Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK

A semi-automatic annotation tool for image data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik & Digitale Bildverarbeitung

ausgeführt von

Dawid Sip
Matrikelnummer 0126051

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: A.o.Univ.-Prof.Dipl.-Ing.Dr.techn. Markus Vincze
Mitwirkung: Dipl.-Ing. Horst Wildenauer

Wien, 19.01.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

A semi-automatic annotation tool for image data

MASTER THESIS

for the completion of the degree requirements

Diplom-Ingenieur

in the study of

Computer graphics & digital image-processing

by

Dawid Sip
Matriculation Number 0126051

at the Automation and Control Institute at Vienna University of Technology

Supervision:
Supervisor: A.o.Univ.-Prof.Dipl.-Ing.Dr.techn. Markus Vincze
Supervisor's assistant: Dipl.-Ing. Horst Wildenauer

Vienna, 19. 01. 2009

(Authors signature) (Supervisors signature)

I would like to dedicate this work to my mother
- Irena Sip

Abstract

To introduce mobile robots into a home environment, ways for interaction with this environment need to be provided. One attempt is to teach robots to recognize objects with the help of learning algorithms. To train these algorithms to efficiently recognize objects like chairs, couches or tables, a large database of images containing these objects is required.

The existence of certain objects within an image can be expressed with image annotation. The term annotation generally denotes an extra information added to a document. Therefore image annotation is additional information superimposed on an image.

Images can be annotated in a variety of ways. One way would be to use a graphics manipulation program to add a named layer on top of the object. The drawback of such approach is that loading and processing of the individual images takes too long time. Thus, a tool for efficient annotation of large image databases is necessary.

Nowadays many tools exist for applying annotation to images and creating annotation databases. Unfortunately, most of those tools weren't made available outside the research groups where they were developed. Tools that are available are often designed to solve a specific problem and are not easily applicable for more general tasks.

In this thesis we present Ultimate Annotation, a new application for efficient, semi-automatic annotation of image data. The way we attempt to annotate images is through the use of polygons. Polygons allow to tightly surround the object's silhouette, and efficiently represent annotation. We make use of a line approximation procedure and a segmentation procedure to achieve a semi-automatic behavior. We provide means for quick browsing through a large number of images. The user interface is designed to support quick access to application functionalities, mainly through mouse and keyboard. The project is open and has a modular composition which allows other groups to contribute and add extensions. All these features combined help to efficiently build large databases of annotated images.

Acknowledgements

I would like to express my gratitude to the people at Automation and Control Institute and especially the robots@home group for their hospitality and for teaching me how to play table soccer.

I would like to give my special thanks to my family, my mother, father and sister, who were always very supportive along my studies. I also thank my friends, for their general support, especially Angelo Ferrari for helping me proof reading this work.

Contents

Abstract i

Acknowledgements ii

1. Introduction 1

1.1. Outline of the document . . . 1
1.2. Motivation . . . 2
1.3. Related work . . . 4
1.3.1. LabelMe . . . 6
1.3.2. Image Markup Tool . . . 9
1.3.3. Image Manager . . . 12
1.4. Requirements . . . 14

2. Implementation 17

2.1. Project environment . . . 17
2.1.1. Language . . . 17
2.1.2. Graphics framework . . . 18
2.2. User interface design . . . 19
2.2.1. Graphics view . . . 19
2.2.2. Zooming . . . 22
2.2.3. View-port transformation . . . 24
2.3. Graphical User Interface . . . 25
2.3.1. Main application window . . . 26
2.3.2. Dialog windows . . . 29
2.4. Annotation . . . 31
2.4.1. Simple polygon . . . 32
2.4.2. Polygon approximation . . . 36
2.4.3. Polygon from SIOX . . . 39

2.5. Managing the annotation . . . 43

3. Future work 45
3.1. More objects for annotation . . . 45
3.2. Hierarchies . . . 48

4. Conclusion 50

A. Acronyms 52

B. Users Manual 53
B.1. System requirements . . . 53
B.2. Launching the Ultimate Annotation . . . 53
B.3. Closing the Ultimate Annotation . . . 53
B.4. Choosing the image . . . 54
B.5. Zooming . . . 55
B.6. Virtual Desktop . . . 55
B.7. Annotation . . . 55
B.7.1. Choosing polygon options . . . 56
B.7.2. Simple polygon . . . 56
B.7.3. Curve Polygon . . . 57
B.7.4. SIOX Polygon . . . 57
B.8. Polygon manipulation . . . 58
B.8.1. Select a polygon . . . 58
B.8.2. Delete a polygon . . . 59
B.8.3. Looking up the polygon properties . . . 59
B.8.4. Changing depth . . . 59
B.8.5. Select a Control Point . . . 59
B.8.6. Move a Control Points . . . 60
B.8.7. Add a Control Point . . . 61
B.8.8. Delete a Control Point . . . 61

C. Java class diagram 62

D. Java class documentation 63
D.1. Class `AnnoTool` . . . 63
D.2. Class `ImageTableModel` . . . 63

D.3. Class LazyPixmap 64
D.4. Class MainWindow 66
D.5. Class MyGraphicsPolygonItem 68
D.6. Class MyGraphicsView 68
D.7. Class MyPolygon 70
D.8. Class Utility 71

E. Bibliography **72**

List of Figures

1.1. (a) Rosie the robot-maid from the popular '60 show - The Jetsons, proves that the concept of artificial intelligence living among people is not a young one. (b) A lawn mower from HUSQVARNA. (c) Korean robot serving soft-drinks. (d) Roomba, the robot vacuum cleaner. 3
1.2. (a) James - the robot butler, can negotiate its way around obstacles and can grasp objects. (b) A robot that can make a cup of coffee for you. . . 4
1.3. Lego Mindstorms NXT Robot. A robot-kit available at app. $200 5
1.4. Image with superimposed annotation in the LabelMe framework. 9
1.5. The Image Markup Tool annotating itself (the image for annotation is the IMT's main application window screen-shot). 12
1.6. The main window of the Image Manager from Softplot Gmbh demonstrating its rich interface. 14

2.1. This figure demonstrates the extra space around the image. 22
2.2. The figure presents how the offset for pointed zooming is calculated. . . 23
2.3. It illustrates the first part of equation 2.1 on how to calculate the start point S for fitting the view-port. Normally vectors D and D' are overlapping, but we have put them paralleled for better understanding. $(0,0)$ is the scene's origin. 25
2.4. The image showing the front-end (Graphical User Interface) with comments to individual parts. 27
2.5. (a) Is a dialog for choosing the annotation category for the currently annotated object. The category can be i.e. a chair, a door etc. (b) Is the About dialog showing general information about the application i.e. release date, author(s)... 31

2.6. (a) The picture shows a person lying on the floor. Around the robot a path is being drawn upon which the polygon will be constructed. One end of the path is attached to the mouse pointer and builds a rubber band line from the pointer to the preceding vertex. (b) The final polygon with the randomly calculated transparent color and black border line. . 35
2.7. (a) In this picture we draw a curve around the person lying down. The curve drawing process is underway. (b) The completed curve and its initial polygonal approximation. 38
2.8. The starting curve (in black) is an ordered set of points. The original curve is shown in 0 and the final output curve is shown in blue on row 4. 39
2.9. (a) Here, a closed curve has been drawn to separate the background from the *unknown region*. (b) The curve is expressed as a set of points in the Confidence Matrix. (c) The curve's points are connected with the Bresenham algorithm to build a closed region. (d) This region is then filled with the value of 0.5. 42
2.10. (a) We select the *foreground region* with a square-shaped brush. (b) The foreground region in the Confidence Matrix is marked with the value of 1.0 (white). The *unknown region* has the value 0.5 (grey) and the black background is simply 0.0. (c) Shows the result of the segmentation - a binary mask where white (1.0) corresponds to the foreground object. (d) We use the binary mask to construct a polygon. 42

3.1. (a) An intuitive process of how an axis-aligned rectangle can be created. After clicking, the creation process begins (step 1). Through steps 2 and 3 the rectangle is stretched to find the right size. In step 4 the process is finished and CPs are created. (b) An axis-aligned ellipse is created similarly to a rectangle. 48
3.2. (a) A tree showing an example hierarchy in a car. (b) The corresponding annotation of object parts within a car. 49

B.1. (a) In this table images from a selected folder are listed. (b) This directory tree lets the user navigate to the image folder of interest. 54
B.2. (a) Roll the mouse wheel away from the user to zoom in. (b) Roll the mouse wheel towards the user to zoom out. 55

B.3. (a) The group of polygon annotation methods. (b) The option page for the *Polygon* (blank for the moment). (c) The option page for the *Poly Curve*. (d) The option page for the *Poly SIOX*. 56
B.4. A dialog for choosing the annotation category for the currently annotated object. The category can be e.g. a chair, door etc. 57
B.5. (a) A smooth approximation of the curve. (b) A rougher approximation of the curve. 58
B.6. (a) When a polygon item is selected the brush pattern changes to a checker pattern. (b) The information bobble displays the annotation category and the item's depth. Here, category - Person, depth - 1.0. . . . 60
B.7. The rubber-band tool for selecting groups of control points. 61

Chapter 1.

Introduction

1.1. Outline of the document

The first chapter of this thesis includes the motivation behind image annotation with regard to state of the art robot research. In section 1.3 the related work shows different approaches to the topic of image annotation, whereas in section 1.4 the requirements for our tool are given.

In chapter 2 we give an exhaustive description on the implementation details of our tool. This part of the thesis focuses mainly on explaining the functionality of the annotation methods in section 2.4. Another major point of this chapter is the presentation of the graphical user interface in sections 2.3 and 2.2.

In chapter 2 we discuss how the tool was implemented. what programming framework, algorithms and ideas have been used and how they work together.

Since our application has a brood scope for extensibility, in chapter 3 we will discuss possible enhancements and improvements to the programm and explain how they could be implemented.

The thesis is concluded with chapter 4.

In the Appendix a Users Manual is given to show how the user should work with the Tool and explaining each single functionality of Ultimate Annotation. The Java class diagram in Appendix C together with the Java documentation in Appendix D give an insight into the application's architecture and provide help for further development.

1.2. Motivation

The idea of introducing robots into our homes and thus the every-day life is not as far away as one might think. In the future, machines could aid us in simple tasks like serving food, lawn mowing, cleaning, elderly care or simply provide a more secure environment in homes (see figure 1.1).

Researchers and organizations across the world already pursue the goal of developing of such house-maids. Japan and USA traditionally have strong robotics research communities that are tightly integrated due to the action of specific funding agencies. Gadgets like Roomba (see figure 1.1 d), a vacuum cleaner, are freely available in the US, whereas Japan has a tradition of introducing new robot toys each year. Robot-kits are available for buying and programming like the one from Lego, depicted in figure 1.3. The EU has also founded a research network and united the major European research groups under EURON - European Robotics Research Network. Among these groups is the Automation and Control Institute at Vienna Technical University (ACIN), where this thesis has been written. Examples of the application areas they are working on can be seen in figure 1.2.

People working in the robots@home group at ACIN are aiming to deliver a domestic, scalable platform equipped with a perception system based on stereo vision, time-of-flight and infrared sensors. The platform should be able to learn and classify the objects like furniture, doors and other domestic equipment and safely navigate through domestic environments. Some of the target milestones for the project are:

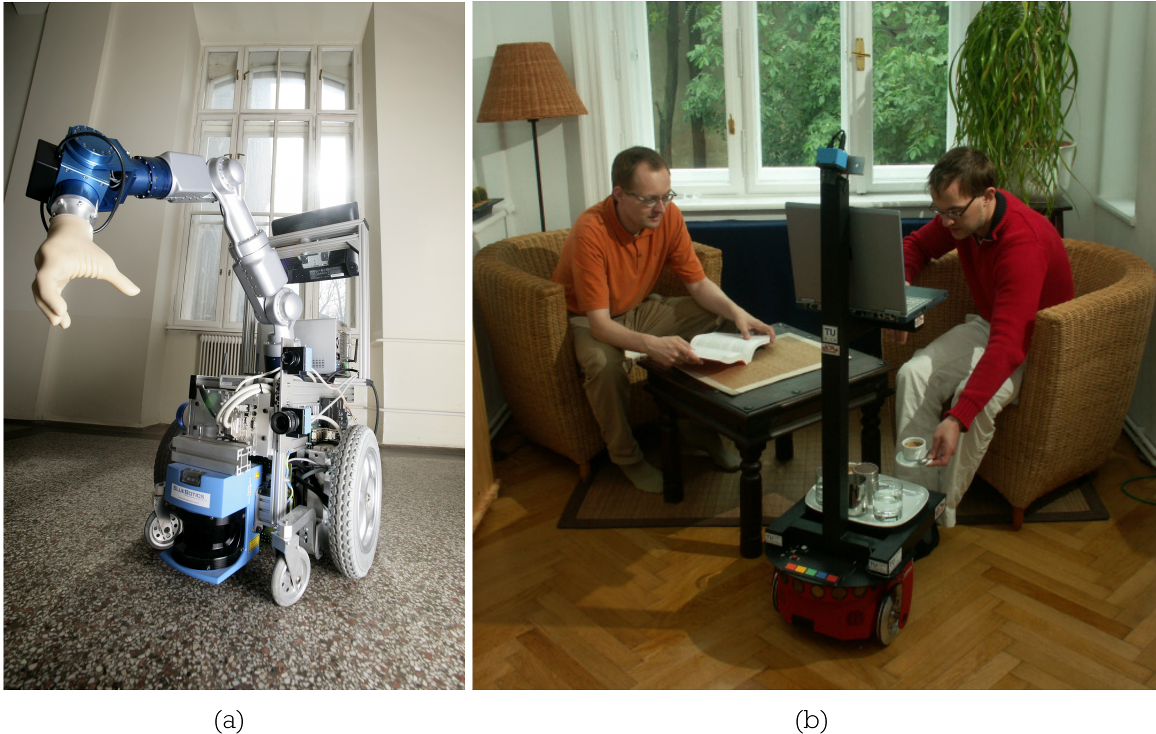
- Coping with basic room and floor types and first table classification.
- Room layout learned by "showing the robot around", classify main types of furniture, and then safely navigate in the home.
- Platforms learn four homes and safely navigate to each room and ten annotated pieces of furniture.

One attempt of achieving the above points is through the use of image processing. Although currently the state of the art computer-vision techniques have difficulties with recognizing objects in images, practical solutions for a few object classes, e.g. human faces or cars exist. Nevertheless the more general problem of recognizing other object classes e.g. chairs, tables, computers, furniture in general, remains unsolved. An

Figure 1.1.: (a) Rosie the robot-maid from the popular '60 show - The Jetsons, proves that the concept of artificial intelligence living among people is not a young one. (b) A lawn mower from HUSQVARNA. (c) Korean robot serving soft-drinks. (d) Roomba, the robot vacuum cleaner.

idea for approaching this problem is to train classification algorithms on large image databases with annotations for such objects. The database should include as many instances of furniture as possible to built a large list of e.g. tables, doors etc.

In this work we present a software tool for applying annotations to objects in images. With the help of this tool we aim to provide the researchers with means to quickly and easily add annotations to create large databases of annotated images. A large enough database of annotated images is a key component used to train the computer's model of each object. The central point of the application is placed on the efficiency of the annotation process. We need to quickly add annotation and manipulate the annotation data.



(a) (b)

Figure 1.2.: (a) James - the robot butler, can negotiate its way around obstacles and can grasp objects. (b) A robot that can make a cup of coffee for you.

1.3. Related work

According to Wikipedia

> In the digital imaging community the term image annotation is commonly used for visible meta-data superimposed on an image without changing the underlying raster image, such as sticky notes, virtual laser pointers, circles, arrows, and black-outs

This way of applying annotation information to images can be accomplished with graphics manipulation tools like e.g. *Photoshop*. The drawback of such an attempt is that the user needs to load each image separately and add e.g. a named layer on top of an object. This can take a significant amount of time when one considers thousands of images. It would be better to use an application specifically designed and addressed to add annotations to images. Many such tools exist, however most of them weren't made available outside the research groups where they were developed. Nevertheless there

Figure 1.3.: Lego Mindstorms NXT Robot. A robot-kit available at app. $200

are tools that are available to a broader audience.

One examples of an image annotation software is the Image Manager developed by Softplot (see [Gmb]). This is a commercial software that has a rich user interface giving a large variety of annotation options. It is mostly used to annotate blue-prints. Another example is the Image Markup Tool [Hol], which allows to annotate by adding rectangles upon objects and defining annotation categories. The iNote [WWWe] is a simple tool that can add rectangular annotations to images. The web-based LabelMe [BCR08], allows you to add and edit complex polygonal shapes. Finally, the Aktive Media developed at the University of Sheffield [WWWa], advertises itself as multimedia annotation tool. Indeed it's scope is relatively broad and includes not only image but also text and even 3D annotation. The tool is at an early stage of development thus its practical application is currently difficult. Also the recent image library managers or image browsers like WWMX or PhotoMesa (see [BS06]), which apart from offering efficient browsing through large image data-banks also provide simple image annotation and retrieval tools.

Video annotation tools can be seen as an extension to image annotation. They represent a branch where annotation can be multi-modal and incorporates, besides image, also

sound, gesture or pose information. One of the main representatives among tools that handle video annotation is Anvil - [Kip01]. Anvil focuses on annotation of multiple channels of communication like dialogue acts, rhetorical structure, words, gesture, graphics. Another good example is SVAS - Semantic Video Annotation Suite - [oISIMJRFm]. This academic project provides means to annotate graphic information in video footage. It uses modules for automatic recognition and search to find specific objects in the whole document, or adding textual and semantic annotation.

We will present advantages and disadvantages for the three tools that we found to be most relevant. This will later allow to answer which of the ideas found in these tools we would like to have implemented in our tool.

1.3.1. LabelMe

LabelMe is a web based application for image annotation developed at the Massachusetts Institute of Technology. The annotated images are kept on an Apache based server and can be accessed at anytime. Thus, LabelMe is not just an image annotation tool but also a database of annotated images. The central concept of LabelMe is to provide means that allow researchers to label images and share the annotations with the rest of the community. To work with the tool a web browser is required, so anyone with an internet connection can add his own annotation. This makes the LabelMe database dynamic and open for public contribution, which is reflected by the following numbers. As of May 27, 2008, LabelMe has 163,054 images, 43,244 annotated images, and 242,249 labeled objects [WWWo].

The motivation behind creating LabelMe was to give the computer vision researchers a more redundant image database. LabelMe was designed to recognize a class of objects instead of single instances of an object. For example, a traditional data-set may have contained images of cars, each of the same size and orientation. In contrast, LabelMe contains images of cars in multiple sizes, and orientations. Moreover, LabelMe recognizing objects embedded in arbitrary scenes instead of images that are normalized, and/or resized to display an individual object. In LabelMe annotation is achieved by adding polygons containing the object of interest. So instead of associating labeles to images, which also limits each image to containing a single object, LabelMe allows to annotate multiple objects within an image.

Annotation

The process of adding object labels works as follows. When a user enters the tool, a random image from the database is loaded and shown. If the image has already annotated objects associated with it, they will also be displayed. Annotation takes place by adding polygons on top of objects in the image using a mouse. Users start on the outline of an object and continue to add control points which eventually build a polygon around the object's silhouette. After the polygon is ready, a window pops-up where the user enters the class name for this object. Once the polygon is finished no control point can be added or deleted afterwards. Only while the polygon is drawn a predeceasing control point can be deleted. The only modification available after polygon creation is the deletion of the entire polygon and the repositioning of individual control points. The polygons are saved in an .xml file named after the image file, allowing for simple accessing and management.

Features

Besides adding polygons, users can zoom in or out, pan the view or display a 3D-like representation of the images. Also uploading and annotation of own images is possible. The user has a convenient access through links to the original image and the .xml file containing the annotation information. The objects that were annotated in the currently viewed image are presented in a list, which helps to keep track of the annotation. This and other user interface options are depicted in figure 1.4.

Another feature of LabelMe is the ordering of the objects according to their depth within the scene. Furthermore users can establish object part-hierarchies. Often objects of class *wheel* are part of objects assigned to classes like *car* or *bicycle*. If polygons overlap to a high degree they are classified as parts of an object.

Furthermore, LabelMe features a search option. A user can enter an annotation class (label name) to find and display all the objects associated with that category.

LabelMe project provides a Matlab toolbox for managing the LabelMe database. It allows to download the database to be used off-line. Since computer-vision researches often do their research in Matlab, this feature allows to integrate the database with existing tools.

Summary

One of the major drawbacks of the open and dynamic architecture of LabelMe is the lack of control over the remote server and the dependency on an internet connection. If the server goes down or the internet connection is broken the annotation tool can not be used. The second drawback is that the annotation entered by random users is not validated. Moreover, the names for the labels are freely chosen by the user, so one time a car can be named *car*, another time it can be an *automobile*, and another time a *ride*. The attempt to randomly chose an image for annotation can be difficult for researchers who want to work on a specific problem e.g. annotating cars. Fortunately, the possibility of uploading images from a user's hard drive is also given.

We summarize the advantages and disadvantages of LabelMe in the following lists.

Advantages:

- Annotation based on polygons.
- XML for saving annotation information.
- Object classes can be associated with the polygons.
- Depth ordering of annotated objects.
- Friendly and efficient user interface.
- Search function.

Disadvantages:

- System is vulnerable to network failures.
- Polygons have little modification possibilities.
- Randomly chosen images.
- No quality control.

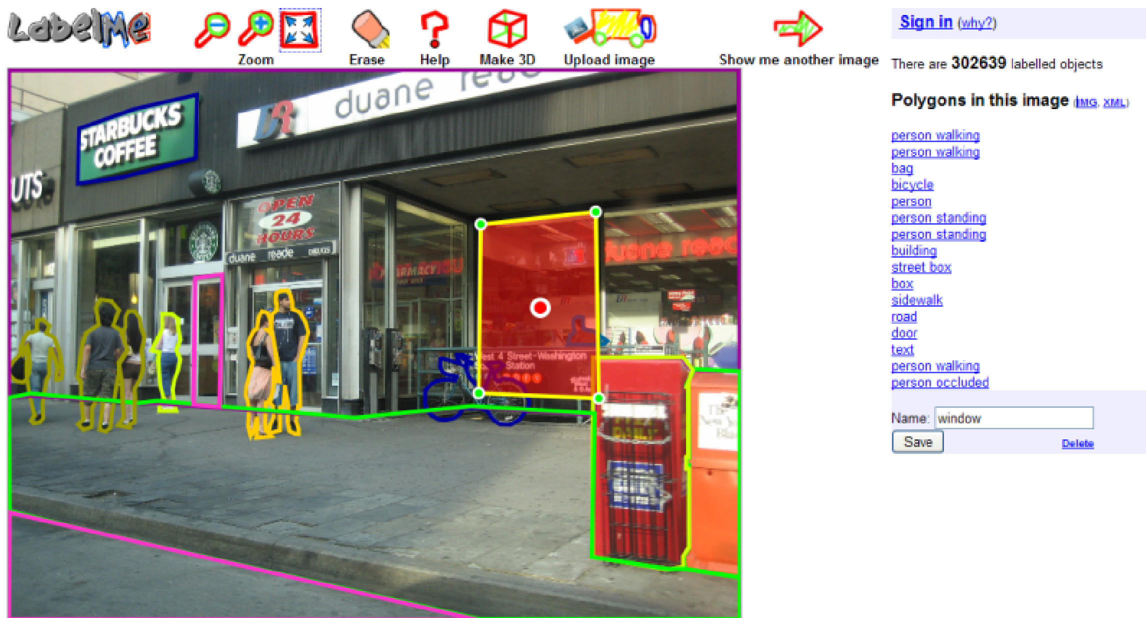


Figure 1.4.: Image with superimposed annotation in the LabelMe framework.

1.3.2. Image Markup Tool

Image Markup Tool (IMT) developed at the University of Victoria in Canada by Martin Holmes is a quit a recent software tool for adding annotations to images. The main focus is put on allowing inexperienced users to easily annotate objects. Another central issue of the IMT is the underlying file format for saving annotation data. The TEI schema XML meta-data format (see [WWWn]). TEI stands for Text Encoding Initiative and is an extensible schema that defines and documents a markup language for representing the structural, renditional, and conceptual features of texts. However in case of the IMT the TEI schema in combination with the SVG [WWWl] schema is used to mark-up images. The users do not necessarily have to know much about the TEI format to be able to use the tool.

Annotation

The labeling of objects offered by IMT is rather simple. Other than by LabelMe, IMT only allows to add a rectangle, ellipse, cross or a spiral on top of an object. IMT's user

interface is mainly focused on creating and managing of the annotation categories. To add annotation first a category needs to be created. The creation of categories is done in a separate dialog window where a user can specify new categories and annotation instances which are assigned to one of the existing categories. These annotation instances are objects (rectangles, ellipses etc.). When an annotation instance is defined it can be added onto the image and moved to the position the user wants to annotate. The possibility to resize the primitives assigned to a category, be it a square, ellipse, cross or spiral, allows to better fit it to the area occupied by the object. Figure 1.5 presents the result of annotating the screen-shot of the IMT itself.

Features

The tool provides an XML editor. This editor lets the users make adjustments to the TEI file. Using the editor requires basic knowledge in handling meta-data languages (e.g. XML, HTML). However if the user makes an error in the edited file, the editor will check if the .xml file is well-formed and repairs the errors if necessary.

At any point of the annotation process a web view representation of the image annotation can be generated based on the underlying TEI file. This web-page uses Java-Script to display an interactive representation of the annotated image. It also displays a hierarchy tree which is interconnected with the annotation primitives. This tree helps in navigating through the annotation hierarchies.

IMT allows for depth ordering of the annotation objects. It provides mechanisms for bringing overlapped rectangles to the front easily.

Furthermore the tool allows annotations to be grouped into categories, distinguished by name and appearance, to allow different classes of annotation.

IMT is an open-source project released under the Mozilla Public License [WWWf]. It was written in Delphi and is thus available only for Windows. However, since the sources are available for downloading it is possible to port the tool to other platforms than Windows. The IMT packaged comes with an installer, Help file, and basic tutorial, making it easy for new users to quickly learn how to work with the tool.

Summary

The tool offers rich options for categorization and naming of the annotation. One can create and delete annotation categories, assign color, give unique ID, chose shapes (rectangle, cross, etc.) for a specific category. Unfortunately this slows down the process of adding annotation, when compared to LabelMe. In IMT the user has to first specify all the details of the annotation category before start with the annotation. LableMe allows to annotate images right after starting the application.

Moreover the annotation primitives like squares or ellipses, are not as precise as e.g. polygons in LabelMe. They only offer a general approximation of the area where the object resides within an image.

The IMT project is a serious long-term project. It is continuously being developed and new versions are released frequently.

Below we list the advantages and disadvantages of IMT.

Advantages:

- Annotation category management.
- XML editor.
- Open-source.

Disadvantages:

- Annotation primitives - only rectangle, circle, spiral and cross are available.
- Complicated process of specifying annotation categories.

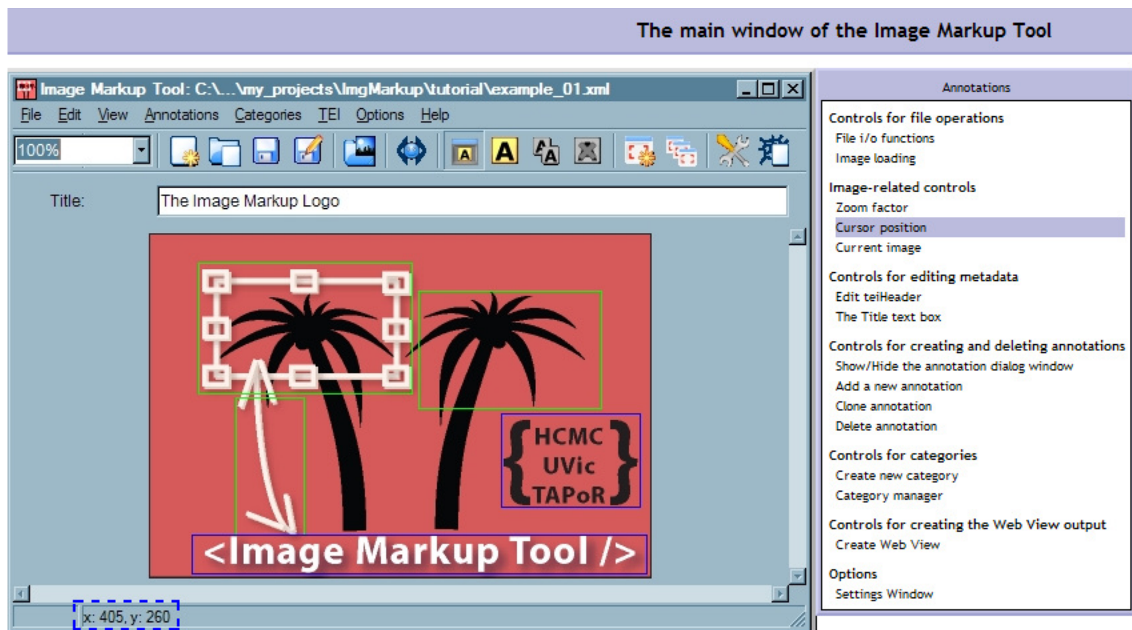


Figure 1.5.: The Image Markup Tool annotating itself (the image for annotation is the IMT's main application window screen-shot).

1.3.3. Image Manager

The creator of this tool - Softlot [Gmb] advertise their product as *The Professional Scan, Copy and Raster Processing Software*. This commercial software is mainly addressed to engineering offices. Its usefulness for engineering staff is reflected by the rich support for output devices like printers, plotters and scanners. Especially the support for scanners and plotters is important for people working on e.g. construction plans, blue-prints etc.

Annotation

The application is equipped with a variety of methods for applying annotation, including rectangles and ellipses. These are used typically used to cover an area belonging to an object. A textual description can be added afterwards with commentary. Also lines and arrows can be drawn on top of an image. These allow for example to connect associated objects. Colored half-transparent markers work like a paint-brush. The user can paint over an object and then write a descriptive text. Textual annotation is

also available through the use of thumbnails, which can be quickly placed on an arbitrary position in the image together with a textual information.

Features

Other than in the two applications discussed above, which are typical university projects, this program strikes with its rich user interface resembling professional graphics manipulation tools like e.g. Photoshop. This is reflected in the functionality regarding the raster processing offered in Image Manager. Users can rotate images, change resolution, transform to gray colors, mirror, invert, clip, resize etc. The tool even allows for clearing of image noise.

Furthermore Image Manager has its own scanner interface allowing for operations like duplex scanning, automatic recognition of paper formats, scanning of oversized documents, scanning of defined sections of the drawing etc. The plotter interface offers an equally vast set of functionality including printing of defined sections of a drawing, optimization of contrast for monochrome documents, support for all Windows-based printers etc.

Summary

Although Image Manager offers various functionalities regarding image annotation it is less practical and useful for researchers. The reason is in the way the annotation information is saved. The results of annotation are merged together with the underlying image bitmap, thus the later manipulation on the data is limited. The tool could theoretically be extended to support writing in a meta-data format like .xml but this requires a request on the software developer. Another drawback of Image Manager is that in opposite to the two other tools we have described it is not free and requires buying a licence.

Below we summarize our review on Image Manager.

Advantages:

- Rich image manipulation functionality.
- Multiple annotation options.

Disadvantages:

- Commercial licence.
- Merging annotation and image data.
- No polygons.

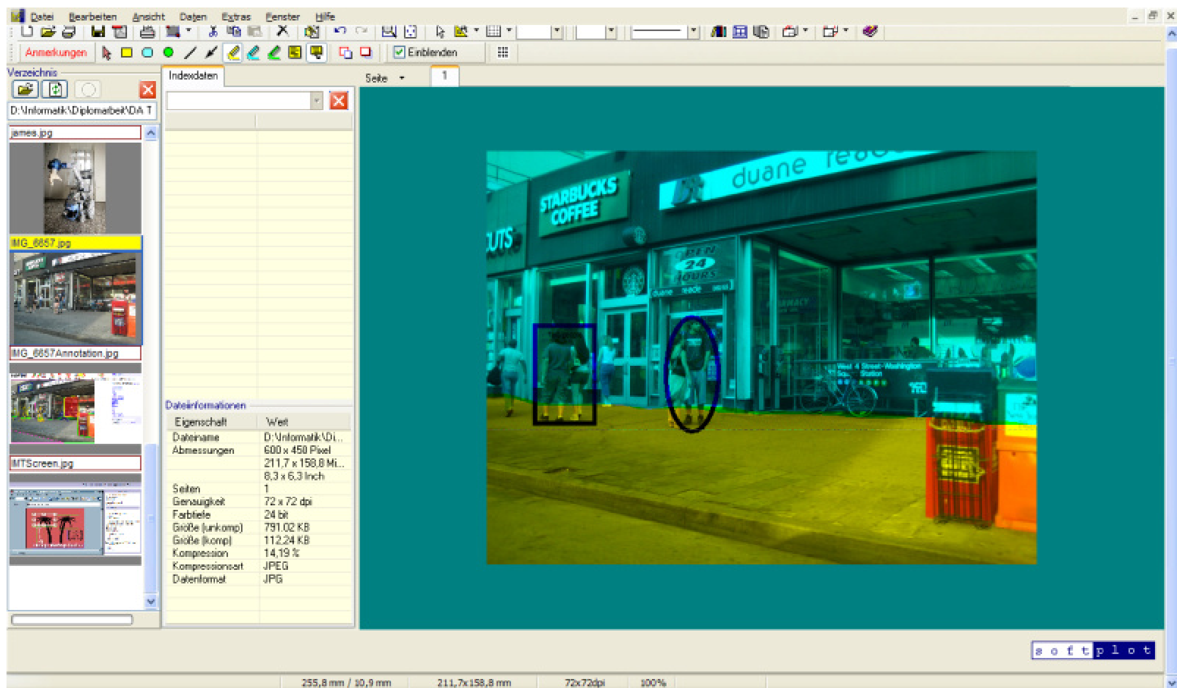


Figure 1.6.: The main window of the Image Manager from Softplot Gmbh demonstrating its rich interface.

1.4. Requirements

None of the tools discussed earlier enable efficient and precise image annotation allowing to quickly populate large image annotation databases. That is why requirements for a new tool were formulated. We will present these requirements in the following paragraphs.

One of the most important goals is to "keep it simple". The applications should be a stand-alone tool. No distributed elements should be considered e.g. no DBMS back-

bone for storing annotation information. This is why we choose XML as the information holder for our annotations. This meta-data format uses human, not computer language. This makes XML readable (and understandable, even by novices) and no more difficult to work with than HTML. XML is extendable and user can easily create new tags. Furthermore a vast variety of applications and programming languages support XML e.g. JAVA, C++, Internet Explorer.

Polygons allow the users to express annotation more precisely than rectangles or ellipses. Tight outlining of objects with polygons makes the annotation more valuable for research. We need to be able to add polygons which are efficient to manipulate on. It should be possible to reposition polygons, delete, create and move individual vertices.

In order to speed-up the process of annotation we need to implement algorithms to create polygons in a more effective way than by clicking from one vertex to the next one. This calls for a semi-automatic behavior which can be achieved i.e. by polygon approximation of hand-drawn curves [DD73], or through the use of concepts like the *magnetic lasso* [WWWi] or the *magic wand* [WWWh] available in image manipulation tools like Photoshop [WWWg].

The application should not only allow adding annotation, but also read and display the already annotated images with the corresponding annotations. This should not only work for application's native XML format but if possible also for other similar tools.

The time users require to add annotation should be minimal. That is why the user-interface should be simple and should support efficient and fast annotation assisted by mouse and keyboard. Also a fast method for navigating through the image database should be given to allow users for efficient work with multiple images.

The development of the application should be an open, long-term process and should allow other parties to join and easily contribute to the project. That is why in order to support this we have chosen Java as the programming language together with the QT-Jambi application framework. The object orientated architecture of Java makes it easy to programm and maintain the code. Another advantage is the platform independency of Java. QT-Jambi on the other hand allows to create professional user interfaces with complex graphics support. Both, Java and QT-Jambi have integrated modules for XML support.

Often the image data that needs to be annotated is taken from a stereo camera setup (two or more cameras). It would be helpful to only have to add annotation to one, say the left-eye image and propagate the annotation information to the next images provided that the offset information from a disparity image could be extracted. It should be possible for such extensions to be implemented.

Chapter 2.

Implementation

2.1. Project environment

In order for our project to be extendible with as little effort as possible we need a language platform which enables that. At the same time we want the language to provide a degree of platform independency for the project. We took three of today's modern object oriented languages under consideration: C++, Java, and C#. The architecture of these languages supports modular composition and enables easy extension and maintenance of the programm.

Not only the selection of the programming language is important, but we also need to choose a graphics framework. The graphics package should be freely available. It should have a sufficient documentation and support. It should also be efficient and widely used and tested by many individuals and generally by the software developers community.

2.1.1. Language

The strongest premise that made C++ a possible choice was its incomparable speed. When compared to Java, C++ is significantly faster, however not so much faster when compared to C# (see [Mor], [WWW05]). C# is, unfortunately, platform dependant. The drawback of C++ is it's weak object orientation which makes it more complicated to programm and time consuming when extensions to the existing code have to be

made. In contrast, Java and C# present themselves as the real object oriented languages, where the developer can focus on programming rather than on code management.

2.1.2. Graphics framework

There is a strong competition among the graphics packages because of the growing demand on applications with rich and efficient graphical layer, especially in the field of portable devices (PDAs, Cellular phones). This makes the choice of the most suitable package harder, since all frameworks offer a similar functionality. Among the most interesting packages are Piccolo2D [WWWj], SWT [WWWm] and QT [WWWk] - all being available for free. Piccolo is distributed for Java and C# and is written as a layer on top of the native libraries of Java or C#. SWT on the other hand, is addressed only for use with Java. It is not as much a graphical package like Piccolo2D and QT but rather a set of user interface objects (widgets). It does not have any 2D graphics support. Nevertheless 2D graphics can be controlled with the Java's native AWT package. The last package that we describe is QT. Although it is a commercial software package, a free version under the General Public License is also available. QT is destined to be used together with C++ but recently a Java version has been released under the name QT-Jambi. Inside the package we have modules for XML, advanced 2D graphics, OpenGL, SQL, SVG - making it more than just a graphics package but rather a complete programming framework. QT-Jambi combines all the advantages of QT (performance, completeness) and Java (object orientation). QT-Jambi wraps over the native QT functionality which is embedded in C++. This allows to programm under Java and still have the performance comparable to the original QT for C++.

The decision was to take the QT-Jambi graphics framework which than determined the use of Java as the programming language for our project. The powerful and efficient QT-Jambi with the combination of Java provide the necessary support for future extension and further development of our tool.

2.2. User interface design

The user interface is one of the most important topics in this work. The development of the application was intended for people who deal with problems of computer vision on a daily basis. Based on that fact we have decided that the user interface should differ from the common way similar applications for a wider audience are designed. This difference lies mainly in the speed in which elements of the GUI (Graphical User Interface) are accessed. Our default user is an engineer who has a lot of experience with computer programs and can easily deal with configuration files editing, or find and use typical tool functionalities. Keeping this in mind we tried to achieve high accessability with as little user interaction as possible.

In this section we give a detailed description of the design and the implementation of all the functionalities which make the user interface of our tool an efficient work-space for our default user.

2.2.1. Graphics view

MyGraphicsView

The term - graphics view refers to where the images and the associated graphic elements (polygons, control points etc.) are being displayed. It defines a scene and a view-port where graphics items can be viewed.

There is an object class `QGraphicsView` in QT-Jambi upon which we have built an extension class - `MyGraphicsView`. In this class we extend the basic functionality of `QGraphicsView`. From now on, we will be using simply GV when referring to the `MyGraphicsView` class.

When an image is loaded into our GV, its scene and view-port are set according to the image dimensions. The scene's width and hight is set to accommodate 4 times the width and height of the image rectangle. The view-port is set so that the image is scaled ,with the aspect ratio preserved, to fit into the graphics view widget-space with some extra 25% margin of the image width or height, depending on the image orientation (horizontal or vertical). For better comprehension see figure 2.4.

Application states

Our graphics view class defines most of the application states. These are the states that are related to mouse manipulation and polygon creation functionality. The reason this concepts are hosted by the GV is because they take place within the graphics view window. The application states are listed below:

The states related to polygons will be covered in the later section 2.4. Before proceeding to the next sub-section, we will describe the last two states from the above table: the SELECT_DRAG and the SCROLL_DRAG.

When the SELECT_DRAG state is active, it indicates that from now on the mouse pointer can be used to stretch a so called rubber-band over the graphics items. When the left mouse button is pressed at an arbitrary point in the scene, a rectangular shape, the rubber band, is being painted from that point to the current mouse position. When the mouse button is released the graphics items (polygons and control points) under the rubber-band will be selected. This is helpful when the user wants to manipulate on more than just one item.

We use the state SCROLL_DRAG for view-port manipulation. We looked up this idea from popular 3D modeling programs like Autodesk Maya and 3D Studio Max (see [WWWb]). In these programs we can do planar movement of the view-port by pressing the Control key and a mouse button. We use the same concept, only that the scene is in 2D. We can crab[1] and boom[1] the camera's view-port by holding the Control key and moving the mouse with a mouse button pressed. The mouse pointer changes its shape into a hand to indicate the current state. The transformation of the view-port position happens in real-time and the user immediately sees the effects. We save the value that refers to the dragging offset in the `deltaMouseDragOffset` variable, which is later used for the view-port transformation.

The state SCROLL_DRAG is the reason for which we setup the scene dimensions to 4 times the image dimensions. We simply need the space in the scene onto which the view-port can 'look at' besides the image itself and the margin (see figure 2.1). This

[1] Crab and boom together with tilt, pan and dolly are names of the camera movements. Crab means moving the camera sideways from e.g. left to right. Boom is moving the camera up or down.

STATE NAME		
Description	How state is activated	Next state

POLYGON		
The state where the Polygon mode is selected	Select the menu bar option *Polygon*	POLYGON_DRAWING
POLYGON_DRAWING		
The state where a polygon is being drawn	Pressing the mouse button	After pressing Enter/Space the current state returns to POLYGON state
POLYGON_CURVE		
The state where *Poly Curve* mode is selected	Select the menu bar option *Poly Curve*	POLYGON_CURVE_DRAWING
POLYGON_CURVE_DRAWING		
The state where a curve is being drawn	Press&hold the mouse button	After releasing the mouse button the current state return to POLYGON_CURVE
POLYGON_SIOX		
The state where *Poly SIOX* mode is selected	Select the menu bar option *Poly SIOX*	POLYGON_SIOX_Unknown_Region_DRAWING
POLYGON_SIOX_Unknown_Region_DRAWING		
The state where the unknown region is being drawn	Press&hold the mouse button to draw a closed curve which defines a closed region	When the mouse button is released and the curve closed the current state turns to POLYGON_SIOX_Known_Foreground_Region
POLYGON_SIOX_Known_Foreground_Region		
The state where the cursor turns into a square and the user is about to mark the foreground	The state is active after the unknown region has been defined	When the mouse button is pressed the current state turns to POLYGON_SIOX_Known_Foreground_Region_DRAWING
POLYGON_SIOX_Known_Foreground_Region_DRAWING		
The state where the foreground region is being drawn	The state is active when the user presses&holds the mouse button to mark the foreground region	When the mouse button is released the current state returns to POLYGON_SIOX_Known_Foreground_Region
POLYGON_SIOX_PolygonyzeOutlin		
The state where an outline is polygonized	The state is activated after the segmentation process has finished and the polygon approximation can be chosen	When the user presses Apply to accept the polygon the current state returns to POLYGON_SIOX
SELECT_DRAG		
The state where a user stretches a rubber-band over polygon/control-points	The state is activated when the shift key + mouse button is pressed	The state returns to one of the three states POLYGON, POLYGON_CURVE or POLYGON_SIOX
SCROLL_DRAG		
The state where user can drag the view-port (Camera movement *crab* and *boom*)	The state is activated when the control key + mouse button is pressed	The state returns to one of the three states POLYGON, POLYGON_CURVE or POLYGON_SIOX

Table 2.1.: The table presenting the application states.

additional space is also utilized by our zooming procedure, which we discuss in detail in the following section.

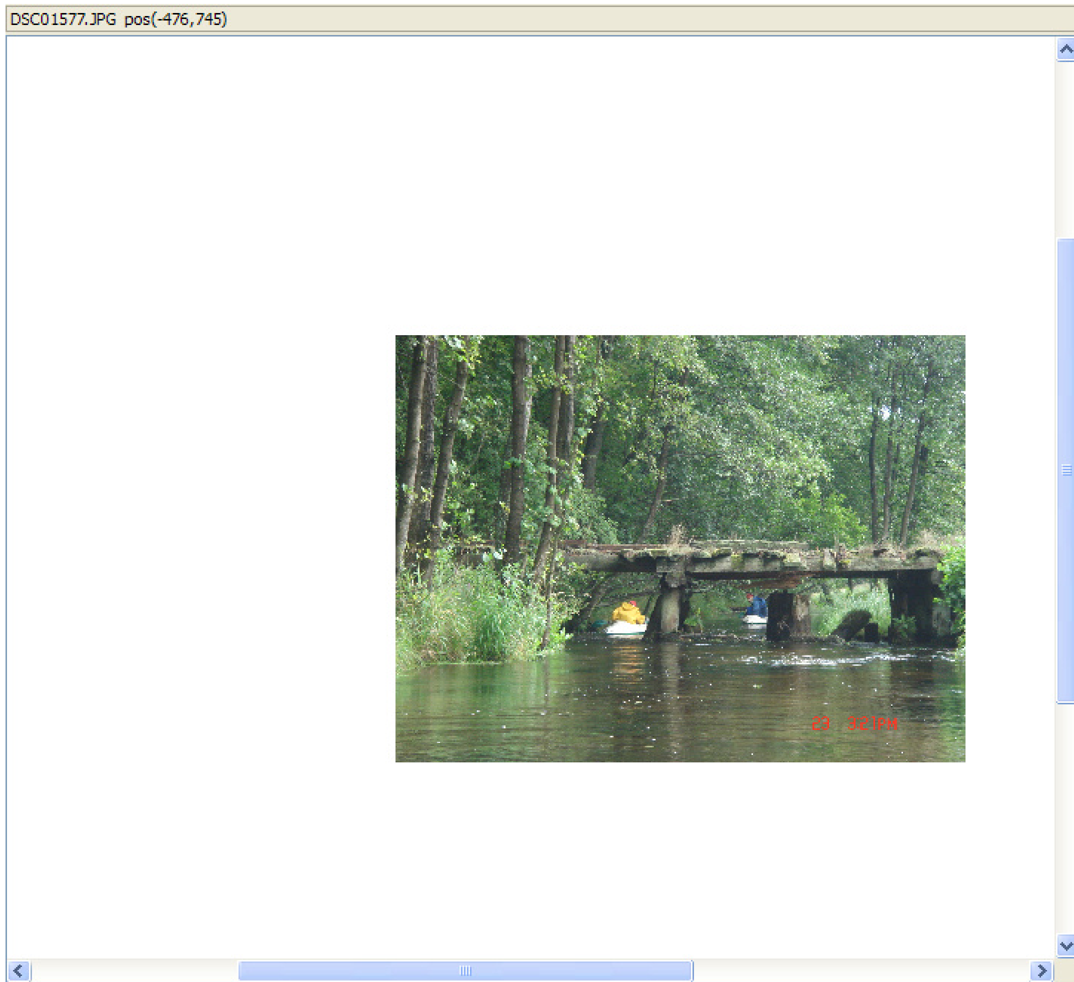


Figure 2.1.: This figure demonstrates the extra space around the image.

2.2.2. Zooming

Zooming does not have a separate state at the time, because it is executed instantly on each change of the mouse-wheel position. As no other procedure uses the mouse wheel, there is no need to isolate it from other application states. Now we know that zooming is triggered through mouse wheel event. This intuitive idea was looked up from the already mentioned 3D modeling applications, where the view-port can be moved to or from the scene, simulating zooming in and out with a mouse wheel. How-

ever, what is interesting about this procedure, is our extension to this idea. We not only allow to zoom in/out, but while zooming the view-port position follows the mouse pointer. This enables the user to show the spot on where he wants to zoom, by moving the mouse and turning its wheel. In combination with the SELECT_DRAG, the pointed zooming, as we refer to this way of zooming, makes it extremely easy and efficient to bring the view-port on the desired spot.

With the help of figure 2.2 we explain how the pointed zooming is implemented in the GV. When the image is loaded into the GV, the image's upper-left corner is always placed in the origin ($x = 0$, $y = 0$) of the scene's coordinate system. We first calculate the image middle point. Second, we take a vector showing from the origin to the current mouse position. Now we can calculate the directed offset by subtracting these two vectors. As depicted in figure 2.2 the subtraction gives us the desired vector `deltaZoomConvergenceOffset` which we use for the later viewing transformation.

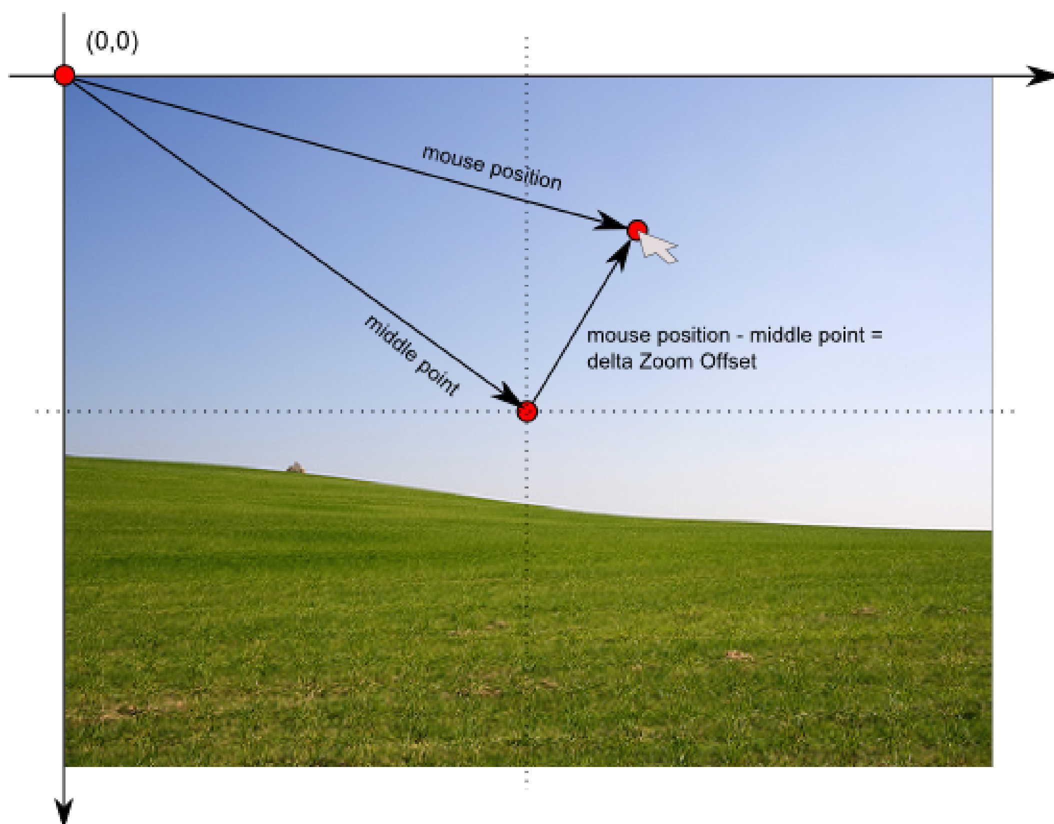


Figure 2.2.: The figure presents how the offset for pointed zooming is calculated.

2.2.3. View-port transformation

Having described in the previous sections how the two vectors `deltaMouseDragOffset` and `deltaZoomConvergenceOffset`, related to the viewing transformation, are derived, we can present how the viewing transformation is done.

The function responsible for the viewing transformation in the GV is called `pixmapTransformFit`. The most important part includes the calculation of the starting-position vector, for the image-rectangle that is to be fitted into the GV's window space. This calculation is expressed with the following equation.

$$S = \frac{D - D'}{2} + \Delta Drag + \Delta Zoom \tag{2.1}$$

where $\boldsymbol{D}$ is the diagonal vector of the image, $\boldsymbol{D'}$ is the diagonal vector of the rectangular area that is to be fitted into the view-port. It is derived from $\boldsymbol{D}$ and the current scalar value of the zoom factor z : $\boldsymbol{D'} = \boldsymbol{D} * z$. Vectors Δ Drag and Δ Zoom refer respectively to vectors `deltaMouseDragOffset` for the drag offset and `deltaZoomConvergenceOffset` for the offset associated with zooming.

The first part of equation 2.1 is depicted in figure 2.3. In the second part we add-up the Δ Drag and Δ Zoom (figure 2.2) vectors in which the drag and the pointed-zoom information is being held.

Now we can call the QGraphicsView's native function `fitInView`, which is a convenience function provided for a higher abstraction level of the transformation matrix manipulation. The call looks like this:

```
> fitInView(start_x, start_y, rect.width()*zoomFactor,
rect.height()*zoomFactor, AspectRatioMode.KeepAspectRatio);
```

Here $start_x$ and $start_y$ correspond to the component of the vector $\boldsymbol{S}$ from the previous equation 2.1. The double values from $rect.width() * zoomFactor$ and $rect.height() * zoomFactor$ give the width and the hight of the rectangular scene area. This area needs to be fitted into the view-port. The $rect$ variable corresponds to image-rectangle. The $zoomFactor$ of type *double* is a scalar z in equation 2.1 and $AspectRatioMode.KeepAspectRatio$ is a flag for preserving the image dimension ratio.

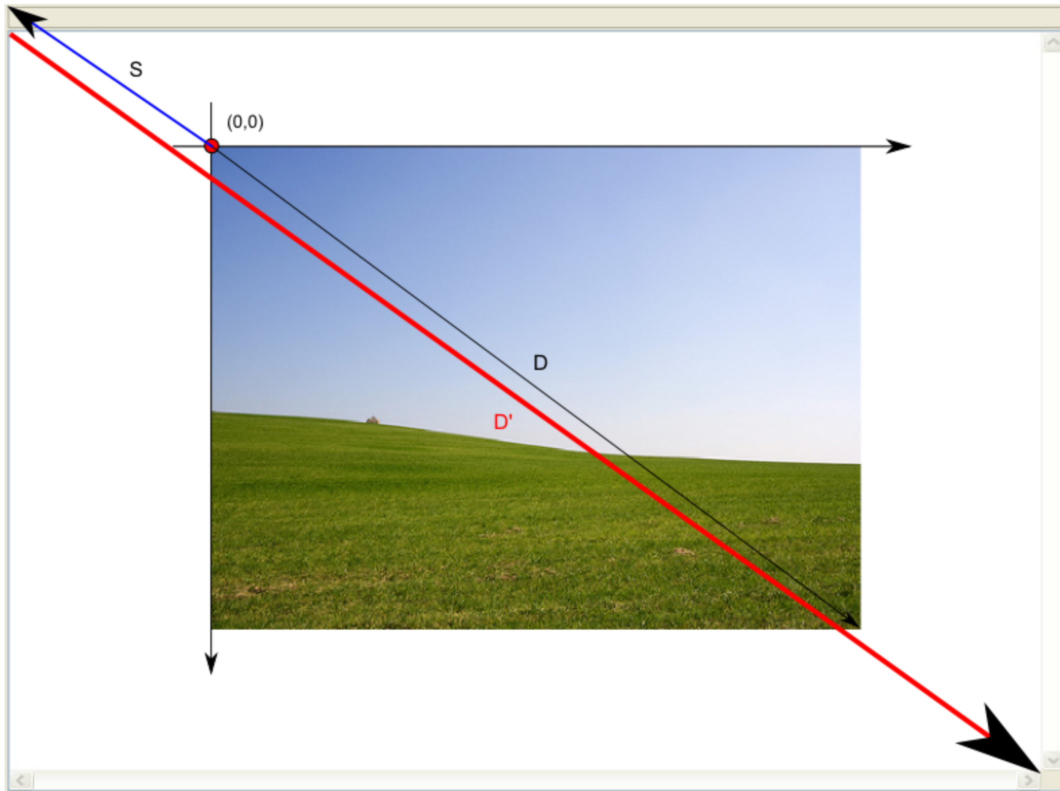


Figure 2.3.: It illustrates the first part of equation 2.1 on how to calculate the start point S for fitting the view-port. Normally vectors D and D' are overlapping, but we have put them paralleled for better understanding. $(0,0)$ is the scene's origin.

2.3. Graphical User Interface

In this section we will take a closer look at the features offered by our application's GUI. We will describe the most top-level widgets with which the user interacts. We do not want it to become a users manual of any sort. However, the study of this section with the combination of the appendix B gives a complete idea of how the tool works from a user's perspective.

As we discuss the details of the GUI we often reference to concrete java classes. In table 2.2 we give a short description to these classes.

QTableView	This class provides a default model/view implementation of a tree view. It implements a tree representation of items from a model.
ImageTableModel	This class provides a standard interface for models that represent image data as a two-dimensional array of items.
QDirModel	This class provides a data model for the local filesystem. It gives access to the local filesystem, providing functions for renaming and removing files and directories, and for creating new directories.
PolygonOptionPage	This class defines the options available while creating polygons using the Polygon mode.
PolygonCurveOptionPage	This class defines the options available while creating polygons using the Poly Curve mode.
PolygonSIOX_OptionPage	This class defines the options available while creating polygons using the Poly SIOX mode.

Table 2.2.: This table describes classes used to describe the GUI functionality.

2.3.1. Main application window

We start with the applications main window which's screen-shot can be seen in figure 2.4. The commented rectangles in the figure describe parts of the GUI. Our main window is equipped with many concepts that are often found in other application software. These are in particular the menu bar, depicted in the upper-left corner, where all the options reside like i.e quit, save, help. For convenient accessing, some of these options are also available from the toolbars placed right below the menu bar. Just like our tool, many applications provide a so called status bar. The status bar is placed at the bottom of the main window and has a passive functionality. It informs of the recent events by displaying messages, like e.g. images are being loaded, the polygon needs at least 3 vertices, hints on what to do, etc.

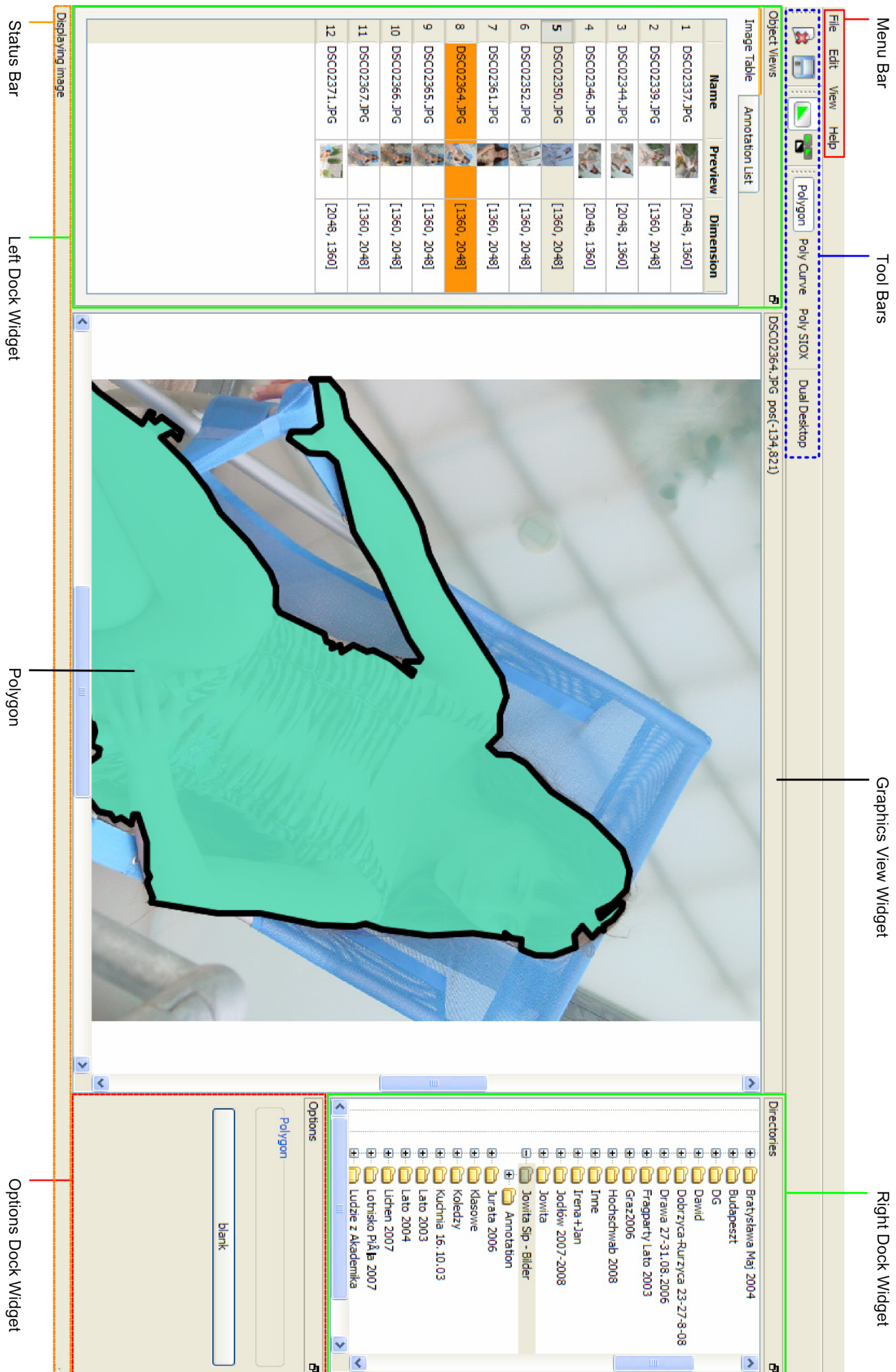


Figure 2.4.: The image showing the front-end (Graphical User Interface) with comments to individual parts.

Central area

In the main application window's central area (`QMainWindow.centralWidget()`) we have a dock widget [Tro] with embedded GV. Up to three such dock widgets with GVs can reside in that central widget. This refers to the menu option - *stereo mode* which is not entirely implemented. The assumption was to be able to annotate simultaneously 2 stereo images based on a disparity image. The idea is to do annotation to one, say the left-eye image and propagate the annotation tho the right-eye image with the help of the information provided by the disparity image. Yet this has proven to be a more complex problem and on this stage of the development this concept was not implemented. Each GV has a dock widgets assigned to it, where differen images can be loaded independently and manipulated individually. Each of these dock widgets has a little bar where the image file name is displayed followed by the current mouse position on the associated GV. On top of the loaded images we can add polygons in 3 different ways and manipulate them. In figure 2.4 we see one GV in a dock widget pair inside of the central widget with an image loaded and annotated by a cyan polygon lied over a person. This refers to the menu option - *mono mode* and is the default setting on program startup.

Sides

We also distinguish the three side dock widgets, the left dock widget (Objects View), the right dock widget (Directories) and the Options widget. The Directories dock widget is hosting the `QTreeView` which displays the contents of the `QDirModel` interface object for directories and hard drive information. The tree view object allows users to choose an image directory to work with by simple double click on the directory entry. After the directory was chosen its images are displayed in the Image Table.

The object View widget embeds two tab widgets. In the first tab a table Image Table is embedded, and in the second tab - Annotation List, nothing is currently embedded. The Image Table uses QT-Jambi's `QTableView` class together with a class from our package - `ImageTableModel`. These two classes provide the functionality of the Model-view-controller (MVC) design pattern (see [Ree03]) that is natively implemented in QT-Jambi as the Model/View model. The class `ImageTableModel` was borrowed from an example application - Image Viewer from the QT-Jambi's example

package and slightly modified. The modification includes the orange highlighting of the image entries with underlying annotation information. In figure 2.4 in the Image Table we see one orange-highlighted table entry corresponding to the image displayed on the current GV. The other entries in the table have no annotation sofar. The idea behind the empty Annotation List tab is to have a list of annotation objects, be it a polygon, square etc. sorted by e.g. depth value or corresponding color information. For example in case of the polygon in figure 2.4 a cyan square could appear in the corresponding list entry. The idea is to have the objects and the Annotation List work in the Model/View model. This would allow for e.g. deleting an entry from the Annotation List to make the corresponding polygon disappear. This idea has not been implemented and remains to be investigated.

The options dock widget is related and dependent on the polygon modes: *Polygon*, *Poly Curve* and *Poly SIOX*. It uses the `QStackedWidget` widget which allows for switching option pages according to the currently chosen polygon option. We have written three classes of widgets for the option pages: PolygonOptionPage, PolygonCurveOptionPage and PolygonSIOX_OptionPage which correspond to the polygon modes in the menu bar/task bar. Now, when the user clicks to chose the polygon mode from the menu bar or task bar, the `QStackedWidget` will automatically change to the corresponding page offering the options for that particular mode. This allows for a very swift and efficient option displaying.

2.3.2. Dialog windows

Besides the main application window we discriminate two other windows: the About Dialog and the Object Category Dialog which are displayed respectively in figure 2.5 *a* and 2.5 *b*. The About Dialog is entirely composed inside our `MainWindow` class. Its purpose is to tell about the application. Inside, we present the logo of our tool - Ultimate Annotation and below we give a textual information on what the application does, what features are offered, information about the authors and the release date.

The Object Category Dialog is a window dialog defined in a separate class `TypeDialog`. The purpose of it is to display a list of object categories and allow the user to choose one for the currently annotated object. The categories are saved in the file *objects.xml*. This dialog is presented each time the user enters a new annotation. A handy feature of this dialog is that it remembers the last chosen category. This is

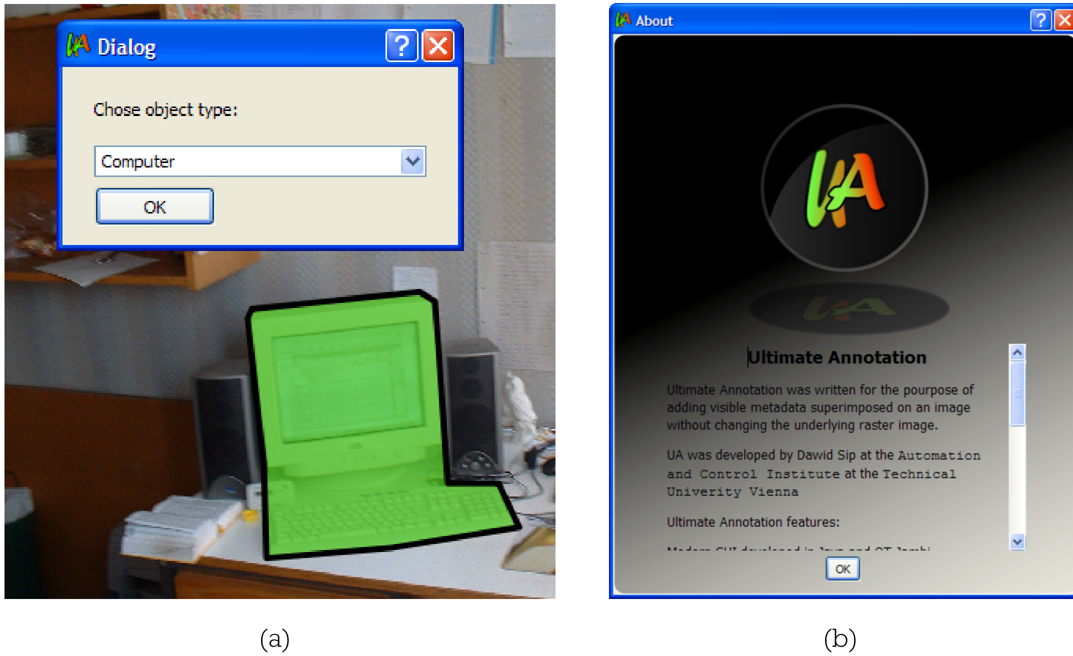
helpful when a user in a typical use case wants to annotate the same object category in subsequent images, like for example doors. For the first door the category has to be manually found in the listing, but for the further annotations the category will already be set to 'door', thus the required interaction is minimized.

Another feature of this dialog is the possibility to enter own categories that are not (yet) present in the *objects.xml* file. The user can click on the text field where the categories are displayed, delete the present category name and enter a new one followed by the OK button assigning the category to the current annotation object. This action however will not save the category in the file. If the user wants to have this category stay permanent, the underlying *object.xml* file has to be extended manually. Since the list is loaded from the file on each call to the Object Category Dialog the user does not need to restart the application for the changes in the file to be visible.

In the following we show how the current *object.xml* file looks like:

```
<?xml version="1.0"?>
<Objects>
    <para>Other</para>
    <para>Table</para>
    <para>Floor</para>
    <para>Wall</para>
    <para>Chair</para>
    <para>Person</para>
    <para>Couch</para>
    <para>Door</para>
    <para>Stairs</para>
    <para>Window</para>
</Objects>
```

To enter a new category the user needs to simply edit this .xml structure. The new category can be entered on any position between the `<Objects>` tag pair. The name for the new category should be chosen in a way that other users easily know what that name is describing. It should be general and descriptive on the same time. That name is then entered between the `para` tag pair.



(a) (b)

Figure 2.5.: (a) Is a dialog for choosing the annotation category for the currently annotated object. The category can be i.e. a chair, a door etc. (b) Is the About dialog showing general information about the application i.e. release date, author(s)...

2.4. Annotation

This is the most important section of this work, where we discuss and explain the annotation methods implemented into the tool. The tool offers three ways of creating polygons. These three ways include: the simple *Polygon*, the polygon derived from a curve - simply named *Poly Curve*, and *Poly SIOX*, which makes use of the quite recent segmentation algorithm - SIOX (see [GF05]). The order in which we present these methods in the following subsections is not chosen by accident. The reason for this is that the subsequent polygon options are built on top of their predecessors. This means the *Poly SIOX* is built upon the functionality of *Poly Curve* and the *Polygon*, whereas *Poly Curve* is built on top of the *Polygon*. The simple *Polygon* is thus the fundamental base class for all other polygon methods available in our tool. This difference is also reflected in the source code, where these three polygon classes are distinguished respectively: `MyPolygon`, `MyPolygonCurve` and `MySIOX_Polygon`. These and other

classes we will use to describe the annotation process are presented in table 2.3.

2.4.1. Simple polygon

The *Polygon* option is the initial method of creating polygons set by default on program startup. The entire functionality is embedded in the object class `MyPolygon`. We explain the functionality of this class by describing the creation process of a polygon and the behavior of its Control Points.

Creation process

The process begins by clicking the mouse button on the outline of an interesting object in the image. After the first point has been defined, the user proceeds to the next polygon vertex and while this is done an animation of a black rubber-line assists the user to decide where the next vertex should be placed. The rubber-line is stretched from the last entered vertex to the current mouse position and is attached to the mouse pointer. The concept of the rubber-line band is common among graphics manipulation applications and is known to the average user. The figure 2.6 a explains this concept. When the user is finished outlining the object, hitting the Enter/Space keyboard key closes the path and removes the rubber-line. What happens next is the deletion of the path item (`QGraphicsPathItem`) object followed by the construction of the `MyGraphicsPolygonItem` object. This object class was derived from the QT-Jambi's native `QGraphicsPolygonItem` class and is the actual graphical representation for polygon items within the GV's scene. After creating the polygon item object we adjust its settings. First we choose a color for the polygon with the `MyPolygon.randomColor` method. This method returns a randomly calculated color with a constant alpha value of 0.7 which was chosen empirically. The transparency is important when one considers that there can exist multiple overlapped polygon items in the scene and the user still needs to be able to see what is in the image. Another issue that needs to be set are the thickness and the color of the polygons border. Since most of the images are taken by day-light and black is more in contrast with bright colors, we have predefined that the outline border should always be black. The thickness of the border was predefined to a 0.000062 fraction of the image rectangle's area. This value was chosen empirically to increase visibility. Lastly, the z-value (depth) is

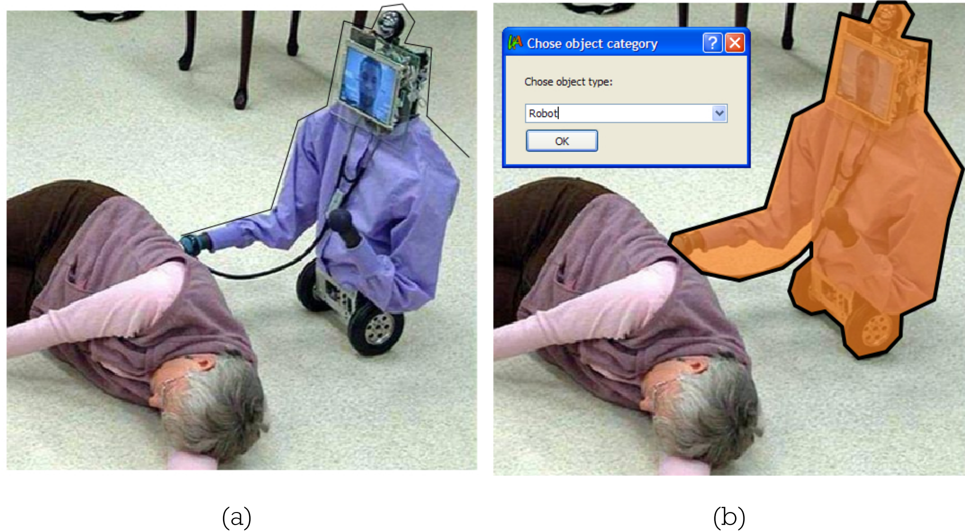
`MyPolygon`	This class organizes the creation and manipulation of the polygon objects in the GV.
`MyPolygonCurve`	This object class is used for the *Poly Curve* option mode. It is closely related to `MyPolygon` which it uses to construct the final polygon. Intermediate steps incorporate drawing of a closed line which uses the `QGraphicsPathItem` object.
`MySIOX_Polygon`	The object class used for the Polygon SIOX option mode. It is closely related to `MyPolygon` and `MyPolygonCurve` which are uses to construct the final polygon. Intermediate steps incorporate drawing a delimiter dividing the background from the unknown region, marking the known foreground region and finally producing an outline which we simplify with Douglas-Peucker to receive the final polygon.
`QGraphicsPathItem`	This object is used in the intermediate steps of creating polygons. In *Poly Curve* mode we use it to depict a curve. In *Poly SIOX* mode it is also used to depict a curve and to depict the foreground region marking.
`MyGraphicsPolygonItem`	This class is a fine tuned QT-Jambi's native `QGraphicsPolygonItem` class for displaying a polygon object in the scene. It differs form the original class by the way it reacts on the *select* event.
`MyGraphicsEllipseItem`	This class is a fine tuned QT-Jambi's native `QGraphicsEllipseItem` class used to display Control Points for a polygon. This extension to the original class adds Control Point management and makes the shape appear as a circle.

Table 2.3.: This table describes classes used to implement the annotation functionality.

automatically set for the new polygon. The GV keeps track of the last added polygon's depth values. This value increased by one is assigned to the new polygon.

Since we want the annotation to refer to image data the vertices of the polygon should be placed within the image borders. If the vertices are outside the image rectangle clipping of the polygon needs to be performed. This is done automatically before adding the polygon object into the scene. First we check if all polygon vertices are within the image rectangle. If one or more vertices are outside, we delete them. Then we use the `QPolygonF.intersected` method to calculate a new polygon by intersecting the current polygon with the image rectangle. The received polygon is added to the scene together with the newly calculated CPs. An example of how clipping works can be seen in figure 2.6 ***b***. Notice that the polygon region around the robot's head was clipped and is now cut to fit into the image area, unlike the path item in figure 2.6 ***a***.

To finish the creation process we need to choose a category (e.g. chair, couch etc.) for the annotated object. This is when the Category Dialog (see figure 2.6 ***b***) described in section 2.3.2 pops up. The user can define a category in two ways. Either chose a predefined category, or enter a new one in the text field. Since the category "Robot" has not been defined yet, the user would chose the second option for annotating the object from figure 2.6 ***b***.



(a) (b)

Figure 2.6.: (a) The picture shows a person lying on the floor. Around the robot a path is being drawn upon which the polygon will be constructed. One end of the path is attached to the mouse pointer and builds a rubber band line from the pointer to the preceding vertex. (b) The final polygon with the randomly calculated transparent color and black border line.

Control Points

As the user subsequently adds new vertices to the path item, on each of those vertex positions a Control Point (CP) is created. These are items that allow the user to manipulate on the polygon's shape by moving, adding or deleting the vertices. The CPs appear in the scene as circles. They are defined in the `MyGraphicsEllipseItem` class which is derived form QT-Jambi's native `QGraphicsEllipseItem` class.

The initial CPs are created along with the polygon in the `MyPolygon` class. However the addition of new CPs is accomplished in the `MyGraphicsPolygonItem` class. This is because we want the new CPs to be added on the polygon item's double-click mouse event activation. On that event the current mouse position is used to calculate a distance from that point to the nearest edge of the polygon. Once the edge with the minimum distance to the mouse position has been determined, we find the point on this edge with the shortest orthogonal distance from the mouse position. This calculated coordinates are the position for the new CP.

Control Point behavior

We discriminate two ways in which individual CPs can be selected. The first one is through the rubber-band, that was described in section 2.2.1. We turn the application state to SELECT_DRAG by pressing and holding the Shift key and instead of stretching the rubber-band we select a CP with a single mouse click. The second method is accessed by turning the application state to SCROLL_DRAG by pressing and holding the Control key. Normally this state is used to move the image in the scene space as described in 2.2.1, but in CP context it enables to select a CP with a single mouse click. Until now this is no different from the first CP selection method. The difference occurs when the user needs to select not one but a few CP. While the Control key is pressed the user can click on individual CPs to form a group of selected CPs.

The Control Points are presented to the user in different ways depending on the context. Normally the CPs are not visible in GV because the alpha channel of the brush is set to 0 (fully transparent). This is convenient when users are only concerned in watching the annotation polygons in the image and when showing a CP would just clutter the view. However, if the user wishes to work with the polygon, he does so by manipulating individual or groups of CPs. A CP is visible only in two cases. A CP was selected or the mouse pointer hovers over it. For these two cases separate brushes have been created. To indicate that the CP was selected we use the red half-transparent brush, whereas to show the hover event we use a half-transparent bluish brush. Similarly to the way in which we have calculated the thickens for the polygon items border, we calculate the hight and width of a CP. We taking a 0.00030 fraction of the image area. When a CP is being created the height and the width for the ellipse are set to that value which make the ellipse appear as a circle. QT-Jambi does not have a circle class, so this is the only way to achieve a circular CP appearance.

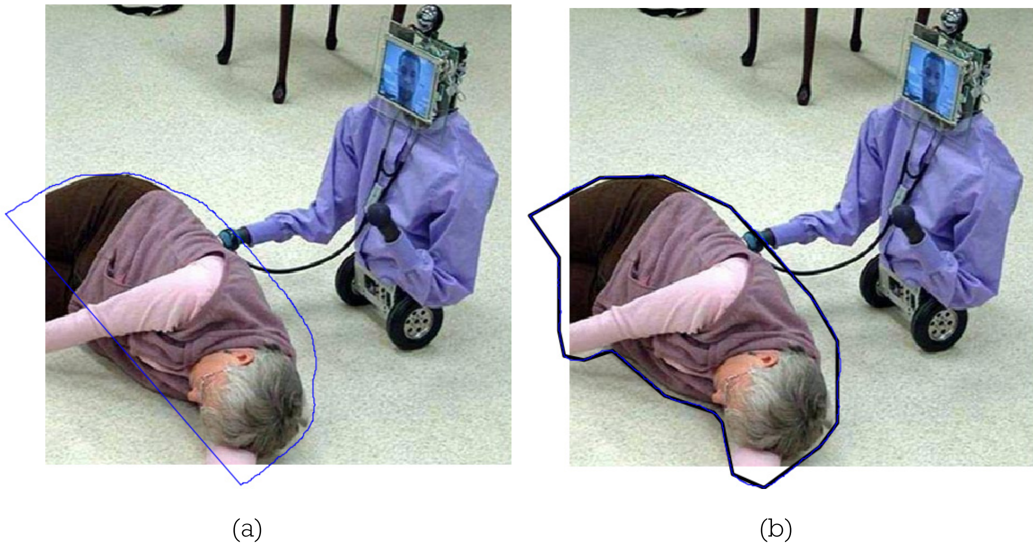
2.4.2. Polygon approximation

A polygon derived from a curve is the second method of adding annotation that we have implemented in the course of the application development. The main idea in this method is to use the Douglas-Peucker (explained in section 2.4.2) recursive curve subdivision algorithm for so-called curve fitting or curve approximation. This procedure

has been successfully used since early '70 for simplification of discreet curves in cartography (see [DD73]). The users annotate objects by drawing a curve and then use the algorithm to turn the curve into a polygon. We explain the process of creating a polygon form a curve in a typical use-scenario.

Creation process

After the user has selected the *Poly Curve* option from the menu or the task bar, the application state turns to POLYGON_CURVE. Next, the user needs to press the mouse button close to the silhouette of an interesting object to turn the state to POLYGON_CURVE_DRAWING and begin drawing a curve. The mouse button has to be pressed while moving the mouse around the object's outline. The curve itself is actually a very fine (smooth) polygon, that appears to the users as a continuous curve because of the small vertex distances. When the outlining is finished the user simply releases the mouse button and the curve is automatically closed. This state is visualized in figure 2.7 ***a*** for creating curve and ***b*** for the finished the curve. In that figure one can also see that an initial polygon approximation has been calculated and superimposed on the curve. The degree of curve simplification depends on the Tolerance parameter. The value for error tolerance can be adjusted within the Option Dock widget, by explicitly entering a value between 1 and 100 or by manipulating the `QDoubleSpinBox` widget. The newly entered tolerance value immediately triggers the simplification algorithm to create the corresponding polygon approximation and to display it. When the user decides what level of polygonalization for the outline curve is sufficient, the Apply button is used to confirm the choice. This triggers the `MyGraphicsPolygonItem`'s method for finalizing the polygon creation process. That finalization includes the deletion of the bluish path item for curve visualization and setting the polygon color, z-value, outline thickness and category, as was described in section 2.4.1.



(a) (b)

Figure 2.7.: (a) In this picture we draw a curve around the person lying down. The curve drawing process is underway. (b) The completed curve and its initial polygonal approximation.

Douglas-Peucker

For a detailed explanation of the Douglas-Peucker poly-line (curve) simplification procedure see [DD73]. Here, supported by figure 2.8, we discuss the concept behind this algorithm. The purpose of the algorithm is that given a curve composed of line segments to find a curve not too dissimilar but that has fewer points. The dissimilarity is defined based on the maximum distance between the original curve and the smoothed curve. The smoothed curve is composed of a subset of the points that build the original curve.

The algorithm performs a recursive line subdivision. It starts with all the points of the original curve. The first and last points are always kept throughout the algorithm steps (see figure 2.8). It begins by finding the furthest point from the line segment defined through the first and last points. Then we check if that point meets the dissimilarity level defined by the tolerance value $\varepsilon > 0$. If the found point is closer than ε to the line segment then any points not currently marked to keep can be discarded without the smoothed curve being worse than the tolerated approximation of ε. However, if that

point is greater than ε then that point is kept. Now the algorithm can recursively call itself with the first and the found point, and then with the found and the last point.

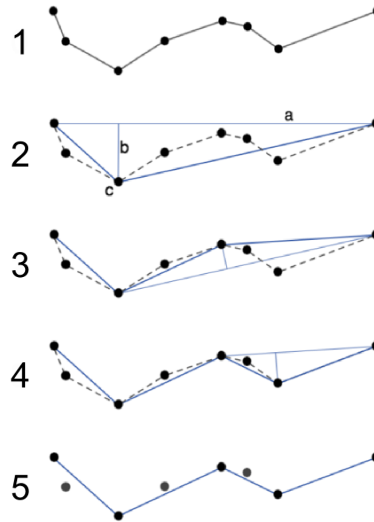


Figure 2.8.: The starting curve (in black) is an ordered set of points. The original curve is shown in 0 and the final output curve is shown in blue on row 4.

2.4.3. Polygon from SIOX

The third way for applying annotation through polygons is called *Poly SIOX*. It uses the SIOX - Simple Interactive Object Extraction [GF05] algorithm for foreground segmentation together with the Douglas-Peucker polygonalization procedure to establish polygons. In the following we discuss the concept behind SIOX, and then show how the algorithm was integrated into our tool.

SIOX

The SIOX algorithm is an image segmentation procedure. Its task is to separate the background from the foreground in a given image. We define foreground to be a single object that is of interest to the user. The rest of the image is considered background. The user has to specify at least the superset region of the foreground object and the algorithm returns the extracted foreground region.

The algorithm's input is composed of three user specified region: *known background*, *unknown region*, and *known foreground*. We call these regions the *trimap*. Internally, the *trimap* is mapped into a *Confidence Matrix*, where each element of the matrix correspondents to a pixel in the image (see figure 2.10 *b*). This input is provided through the following user interaction. The user uses a mouse to draw a closed curve. The outside of the curve defines the *known background* region while the inside defines the superset of the foreground object, i.e. the *unknown region*. Next, the user defines the *known foreground* by adding further selections within the *unknown region*.

The *trimap* (*Confidence Matrix*) and the image are the input for the actual segmentation. The algorithm first converts the image into the CIELAB color space. According to the authors of SIOX, the Euclidean distance between two colors in this space better approximates a perceptually uniform measure for color differences than in any other color space, like YUV, HSI, or RGB. Next, the algorithm creates a signature of the *known background* region and uses it to classify the pixels into those belonging to the signature and those not belonging to it. This is achieved through clustering of the background sample into equal clusters. In CIELAB space specifying a cluster size means assuming to specify a certain perceptual accuracy. A two-stage k-d tree algorithm is used for clustering, where the splitting is done through the division of two equally sized subintervals. In the first stage, the clusters are found by creating the tree and stopping when the interval at a node is smaller than the allowed diameter value. The clusters may be divided in several nodes at this point. In the second phase, nodes that belong to several clusters are recombined. Another k-d tree is build based on the cluster centroids from the previous tree. In this k-d tree the interval boundaries are stored in the nodes. Given a pixel, all that has to be done is to traverse the tree to decide whether it belongs to one of the *known background* clusters or not. A third k-d tree is built for the *known foreground*. Each pixel is then also checked against the *known foreground*. If it does not belong to either one of the cluster trees, it is assumed to belong to the cluster with the minimum Euclidean Distance between the pixel and each cluster's centroid.

For a detailed description of SIOX see [GF05]. The result of the segmentation is a closed region corresponding to the foreground object, like the one in figure 2.10 *c*.

Creation process

After the *Poly SIOX* has been selected, the application state turns to POLYGON_SIOX (see table 2.1). The first thing the user is confronted with, when doing the annotation in the *Poly SIOX* mode is to separate the background from the *unknown region*. The user has to draw a curve similarly as was done in the *Poly Curve* mode. Clicking near the object and holding the mouse button while dragging the pointer around the object's silhouette paints a curve like the one shown in figure 2.7 a. At that time the application state is set to POLYGON_SIOX_Unknown_Region_DRAWING (see table 2.1). When the user is ready drawing the curve, the mouse button is released and the curve is automatically closed. The region outside the curve - the *known background*, is blurred with a semi-transparent white color leaving the inside - the *unknown region* transparent. Figure 2.9 a illustrates this situation. The provided input for the *unknown region* needs to be expressed in the *Confidence Matrix*. The curve is composed of a set of points that are not adjacent and do not have the same distance from each other (see figure 2.9 b). These points need to be, first, connected together into a closed discrete curve and second, this discrete curve needs to be solidly filled. To connect the points we use the Bresenham's line-drawing algorithm - one of the most popular and basic procedures for creating straight line segments (see [Bre65]). The result of using the Bresenham algorithm to connect each pair of points is shown in figure 2.9 c. Now, having a closed region within our Confidence Matrix we fill it with the value of 0.5 that refers to the *unknown region*. For this operation we use the Flood Fill algorithm [Van].

After separating the *unknown region* from the *known background* the user is confronted with the selection of the *foreground region*. The state at that point is changed to POLYGON_SIOX_Known_Foreground_Region (see table 2.1). The mouse pointer is also changed. Now instead of the standard arrow shape it is assigned a square. This square is interpreted as a brush mask, and the size of this mask can be changed with a slider in the option page. Once the appropriate size for the brush has been selected, the user starts to draw within the *unknown region* (see figure 2.10 a). This changes the application state to POLYGON_SIOX_Known_Foreground_Region_DRAWING (see table 2.1). In the process of foreground marking one or more *foreground regions* are created. These regions are assigned the value of 1.0, which denotes the sure foreground as is shown in figure 2.10 b.

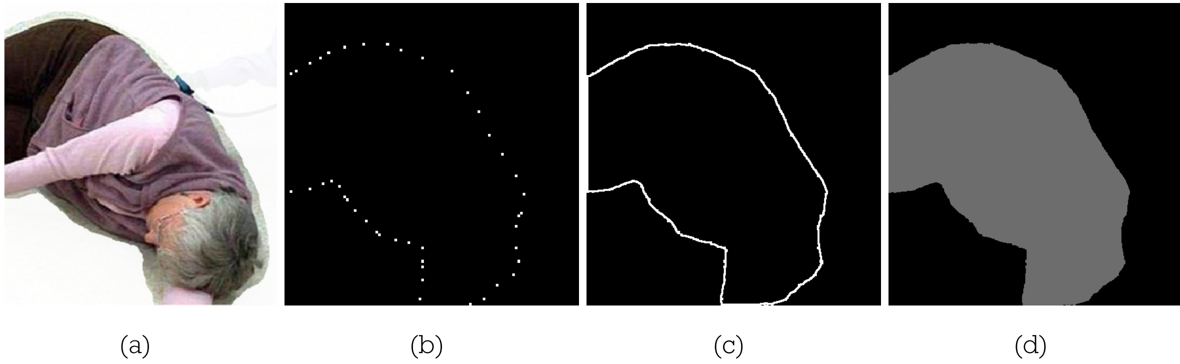


Figure 2.9.: (a) Here, a closed curve has been drawn to separate the background from the *unknown region*. (b) The curve is expressed as a set of points in the Confidence Matrix. (c) The curve's points are connected with the Bresenham algorithm to build a closed region. (d) This region is then filled with the value of 0.5.

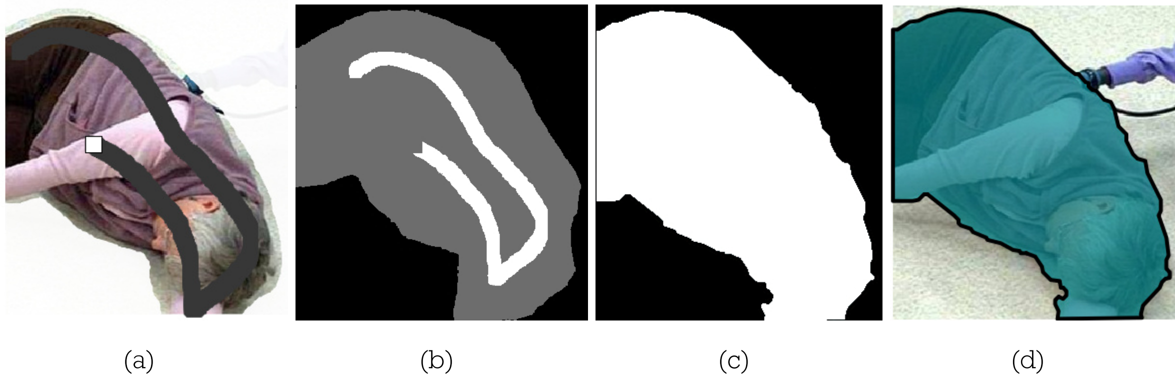


Figure 2.10.: (a) We select the *foreground region* with a square-shaped brush. (b) The foreground region in the Confidence Matrix is marked with the value of 1.0 (white). The *unknown region* has the value 0.5 (grey) and the black background is simply 0.0. (c) Shows the result of the segmentation - a binary mask where white (1.0) corresponds to the foreground object. (d) We use the binary mask to construct a polygon.

At that point the Confidence Matrix for the image is finished and ready for further processing with the SIOX segmentation procedure. We use the java package available at the SIOX project's web-site (see [GF]). The package provides an object class `SioxSegmentator` that represents the SIOX segmentation process. We call the

`SioxSegmentator.segmentate` method and pass the image-data and the Confidence Matrix followed by parameters used in the postprocessing step (see [GF05]). As a result of the segmentation we receive the altered Confidence Matrix with the values: 0.0 for the background and 1.0 for the extracted foreground region (see figure 2.10 *c*).

Now having the Confidence Matrix defining the foreground object, we use it to construct a polygon. To do this, we first trace the foreground-blob's boundary using the inner-boundary trace algorithm (see [WWWd]) to obtain a curve. Secondly, we use the Douglas-Peucker curve approximation procedure in the same way we did in the *Poly Curve* mode. In the option page we can adjust the smoothness of the approximation by giving the error tolerance. Once the user has found the desired approximation hitting the Apply button finalizes the polygon creation (see figure 2.10 *d*).

2.5. Managing the annotation

As was already mentioned in section 1.4 we do not want to save the results of annotation in a DBMS. We assume that image files are gathered in folders and associate annotation files with individual images. The association is done by giving the individual annotation files the same name as the corresponding image file. In order to achieve more clarity we store the annotation files in a sub-directory of the image folder and name it *Annotation*. The annotation files are saved in the Extensible Markup Language, short XML (see [WWWc]).

We show an example annotation file and discuss the individual tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<Annotation>
    <Object>
        <Type>Floor</Type>
        <RGBA>-1265614394</RGBA>
        <zValue>1.0</zValue>
        <Polygon>
            <pt>
                <x>304.0</x>
                <y>2.5</y>
            </pt>
            <pt>
                <x>336.5</x>
                <y>50.0</y>
            </pt>
            <pt>
                <x>412.5</x>
                <y>280.0</y>
            </pt>
        </Polygon>
    </Object>
</Annotation>
```

The first line is the header declaring of the XML format. In the second line we have the root tag - *Annotation* required for a well-formed XML file format (see [WWWc]). The next tag - *Object* denotes a single object annotation instance. In the next three tags: *Type*, *RGBA* and *zValue* we save respectively the objects annotation category, the color value and the depth (see section 2.4.1). Next follows the *Polygon* tag which denotes the way how this object has been annotated. Other way could include ellipses, rectangles or ellipses (see chapter 3). Within the scope of the *Polygon* tag, series of *pt* tags denote the vertices for that polygon. Each of the *pt* tags has a *x* and a *y* tag for storing the cartesian vertex coordinates.

Chapter 3.

Future work

While discussing the requirements in section 1.4 we stated that we want the project to be open for extensions. To support this openness we made choices regarding the environment in which the tool has been developed. The choice was to use the platform independent Java language together with QT-Jambi application framework. This allows to easily add new options and functionalities to the tool. In this chapter we want to encourage the tool-users to take part in the project by presenting some of the ideas for next extensions.

3.1. More objects for annotation

Currently the Ultimate Annotation supports image annotation based on creating polygons. Although this method of applying annotation is relatively flexible and adaptable to a variety of use-case scenarios, sometimes a more specific approach can be more useful to the user. A simple annotation method can be interesting for users that want to quickly add annotation and do not need high precision. We propose the implementation of two such approaches: the rectangle and the ellipse.

Rectangles

While the rectangle can be easily created form a polygon, quickly drawing a rectangle for a specific algorithm is faster. Such an algorithm is for example a face detection

algorithm [PV], for which the user needs to annotate a persons faces, and where rectangular input is preferred.

The QT-Jambi package does not have a class for displaying rectangles like it does e.g. for ellipses - `QGraphicsEllipseItem`. However, since a rectangle is a specific polygon, we can utilize the `QGraphicsPolygonItem` class to display and manage rectangles in a scene. So a possible solution would be to create a class, say `MyRectangle` to represent the annotation rectangle entity in the application. Additionally a `MyGraphicsRectangleItem` class inheriting from the `QGraphicsPolygonItem` class, for displaying and managing rectangles in the scene should be created. This concept is similar to how the simple Polygon annotation mode (section 2.4.1) was added to the application.

Besides writing new classes, at least two new application states should be introduced to the class `MyGraphicsView`. The first one, the RECTANGLE is for the application to know which tool mode is currently selected. If this state is active, the application is ready to add a new rectangle. The creation process can begin by pressing the mouse button at a position in the image interpreted as the upper-left corner of the rectangle. At this point the application would change to the second state - RECTANGLE_DRAWING denoting the process of drawing of the rectangle and the class `MyRectangle` would be created. The bottom-right corner of the rectangle could be defined in the position where the mouse button has been released. Two diagonal corner points are sufficient to construct a rectangle. After releasing the mouse button, the state would return to RECTANGLE and the `MyGraphicsRectangleItem` object would be created for displaying and user interaction. Then, on each of the four corners CPs can be created. But other than in `MyPolygon` class these CPs cannot be manipulated individually. The movement of one CP should update the positions of the adjacent CPs so that the rectangular shape is preserved. Since rectangles have always four vertices, the addition or deletion of CPs would be a consistency violation. That is why these editing options should be disallowed. For better understanding see figure 3.1 a.

In the above paragraphes we have describe how to implement an axis-aligned rectangle. But since all the GraphicsItem classes in QT-Jambi implement a rotation function, it should be easy to extend the tool and use `MyGraphicsRectangleItem` to allow creating more general rectangles.

Ellipses

There are cases in which the use of an elliptical shape is precise enough and thus satisfactory for the user. This can be the case when the user wants to annotate e.g. a car. QT-Jambi provides a class for drawing ellipse items - `QGraphicsEllipseItem`. Similar as for the rectangle we can create two classes: `MyEllipse` and `MyGraphicsEllipseAnnoItem`. An important notion about an ellipse is that each ellipse can be enclosed by a rectangle. So, by defining a rectangle we define an ellipse as well. Thus, the process of creating an ellipse can be similar to the creation process of a rectangle. Other possibility for defining an ellipse would be by e.g. deforming a circle. But since QT-Jambi does not have a class for drawing circles probably the easiest way of constructing ellipses is through a rectangle. As in case of a rectangle, we create an axis-aligned Ellipse, which can easily be extended to support rotations. Also, similarly as for the rectangle, to implement the ellipse tool mode, two additional states should be added to the list of the application states. The first one, the ELLIPSE is for the application to know which tool mode is currently selected. We can start the creation by pressing the mouse button to define one of the rectangle's corner. At this point the application would change to the second state - ELLIPSE_DRAWING denoting the process of drawing of the ellipse and the class `MyEllipse` would be created. Then, as we drag the mouse pointer, we define a rectangle between the position where the mouse button was pressed and the current mouse position. This rectangle defines our ellipse item. When the mouse button is released the definition of the ellipse is finished. Now the state would return to ELLIPSE and the `MyGraphicsEllipseAnnoItem` object would be created for displaying and user interaction. Next, we can add CPs. However, other like in case of a rectangle, these CPs cannot be positioned on the corners since ellipses have no corners. Instead, we can establish the endpoints of the minor and major axes of the ellipse by using the dimensions of the `QRectF` object for defining an ellipse, and place the CPs there. Like for the `MyRectangle`, the manipulation of individual CP would have to take into account the need to update the adjacent CPs. Deletion or addition of CPs should be forbidden as it would violate the shape consistency. Figure 3.1 *b* visualizes the process of creating an ellipse objects.

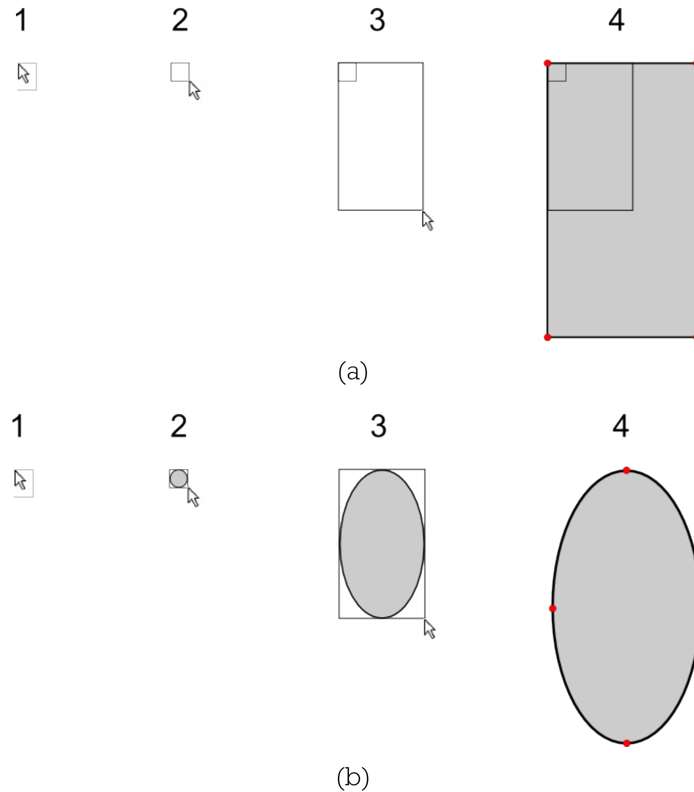


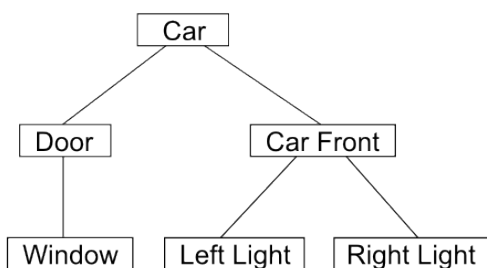
Figure 3.1.: (a) An intuitive process of how an axis-aligned rectangle can be created. After clicking, the creation process begins (step 1). Through steps 2 and 3 the rectangle is stretched to find the right size. In step 4 the process is finished and CPs are created. (b) An axis-aligned ellipse is created similarly to a rectangle.

3.2. Hierarchies

For the moment Ultimate Annotation does not allow to define relations between objects. If an annotated object in the image is composed of parts and these parts are also annotated, this relationship information is not explicitly expressed in the tool or in the XML file. However, such dependency information is needed if the user wants to later use the annotations to build more specific databases, query the databases or use the annotation data e.g. in part-based object recognition [DS06], [AQ]. In particular the part-based object recognition where various parts of the image are used separately in order to determine if and where an object of interest exists can benefit from a detailed partitioning. To support these applications we propose the introduction of hierarchies to the tool.

The idea for implementing hierarchies is to incorporate a parent-child relationship model between annotation objects (i.e. polygons) in a form of a tree (see figure 3.2). In order to be able to have this relationship model we need to extend how annotation objects are represented in a XML file. For the moment, in each object we only save the annotation category, color, depth and the figure representing the annotation (see section 2.4.1). To this list we add an identifier (ID), which provides uniqueness for the annotation objects. For our relations to form trees we need the objects to have no more than one parent and have none, one or more children. Now we can build relationships by exchanging IDs between objects. For the child object we save the parent's ID, whereas for the parent object children IDs are saved.

To visualize the application of this relationship in the tool, new application states should be added. The first state - SELECT_CHILD can be accessible through a menu bar button or by pressing a keyboard key, i.e. *R* for **R**elation. In this state the user selects the child object by simply clicking on it with a mouse button. To indicate the child object being selected the color or the pattern of the object's brush can change to e.g. solid opaque. After the child has been selected the application state changes to SELECT_PARENT. This state denotes the selection of the parent object for the child object selected in the previous step. The user drags the mouse pointer over the object and clicks the mouse button. To indicate that the parent-child association has been successfully established, the brush color and/or pattern of the parent and child objects can be set to the same values for a glimpse of a second and then change back.



(a)

(b)

Figure 3.2.: (a) A tree showing an example hierarchy in a car. (b) The corresponding annotation of object parts within a car.

Chapter 4.

Conclusion

The objective of this master thesis was to develop a software tool for adding annotation to images. The tool is intended to be used for building large databases of annotated images which can later be applied for example in robot learning [MLC]. The way we attempt to annotate these images is through the use of polygons whose edges coincide with the objects silhouette.

Currently existing tools for image annotation like LabelMe [BCR08] or Image Markup Tool [Hol] attempt to solve a specific problem. Since it is often difficult to utilize these tools for other applications, a need for a new tool has been recognized. In this document, we present *Ultimate Annotation*, a new application for efficient, semi-automatic annotation of image data.

Our tool is mainly directed to researchers working in the field of computer-vision, which naturally have experience in handling computer programs. Therefore we were able to simplify the user interface and provide access to functionalities through mouse and keyboard events, rather than by navigating through complicated menus.

We want the research community to be able to easily add extensions and participate in the project. To provide support for new enhancements, we have chosen Java as the programming platform together with the Qt-Jambi - an open source framework for application development well known among the software developers. Also, an extensive Java-class documentation gives a rich source of information on the project.

Three ways of applying polygons to the objects within an image have been implemented. The first method is a simple selection of vertex positions which at the end form a polygon. The second way of creating a polygon is by drawing a curve around

the object and then simplifying the curve with the Douglas-Peucker [DD73] procedure. In the third method we use a semi-automatic foreground extraction algorithm - SIOX [GF05] to separate the object of interest from the background. Followed by the application of the Douglas-Peucker to approximate its outline and construct a polygon.

The annotation data produced by users is saved in a XML file. This widely used standard for saving meta-data is supported by a variety of software packages facilitating effective manipulation of the annotation data. Another advantage of XML is the easiness of reading files. The clear structuring of XML allows the users to quickly find the interesting information.

Future enhancements may include a larger set of geometric objects including e.g. ellipses and rectangles, which can speed-up the annotation process for a large diversity of use scenarios. Also, the introduction of object-part hierarchies has been proposed. This extension allows to create relationships between objects necessary e.g. in part-based object recognition [DS06]. From a perspective of a database of annotated images, the presence of relationships provides means to construct queries upon the database, and thus create more specialized databases.

Appendix A.

Acronyms

ACIN Automation and Control Institute

EURON European Robotics Research Network

TEI Text Encoding Initiative

SVG Scalable Vector Graphics

IMT Image Markup Tool

DBMS Data Base Management System

GV Graphics View

CP Control Point

SIOX Simple Interactive Object Extraction

Appendix B.

Users Manual

B.1. System requirements

The current version of Ultimate Annotation has beeen deployed for the 32-bit Windows platform. It requires the Java 1.6 Runtime Environment. The system memory should be at least 1024 MB since the application will attempt to reserve 512Mb to function properly.

B.2. Launching the Ultimate Annotation

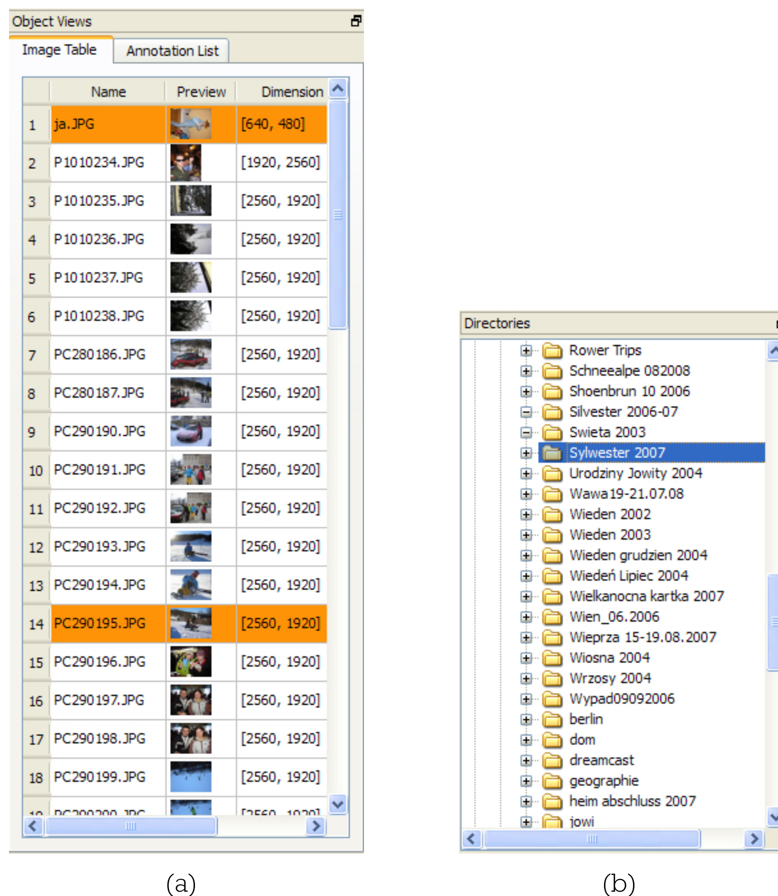
Navigate to the directory where the application folder is kept. Inside the *Ultimate Annotation* folder there are three files. To start the application simply launch the *ua.exe* file.

B.3. Closing the Ultimate Annotation

In the menu bar *File* section, and on the tool bar there is an *Exit* entry for a clean application closing. Be sure to save the annotation data and/or any changes before leaving the application. The annotation data will be lost otherwise.

B.4. Choosing the image

Before the annotation can begin, the user has to select the image to work with. First the user navigates through the directory tree on the right side of the application's main window to find the appropriate folder in which images are stored (see fig. B.1 *b*). A mouse double click or hitting Enter on that folder will activate the Image Table on the left side of the application's main window (see fig. B.1 *a*). In this table, all images from the selected folder are being displayed. Each entry in the table includes the image's file name, resolution and a small thumbnail image. The table entries that already have .xml annotation files associated with the image are highlighted in orange. A mouse double click or hitting the Enter key upon the selected table entry will load the selected image into the *Graphics View* window.

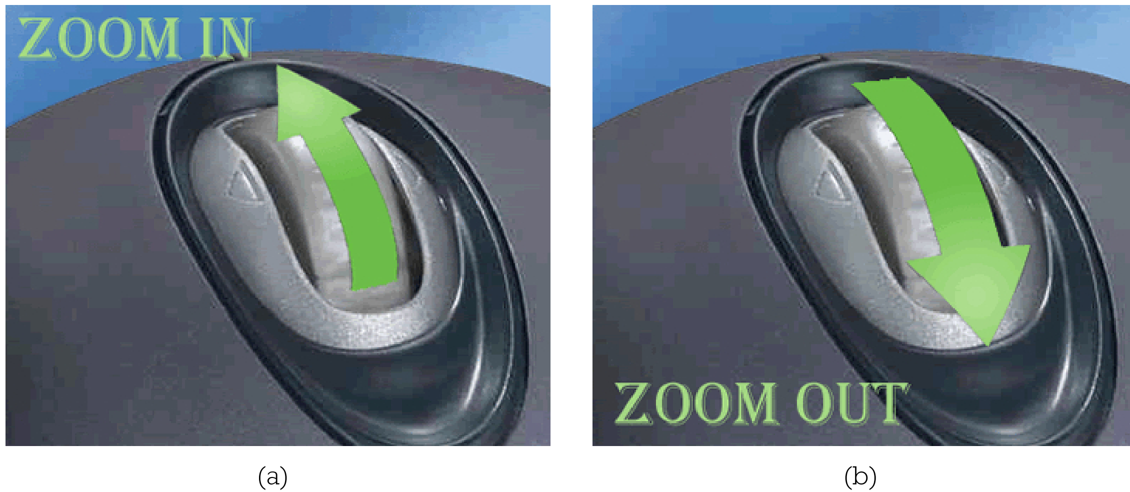


(a) (b)

Figure B.1.: (a) In this table images from a selected folder are listed. (b) This directory tree lets the user navigate to the image folder of interest.

B.5. Zooming

To zoom in or out only a mouse is needed. At the time of zooming, the mouse pointer has to be inside of the *Graphics View* window. The mouse wheel triggers the actual zooming as explained in figure B.2. The mouse pointer is also used in the process of zooming. The user can place the pointer on an arbitrary position and while zooming the view-port converges to or from that position.



(a) (b)

Figure B.2.: (a) Roll the mouse wheel away from the user to zoom in. (b) Roll the mouse wheel towards the user to zoom out.

B.6. Virtual Desktop

To utilize a two monitor environment combined into a virtual desktop, a user can move parts of the application's main window to the secondary screen with the *Dual Desktop* option button.

B.7. Annotation

The annotation is in the center of attention in Ultimate Annotation. That is why one of the three methods for applying annotation is always selected. Thus, the mouse pointer

is always assigned and ready to use with one of these three annotation methods.

B.7.1. Choosing polygon options

In the menu/tool bar the polygon methods are organized in a group (figure B.3 *a*). Individual methods within this group are associated with separate option pages (see figure B.3 *b*, *c*, *d*).

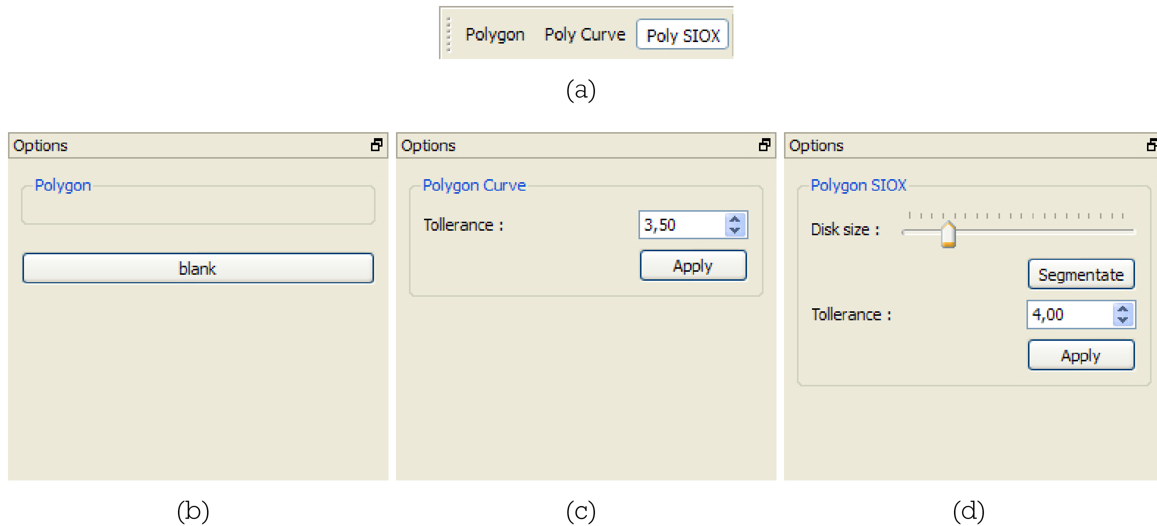


Figure B.3.: (a) The group of polygon annotation methods. (b) The option page for the *Polygon* (blank for the moment). (c) The option page for the *Poly Curve*. (d) The option page for the *Poly SIOX*.

B.7.2. Simple polygon

The first method is a simple selection of vertex positions which at the end form a polygon. This polygon option is designated with the word *Polygon* in both, the menu and the tool bar. It is the default, initial method available on application start.

We begin the polygon creation by clicking on the image to select the first polygon vertex. Then, by moving the mouse pointer to the next position and clicking the mouse button, the next vertex is selected, and so on... As the user is finished creating vertices, it is time to close the figure and finalize the polygon. Hitting Enter/Space will close the polygon, choose a random semi-transparent color and clip it if necessary. The user will

then see a dialog window for choosing the annotation category for the object (see fig. B.4).

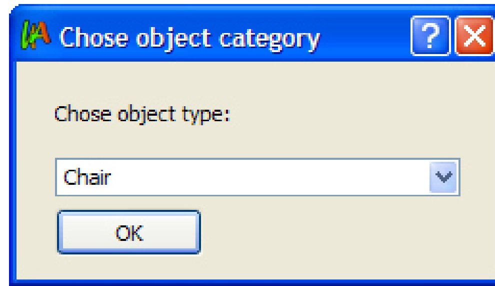


Figure B.4.: A dialog for choosing the annotation category for the currently annotated object. The category can be e.g. a chair, door etc.

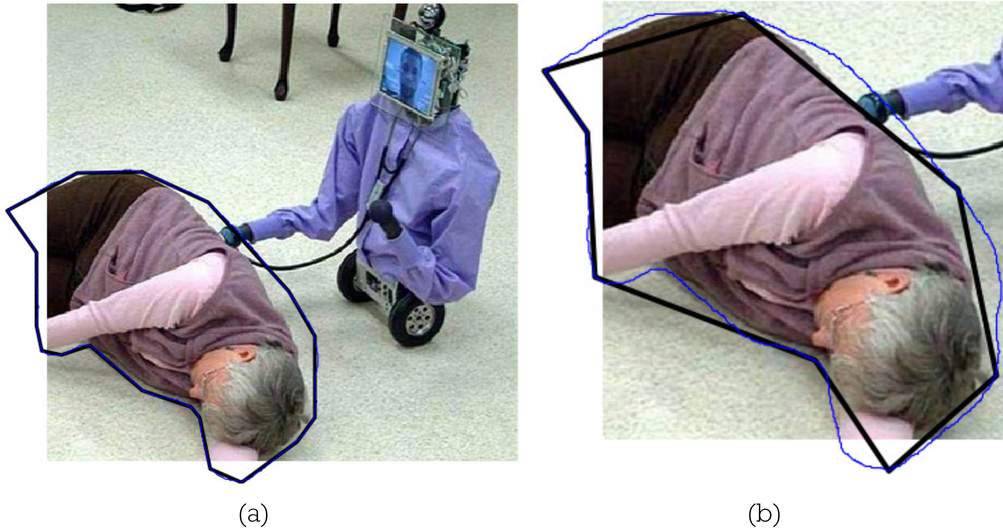
B.7.3. Curve Polygon

The second way of creating a polygon is by drawing a curve around the object and then simplifying that curve with the Douglas-Peucker procedure. In the manu/tool bar this method is denoted with the word *Poly Curve*.

We start by pressing and holding the mouse button on a an image position near the silhouette of an interesting object. By dragging the mouse pointer around the object, a blueish curve is being drawn. The starting and the pointer position are always connected with a straight line segment to visualize how the closing of the curve would look like at the current pointer position. Once the user is done drawing the curve the mouse button is released. Immediately after releasing the button an initial polygon approximation is calculated from the curve (see figure B.5). The smoothness of the approximation is controlled by the **Tolerance** in the option page in figure B.3 c. When the satisfactory approximation has been found the user needs to press the **Apply** button to commit the selected degree of approximation and finalize the polygon. The last thing to do is to chose the annotation category from the dialog window in figure B.4.

B.7.4. SIOX Polygon

In the third method we use a foreground extraction algorithm - SIOX to separate the interesting object from the background and then use Douglas-Peucker to approximate



(a) (b)

Figure B.5.: (a) A smooth approximation of the curve. (b) A rougher approximation of the curve.

its outline with a polygon. In the manu/tool bar this method is designated with the word *Poly SIOX*.

We start by drawing a curve around the interesting object, similarly like in the *Poly Curve* method. With the help of this curve, we select the super set of where the object resides. Now, we want to find the object. The cursor will change into a square brush. We use this brush to mark over the object and try to mark the most representative colors for the object of interest. The brush size is adjustable from the option page in figure B.3. Once the marking is finished the user hits **Segmentate** to launch the SIOX segmentation. After the segmentation is done, the user gets an initially approximated polygon from the outline curve. We can adjust the approximation with the **Tolerance** value. The **Apply** button finalizes the polygon creation.

B.8. Polygon manipulation

B.8.1. Select a polygon

To select an existing polygon item within a scene the user holds the **Shift** key and presses a mouse button upon the item. When selected, the polygon item changes its

brush pattern into a fine checker to visually emphasize the item being selected (see figure B.6 *a*).

B.8.2. Delete a polygon

To delete an existing polygon item the user needs to first select it (as was describe above B.8.1). When the polygon is selected the user presses the **Delete** key to erase the polygon item.

B.8.3. Looking up the polygon properties

To see the current annotation category and the depth of an existing polygon, the user simply hovers over the polygon item with the mouse pointer to trigger a small information bobble pop-up. The bobble shows respectively, the category and the depth (see figure B.6 *b*).

B.8.4. Changing depth

The depth of a polygon item can be changed. The user can look the current depth value up as described above (see B.8.3), and decide to lower or heighten the value by pressing, respectively, **Page Down** or **Page Up** keys.

B.8.5. Select a Control Point

The control points that lie on the polygon's vertices are initially hidden. To make a control point appear, the user hovers the mouse pointer over an interesting vertex.

To select a control point the user holds the **Control** key and presses the mouse button over the circle representing the control point.

To deselect the selected control point the user can simply press the **Control** key and click on a an arbitrary position with no items.

The user can select a group of control points by clicking on subsequent control points with the **Control** key pressed.

(a) (b)

Figure B.6.: (a) When a polygon item is selected the brush pattern changes to a checker pattern. (b) The information bobble displays the annotation category and the item's depth. Here, category - Person, depth - 1.0.

Another way of selecting a group of control points is with the rubber-band tool. The rubber-band is activated by pressing and holding the **Shift** with a mouse button, and dragging the mouse pointer. This stretches a rectangular shape between the point where the mouse button was pressed and the current pointer position. When the mouse button is released, the control points underneath the rectangle will appear selected (see figure B.7).

B.8.6. Move a Control Points

To move an individual control point it first has to be selected (see B.8.6). Then, by pressing **Shift/Control** with a mouse button on the selected control point and dragging the mouse pointer the control points are moved until the mouse button is released.

Moving a group of control points is achieved in the same way as moving of an individual control point. A group of control points has to be selected and the user moves only one of them, while the rest follows.

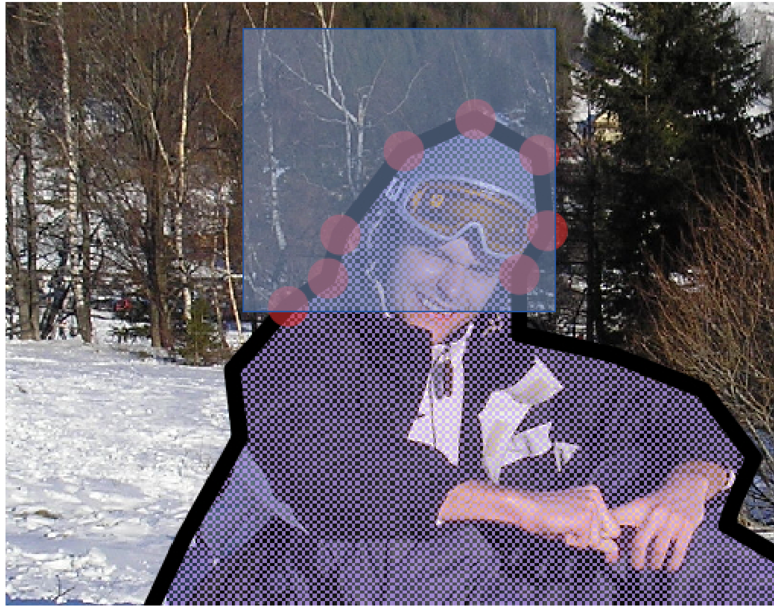


Figure B.7.: The rubber-band tool for selecting groups of control points.

B.8.7. Add a Control Point

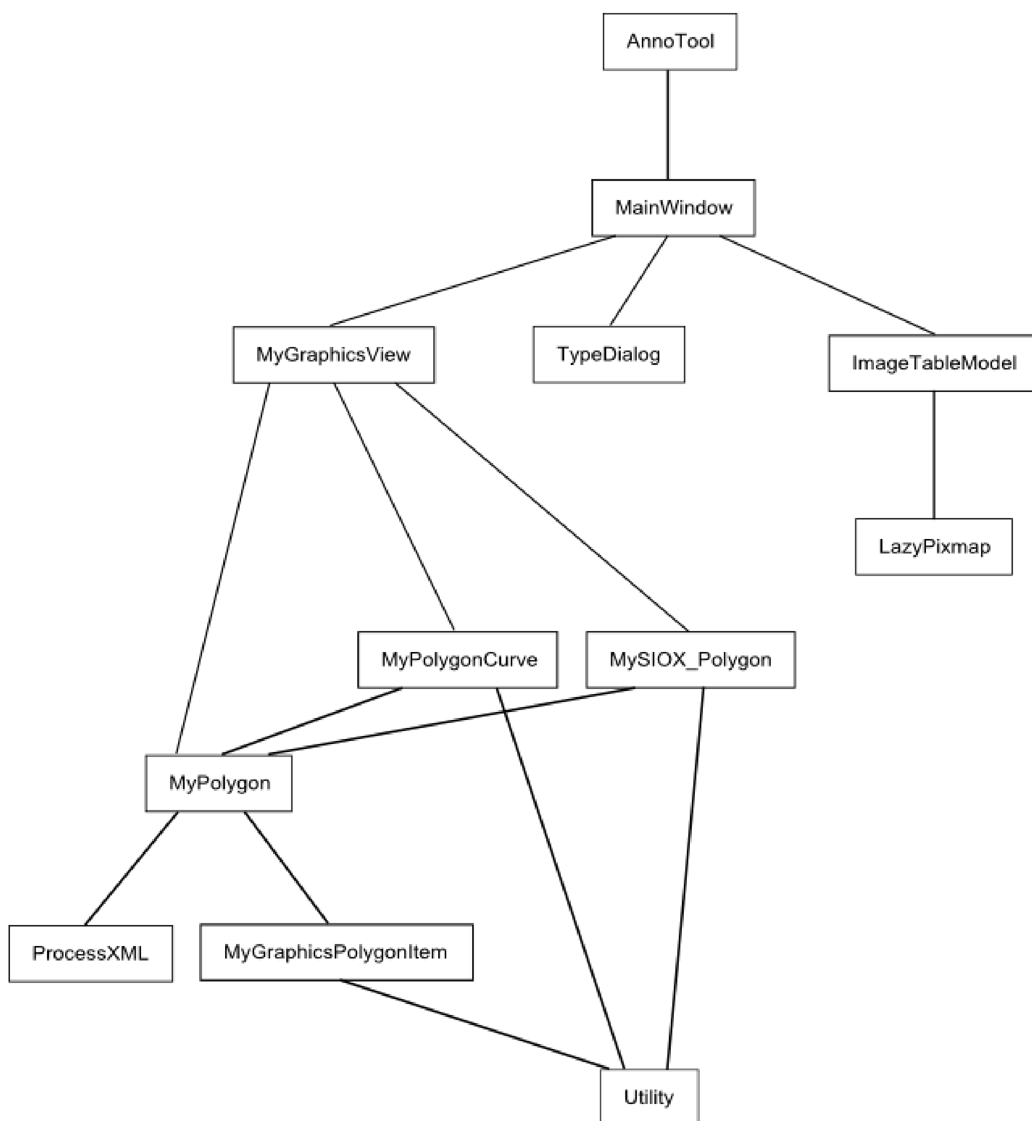
To add a new control point to an existing polygon the user needs to move the mouse pointer near the position where the control point should be created within the polygon's area. The **Shift/Control** + double mouse click will create and add a new control point on a position nearest to the mouse pointer.

B.8.8. Delete a Control Point

Deletion of an existing control point is strait forward. Select the control point to be erased e.g with the **Control** + mouse click, and hit the **Delete** key. The control point will disappear and the polygon will update to the new set of control points.

Appendix C.

Java class diagram



Appendix D.

Java class documentation

D.1. Class `AnnoTool`

This is the starting point of the application.

Fields
`MainWindow testAnnotation`
The reference to the MainWindow object building the main gui window.
`Runtime runtime`
The static Runtime object representing the current runtime. Will later be used for example to manually start garbage collection.

Methods
`void main(String[] args)`
The main method initializing and starting the application.

D.2. Class `ImageTableModel`

This class defines the behavior and occurrence of the tableView.

Constructors
`ImageTableModel (QObject parent)`
Constructor runs the thread and loads the thumbnails into the table.

Fields
`QColor annoted_RowColor`
The constant field representing the color of the row when the there is an underlying annotation file (.xml).
`List<QFileInfo> infos`
The list storing file information objects.
`LazyPixmap[] pixmaps`
The list storing the LazyPixmap objects.

Methods
`QDir getDirectory()`
Gets the current directory.
`String getFileName(int row)`
Gets the file name of the file associated with the given row.
`QImage imageAt(int row)`
Gets the image at a given row index.
`void setDirectory(QDir directory)`
Sets the current directory.
`void update()`
Updates the model.
`void updateRow(int i)`
Updates the given row.

D.3. Class `LazyPixmap`

This class is responsible for loading the image and storing a thumbnail version of that image for the ImageTableModel.

Constructors
`LazyPixmap(String fileName)`
The constructor taking the file name string.

Fields
`String fileName`
The file name string.
`QSignalEmitter.Signal0 loaded`
Signal for confirming the readiness of the image loading.
`QSize size`
The size of the Thumb-Nail image.
`QSize SMALL_SIZE`
The constant field defining the size of thumbnail image.
`QImage thumbNail`
The Thumb-Nail image.

Methods
`QImage image()`
Returns the original image.
`boolean isValid()`
Validation of the existence of the thumbnail image.
`void loadThumbNail()`
Loads the original image after which it constructs the thumbnail and emits loaded.
`QSize size()`
Gives the size of the original image.
`QImage thumbNail()`
Returns the thumbnail image.

D.4. Class MainWindow

The object class building the main window for the application and its basic functionality.

Fields
Used for Ctrl+s which leads to saving the annotation information.
TypeDialog dia
A dia object used on the typeD user interface object.
QTreeView dirView
The QTreeView object for the dirModel used for treeDock QDockWidget.
List<MyGraphicsView> graphiscList
Stores currently displayed MyGraphicsView object (max 3).
int imageInCentralWidget
This integer holds the current number for the revolving MyGraphicsView objects.
ImageTableModel imageModel
This the ImageTableModel object which models the image data which are then shown in the tableView.
QPixmap origPixmap
The QPixmap object holding the image data.
QStackedWidget pagesWidget
This Widget switches chooses which option to show to the user when one of three Polygon modes is selected.
QActionGroup polygonsGroup
Hold together the three QActions responsible for switching between 3 Polygon Modes.
QTableView tableView
An instance of QTableView for the imageModel.
QTabWidget tabWidget
The widget responsible for tabs in the leftDock QDockWidget.
Ui_TypeDialog typeD
A Ui_TypeDialog user interface object (generated by QT Jambi).
Ui_AnnotationClass ui
User Interface for main window object (generated by QT Jambi).

Methods
`void documentWasModified(List<QRectF> li)`
This function is not yet used since there is a bug in QGraphicsScene changed signal.
`void fileLoad_LabelMe(String fileName, QDir dir)`
Loads the contents of a LabelMe .xml file specified by:
`void fileLoad(String fileName, QDir dir)`
Loads the contents of a native UA .xml file
`void fileSave()`
Saves all MyPolygon objects drawn on the MyGraphicsView objects listed in graphiscList.
`void on_action_About_triggered()`
This action launches a Window with information about the application and credits.
`void on_action_Exit_triggered()`
Controls what happens after the Exit Action has been activated.
`void on_action_Poly_Curve_triggered()`
Handles the action of switching to Polygon Curve mode.
`void on_action_Poly_SIOX_triggered()`
Handles the action of switching to Polygon SIOX mode.
`void on_action_Save_triggered()`
Launches the fileSave() methode.
`void on_actionDual_Desktop_toggled()`
Handles the activation of the Dual Desktop action for virtual desktops (more than one screen for workspace).
`void on_actionMono_triggered()`
Controls the Mono Action.
`void on_actionPolygon_triggered()`
Handles the action of switching to Polygon mode.
`void on_actionStereo_triggered()`
Controls the Stereo Action.
`void on_dirView_activated(QModelIndex index)`
Handles the activation of the dirView (clicking on the directory in the ui.treeDock).
`void on_tableView_activated(QModelIndex index)`
This method handles the process of clicking on the tableView.
`void setupDirModel()`
Creates and sets up the QDirModel for the dirView.

D.5. Class MyGraphicsPolygonItem

This class inherits form the QGraphicsPolygonItem and is a fine tuned objects class for the MyPolygon.

Constructors
`MyGraphicsPolygonItem(QPolygonF polygon, QGraphicsItemInterface parent, QGraphicsScene scene, MyPolygon myPolygon)`
The constructor initializes objects.

Methods
`Object itemChange(QGraphicsItem.GraphicsItemChange change, java.lang.Object value)`
We overwrite this method to make it possible to change the appearance of the chosen polygon item.
`void mouseDoubleClickEvent(QGraphicsSceneMouseEvent event)`
This overwritten method takes care of adding an extra control point to the polygon item when double-clicked upon.
`void zValue1Down()`
This method sets the zLevel of this polygon item to a value smaller by one.
`void zValue1Up()`
This method sets the zLevel of this polygon item to a value grater by one.

D.6. Class MyGraphicsView

This class enhances and modifies the functionality of the standard QGraphicsView too meet our requirements.

Constructors
`MyGraphicsView()`
Default initialization.

Fields
`List<MyPolygon> polyList`
List holding references to MyPolygon objects in the scene.
`MyGraphicsPolygonItem selectedPolyItem`
The MyGraphicsPolygonItem currently being selected in the scene.

Methods
`void addOriginalPixmap(QPixmap origPix)`
Adds the original pixel map representation of the image in the scene of this MyGraphicsView.
`void createNewMyPolygon(java.util.List<QPointF> plist, java.lang.String type, int rgba, double z)`
Creates a new MyPolygon object (for polygon red from .xml file).
`double getCurrentZ()`
Gets the currently greatest-tracked z value.
`QDir getImageDir()`
Gets image directory.
`String getImageFileName()`
Gets image file name.
`double nextZValue()`
Calculates the next Z value for the created MyPolygon.
`void resizeEvent(QResizeEvent e)`
Performes the necessary transformations upon resizing of this MyGraphicsView.
`void setCurrentPolygon(MyPolygon currentPolygon)`
Sets the currentPolygon for this MyGraphicsView.
`void setCurrentZ(double currentZ))`
Sets the currently greatest-tracked z value.
`void setImageDir(QDir directory)`
Sets image directory.
`void setImageFileName(java.lang.String name)`
Sets image file name.

D.7. Class MyPolygon

This class organizes the creation and manipulation of the polygon objects on the scene of a MyGraphicsView.

Constructors
MyPolygon(QGraphicsItemInterface parent, MyGraphicsView view, QGraphicsScene scene, java.util.List<QPoint> plist, double Z)
This special constructor is constructing the polygon only partially and is supplemented through the finalizePolygonCreation(List) method.
MyPolygon(QGraphicsItemInterface parent, MyGraphicsView view, QGraphicsScene scene, java.util.List<QPointF> plist, java.lang.String type, int rgba, double Z)
This constructor is used when a polygon is loaded from an .xml file.
MyPolygon(QGraphicsItemInterface parent, MyGraphicsView view, QGraphicsScene scene, QPointF mousePos, double Z)
The main constructor for this object.

Fields
List<MyPolygon.MyGraphicsEllipseItem> cpList
List of control points.
QPolygonF poly
This is a reference for the polygon.
MyGraphicsPolygonItem polyItem
The reference for the actual polygon item of our customized type - MyGraphicsPolygonItem.

Methods
`void addControlPoint(QPointF pos, boolean selected)`
Constructs a new control point on a specified position.
`QPolygonF clipPolygon()`
Clips the polygon item against the image rectangular boundary.
`void finalizePolygonCreation(java.util.List<QPoint> plist)`
This function finishes the partial creation of the MyPolygon started with the constructor MyPolygon(QGraphicsItemInterface, MyGraphicsView, QGraphicsScene, List, double)
`String getThisPolysType()`
Gets this instances object type string.
`QColor randomColor()`
Chooses a random half-transparent color for the polygon item.
`void setThisPolysType(java.lang.String thisPolysType)`
Sets this instances object type string.

D.8. Class `Utility`

Utility class is a place holder for some small algorithms.

Methods
`QPointF dotLineIntersection(double x, double y, double x0, double y0, double x1, double y1)`
The algorithm computing the intersection coordinates of a point built form projectiong a perpendicular line form a given point to a line defined by two points.
`List<QPoint> DouglasPeuckerReduction(java.util.List<QPoint> Points, double Tolerance)`
Uses the Douglas Peucker algorithm to reduce the number of points.
`void DouglasPeuckerReduction(java.util.List<QPoint> points, int firstPoint, int lastPoint, java.lang.Double tolerance, java.util.List<java.lang.Integer> pointIndexsToKeep)`
Douglases the peucker reduction.
`double PerpendicularDistance(QPoint Point1, QPoint Point2, QPoint Point)`
The distance of a point from a line made from point1 and point2.

Appendix E.

Bibliography

[AQ] Trevor Darrell Ariadna Quattoni, Michael Collins. Conditional random fields for object recognition.

[BCR08] K. P. Murphy W. T. Freeman B. C. Russell, A. Torralba. Labelme: a database and web-based tool for image annotation. International Journal of Computer Vision, pages 157-173, Volume 77, Numbers 1-3, May 2008.

[Bre65] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4:25–30, 1965.

[BS06] Steven M. Drucker Ben Shneiderman, Benjamin B. Bederson. Interface strategies to annotate, browse, and share. *COMMUNICATIONS OF THE ACM*, 49(4), April 2006.

[DD73] T.K. Peucker D.H. Douglas. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographer*, 10(2):112–122, 1973.

[DS06] Frank DiMaio and Jude Shavlik. Belief propagation in large, highly connected graphs for 3d part-based object recognition. In *Proceedings of the Sixth International Conference on Data Mining (ICDM'06)*, 2006.

[GF] Lars Knipping Tobias Lenz Gerald Friedland, Kristian Jantz. *Simple Interactive Object Extraction (SIOX)*. www.siox.org.

[GF05] L. Knipping R. Rojas G. Friedland, K. Jantz. Image segmentation by uniform color clustering – approach and benchmark results. June 2005.

[Gmb] Softplot GmbH. *Softplot Image Manager*. http://www.softplot.de.

[Hol] Martin Holmes. *Image Markup Tool*. http://tapor.uvic.ca/~mholmes/image_markup/index.php.

[Kip01] Michael Kipp. Anvil - a generic annotation tool for multimodal dialogue. In *Proceedings of the 7th European Conference on Speech Communication and Technology (Eurospeech)*, pages 1367–1370, 2001.

[MLC] Daniel H. Grollman Jonas N. Schwertfeger Theodora R. Hinkle Micah Lapping Carr, Odest Chadwicke Jenkins. Wiimote interfaces for lifelong robot learning.

[Mor] S.P.; Gupta M. Moreira, J.E.; Midkiff. A comparison of java, c/c++, and fortran for numerical computing. *Antennas and Propagation Magazine*, 40:102–105.

[oISIMJRFm] Institute of Information Systems & Information Management JOANNEUM RESEARCH Forschungsgesellschaft mbH. *Semantic Video Annotation Suite*. http://www.joanneum.at.

[PV] Michael J. Jones Paul Viola. Robust real-time face detection. *International Journal of Computer Vision*, 57.

[Ree03] Trygve Reenskaug. The model-view-controller (mvc) its past and present. August 2003.

[Tro] Trolltech. *QDockWidget class*. http://doc.trolltech.com/qtjambi-4.4/html/com/trolltech/qt/gui/QDockWidget.html.

[Van] Lode Vandevenne. *Flood Fill*. http://student.kuleuven.be/~m0216922/CG/floodfill.html.

[WWWa] *AKTive Media - Ontology based annotation system*. http://www.dcs.shef.ac.uk/~ajay/html/cresearch.html.

[WWWb] *Autodesk 3D Studio Max, Maya*. http://www.autodesk.com.

[WWWc] *Extensible Markup Language*. http://www.w3.org/XML/.

[WWWd] *Inner boundary tracing*. http://iria.pku.edu.cn/~jiangm/courses/dip/html/node127.html.

[WWWe] *Inote: An Image Annotation Tool in Java*. http://www.iath.virginia.edu/inote/.

[WWWf] *Mozilla Public License Version 1.1*. http://www.mozilla.org/MPL/MPL-1.1.html.

[WWWg] *Photoshop*. http://www.adobe.com/products/.

[WWWh] *Photoshop Magic Wand Tool*. http://simplephotoshop.com/photoshop_tools/magic_wandf.htm.

[WWWi] *Photoshop Magnetic Lasso Tool*. http://www.computerbooters.org/Photo_shop/PESIG_jun05.pdf.

[WWWj] *Picolo2D - A Structured 2D Graphics Framework*. http://www.piccolo2d.org.

[WWWk] *Qt - Cross-Platform Application Framework* . http://trolltech.com/.

[WWWl] *SVG - Scalable Vector Graphics* . http://www.w3.org/Graphics/SVG/.

[WWWm] *SWT - The Standard Widget Toolkit*. http://www.eclipse.org/swt/.

[WWWn] *TEI - Text Encoding Initiative*. http://www.tei-c.org.

[WWWo] *Wikipedia; LabelMe*. http://en.wikipedia.org.

[WWW05] *Language Comparison, C, C++ and Java*. http://www.kuro5hin.org/story/2002/6/25/122237/078, Jun 2005.