		Mode_2	[2]	640 X 480 YUV(4:1:1) Mode (12bit/pixel)
		Mode_3	[3]	640 X 480 YUV(4:2:2) Mode (16bit/pixel)
		Mode_4	[4]	640 X 480 RGB Mode (24bit/pixel)
		Mode_5	[5]	640 X 480 Y (Mono) Mode (8bit/pixel)
		Mode_6	[6]	640 X 480 Y (Mono16) Mode (16bit/pixel)
		Mode_x	[7]	Reserved for another Mode
		-	[8..31]	Reserved
184h	V_MODE_INQ_1 (Format_1)	Mode_0	[0]	800 X 600 YUV(4:2:2) Mode (16bit/pixel)
		Mode_1	[1]	800 X 600 RGB Mode (24bit/pixel)
		Mode_2	[2]	800 X 600 Y (Mono) Mode (8bit/pixel)
		Mode_3	[3]	1024 X 768 YUV(4:2:2) Mode (16bit/pixel)
		Mode_4	[4]	1024 X 768 RGB Mode (24bit/pixel)
		Mode_5	[5]	1024 X 768 Y (Mono) Mode (8bit/pixel)
		Mode_6	[6]	800 X 600 Y (Mono16) Mode (16bit/pixel)
		Mode_7	[7]	1024 X 768 Y (Mono16) Mode (16bit/pixel)
		-	[8..31]	Reserved
188h	V_MODE_INQ_2 (Format_2)	Mode_0	[0]	1280 X 960 YUV(4:2:2) Mode (16bit/pixel)
		Mode_1	[1]	1280 X 960 RGB Mode (24bit/pixel)
		Mode_2	[2]	1280 X 960 Y (Mono) Mode (8bit/pixel)
		Mode_3	[3]	1600 X 1200 YUV(4:2:2) Mode (16bit/pixel)
		Mode_4	[4]	1600 X 1200 RGB Mode (24bit/pixel)
		Mode_5	[5]	1600 X 1200 Y (Mono) Mode (8bit/pixel)
		Mode_6	[6]	1280 X 960 Y (Mono16) Mode (16bit/pixel)
		Mode_7	[7]	1600X 1200 Y (Mono16) Mode (16bit/pixel)
		-	[8..31]	Reserved
18Ch : 197h	Reserved for other V_MODE_INQ_x for Format_x.			
198h	V_MODE_INQ_6 (Format_6)	Mode_0	[0]	Exif format
		Mode_x	[1..7]	Reserved for another Mode
		-	[8..31]	Reserved
19Ch	V_MODE_INQ_7 (Format_7)	Mode_0	[0]	Format_7 Mode_0
		Mode_1	[1]	Format_7 Mode_1
		Mode_2	[2]	Format_7 Mode_2
		Mode_3	[3]	Format_7 Mode_3
		Mode_4	[4]	Format_7 Mode_4
		Mode_5	[5]	Format_7 Mode_5
		Mode_6	[6]	Format_7 Mode_6
		Mode_7	[7]	Format_7 Mode_7
		-	[8..31]	Reserved

0-7	8-15	16-23	24-31
V_MODE_INQ	Reserved		

Initial values	System dependent
Read values	System dependent. Same value to Initial value
Write effect	Ignored

Appendix B

Philips 7023 Datasheet

7023 100W GY6.35 12V 1CT

Product family description
Low-voltage, flat-filament quartz halogen lamps

Product Features

- Small bulb shape and high luminous intensity
- Lamps with prefocus base have a better-defined position of the filament in the system, allowing lamp replacement without adjustment
- XHP lamps are optimised for maximum light output by using xenon filling gas within IEC limits for these types

Product Benefits

- Specially suited for use in compact, efficient projection systems
- Constant, high light output during lifetime
- Distortion-free quartz bulb for optimal beam performance

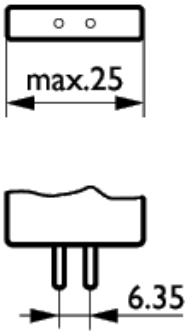
Application

- Studio, film, theatre and disco lighting
- Slide, overhead, profile and 8/16 mm film projectors
- Microfilm readers and reader printers
- Dental lights
- Microscopes and endoscopes

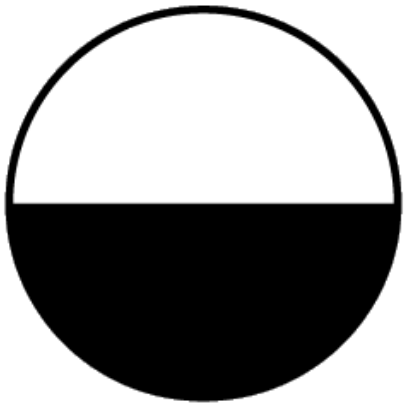
Product data	
Order code	409812 50
Full product code	871150040981250
Full product name	7023 100W GY6.35 12V 1CT
Order product name	7023 100W GY6.35 12V 1CT/10X10F
Packing type	1 Carton
Pieces per pack	1
Packing configuration	10X10F
Packs per outerbox	100
Bar code on pack - EAN1	8711500409812
Bar code on intermediate packing - EAN2	8711500420015
Bar code on outerbox - EAN3	8711500423269

Product data	
Logistic code(s) - 12NC	9238 700 17103
ILCOS code	
Net weight per piece	2.640 GR
Successor order code	
Philips Code	7023
ANSI Code	FCR
LIF Code	A1/215
Cap-Base	GY6.35
Bulb Material	Quartz-UV Open
Filament Shape	Flat
Burning Position	s90
Main Application	Projection
Life to 50% failures	50 hr
Rated Lamp Wattage	100W
Voltage	12V
Color Rendering Index	100 Ra8
Lamp Luminous Flux	3400 Lm
Filament Dimensions (WxH) [mm]	4.2x2.3

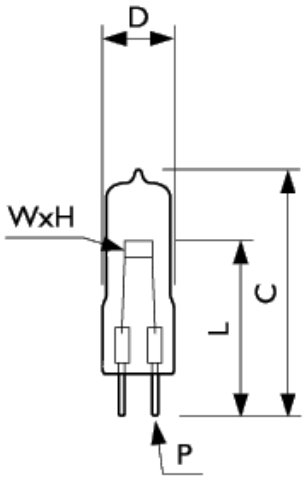
XHMPSEFF GY6.35



Cap-Base GY6.35



Burning Position s90



XHMPSEFF G6.35/GY6.35

	C	D	L	L	L	P	P	P
Full product name	Max	Max	Min	Nom	Max	Min	Nom	Max
7023	44	11.5	29.75	30	30.25	1.20	1.25	1.30

	C	D	L	L	L	P	P	P
Full product name	Max	Max	Min	Nom	Max	Min	Nom	Max
100W GY6.35 12V 1CT								

Appendix C

Conversion functions to (A)RGB32

```
/*
 *     myconv.h - headerfile - different types of conversion functions
 *     Leemans Jasper
 *     De Nayer Instituut / Visics Research Group / 2007-2008
 */

#ifndef MYCONV_H          //ensure header file is only processed once
#define MYCONV_H

//Qt includes
#include <QtGui>

///libdc1394 includes
#include <dc1394/dc1394.h>

//to rgb32
void RGB24_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height);
void MONO8_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height);
void YUV444_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height);
void YUV411_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height);
void YUV422_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height, uint32_t byte_order);
void RGB16_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height, uint32_t bits);
void MONO16_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height, uint32_t bits);

#endif
```

```
/*
*    myconv.cpp - different types of conversion functions
*    Leemans Jasper
*    De Nayer Instituut / Visics Research Group / 2007-2008
*/

#include "myconv.h"

/*****************************************************************
*
* CONVERSION FUNCTIONS TO RGB 32bit QImage 0xffRRGGBB/0xAARRGGBB
*
*****************************************************************/

void RGB24_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height){

register int i = (width*height) + ( (width*height) << 1 ) - 1; //start at the b
plane of the source
register int j = ((width*height) << 2) - 1; //4 channels
register int r, g, b, alpha = 255; //variables for switching, opacity 100%

while (j >= 0) {

    b = (uint8_t) src[i--]; //blue
    g = (uint8_t) src[i--]; //green
    r = (uint8_t) src[i--]; //red

    dest[j--] = alpha;
    dest[j--] = r;   //RED
    dest[j--] = g; //GREEN
    dest[j--] = b; //BLUE
    }
}

//This is the mac version
void MONO8_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height){

  register int i = (width*height) - 1;
  register int j = ((width*height) << 2) - 1; //4 channels
  register int y, alpha = 255;

while (i >= 0) {
    y = (uint8_t) src[i--];
    dest[j--] = alpha;
    dest[j--] = y;   //RED
    dest[j--] = y; //GREEN
    dest[j--] = y; //BLUE
    }
}

void YUV444_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
```

```
uint32_t width, uint32_t height){

  register int i = (width*height) + ( (width*height) << 1 ) - 1;
  register int j = ((width*height) << 2) - 1; //4 channels
  register int y, u, v;
  register int r, g, b, alpha = 255;

  while (i >= 0) {
    v = (uint8_t) src[i--] - 128;
    y = (uint8_t) src[i--];
    u = (uint8_t) src[i--] - 128;
    YUV2RGB (y, u, v, r, g, b);
    dest[j--] = alpha;
    dest[j--] = r;  //RED
    dest[j--] = g; //GREEN
    dest[j--] = b; //BLUE
  }
}

void YUV411_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height){

  register int i = (width*height) + ( (width*height) >> 1 ) - 1;
  register int j = ((width*height) << 2) - 1; //4 channels
  register int y0, y1, y2, y3, u, v;
  register int r, g, b, alpha = 255;

  while (i >= 0) {
    y3 = (uint8_t) src[i--];
    y2 = (uint8_t) src[i--];
    v  = (uint8_t) src[i--] - 128;
    y1 = (uint8_t) src[i--];
    y0 = (uint8_t) src[i--];
    u  = (uint8_t) src[i--] - 128;
    YUV2RGB (y3, u, v, r, g, b);
    dest[j--] = alpha;
    dest[j--] = r;  //RED
    dest[j--] = g; //GREEN
    dest[j--] = b; //BLUE
    YUV2RGB (y2, u, v, r, g, b);
    dest[j--] = alpha;
    dest[j--] = r;  //RED
    dest[j--] = g; //GREEN
    dest[j--] = b; //BLUE
    YUV2RGB (y1, u, v, r, g, b);
    dest[j--] = alpha;
    dest[j--] = r;  //RED
    dest[j--] = g; //GREEN
    dest[j--] = b; //BLUE
    YUV2RGB (y0, u, v, r, g, b);
    dest[j--] = alpha;
    dest[j--] = r;  //RED
    dest[j--] = g; //GREEN
```

```
    dest[j--] = b; //BLUE
  }
}

void YUV422_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height, uint32_t byte_order){

  register int i = ((width*height) << 1) - 1;
  register int j = ((width*height) << 2) - 1; //4 channels
  register int y0, y1, u, v;
  register int r, g, b, alpha = 255;


  switch (byte_order) {
  case DC1394_BYTE_ORDER_YUYV:
    while (i >= 0) {
        v  = (uint8_t) src[i--] -128;
        y1 = (uint8_t) src[i--];
        u  = (uint8_t) src[i--] -128;
        y0  = (uint8_t) src[i--];
        YUV2RGB (y1, u, v, r, g, b);
        dest[j--] = alpha;
        dest[j--] = r;  //RED
        dest[j--] = g; //GREEN
        dest[j--] = b; //BLUE
        YUV2RGB (y0, u, v, r, g, b);
        dest[j--] = alpha;
        dest[j--] = r;  //RED
        dest[j--] = g; //GREEN
        dest[j--] = b; //BLUE
    }
    break;
  case DC1394_BYTE_ORDER_UYVY:
    while (i >= 0) {
        y1 = (uint8_t) src[i--];
        v  = (uint8_t) src[i--] - 128;
        y0 = (uint8_t) src[i--];
        u  = (uint8_t) src[i--] - 128;
        YUV2RGB (y1, u, v, r, g, b);
        dest[j--] = alpha;
        dest[j--] = r;  //RED
        dest[j--] = g; //GREEN
        dest[j--] = b; //BLUE
        YUV2RGB (y0, u, v, r, g, b);
        dest[j--] = alpha;
        dest[j--] = r;  //RED
        dest[j--] = g; //GREEN
        dest[j--] = b; //BLUE
    }
    break;
  default:
    fprintf(stderr,"Invalid overlay byte order\n");
    break;
```

```
  }
}

/**********************************************************************
*
* 16BIT functions are experimental = not tested but should work!
*
**********************************************************************/

void RGB16_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height, uint32_t bits){

  register int i = (((width*height) + ( (width*height) << 1 )) << 1) - 1;
  register int j = ((width*height) << 2) - 1;
  register int r, g, b, alpha = 255;

while (i >= 0) {
    b = src[i--];
    b = (b + (src[i--]<<8))>>(bits-8);
    g = src[i--];
    g = (g + (src[i--]<<8))>>(bits-8);
    r = src[i--];
    r = (r + (src[i--]<<8))>>(bits-8);
    dest[j--] = alpha;
    dest[j--] = r;
    dest[j--] = g;
    dest[j--] = b;
    }
}

void MONO16_to_RGB32(uint8_t *restrict src, uint8_t *restrict dest,
uint32_t width, uint32_t height, uint32_t bits){

  register int i = ((width*height) << 1) - 1;
  register int j = ((width*height) << 2) - 1;
  register int y, alpha = 255;

while (i > 0) {
        y = src[i--];
        y = (y + (src[i--]<<8))>>(bits-8);
        dest[j--] = alpha;
        dest[j--] = y;
        dest[j--] = y;
        dest[j--] = y;
    }
}
```