Development of a Guided Tagging Tool for Whole Slide Images

Master Thesis

Submitted in partial fulfillment of the requirements for the degree of

Master of Science (M.Sc.)
in Applied Computer Science

at the

Berlin University of Applied Sciences (HTW)

First Supervisor: Prof. Dr. Peter Hufnagl

Second Supervisor: Diplom Informatiker Benjaming Voigt

Submitted by:
Sascha Nawrot (B.Sc.)

Berlin, September 15, 2016

Preface

This thesis was created for and in support with the digital pathology department of the Charité - Universitätsmedizin Berlin. The tools created in this thesis will be used there to further advance the state of research on how to use neural networks in the context of digital pathology. All tools are open source and available at GitHub. The further development and diversification of them is highly encouraged.

When starting on this project, I had very limited knowledge of medical image processing, neural networks and digital pathology. Because of that, writing this thesis was an interesting, educational, at times frustrating, but always exciting and rewarding task.

I thank my supervisors, Prof. Dr. Peter Hufnagl and Benjamin Voigt, for the effort, time and patience they invested into mentoring me. Additional thanks goes to Dr. Iris Klempert, who took the time to try the developed tools and give constructive feedback on them. I also want to thank John Cupitt, who, without knowing me, answered all my emails with questions concerning VIPS within hours. Last, but not least, I want to thank Maximilian Mackeprang for his aware eyes.

Abstract

The digitalization of pathologic tissue samples (so called whole slide images) enables the use of image processing and analysis techniques to support pathology experts in their work. The area of digital pathology can greatly benefit from introducing neural networks, which have proven to be ideally suited for tasks like image classification and pattern recognition. In order to be used efficiently, they need training. To do so, training samples and ground truth are needed. Unfortunately, context specific training samples are hard to come by in this context, thus the needed samples must be created. One way to create training samples are image annotations.

The lack of an official standard resulted in a number of proprietary solutions and image formats by different vendors, which require the use of proprietary viewers and annotation tools that are often windows-only. Those tools are not specialized in annotating images and include many other functions, making them difficult to use. This makes the annotation of whole slide images difficult and cumbersome.

Therefore, the goal of this thesis is to introduce novel, lightweight open source annotation tools for whole slide images that enable deep learning and pathology experts to cooperate in creating training samples by annotating regions of interest in whole slide images, regardless of platform and format, in a fast and easy manner. Made annotations will be extracted into regular image (JPEG, PNG) and text (TXT) files, to ensure easy-to-use and non-proprietary training samples.

Contents

1 Introduction **4**
- 1.1 Motivation . . . 4
- 1.2 Research Objective . . . 5
- 1.3 About this thesis . . . 6

2 Background **7**
- 2.1 Whole Slide Image Formats . . . 7
  - 2.1.1 DICOM Supplement 145 . . . 7
  - 2.1.2 Proprietary Formats . . . 8
  - 2.1.3 Open Formats . . . 11
- 2.2 Short Introduction to Neural Networks . . . 18
  - 2.2.1 Methods of Learning . . . 19
  - 2.2.2 The Perceptron . . . 20
  - 2.2.3 Multi-layered Neural Networks . . . 21
- 2.3 Microservices . . . 23
  - 2.3.1 Definition . . . 23
  - 2.3.2 Advantages and Disadvantages . . . 25
  - 2.3.3 Conclusion . . . 25
- 2.4 Process Chain . . . 26
  - 2.4.1 Conversion Service . . . 28
  - 2.4.2 Annotation Service . . . 28
  - 2.4.3 Tessellation Service . . . 29

3 Conversion Service **30**
- 3.1 Methodology . . . 30
  - 3.1.1 Selection of Image Format . . . 31
  - 3.1.2 Deepzoom.py . . . 32
  - 3.1.3 VIPS . . . 34
- 3.2 Implementation . . . 35
- 3.3 Test . . . 37
  - 3.3.1 Setup . . . 37
  - 3.3.2 Result . . . 38

4 Annotation Service **39**
4.1 Objective of the Annotation Service . . . 39
4.2 Methodology . . . 39
4.3 Parts of the Annotation Service . . . 41
4.3.1 Annotation Service Server . . . 41
4.3.2 Annotation Service Viewer . . . 43
4.4 Annotation Service Server Implementation . . . 45
4.4.1 Flask . . . 45
4.4.2 OpenSlide Python . . . 47
4.4.3 Structure and Setup . . . 49
4.4.4 RESTful API . . . 51
4.5 Annotation Service Viewer Implementation . . . 53
4.5.1 Frameworks . . . 54
4.5.2 Definition: Region . . . 58
4.5.3 Graphical User Interface . . . 60

5 Tessellation Service **62**
5.1 Objective of the Tessellation Service . . . 62
5.2 Methodology . . . 62
5.3 Implementation . . . 65
5.3.1 Execution . . . 65
5.3.2 Extraction process . . . 66
5.3.3 Extraction without Tessellation . . . 69
5.3.4 Extraction with Tessellation . . . 72
5.4 Test . . . 73
5.4.1 Setup . . . 73
5.4.2 Result . . . 75

6 Conclusion **79**
6.1 Results . . . 79
6.2 Conclusion . . . 80
6.3 Future work . . . 80

Appendices **82**

A Test Data **83**
A.1 Files on disc . . . 83
A.2 Free Whole Slide Images . . . 84
A.2.1 Aperio (.svs) . . . 84
A.2.2 Generic Tiled tiff (.tiff) . . . 84
A.2.3 Hamamatsu (.ndpi) . . . 85
A.2.4 Hamamatsu (.vms) . . . 85
A.2.5 Leica (.scn) . . . 86
A.2.6 Trestle (.tiff) . . . 86
A.2.7 Ventana (.bif) . . . 86
A.2.8 Mirax (.mrxs) . . . 87

B Annotation Service Documentation **88**
B.1 Annotation Service Server 88
B.2 Annotation Service Viewer 95
B.2.1 Initialization functions 95
B.2.2 Data management functions 97
B.2.3 GUI functions 100
B.2.4 Region functions 103
B.2.5 Interaction functions 105
B.2.6 Internal functions 106
B.2.7 Key listener 109

C Tessellation Service Documentation **111**
C.1 Main 111
C.2 WSI 112
C.3 DZI 115
C.4 Utility 118

Bibliography **124**

List of Figures **131**

List of Tables **133**

Nomenclature **135**

Statutory declaration **136**

Chapter 1

Introduction

1.1 Motivation

The medical discipline of pathology is in a digital transformation. Instead of looking at tissue samples through the means of traditional light microscopy, it is now possible to digitize those samples. This digitalization is done with the help of a so called slide scanner. The result of such an operation is a *whole slide image* (WSI) [11]. The digital nature of WSIs opens the door to the realm of image processesing and analysis which yields certain benefits, such as the use of image segmentation and registration methods to support the pathologist in his/her work.

A very promising approach to image analysis is the use of deep learning, also known as *neural networks* (NN). These are a group of computational models inspired by our current understanding of biological NN. The construct of many interconnected neurons is considered a NN (both in the biological and artificial context). Each single one of those neurons has input values and an output value. Once the input reaches a certain trigger point, the cell in the neuron sends a signal as output. The connections between the neurons are weighted and can dampen or strengthen a signal. Because of this, old pathways can be blocked and new ones created. In other words, a NN is capable of "learning" [60]. This is a huge advantage compared to other software models. While certain problems are "easier" to solve in a sequential, algorithmic fashion (say an equation or the towers of hanoi), certain problems (e.g. image segmentation or object recognition) are very complex, so that new approaches are needed, while other problems can not be solved algorithmically at all. With the use of adequate training samples, a NN can learn to solve a problem, much like a human.

In the recent past the use of NN enabled major breakthroughs, especially in the area of image classification and object recognition. Karpathy and Fei-Fei, for example, created a NN that is capable of describing an image or a scene using natural language text blocks [2] (see fig. 1.1 for a selection of examples).

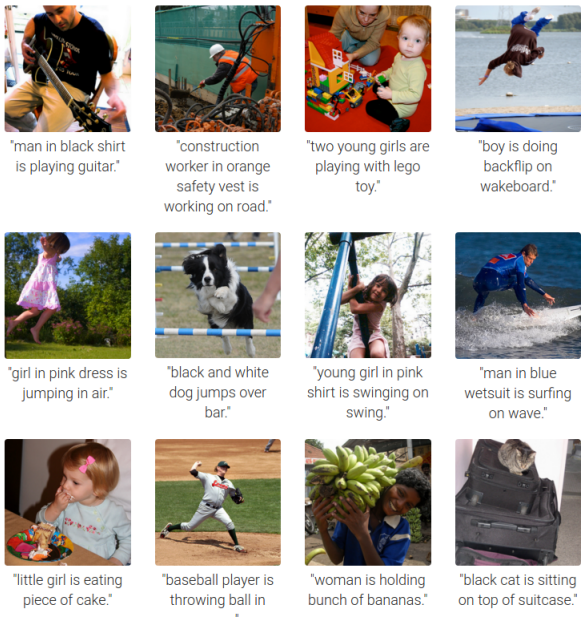


Figure 1.1: Example results of the in [2] introduced model (source: `http://cs.stanford.edu/people/karpathy/deepimagesent/`)

There is enormous potential in the use of NN in the digital pathology as well, but to transfer these models and technologies, certain obstacles must be overcome. One of those is the need for proper training samples. While generally there are large amounts of WSIs (e.g. publicly available at the Cancer Genome Atlas[1]), most of them will not be usable as a training sample without further preparation.

A possible way to prepare them is by using image annotation: tagging regions of interest (ROI) on an image and assigning labels or keywords as metadata to those ROIs. These can be added to the WSIs, stored and later used for training. The result of such an approach could be similar to the one of Karpathy and Fei-Fei [2], but with a medical context instead of daily situations.

1.2 Research Objective

The goal of this thesis is the conceptualization and implementation of tools to enable deep learning and pathology experts to cooperate in annotating WSIs to create a ground truth for the later use in NNs. In order to do so, it must be possible to open a given WSI with a viewer, add annotations to it and persist

[1] `https://gdc-portal.nci.nih.gov/`

those annotations. Additionally, persisted annotations must be extracted from the WSI to be used as ground truth.

This thesis has three objectives:

(1) There is no standard for WSI files, therefore vendors developed their own proprietary solutions [11]. This either leads to a vendor lock-in or separate handling of each proprietary format. To avoid both cases, a conversion tool that turns proprietary WSI formats into an open format must be introduced.

(2) The deployment of a WSI viewer tool. This WSI viewer must be capable of adding annotations to a WSI and persisting them. As stated earlier, the tool is intended to be used by deep learning and pathology experts. An intuitive and easy-to-understand graphical user interface (GUI) is necessary to avoid long learning periods and create willingness to actually use the tool.

(3) The implementation of a tool that is capable of turning persisted annotations into a format usable as ground truth.

It is explicitly stated, that the intention of this thesis is to introduce tools that are used by deep learning and pathology experts to create a ground truth for NN. The intention is **not** to create a tool for analyzing and diagnosing WSIs, that is capable of competing with existing industry solutions.

1.3 About this thesis

This thesis contains 6 chapters.

Chapter 1 - Introduction and *2 - Background* address the scope, background and vocabulary of this thesis.

The chapters 3 to 5 address the components described in the last section: *chapter 3 - Conversion Service* will describe a tool for image conversion, *chapter 4 - Annotation Service* will describe a tool for image annotation and *chapter 5 - Tessellation Service* will describe an extraction tool, to prepare the annotations made with the Annotation Service for the use in a NN.

Finally, *Chapter 6 - Conclusion* will discuss and conclude the findings of the aforementioned chapters.

Chapter 2

Background

2.1 Whole Slide Image Formats

Due to the amount of data stored in a raw, uncompressed WSI[1], file formatting and compression are required to make working with WSIs feasible. Since there is no standardized format for WSIs, vendors came up with their own, proprietary solutions, which vary greatly [11]. Efforts of standardization are being made through the *Digital Imaging and Communications in Medicine* (DICOM) Standard [16].

Usually, WSI files are stored as a multitude of single images, spanning multiple folders and different resolutions. Those files are used to construct a so called *image pyramid* [30] (see fig. 2.1 and subsection 2.1.1).

2.1.1 DICOM Supplement 145

Singh et al. [33] describe DICOM as follows:

> "Digital Imaging and Communications in Medicine (DICOM), synonymous with ISO (International Organization for Standardization) standard 12052, is the global standard for medical imaging and is used in all electronic medical record systems that include imaging as part of the patient record."

Before *Supplement 145: Whole Slide Microscopic Image IOD and SOP Classes*, the DICOM Standard did not address standardization of WSI. Among others, the College of American Pathologist's Diagnostic Intelligence and Health Information Technology Committee is responsible for the creation and further advancement of this supplement [33].

It addresses every step involved in creating WSIs: image creation, acquisition, processing, analyzing, distribution, visualization and data management

[1] A typical 1,600 megapixel slide requires about 4.6 GB of memory on average [30]. The size of a H&E (hematoxylin and eosin) stained slide ranges typically from 4 to 20 GB [33].

[16]. It impacted the way how data is stored greatly [33], due to the introduction of a pyramid image model [16] (see fig. 2.1).

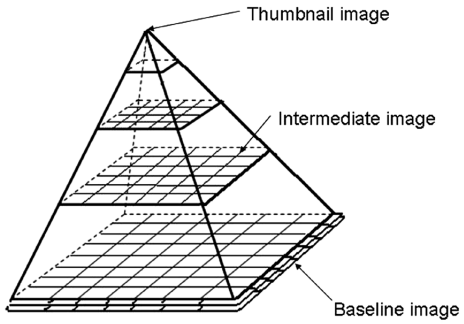


Figure 2.1: DICOMs image pyramid (source: [33])

The image pyramid model facilitates rapid zooming and reduces the computational burden of randomly accessing and traversing a WSI [33], [35]. This is made possible by storing an image in several precomputed resolutions, with the highest resolution sitting at the bottom (called the *baseline image*) and a thumbnail or low power image at the top (compare fig. 2.1) [16]. This creates a pyramid like stack of images, hence the name "pyramid model". The different resolutions are referred to as *layers* [16] or *levels* [33] respectively.

Each level is tessellated into square or rectangular fragments, called tiles, and stored in a two dimensional array [30].

Because of this internal organization, the tiles of each level can be retrieved and put together separately, to either form a subregion of the image or show it entirely. This makes it easy to randomly access any subregion of the image without loading large amounts of data [33].

2.1.2 Proprietary Formats

Vendors of whole slide scanners implement their own file formats, libraries and viewers (see tab. 2.1 for a list of vendors and their formats). Because of this, they can focus on the key features and abilities of their product. This generally leads to a higher usability, ease-of-use and enables highly tailored customer support. Furthermore, in comparison to open source projects, the longevity of proprietary software is often higher [48].

Since the proprietary formats have little to no documentation, most of the subsequently presented information was reverse engineered in [21] and [67]. All proprietary formats listed here implement a modified version of the pyramid model introduced in 2.1.1

vendor	formats
Aperio	SVS, TIF
Hamamatsu	VMS, VMU, NDPI
Leica	SCN
3DHistech/Mirax	MRXS
Philips	TIFF
Sakura	SVSLIDE
Trestle	TIF
Ventana	BIF, TIF

Table 2.1: File formats by vendor

Aperio

The SVS format by Aperio is a TIFF-based format, which comes in a single file [21]. It has a specific internal organization in which the first image is the baseline image, which is always tiled (usually with 240x240 pixels). This is followed by a thumbnail, typically with dimensions of about 1024x768 pixels. The thumbnail is followed by at least one intermediate pyramid image (compare fig. 2.1), with the same compression and tile organization as the baseline image [67]. Optionally, there may be a slide label and macro camera image at the end of each file [67].

Hamamatsu

Hamamatsu WSIs come in 3 variants:

(1) VMS

(2) VMU

(3) NDPI

(1) and (2) consist of an index file ((1) - [file name].vms, (2) - [file name].vmu) and 2 or more image files. In the case of (2), there is also an additional optimization file. (3) consists of a single TIFF-like file with custom TIFF tags. While (1) and (3) contain JPEG images, (2) contains a custom, uncompressed image format called *NGR*[2] [67].

The random access support for decoding parts of jpeg files is poor [67]. To work around this, so called *restart markers* are used to create virtual slides. Restart markers were originally designed for error recovery. They allow a decoder to resynchronize at set intervals throughout the image [21]. These markers are placed at regular intervals. Their offset is specified in a different manner, depending on the file format. In the case of (1), it can be found in the index file. In the case of (2), the optimization file holds the information and in the case of (3), a TIFF tag contains the offset [67].

[2]For more information on NGR, consult `http://openslide.org/formats/hamamatsu/`

Leica

SCN is a single file format based on BigTIFF that additionally provides a pyramidal thumbnail image [21]. The first TIFF directory has a tag called "ImageDescription" which contains an XML document that defines the internal structure of the WSI [67].

Leica WSIs are structured as a collection of images, each of which has multiple pyramid levels. While the collection only has a size, images have a size and position, all measured in nanometers. Each dimension has a size in pixels, an optional focal plane number, and a TIFF directory containing the image data. Fluorescence images have different dimensions (and thus different TIFF directories) for each channel [67].

Brightfield slides have at least two images: a low-resolution macro image and one or more main images corresponding to regions of the macro image. Fluorescence slides can have two macro images: one brightfield and one fluorescence [67].

3DHistech/Mirax

MRXS is a multi-file format with complex metadata in a mixture of text and binary formats. Images are stored as either JPEG, PNG or BMP [21]. The poor handling of random access is also applicable to PNG. Because of this, multiple images are needed to encode a single slide image. To avoid having many individual files, images are packed into a small number of data files. An index file provides offsets into the data files for each required piece of data. [67].

A 3DHistech/Mirax scanner take images with an overlap. Each picture taken is then tessellated without an overlap. Therefore, overlap only occurs between taken pictures [67].

The generation of the image pyramid differs from the process described in 2.1.1 To create the n^{th} level, each image of the $n^{th} - 1$ level is divided by 2 in each dimension and then concatenated into a new image. Where the $n^{th} - 1$ level had 4 images in 2x2 neighborhood, the n^{th} level will only have 1 image. This process has no regards for overlaps. Thus, overlaps may occur in the higher levels of the image pyramid [67].

Philips

Philips' TIFF is an export from the native iSyntax format. An XML document with the hierarchical structure of the WSI can be found over the *ImageDescription* tag of the first TIFF directory. It contains key-value pairs based on DICOM tags [67].

Slides with multiple regions of interest are structured as a single image pyramid enclosing all regions. Slides may omit pixel data for TIFF tiles not in an ROI. When such tiles are downsampled into a tile that does contain pixel data, their contents are rendered as white pixels [67].

Label and macro images are stored either as JPEG or as stripped TIFF directories.

Sakura

WSIs in the SVSLIDE format are SQLite 3 database files. Their tables contain the metadata, associated images and tiles in the JPEG format. The tiles are addressed as a tupel of ⟨ focal plane, downsample, level-0 X coordinate, level-0 Y coordinate, color channel ⟩. Additionally, each color channel has a separate grayscale image [67].

Trestle

Trestles TIF is a single-file TIFF. The WSI has the standard pyramidic scheme and tessellation. It contains non-standard metadata and overlaps, which are specified in additional files. The first image in the TIFF file is the baseline image. Subsequent images are assumed to be consecutive levels of the image pyramid with decreasing resolution [67].

Ventana

Ventanas WSIs are single-file BigTIFF images, organized in the typical pyramidical scheme. The images are tiled and have non-standard metadata, as well as overlaps. They come with a macro and a thumbnail image [67].

2.1.3 Open Formats

As mentioned in 2.1.2, proprietary formats typically come without much or any documentation. Furthermore, a vendors viewer is usually the only way of viewing WSIs of a particular format. This creates a vendor lock-in, where users can not take advantage of new improvements offered by other vendors. Furthermore, most viewers only provide support for Windows platforms. While, in a clinical setting, Windows may dominate the market, a significant amount of users in medical research prefer Linux or Mac OS X [21]. The use of mobile platforms, such as iOS or Android tablets may also have a great influence of the work flow in the future. Some vendors try to compensate for this fact with a server-based approach, which hurts performance by adding a network round-trip delay on every digital slide operation [21].

To resolve these issues, open image formats have been suggested, which will be discussed further in the following subsections.

Deep Zoom Images

The DZI format is an XML-based file format, developed and maintained by Microsoft [64]. A DZI is a pyramidcal, tiled image (see fig. 2.2), similar to the one described in 2.1.1 (compare 2.1 and 2.2), with two exceptions:

1. the baseline image is referred to as the highest level, instead of the lowest; this either turns the image pyramid or its labeling upside down
2. tiles are always square, with the exception of the last column/row

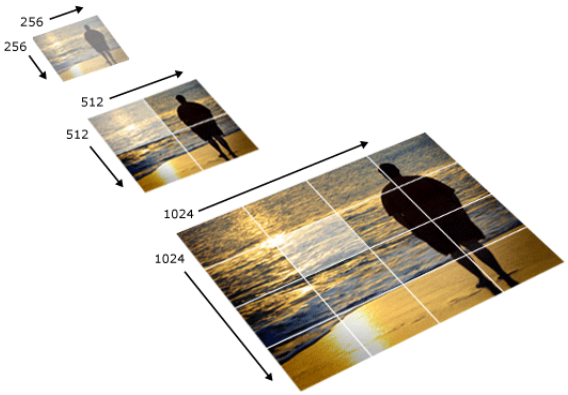


Figure 2.2: DZI pyramid model example (source: [64])

A DZI consists of two main parts [64]:

(1) a describing XML file ([file name].dzi) with the following metadata:

- format of individual tiles (e.g. JPEG or PNG)
- overlap between tiles
- size of individual tiles
- height and width of baseline image

(2) a directory ([file name]_files) containing image tiles of the specified format

(1) and (2) are stored "next" to each other, so that there are 2 spearate files. (2) contains sub directories, one for each level of the image pyramid. The baseline image of a DZI is in the highest level. Each level is tessellated into as many tiles necessary to go over the whole image, with each tile having the size specified in the XML file. If the image size is no multiple of the specified tile size, the width of the n^{th} column of tiles will be ($width$ mod $tile\ size$) pixels. Equally, the height of the m^{th} row will be ($height$ mod $tile\ size$) pixels. Thus, the outermost right bottom tile $t_{n,m}$ will be of ($width$ mod $tile\ size$) x ($height$ mod $tile\ size$) pixels.

International Image Interoperability Framework

The International Image Interoperability Framework (IIIF) is the result of a cooperation between The British Library, Stanford University, the Bodleian Libraries (Oxford University), the Bibliothèque Nationale de France, Nasjonalbiblioteket (National Library of Norway), Los Alamos National Laboratory Research Library and Cornell University [12]. Version 1.0 was published in 2012.

IIIFs goal is to collaboratively produce an interoperable technology and community framework for image delivery [46]. To achieve this, IIIF tries to:

(1) give scholars access to image-based resources around the world

(2) define a set of common APIs to support interoperability between image repositories

(3) develop and document shared technologies (such as image servers and web clients), that enable scholars to view, compare, manipulate and annotate images

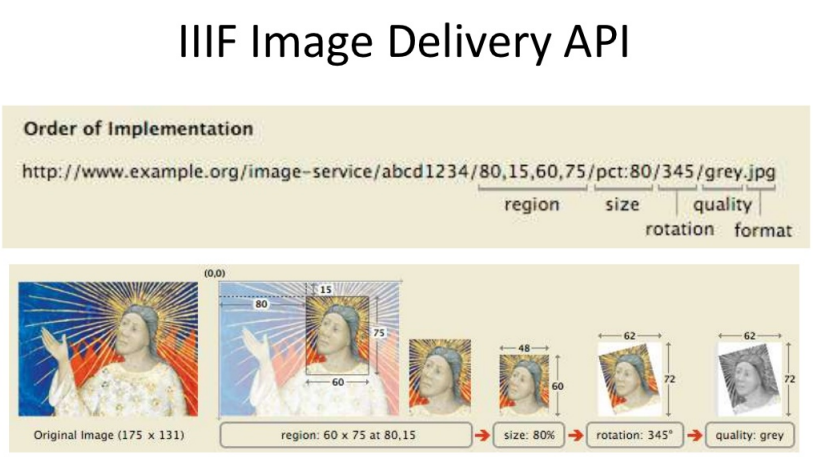


Figure 2.3: Example of iiif request (source:`http://www.slideshare.net/Tom-Cramer/iiif-international-image-interoperability-framework-dlf2012?ref=https://www.diglib.org/forums/2012forum/transcending-silos-leveraging-linked-data-and-open-image-apis-for-collaborative-access-to-digital-facsimiles/`)

The part relevant for this thesis is (2), especially the image API [27]. It specifies a web service that returns an image in response to a standard web request. The URL can specify the region, size, rotation, quality and format of the requested image (see 2.3). Originally intended for resources in digital image repositories maintained by cultural heritage organizations, the API can be used to retrieve static images in response to a properly constructed URL [45]. The URL scheme looks like this [3]:

[3]For detailed information on all parameters see the official API: `http://iiif.io/api/image/2.0`

```
{scheme}://{server}{/prefix}/{identifier}/{region}/{size}/{rotation}/{quality}.{format}
```

The *region* and *size* parameters are of special interest for this thesis. With them, it is possible to request only a certain region of an image in a specified size.

The region parameter defines the rectangular portion of the full image to be returned. It can be specified by pixel coordinates, percentage or by the value "full" (see tab. 2.2 and fig. 2.4).

Form	**Description**
full	The complete image is returned, without any cropping.
x,y,w,h	The region of the full image to be returned is defined in terms of absolute pixel values. The value of x represents the number of pixels from the 0 position on the horizontal axis. The value of y represents the number of pixels from the 0 position on the vertical axis. Thus the x,y position 0,0 is the upper left-most pixel of the image. w represents the width of the region and h represents the height of the region in pixels.
pct:x,y,w,h	The region to be returned is specified as a sequence of percentages of the full image's dimensions, as reported in the Image Information document. Thus, x represents the number of pixels from the 0 position on the horizontal axis, calculated as a percentage of the reported width. w represents the width of the region, also calculated as a percentage of the reported width. The same applies to y and h respectively. These may be floating point numbers.

Table 2.2: Valid values for *region* parameter (source: [27])

If the request specifies a region whose size extends beyond the actual size of the image, the response should be a cropped image, instead of an image with added empty space. If the region is completely outside of the image, the response should be a "404 Not Found" HTTP status code [27].

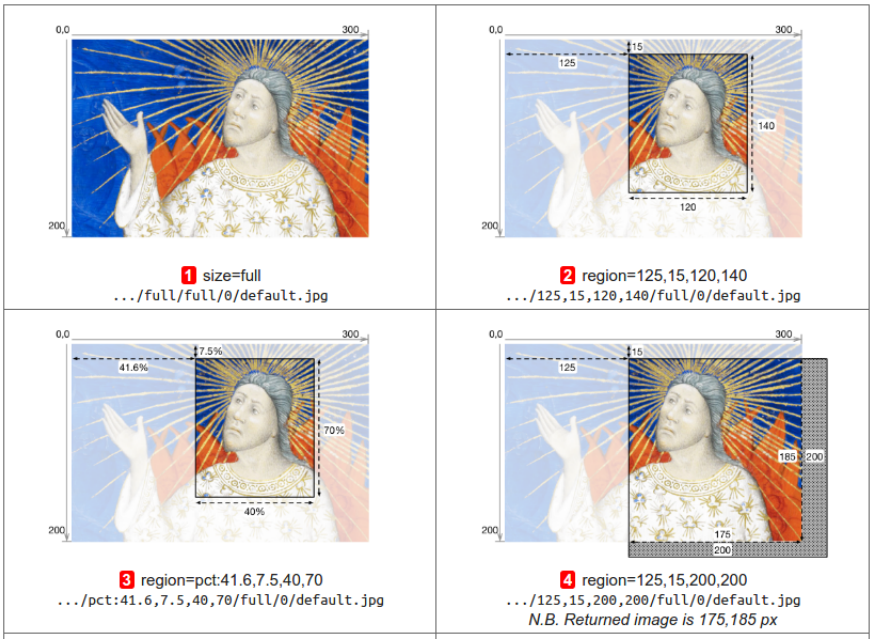


Figure 2.4: Results of IIIF request with different values for region parameter (source: [27])

If a region was extracted, it is scaled to the dimensions specified by the size parameter (see tab. 2.3 and fig. 2.5).

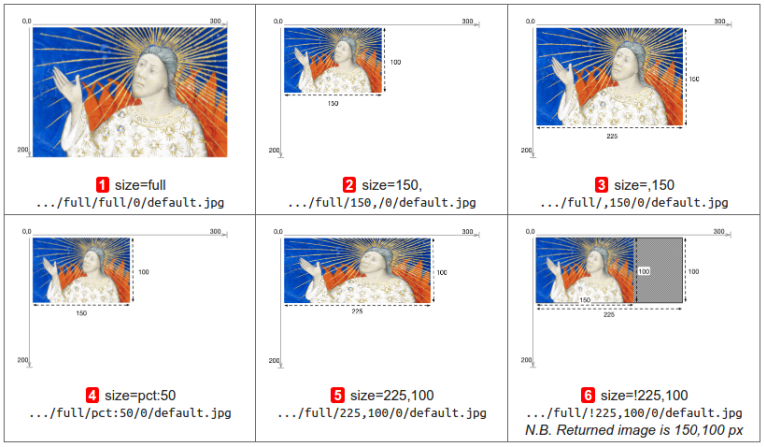


Figure 2.5: Results of IIIF request with different values for size parameter (source: [27])

If the resulting height or width equals 0, then the server should return a "400 Bad Request" HTTP status code. Depending on the image server, scaling above the full size of the extracted region may be supported [27].

Form	**Description**
full	The extracted region is not scaled, and is returned at its full size.
w,	The extracted region should be scaled so that its width is exactly equal to w, and the height will be a calculated value that maintains the aspect ratio of the extracted region.
,h	The extracted region should be scaled so that its height is exactly equal to h, and the width will be a calculated value that maintains the aspect ratio of the extracted region.
pct:n	The width and height of the returned image is scaled to n% of the width and height of the extracted region. The aspect ratio of the returned image is the same as that of the extracted region.
w,h	The width and height of the returned image are exactly w and h. The aspect ratio of the returned image may be different than the extracted region, resulting in a distorted image.
!w,h	The image content is scaled for the best fit such that the resulting width and height are less than or equal to the requested width and height. The exact scaling may be determined by the service provider, based on characteristics including image quality and system performance. The dimensions of the returned image content are calculated to maintain the aspect ratio of the extracted region.

Table 2.3: Valid values for *size* parameter (source: [27])

To use the IIIF API, a compliant web server must be deployed. Loris and IIPImageserver are examples for open source IIIF API compliant systems [45]:

- **Loris**, an open source image server based on python that supports the IIIF API versions 2.0, 1.1 and 1.0. Supported image formats are JPEG, JPEG2000 and TIFF.
- **IIPImage Server**, an open source Fast CGI module written in C++, that is designed to be embedded within a hosting web server such as Apache, Lighttpd, MyServer or Nginx. Supported image formats are JPEG2000 and TIFF [46].

OpenStreetMap/Tiled Map Service

OpenStreetMap (OSM) is a popular tile source used in many online geographic mapping specifications [45]. It is a community driven alternative to services such as Google Maps. Information is added by users via aerial images, GPS devices and field maps. All OSM data is classified as *open data*, meaning that it can be used anywhere, as long as the OSM Foundation is credited [41].

Tiled Map Service (TMS) is a tile scheme developed by the Open Source Geospatial Foundation (OSGF) [45] and specified in [40]. The OSGF is a non-profit organization whose goal it is to support the needs of the open source geospatial community. TMS provides access to cartographic maps of geo-referenced data. Access to these resources is provided via a "REST" interface, starting with a root resource describing available layers, then map resources with a set of scales, then scales holding sets of tiles [40].

Both, OSM and TMS, offer zooming images, which in general, have the functionality necessary, to be of use for this thesis. Unfortunately, they are also highly specialized on the needs of the mapping community, with many features not needed in the context of this thesis.

JPEG 2000

[56] describes the image compression standard JPEG 2000 as follows:

> "JPEG 2000 is an image coding system that uses state-of-the-art compression techniques based on wavelet technology. Its architecture lends itself to a wide range of uses from portable digital cameras through to advanced pre-press, medical imaging and other key sectors."

It incorporates a mathematically lossless compression mode, in which the storage requirement of images can be reduced by an average of 2:1 [15]. Furthermore, a visually lossless compression mode is provided. At *visually lossless compression rates*, even a trained observer can not see the difference between original and compressed version. The visually lossless compression mode achieves compression rates of 10:1 and up to 20:1 [50]. JPEG 2000 code streams offer mechanisms to support random access at varying degrees of granularity. It is possible to store different parts of the same picture using different quality [15].

In the compression process, JPEG 2000 partitions an image into rectangular and non-overlapping tiles of equal size (except for tiles at the image borders). The tile size is arbitrary and can be as large as the original image itself (resulting in only one tile) or as small as a single pixel. Furthermore, the image gets decomposed into a multiple resolution representation [62].

This creates a tiled image pyramid, similar to the one described in subsection 2.1.1.

The encoding-decoding process of JPEG 2000 is beyond the scope of this thesis. Therefore, it is recommended to consult either [50] for a quick overview or [62] for an in depth guide.

TIFF/BigTIFF

The Tagged Image File Format (TIFF) consists of a number of corresponding key-value pairs (e.g. *ImageWidth* and *ImageLength*, who describe the width and length of the contained image) called *tags*. One of the core features of this format is that it allows for the image data to be stored in tiles [20].

Each tile offset is saved in an image header, so that efficient random access to any tile is granted. The original specification demands a use of 32 bit file offset values, limiting the maximum offset to 2^{32}. This constraint limits the file size to be below 4 GB [20].

This constraint led to the development of BigTIFF. The offset values were raised to a 64 bit base, limiting the maximum offset to 2^{64}. This results in an image size of up to 18,000 peta bytes [17].

TIFF and BigTIFF are capable of saving images in multiple resolutions. Together with the feature of saving tiles, the image pyramid model (as described in subsection 2.1.1) can be applied [18].

2.2 Short Introduction to Neural Networks

The objective of the workflows introduced in chapter 1.2 is to create training samples for NNs. Before going into other details, it is necessary to clarify what NNs are, how they work, why they need training samples and what they are used for[4].

Artificial NNs are a group of models inspired by Biological Neural Networks (BNN) . BNNs can be described as an interconnected web of neurons (see 2.6), whose purpose it is to transmit information in the form of electrical signals. A neuron receives input via dendrites and sends output via axons [71]. An average human adult brain contains about 10^{11} neurons. Each of those receives input from about 10^4 other neurons. If their combined input is strong enough, the receiving neuron will send an output signal to other neurons [14].

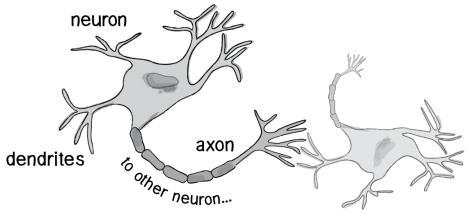


Figure 2.6: Neuron in a BNN (source: [71])

Although artificial NNs are much simpler in comparison (they seldom have

[4] An in-depth introduction into the field of NNs is far beyond the scope of this work. For further information about NNs, consultation of literature (e.g. [8], [14], [28], [60], [71]) is highly recommended.

more than a few dozen neurons [14]), they generally work in the same fashion.

One of the biggest strengths of a NN, much like a BNN, is the ability to adapt by learning (as humans, NN learn by training [71]). This adaption is based on *weights* that are assigned to the connections between single neurons. Fig 2.7 shows an exemplary NN with neurons and the connections between them.

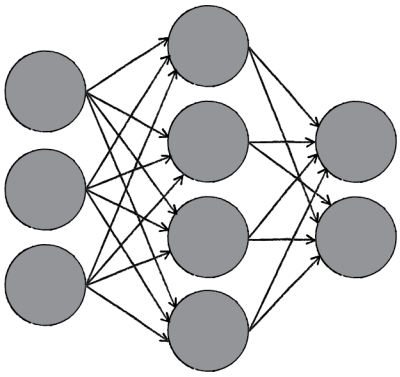


Figure 2.7: Exemplary NN (source: [71])

Each line in fig. 2.7 represents a connection between 2 neurons. Those connections are a one-directional flow of information, each assigned with a specific weight. This weight is a simple number that is multiplied with the incoming/outgoing signal and therefore weakens or enhances it. They are the defining factor of the behavior of a NN. Determining those values is the purpose of training a NN [14].

According to [71], some of the standard use cases for NN are: Pattern Recognition, Time Series Prediction, Signal Processing Perceptron, Control, Soft Sensors, and Anomaly Detection

2.2.1 Methods of Learning

There are 3 general strategies when it comes to the training of a NN [14]. Those are:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning (a variant of Unsupervised Learning [69])

Supervised Learning is a strategy that involves a training set to which the correct output is known (a so called *ground truth*), as well as an observing teacher. The NN is provided with the training data and computes its output. This output is compared to the expected output and the difference is measured.

According to the error made, the weights of the NN are corrected. The magnitude of the correction is determined by the used learning algorithm [69].

Unsupervised Learning is a strategy that is required when the correct output is unknown and no teacher is available. Because of this, he NN must organize itself [71]. [69] makes a distinction between 2 different classes of unsupervised learning:

- reinforced learning
- competitive learning

Reinforced learning adjusts the weights in such a way, that desired output is reproduced. An example is a robot in a maze: If the robot can drive straight without any hindrances, it can associate this sensory input with driving straight (desired outcome). As soon as it approaches a turn, the robot will hit a wall (non-desired outcome). To prevent it from hitting the wall it must turn, therefore the weights of turning must be adjusted to the sensory input of being at a turn. Another example is *Hebbian learning* (see [69] for further information).

In competitive learning, the single neurons compete against each other for the right to give a certain output for an associated input. Only one element in the NN is allowed to answer, so that other, competing neurons are inhibited [69].

2.2.2 The Perceptron

The perceptron was invented by Rosenblatt at the Cornell Aeronautical Laboratory in 1957 [70]. It is the computational model of a single neuron and as such, the simplest NN possible [71]. A perceptron consists of one or more inputs, a processor and a single output (see fig. 2.8) [70].

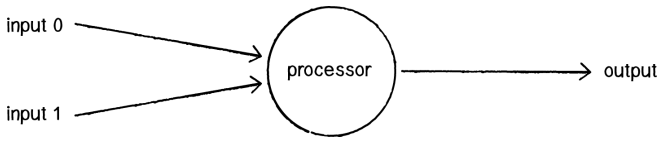


Figure 2.8: Perceptron by Rosenblatt (source: [71])

This can be directly compared to the neuron in fig. 2.6, where:

- input = dendrites
- processor = cell
- output = axon

A perceptron is only capable of solving *linearly separable* problems, such as logical *AND* and *OR* problems. To solve non-linearly separable problems, more then one perceptron is required [70]. Simply put, a problem is linearly

separable, if it can be solved with a straight line (see fig. 2.9), otherwise it is considered a non-linearly separable problem (see fig. 2.10).

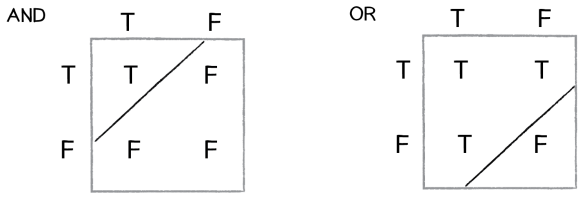


Figure 2.9: Examples for linearly separable problems (source: [71])

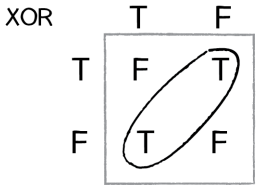


Figure 2.10: Examples for non-linearly separable problems (source: [71])

2.2.3 Multi-layered Neural Networks

To solve more complex problems, multiple perceptrons can be connected to form a more powerful NN. A single perceptron might not be able to solve *XOR*, but one perceptron can solve *OR*, while the other can solve $\neg AND$. Those two perceptrons combined can solve *XOR* [71].

If multiple perceptrons get combined, they create layers. Those layers can be separated into 3 distinct types [8]:

- input layer
- hidden layer
- output layer

A typical NN will have an input layer, which is connected to a number of hidden layers, which either connect to more hidden layers or, eventually, an output layer (see fig. 2.11 for a NN with one hidden layer).

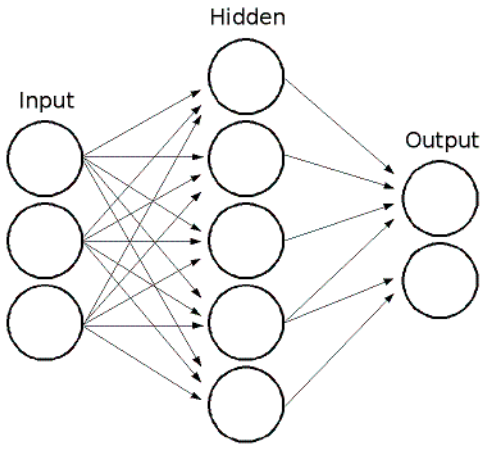


Figure 2.11: NN with multiple layers (source: `http://docs.opencv.org/2.4/_images/mlp.png`)

As the name suggests, the input layer gets provided with the raw information input. Depending on the internal weights and connections inside the hidden layer, a representation of the input information gets formed. At last, the output layer generates output, again based on the connections and weights between the hidden and output layer [8].

Training this kind of NN is much more complicated than training a simple perceptron, since weights are scattered all over the NN and its layers. A solution to this problem is called *backpropagation* [71].

Backpropagation

Training is an optimization process. To optimize something, a metric to measure has to be established. In the case of backpropagation, this metric is the accumulated output error of the NN to a given input. There are several ways to calculate this error, with the *mean square error* being the most common one [14]. The mean square error describes the average of the square of the differences of two variables (in this case the expected and the actual output).

Finding the optimal weights is an iterative process of the following steps:

1. start with training set of data with known output
2. initialize weights in NN
3. for each set of input, feed the NN and compute the output
4. compare calculated with known output
5. adjust weights to reduce error

There are 2 possibilities in how to proceed. The first one is to compare results and adjust weights after each input/output-cycle. The second one is to calculate the accumulated error over a whole iteration of the input/output-cycle. Each of those iterations is known as an *epoch* [14].

2.3 Microservices

The following section elaborates on the concept of *Microservices* (MS), defining what they are, listing their advantages and disadvantages, as well as explaining why this approach was chosen over a monolithic approach. A monolithic software solution is described by [55] as follows:

> "[...] a monolithic application [is] built as a single unit. Enterprise Applications are often built in three main parts: a client-side user interface (consisting of HTML pages and javascript running in a browser on the user's machine) a database (consisting of many tables inserted into a common, and usually relational, database management system), and a server-side application. The server-side application will handle HTTP requests, execute domain logic, retrieve and update data from the database, and select and populate HTML views to be sent to the browser. This server-side application is a monolith - a single logical executable. Any changes to the system involve building and deploying a new version of the server-side application."

2.3.1 Definition

MS are an interpretation of the Service Oriented Architecture. The concept is to separate one monolithic software construct into several smaller, modular pieces of software [79]. As such, MS are a modularization concept. However, they differ from other such concepts, since MS are independent from each other. This is a trait, other modularization concepts usually lack [79]. As a result, changes in one MS do not bring up the necessity of deploying the whole product cycle again, but just the one service. This can be achieved by turning each MS into an independent process with its own runtime [55].

This modularization creates an information barrier between different MS. Therefore, if MS need to share data or communicate with each other, light weight communication mechanisms must be established, such as a RESTful API [68].

Even though MS are more a concept than a specific architectural style, certain traits are usually shared between them [68]. According to [68] and [55], those are:

(a) **Componentization as a Service:** bringing chosen components (e.g. external libraries) together to make a customized service

(b) **Organized Around Business Capabilities:** cross-functional teams, including the full range of skills required to achieve the MS goal

(c) **Products instead of Projects:** teams own a product over its full lifetime, not just for the remainder of a project

(d) **Smart Endpoints and Dumb Pipes:** each microservice is as decoupled as possible with its own domain logic

(e) **Decentralized Governance:** enabling developer choice to build on preferred languages for each component.

(f) **Decentralized Data Management:** having each microservice label and handle data differently

(g) **Infrastructure Automation:** including automated deployment up the pipeline

(h) **Design for Failure:** a consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of single or multiple services

Furthermore, [7] defined 5 architectural constraints, which should help to develop a MS:

(1.) **Elastic**
The elasticity constraint describes the ability of a MS to scale up or down, without affecting the rest of the system. This can be realized in different ways. [7] suggests to architect the system in such a fashion, that multiple stateless instances of each microservice can run, together with a mechanism for service naming, registration, and discovery along with routing and load-balancing of requests.

(2.) **Resilient**
This constraint is referring to the before mentioned trait (h) - *Design for Failure*. The failure of or an error in the execution of a MS must not impact other services in the system.

(3.) **Composable**
To avoid confusion, different MS in a system should have the same way of identifying, representing, and manipulating resources, describing the API schema and supported API operations.

(4.) **Minimal**
A MS should only perform one single business function, in which only semantically closely related components are needed.

(5.) **Complete**
A MS must offer a complete functionality, with minimal dependencies to other services. Without this constraint, services would be interconnected again, making it impossible to upgrade or scale individual services.

2.3.2 Advantages and Disadvantages

One big advantage of this modularization is that each service can be written in a different programming language, using different frameworks and tools. Furthermore, each microservice can bring along its own support services and data storages. It is imperative for the concept of modularization, that each microservice has its own storage of which it is in charge of [79].

The small and focused nature of MS makes scaling, updates, general changes and the deployment process easier. Furthermore, smaller teams can work on smaller code bases, making the distribution of know-how easier [68].

Another advantage is how well MS plays into the hands of agile, scrum and continuous software development processes, due to their previously discussed inherent traits.

The modularization of MS does not only yield advantages. Since each MS has its own, closed off data management (see 2.3.1(f)), interprocess communication becomes a necessity. This can lead to communicational overhead which has a negative impact on the overall performance of the system [79].

2.3.1(e) (*Decentralized Governance*) can lead to compatibility issues, if different developer teams chose to use different technologies. Thus, more communication and social compatibility between teams is required. This can lead to an unstable system which makes the deployment of extensive workarounds necessary [68].

It often makes sense to share code inside a system to not replicate functionality which is already there and therefore increase the maintenance burden. The independent nature of MS can make that very difficult, since shared libraries must be build carefully and with the fact in mind, that different MS may use different technologies, possibly creating dependency conflicts.

2.3.3 Conclusion

The tools needed to achieve the research objective stated in subsection 1.2 will be implemented by using the MS modularization patterns. Due to the implementation being done by a single person, some of the inherent disadvantages of MS are negated (making them a favorable modularization concept):

- Interprocess communication does not arise between the single stages of the process chain, since they have a set order (e.g. it would not make sense trying to extract a training sample without converting or annotating a WSI first).
- Different technologies may be chosen for the single steps of the process chain, however, working alone on the project makes technological incompatibilities instantly visible
- The services should not share functionality, therefore there should be no need for shared libraries

This makes the advantages outweight the disadvantages clearly:

- different languages and technologies can be used for every single step of the process chain, making the choice of the most fitting tool possible
- WSIs take a heavy toll on memory and disk space due to their size; the use of MS allows each step of the chain to handle those issues in the most suitable way for each given step
- separating the steps of the process chain into multiple MS leads to well separated modules, each having a small and therefore easy to maintain codebase
- other bachelor/master students may continue to use or work on this project in the future, further increasing the benefit of a small, easily maintainable code base

2.4 Process Chain

This section and its following subsections are dedicated to establish the process chain necessary to accomplish the research objectives stated in 1.2. The process to create training samples from a set of WSI images is defined as follows:

(1.) convert chosen WSI img_i^{wsi} to open format img_i^{cvrt}

(2.) open img_i^{cvrt} in a viewer V

(3.) annotate img_i^{cvrt} in V

(4.) persist annotations A_i on img_i^{cvrt} in a file $f_{(A_i)}$

(5.) create training sample ts_i by extracting the information of A_i in correspondence to img_i^{cvrt}

While it only makes sense to run (1.) once per img_i^{wsi} to create img_i^{cvrt}, steps (2.) - (4.) can be repeated multiple times, so that there is no need to finish the annotation of an image in one session. That makes it necessary to not only save but also load annotations. Therefore, the loading of already made annotations can be added as step (2.5). This also enables the user of editing and deleting already made annotations. Because of this, step (5.) also needs to be repeatable (see fig. 2.12).

The single steps of the process chain will be sorted into semantic groups. Each group will be realized on its own, in a MS like fashion. The semantic groups are: conversion (1.), extraction (5.) and viewing and annotation (2. - 4.).

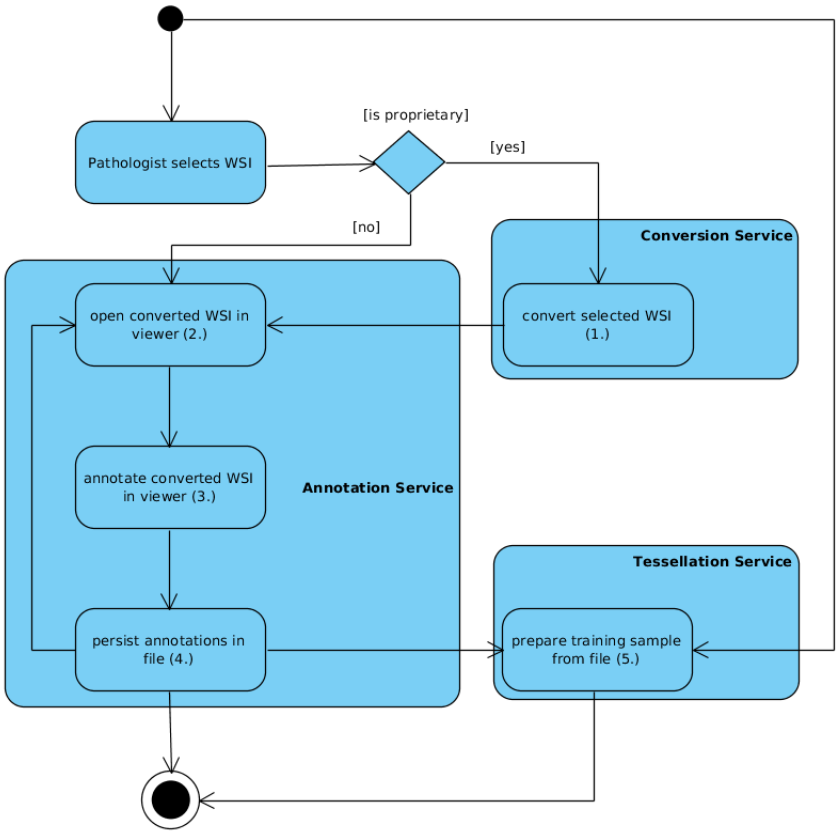


Figure 2.12: Activity diagram of the process chain

A tool will be introduced for each group in the subsections 2.4.1 - 2.4.3). Those are:

- **Conversion Service**
  This service will be responsible of the conversion from img_i^{wsi} to img_i^{cvrt} (1.).
- **Annotation Service**
  This service will offer a GUI to view a img_i^{cvrt}, as well as make and manage annotations (2. - 4.)
- **Tessellation Service**
  This service will be responsible for extracting a ts_i from a given A_i and img_i^{cvrt} (5.).

2.4.1 Conversion Service

The devices which create WSIs, so called *whole slide scanners*, create images in various formats, depending on the vendor system (due to the lack of standardization [11]). The Conversion Service (CS) has the goal of converting those formats to an open format[5] (see fig. 2.13).

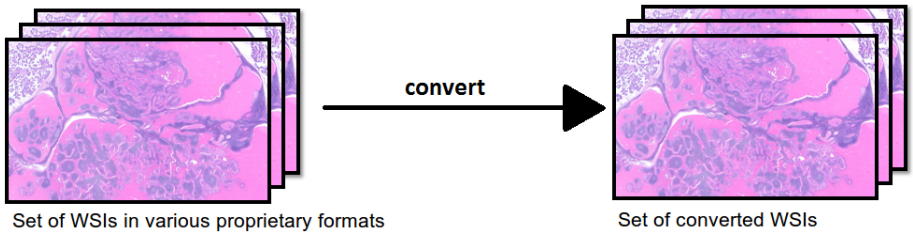


Figure 2.13: Visualization of the Conversion Service

Upon invocation, the CS will take every single WSI inside a given directory and convert it to a chosen open format. The output of each conversion will be saved in another specified folder. Valid image formats for conversion are: BIF, MRXS, NDPI, SCN, SVS, SVSLIDE, TIF, TIFF, VMS and VMU.

2.4.2 Annotation Service

As mentioned in 2.4, the Annotation Service (AS) will provide a graphical user interface to view a WSI, create annotations and manage those annotations. This also includes persisting made annotations in a file (see fig. 2.14).

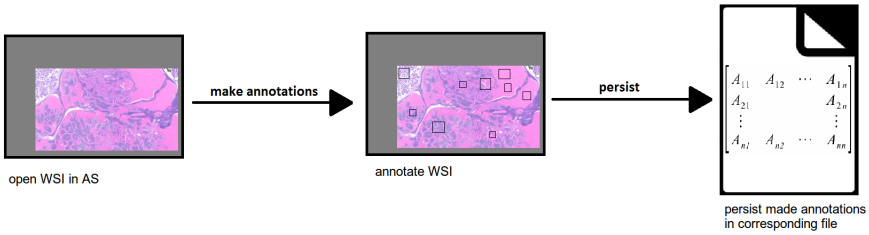


Figure 2.14: Visualization of the Annotation Service

The supplied GUI will offer different tools to help the user annotate the WSI, e.g. a ruler to measure the distance between two points. The annotations themselves will be made via drawing a contour around an object of interest and

[5] Compare subsections 2.1.2 and 2.1.3

putting a specified label on that region. To ensure uniformity of annotations, labels will not be added in free text. Instead they will be selected from a predefined dictionary.

2.4.3 Tessellation Service

The task of the Tessellation Service (TS) is to extract annotations and their corresponding image data to create a ground truth.

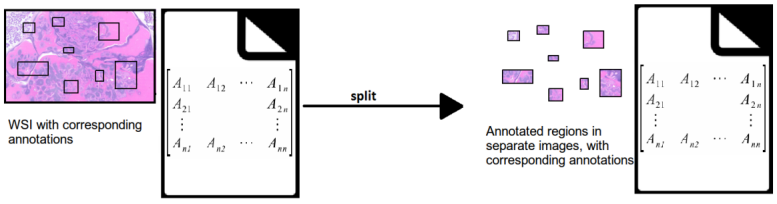


Figure 2.15: Visualization of the Tessellation Service

Let there be a WSI Img and a corresponding set of annotations A. The TS will achieve the extraction by iterating over every $a_i \in A$, creating a sub-image img_i which is the bounding box around the region described by a_i (see fig. 2.15). To be used as training sample, the TS must keep up the relationship between img_i and a_i.

Chapter 3

Conversion Service

3.1 Methodology

As stated in section 2.1, there is no standardized format for WSIs. Supplement 145 of the DICOM standard tries to unify the whole process around WSIs, but vendors still push their proprietary formats. For the reasons mentioned in subsection 2.1.3, it is necessary to establish a common format for all the WSIs which are subject to the process chain established in section 2.4. Therefore, the goal of the CS is to convert WSIs of proprietary formats into a common open format.

To make the conversion as convenient and fast as possible, the CS should only have brief user interaction. For this purpose it will not have a GUI. Instead the CS will be implemented as a console script. Furthermore, the CS should be capable of converting multiple WSIs after one another, so that no restart is necessary between conversions. Therefore, the CS will take an input directory as parameter and convert all WSIs of valid format inside that directory. Another parameter will be the output folder, in which the converted DZIs are stored.

vendor	**formats**
Aperio	SVS, TIF
Hamamatsu	VMS, VMU, NDPI
Leica	SCN
3DHistech/Mirax	MRXS
Philips	TIFF
Sakura	SVSLIDE
Trestle	TIF
Ventana	BIF, TIF

Table 3.1: File formats by vendor

Tab. 3.1 gives an overview of file formats, sorted by vendor, which are viable

as input for the conversion.

3.1.1 Selection of Image Format

A format or service must be chosen as conversion target for the CS. Choices have been established in 2.1.3. These are:

(1) BigTIFF

(2) DZI

(3) IFFF

(4) JPEG 2000

(5) TMS/OMS

To convert a WSI, a conversion tool is needed. Tab. 3.2 shows a listing of possibilities for that purpose. Listed are the name of the tool, the technology used and the output format. The table indicates, that DZI has a great variety of options, while the alternatives have little to none (Map Tiler for TMS, Kakadu for IFFF and none for the others).

tool	**description**	**output**
Deep Zoom Composer	dekstop app for Windows	DZI
Image Composite Editor	dekstop app for Windows	DZI
DeepZoomTools.dll	.NET library	DZI
deepzoom.py	python script	DZI
deepzoom	perl script	DZI
PHP Deep Zoom Tools	PHP script	DZI
Deepzoom	PHP script	DZI
DZT	Ruby library	DZI
MapTiler	desktop app for Windows, Mac, Linux	TMS
VIPS	command line tool, library for a number of languages	DZI
Sharp	Node.js script, uses VIPS	DZI
MagickSlicer	shell script	DZI
Gmap Uploader Tiler	C++ application	DZI
Node.js Deep Zoom Tools	Node.js script, under construction	DZI
OpenSeaDragon DZI Online Composer	Web app (and PERL, PHP scripts)	DZI

Zoomable	service, offers embeds; no explicit API	DZI
ZoomHub	service, under construction	DZI
Kakadu	C++ library	IIIF
PyramidIO	Java tool (command line and library)	DZI

Table 3.2: Overview of conversion options for zooming image formats (source: [45])

Since the CS should only consist of brief user interaction and be as automated as possible, desktop and web applications are not valid as tools for conversion. This excludes *Deep Zoom Composer*, *MapTiler*, *OpenSeaDragon DZI Online Composer* and *Zoomable* as possible choices (therefore also excluding (next to the reasons given in subsection 2.1.3), TMS as possible format).

One of the reasons not to use proprietary formats was the limitation to only certain operating systems, eliminating Windows-only tools. Those are *Image Composite Editor* and *DeepZoomTools.dll*.

Furthermore, reading the proprietary formats is a highly specialized task, eliminating most of the leftover choices: *deepzoom* [5], *DZT* [19], *sharp* [42], *MagickSlicer*, *Node.js Deep Zoom Tools* (both use ImageMagick to read images, which does not support any of the proprietary WSI formats [47]), *Gmap Uploader Tiler* [65], Zoomhub [44] and *PyramidIO* [39].

Kakadu can only encode and decode JPEG 2000 images [45], making it no valid choice either.

This leaves *deepzoom.py* and *VIPS*, both creating DZI as output. Through the use of OpenSlide, they are both capable of reading all proprietary formats stated in tab. 3.1 [67].

3.1.2 Deepzoom.py

Deepzoom.py[1] is a python script and part of Open Zoom[2]. It can either be called directly over a terminal or imported as a module in another python script. The conversion procedure itself is analogous for both methods.

If run in a terminal the call looks like the following:

```
1  $ python deepzoom.py [options] [input file]
```

The various options and their default values can be seen in table 3.3. If called without a designated output destination, deepzoom.py will save the converted DZI in the same directory as the input file.

[1] See `https://github.com/openzoom/deepzoom.py` for further details
[2] See `https://github.com/openzoom` for further details

option	description	default
-h	show help dialog	-
-d	output destination	-
-s	size of the tiles in pixels	254
-f	image format of the tiles	jpg
-o	overlap of the tiles in pixels (0 - 10)	1
-q	quality of the output image (0.0 - 1.0)	0.8
-r	type of resize filter	antialias

Table 3.3: Options for deepzoom.py

The resize filter is applied to interpolate the pixels of the image when changing its size for the different levels. Supported filters are:

- cubic
- biliniear
- bicubic
- nearest
- antialias

When used as module in another python script, deepzoom.py can simply be imported via the usual *import* command. To actually use deepzoom.py, a Deep Zoom Image Creator needs to be created. This class will manage the conversion process:

```
# Create Deep Zoom Image Creator
creator = deepzoom.ImageCreator(tile_size=[size],
  tile_overlap=[overlap], tile_format=[format],
  image_quality=[quality], resize_filter=[filter])
```

The options are analogous with the terminal version (compare tab. 3.3). To start the conversion process, the following call must be made within the python script:

```
# Create Deep Zoom image pyramid from source
creator.create([source], [destination])
```

In the proposed workflow, the ImageCreator opens the input image img^{wsi} and accesses the information necessary to create the describing XML file for the DZI (compare subsection 2.1.3). The needed number of levels is calculated next. For this, the bigger value of height or width of img^{wsi} is chosen (see eq. 3.1) and then used to determine the number of levels lvl^{max} (see eq. 3.2) necessary.

$$max\_dim = max(height, width) \tag{3.1}$$

$$lvl^{max} = \lceil log_2(max\_dim) + 1 \rceil \tag{3.2}$$

Once lvl^{max} has been determined, a resized version img_i^{dzi} of img^{wsi} will be created for every level $i \in [0, lvl - 1]$. The quality of img_i^{dzi} will be reduced according to the value specified for -q/image_quality (see tab. 3.3). The resolution of img_i^{dzi} will be calculated with the *scale* function (see eq. 3.3) for both, height and width. Furthermore, the image will be interpolated with the specified filter (-r/resize_filter parameter, see tab. 3.3).

$$scale = \lceil dim * 0.5^{lvl^{max} - i} \rceil \tag{3.3}$$

Once img_i^{dzi} has been created, it will be tessellated into as many tiles of the specified size (-s/tile_size parameter, see tab. 3.3) and overlap (-o/tile_overlap parameter, see tab. 3.3) as possible. If the size of img^{wsi} in either dimension is not a multiple of the tile size, the last row/column of tiles will be smaller by the amount of ($tile\ size - ([height \text{ or } width] \mod tile\ size)$) pixels.

Every tile will be saved as [column]_[row].[format] (depending on the -f/file_format parameter, see tab. 3.3) in a directory named according to the corresponding level i. Each one of those level directories will be contained within a directory called [filename]_files. The describing XML file will be persisted as [filename].dzi in the same directory as [filename]_files.

3.1.3 VIPS

VIPS (VASARI Image Processing System) is described as "[...] a free image processing system [...]" [13]. It includes a wide range of different image processing tools, such as various filters, histograms, geometric transformations and color processing algorithms. It also supports various scientific image formats, especially from the histopathological sector [13]. One of the strongest traits of VIPS is its speed and little data usage compared to other imaging libraries [52].

VIPS consists of two parts: the actual library (called libvips) and a GUI (called nip2). libvips offers interfaces for C, C++, python and the command line. The GUI will not be further discussed, since it is of no interest for the implementation of the CS.

VIPS speed and little data usage is achieved by the usage of a fully demand-driven image input/output system. While conventional imaging libraries queue their operations and go through them sequentially, VIPS awaits a final write command, before actually manipulating the image. All the queued operations will then be evaluated and merged into a few single operations, requiring no additional disc space for intermediates and no unnecessary disc in- and output. Furthermore, if more than one CPU is available, VIPS will automatically evaluate the operations in parallel [51].

As mentioned before, VIPS has a command line and python interface. In either case, a function called *dzsave* will manage the conversion from a WSI to a DZI. A call in the terminal looks as follows:

```
1 $ vips dzsave [input] [output] [options]
```

When called, VIPS will take the image [input], convert it into a DZI and then save it to [output]. The various options and their default values can be seen in tab. 3.4.

option	**description**	**default**
layout	directory layout (allowed: dz, google, zoomify)	dz
overlap	tile overlap in pixels	1
centre	center image in tile	false
depth	pyramid depth	onepixel
angle	rotate image during save	d0
container	pyramid container type	fs
properties	write a properties file to the output directory	false
strip	strip all metadata from image	false

Table 3.4: Options for VIPS

A call in python has the same parameters and default values. It looks like this:

```
1 image = Vips.Image.new_from_file(input)
2 image.dzsave(output[, options])
```

In line 1 the image gets opened and saved into a local variable called *image*. While being opened, further operations on the image could be done. The command in line 2 writes the processed image as DZI into the specified output location.

3.2 Implementation

The first iteration of the CS was a python script using deepzoom.py for the conversion. This caused severe performance issues. Out of all the image files in the test set (see section 3.3), only *CMU-3.svs* (from Aperio, see appendix A.2.1) could be converted. Other files were either too big, so the process would eventually be killed by the operating system, or exited with an IOError concerning the input file from the PIL imaging library.

The second iteration uses VIPS python implementation, which is capable of converting all the given test images[3].

The script has to be called inside a terminal in the following fashion:

```
1 $ python ConversionService.py [input dir] [output dir]
```

Both the input and the output directory parameter are mandatory, in order for the script to know where to look for images to convert and where to save the resulting DZIs.

Upon calling, the *main()* routine will be started, which orchestrates the whole conversion process. The source code is as follows:

[3] The CS can be found on the disc at the end of this thesis, see appendix A.1.

```
def main():
  path = checkParams()
  files = os.listdir(path)
  for file in files:
    print("----------------------------------------------------")
    extLen = getFileExt(file)
    if(extLen != 0):
      print("converting " + file + "...")
      convert(path, file, extLen)
      print("done!")
```

checkParams() checks if the input parameters are valid and, if so, returns the path to the specified folder or aborts the execution otherwise. Furthermore, it will create the specified output folder, if it does not exist already. In the next step, the specified input folder will be checked for its content. *getFileExt(file)* looks up the extension of each contained file and will either return the length of the files extension or 0 otherwise. Each valid file will then be converted with the *convert(...)* function:

```
# convert image source into .dzi format and copies all header
# information into [img]_files dir as metadata.txt
# param path: directory of param file
# param file: file to be converted
# param extLen: length of file extension
def convert(path, file, extLen):
  dzi = OUTPUT + file[:extLen] + "dzi"
  im = Vips.Image.new_from_file(path + file)
  # get image header and save to metadata file
  im.dzsave(dzi, overlap=OVERLAP, tile_size=TILESIZE)
  # create file for header
  headerOutput = OUTPUT + file[:extLen-1] + "_files/metadata.txt"
  bashCommand = "touch " + headerOutput
  call(bashCommand.split())
  # get header information
  bashCommand = "vipsheader -a " + path + file
  p = subprocess.Popen(bashCommand.split(), stdout=subprocess.PIPE,
      stderr=subprocess.PIPE)
  out, err = p.communicate()
  # write header information to file
  text_file = open(headerOutput, "w")
  text_file.write(out)
  text_file.close()
```

The name for the new DZI file will be created from the original file name, however, the former extension will be replaced by "dzi" (see line 7). *OUTPUT* specifies the output directory which the file will be saved to. Next, the image file will be opened with Vips' Image class. Afterwards, *dzsave(...)* will be called, which handles the actual conversion into the dzi file format. *OVERLAP* and *TILESIZE* are global variables which describe the overlap of the tiles and their respective size. Their default values are 0 (*OVERLAP*) and 256 (*TILESIZE*). The output will be saved to the current working directory of ConversionService.py, appending "/dzi/[*OUTPUT*]/".

When a WSI gets converted into DZI by the CS, most of the image header

information is lost. To counteract this, a file *metadata.txt* is created in the [name]_files directory, which serves as container for the header information of the original WSI (see line 12 and 13).

The console command *vipsheader -a* is responsible for extracting the header information (see line 17 - 18). The read information (*out* in line 18) is then written into the *metadata.txt* file (Line 20 - 22).

3.3 Test

To test the correct functionality of the CS a test data set was needed. OpenSlide offers a selection of freely distributable WSI files[4], which can be used for that purpose.

Because of the size of each single WSI file, only 2 are included on the disc at the end of this thesis (see appendix A.1). The others must be downloaded separately from the OpenSlide homepage. For a complete listing of the used test data see appendix A.2.

3.3.1 Setup

To create a controlled environment for the test, a new directory will be created, called *CS_test*. A copy of ConversionService.py as well as a directory containing all the test WSIs (called *input*) will be placed in that directory.

Input contains the following slides:

(1) CMU-2 (Aperio, .svs)

(2) CMU-1 (Generic Tiled tiff, .tiff)

(3) OS-3 (Hamamatsu, .ndpi)

(4) CMU-2 (Hamamatsu, .vms)

(5) Leica-2 (Leica, .scn)

(6) Mirax2.2-3 (Mirax, .mrxs)

(7) CMU-2 (Trestle, .tif)

(8) OS-2 (Ventana, .bif)

Because of their structure, (4), (6) and (7) will be placed in directories titled with their file extension. Fig. 3.1 shows the content of the input folder.

[4] See `http://openslide.cs.cmu.edu/download/openslide-testdata/` for the test data.

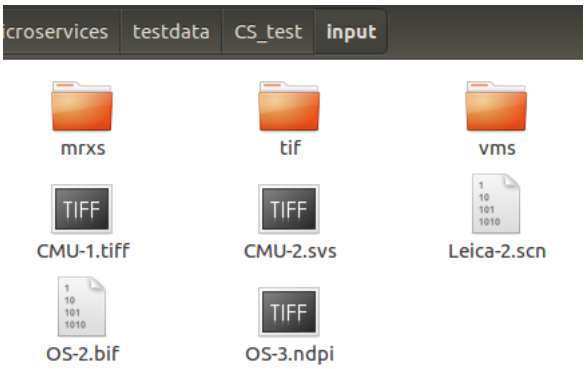


Figure 3.1: Content of input directory

This makes multiple calls of the CS necessary. The calls, in that order, are:

```
$ python ConversionService.py input/ out_1/
$ python ConversionService.py input/mrxs out_2/
$ python ConversionService.py input/tif out_3/
$ python ConversionService.py input/vms out_4/
```

3.3.2 Result

All runs of ConversionService.py were successful. Tab. 3.5 shows an overview of the results:

input	**output**	**time (sec)**
input/	CMU-1.dzi, CMU-2.dzi, Leica-2.dzi, OS-2.dzi, OS-3.dzi	1992
input/mrxs/	Mirax2.2-3.dzi	500
input/tif/	CMU-2.dzi	56
input/vms/	CMU-2-40x - 2010-01-12 13.38.58.dzi	305

Table 3.5: Results of Conversion Service Test

The vast difference in file size of the test data accounts for the different run times of the tests. While the first test converted 5 WSIs (399 sec/WSI), every other test converted a single one. The conversion of (6) was much faster, since the file was smaller in size (304.22 MB) compared to the others (1495.24 MB on average).

Chapter 4

Annotation Service

4.1 Objective of the Annotation Service

As described in 2.4.2, the goal of the AS is to provide a user with the possibility to:

(1) view a WSI

(2) annotate a WSI

(3) manage previously made annotations

In order to achieve objective (1) - (3), a GUI needs to be deployed which supports the user in working on those tasks. (3) also adds the need for file persistence management.

During the development of the AS it became clear that the support of DZI as the only image format was impractical for the real life environment, thus making it necessary to support proprietary formats as well. A solution has to be found, that still addresses the vendor and platform issues stated in 1.2 and 2.1.3.

4.2 Methodology

As stated in 2.1.3, most vendors ship their own implementation of an image viewer tailored to their proprietary image format, thus creating a vendor lock-in. Furthermore most of the vendors only support Windows as a platform, ignoring other operating systems [11], [16], [30]. To avoid vendor- or platform lock-in, a solution must be found that is independent of operating system and vendor.

Independence from an operating system can be achieved by using web technologies, especially when running an application in a web browser, since those are supported by all modern operating systems and even mobile platforms [23].

By choosing the web as target environment for the AS, the service itself becomes subject to security considerations, namely *cross-origin resource sharing* (CORS) [59] and the *same-origin policy* (SOP) [63]. The SOP is a security concept of the web application security model, that only allows direct file access if the parent directory of the originating file is an ancestor directory of the target file [63]. Since the local WSI file will not have the same origin, CORS is needed. CORS is a standard that defines mechanisms to allow access to restricted resources from a domain outside of the origin, when using the HTTP protocol [59]. Since the WSI is a local file, HTTP can not be used to retrieve the file.

The restrictions of SOP and CORS can be worked around by deploying a server as a so called *digital slide repository* (DSR). A DSR manages storage of WSIs and their metadata [11]. This way, WSIs would share the same origin as the viewer and their retrieval would be possible.

Using a DSR has additional advantages:

- WSIs are medical images and as such confidential information. Their access is usually tied to non-disclosure or confidentiality agreements (e.g. [66] or [75]). A DSR eliminates the need to hand out copies of WSIs, which makes it easier to uphold the mentioned agreements.

- WSIs take up big portions of storage [33]. The local systems used by pathologists in the environment of the AS are usual desktop computers and laptops. As such, their storage might be insufficient to hold data in those quantities. A DSR can be set up as a dedicated file server, equipped for the purpose of offering large amounts of storage.

- A DSR enables centralized file management. Pathologists don't access their local version of a WSI and it's annotations, but share the same data pool.

- Depending on the network setup, other advantages become possible, e.g. sharing of rare cases as educational material and teleconsultation of experts independent of their physical position [22].

Chapter 3 established a service to convert WSIs of various, proprietary formats to DZI, addressing the need to implement multiple image format drivers. But, as stated in 4.1, a solution to serve proprietary image formats without explicit conversion is needed as well.

OpenSlides Python provides a DZI wrapper. This wrapper can be used to wrap a proprietary WSI and treat it as a DZI [67]. A DSR can use this to serve a proprietary WSI as DZI to a viewer.

For the reasons mentioned above, the AS will be implemented as a web application. To do so, it will be split into 2 parts: a DSR and a viewer.

4.3 Parts of the Annotation Service

As described in section 4.2, the AS will be realized in 2 separate parts:

- a DSR, called *Annotation Service Server* (ASS) (see subsection 4.3.1)
- a viewer, called *Annotation Service Viewer* (ASV) (see subsection 4.3.2)

The ASS will be responsible for data management, supplying image data and serving the ASV to the client. The ASV will provide a WSI viewer with the tools needed to annotate ROIs in a WSI.

The two components interact as follows: once the client requested a valid image URL, the ASS will check if the requested WSI is a DZI and, if so, render a ASV with the image path, the image's microns per pixel (MPP) and file name. If the WSI is proprietary, it will be wrapped by OpenSlide. The remaining procedure is then identical to the DZI case.

The ASV is served to the client as a web application that requests the data necessary to view the WSI and it's annotations. This includes configurations, previously made annotations (if present), label dictionaries and the image tiles for the current view. Once loaded, the client can change the current view to maneuver through the different levels and image tiles available, which will be requested by the ASS whenever needed. Annotations can be made and persisted at any time.

Fig. 4.1 visualizes the described interaction process in an activity diagram.

4.3.1 Annotation Service Server

As described in section 4.2, the ASS serves as a DZR. As such it is responsible for the storage of WSI files and their related metadata [11]. Additional data managed by the ASS will be:

- annotation data
- the ASV's configuration data
- dictionary data

Communication with the ASS directly is only necessary to request the rendering of an ASV with a WSI. Once the ASV is rendered, communication can be handled through shortcuts in the ASV.

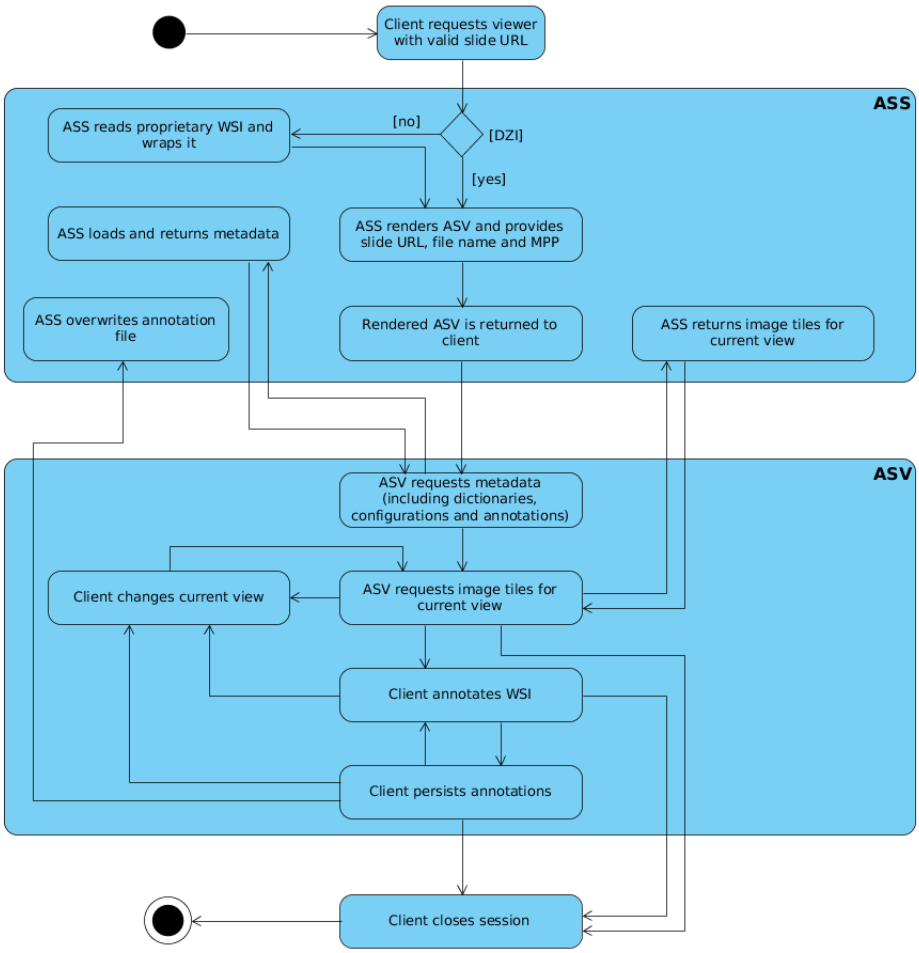


Figure 4.1: Activity diagram of ASS and ASV

Communication between ASS and client will be realized over a *Representational State Transfer* (REST) API offered by the ASS. REST is an architectural style for developing web applications. It was established in 2000 by Fielding in [37]. A system that complies to the constraints of REST can be called RESTful. Typically, RESTful systems communicate via the HTTP protocol [37].

The development of a fully functional web server is not within the scope of this thesis. Therefore, the ASS is intended to run as a local web server. This works around many of the common issues when hosting a web server (e.g. inefficient caching, load balance issues, gateway issues, poor security design, connectivity issues) [3].

4.3.2 Annotation Service Viewer

The ASV is developed to provide a WSI viewer with annotation capabilities. It serves as the main component for interaction with the AS, realizing most of the communication with the ASS (compare fig. 4.1).

The annotation capabilities look as follows:

- Annotations will be represented by so called *regions*. A region is defined by a path enclosing the ROI. This path can be drawn directly onto the WSI. The drawing is done either in *free hand* or *polygon* mode. When drawing free hand, the path will follow the mouse cursor along its way as long as the drawing mode is activated. In polygon mode, segments can be placed, which are connected with a path in the order that they were placed in.
- Each region has an associated *label* that describes the region. A label is a keyword, predefined by a *dictionary*. A dictionary contains a list of keywords that are available as labels.
- New, empty label dictionaries can be created.
- New labels can be added to existing dictionaries.
- Each region has a *context* trait. This trait lists all other regions that
  - touch
  - cross
  - surround
  - are surrounded by

  the region (see fig. 4.2).
- A *point of interest* (POI) is another way to create a region. After selecting a POI (a point coordinate in the image), an external script will start a segmentation and return the image coordinates for an enclosing path to the ASV. The ASV will then automatically create a region based

on the provided information. Since segmentation approaches differ drastically between cases and scenarios [61] (e.g. [25], [26], [36] or [53] for cell segmentation alone), it exceeds the scope of this work by far. To prove basic functionality a dummy implementation will be delivered. The script will be an interchangeable python plug in.

- For annotation support, a distance measurement tool is provided. This tool can measure the distance between 2 pixels in a straight line. The measurement will be realized by the euclidean distance between a pixel p_a and p_b [74].

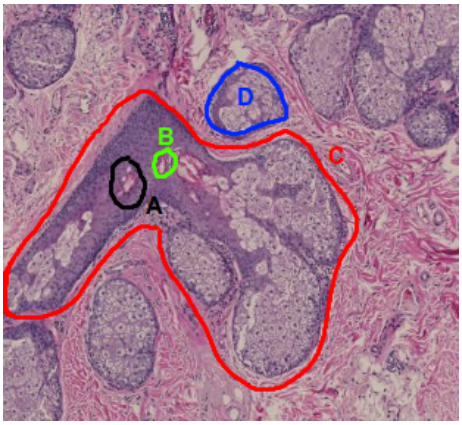


Figure 4.2: Example of context regions (B, C are context of A; A, C are context of B; A, B are context of C; D has no context region)

The ASV uses keywords from a dictionary to label regions. While a free text approach is more flexible and easier to handle for novice users, it encounters difficulties in a professional metadata environment (such as histopathological image annotation). A dictionary-based approach facilitates interoperability between different persons and annotation precision [29]. To increase flexibility, the ASV will offer the possibility of adding new entries to existing dictionaries.

Since the vocabulary may vary strongly between different studies, the ASV offers the possibility to create new dictionaries. This way, dictionaries can be filled with a few case-relevant keywords instead of many generic, mostly irrelevant ones. This explicitly does not exclude the use of a generic dictionary if it should serve a broad series of cases.

The first iteration of the ASV will be based on an open source project called *MicroDraw*[1] (see fig. 4.3 for MicroDraw's GUI).

[1] See `https://github.com/r03ert0/microdraw` for more information on the MicroDraw project

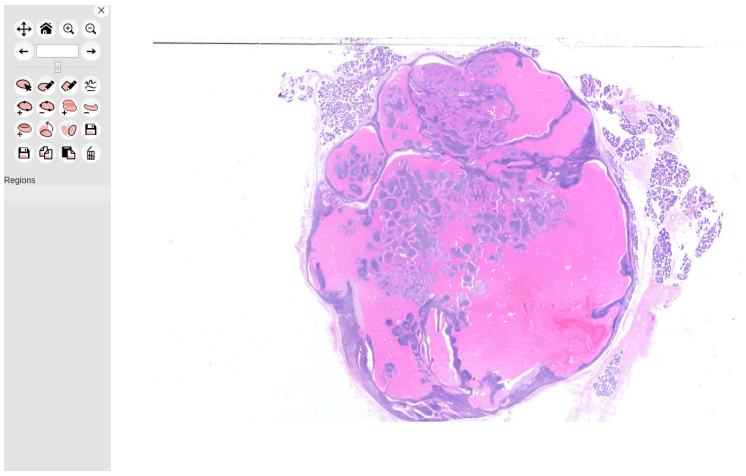


Figure 4.3: Microdraw GUI with opened WSI

MicroDraw is a web application to view and annotate *"high resolution histology data"* [4]. The visualization is based on *OpenSeadragon* (OSD)[2], another open source project. Annotations are made possible by the use of *Paper.js*[3]. This delivers a baseline for the capabilities stated earlier in this section.

Each iteration of the ASV will be reviewed regarding its usability and functionality by a pathologist, thus adjusting it to its real life environment with each iteration.

4.4 Annotation Service Server Implementation

The ASS is a local server, implemented in python (***as_server.py***). It offers a RESTful styled API for communication (see subsection 4.4.4). To improve functionality, the following frameworks were used:

- Flask (see subsection 4.4.1)
- OpenSlide Python (see subsection 4.4.2)

All code snippets in the following subsections have been taken from ***as_server.py***. A detailed documentation of the individual functions can be found in appendix B.1.

4.4.1 Flask

To give ASS its server capabilities, Flask was used[4]. It provides a built-in development server, integrated unit testing, RESTful request dispatching and is

[2] See `https://openseadragon.github.io/` for more information on OSD.
[3] See `http://paperjs.org/` for more information on Paper.js.
[4] See `http://flask.pocoo.org/` for additional information on Flask.

Web Server Gateway Interface (WSGI) compliant [38]. The WSGI is a standard interface for the communication between web servers and web applications or frameworks in python. The interface has a server and application side. The server side invokes a callable object that is provided by the application side. The specifics of providing this object are up to the individual server [6].

Flask's so called *route() decorator* provides a simple way to build a RESTful API for server client communication:

```
1  @app.route('/loadJson')
2  def loadJson():
3    ...
4
5  @app.route('/createDictionary')
6  def createDictionary():
7    ...
8
9  @app.route('/getDictionaries')
10 def getDictionaries():
11   ...
12
13 @app.route("/runSegmentation")
14 def runSegmentation():
15   ...
```

Decorating a function with `@app.route([URL])` will bind it to the supplied URL. When the client requests that bound URL, the server will call the decorated function [38]. The code snippet above shows exemplary how to use decorators.

A bound URL can also contain variable sections, which are marked as <*variable name*>. Optionally, a converter can by used to only accept variables of a certain type. This becomes possible by specifying the converter in front of the variable: <*converter:variable name*> [38]. The following code snippet shows possible examples for URLs with variables (see tab. 4.1 for a list of available converters):

```
1  @app.route('/wsi/<path:file_path>.dzi')
2  def index_dzi(file_path):
3    ...
4
5  @app.route('/wsi/<path:file_path>')
6  def index_wsi(file_path):
7    ...
8
9  @app.route('/<slug>.dzi')
10 def dzi(slug):
11   ...
```

To bind a URL with one or more variables, the corresponding function must be parameterized with the same variables (compare line 1 and 2 or 13 - 14 and 15) [38].

name	accepted input
string	any text without a slash (default)
int	integer values
float	floating point values
path	like string, but also accepts slashes
any	matches one of the items provided
uuid	UUID strings

Table 4.1: Available converters in Flask (source: [38])

HTTP provides different methods for accessing URLs (such as GET or POST) [73]]. By default Flask will answer only GET requests. Any other method is answered with a "405 Method not allowed" HTTP status code [38]. This can be changed by adding the *methods* argument to the decorator:

```
@app.route('/saveJson', methods=['POST'])
def saveJson():
  ...
```

Tab. 4.2 states a number of URLs, that have been bound to a corresponding function. For a detailed documentation of the individual functions, consult appendix B.1.

URL	function
/wsi/<path:file_path>.dzi	`index_dzi(file_path)`
/wsi/<path:file_path>	`index_wsi(file_path)`
/<slug>.dzi	`dzi(slug)`
/<slug>_files/<int:level>_<int:col>_<int:row>.<format>	`tile(slug, level, col, row, format)`
/saveJson	`saveJson()`
/loadJson	`loadJson()`
/createDictionary	`createDictionary()`
/static/dictionaries/<dictionary>	`loadDictionary(dictionary)`
/getDictionaries	`getDictionaries()`
/switchDictionary	`switchDictionary()`
/runSementation	`runSegmentation()`

Table 4.2: ASS' URL-function binding overview

4.4.2 OpenSlide Python

To read a WSI, ASS uses OpenSlide Python, which is a python interface to the OpenSlide C library. Besides providing an interface to read a WSI, it offers a DZI wrapper [67], called *DeepZoomGenerator* (DZG). The DZG can be used

to create Deep Zoom tiles on demand. The following formats are supported by OpenSlide [67]: BIF, NDPI, MRXS, SCN, SVS, SVSLIDE, TIF, TIFF, VMS and VMU. This list is identical to the list of image formats supported by the CS (compare chapter 3). This is due to the fact that *VIPS* uses OpenSlide to read WSIs as well [13].

OpenSlide can read a proprietary WSI as a so called *OpenSlide* object (see line 4):

```
1 from openslide import open_slide
2 from openslide.deepzoom import DeepZoomGenerator
3
4 slide = open_slide(slide path)
5 dzg = DeepZoomGenerator(slide[, tile_size, overlap, limit_bounds])
```

An OpenSlide object offers methods to access available metadata, image tiles, the thumbnail image and associated images, if available. The OpenSlide object can be wrapped with a DZG to enable DZI support (see line 5) [67]. A number of optional parameters can be passed into the constructor as well (see tab 4.3 for parameters and their default values).

parameter	**type**	**description (default value)**
osr	OpenSlide, ImageSlide	the slide object (mandatory)
tile_size	integer	the width and height of a single tile (254)
overlap	integer	the number of extra pixels to add to each interior edge of a tile (1)
limit_bounds	boolean	true to render only the non-empty slide region (false)

Table 4.3: DZG parameters (source: [67])

The DZG[5] generates all data necessary, to work with a proprietary WSI as if it would be a DZI [67]. The DZG's `get_dzi(format)` and `get_tile(level, address)` functions are of special importance. `get_dzi(format)` generates a string containing the complete metadata of a DZI XML file (compare subsection 2.1.3 - Deep Zoom Images). The parameter *format* specifies the format (PNG or JPEG) of the individual Deep Zoom tiles. `get_tile(level, address)` returns an image of the tile corresponding to the provided parameter values (see tab. 4.4). The tiles are returned either as PNG or JPEG, depending on which format was chosen for `get_dzi(format)`,

[5] For an in-depth list of functions, see `http://openslide.org/api/python/`

name	type	description
level	integer	the DZI level to get the tile from
address	tuple	the address of the tile within the level as a (column, row) tuple

Table 4.4: Description of `get_tile(level, address)` parameters (source: [67])

The use of the DZG enables ASS to create DZI metadata and Deep Zoom tiles from proprietary WSI files on demand.

4.4.3 Structure and Setup

Flask requires a specific directory structure to access static files and HTML templates. Static files are served from a *static/* directory, render-able view templates must be stored in a *template* directory. Both of those directories must be at the root level of the ASS [38]. Because of this, the following directory and file structure was chosen:

- AnnotationService/
  - static/
    - css/
    - cursors/
    - dictionaries/
    - img/
    - lib/
    - segmentation/
    - wsi/
    - configuration.json
    - slideDictionaries.json
  - templates/
    - as_viewer.html
  - as_server.py

As mentioned before, **static/** contains all static files required by the ASS, e.g. the CSS files (**css/**), icons (**cursors/**, **img/**) and JavaScript files (**lib/**) needed by the ASV.

To make a WSI accessible to the AS, it must be placed in **wsi/**. From there, the file path is arbitrary. **wsi/** contains also the persisted annotations of the associated WSIs. A WSI file will have a save file for each dictionary used on it.

dictionaries/ contains all dictionaries used by the ASV. **slideDictionaries.json** is a key-value map containing all WSIs opened (none when the AS is newly set up) and the latest dictionaries used for them. The file name of the

WSI is the key, which maps to a dictionary name as the value. When a new WSI is opened for the first time, it will be opened with the dictionary defined in the configuration file **configuration.json**.

configuration.json is the configuration file of the AS. It contains values such as the default dictionary and the default alpha value for regions. All those configurations can be adjusted manually. Tab. 4.5 lists all configurable values.

parameter	**description**	**standard**
defaultFillAlpha	Default alpha value for regions	0.5
defaultStrokeColor	Default color for region strokes	black
defaultStrokeWidth	Default width for region strokes	1
dictionary	Name of default dictionary (whenever a new WSI is opened in the ASV, it will be opened with this dictionary)	example.json
hideToolbar	Hide toolbar when ASV is rendered	false
segmentationScript	Script used for segmentation of POIs	opencv.py

Table 4.5: Configurable parameters in configuration.json

To provide a segmentation script, it must be placed in **segmentation/**. Additionally, the *"segmentationScript"* value in the configuration.json file must be adjusted (see tab. 4.5).

templates/ contains all HTML templates that must be rendered by the ASS. The AS contains only one HTML template, which is used to render the ASV: **as_viewer.html**.

as_server.py is the python script that realizes the ASS. It can be started from a terminal through the use of a python interpreter:

```
$ python as_server.py
```

Alternatively, python's -m switch can be used:

```
$ export FLASK_APP=as_server.py
$ python python –m flask run
```

When started, the server will listen on 127.0.0.1:5000 by default. Another port (-p, –port) or IP address (-l, –listen) can be specified when starting the ASS. Additionally, DZG-related parameters can be changed (see tab. 4.6).

parameter	description	default
-B, –ignore-bounds	render only the non-empty slide region	false
-e, –overlap	set overlap between adjacent tiles in pixels	0
-f, –format	set tile format (PNG or JPEG)	JPEG
-l, –listen	set IP address to listen to	127.0.0.1
-p, –port	set port to listen to	5000
-Q, –quality	set JPEG compression quality in %	100
-s, –size	set tile size	256

Table 4.6: Parameters for as_server.py

To retrieve a WSI from the ASS, the URL must be pointed to it in the following manner: *http://[host]:[port]/wsi/[file path]* (see fig. 4.4 for an example).

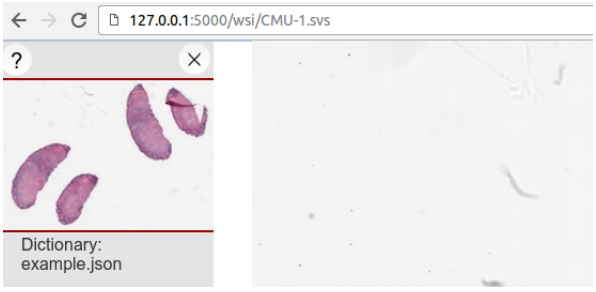


Figure 4.4: Example URL to retrieve WSI (`http://127.0.0.1:5000/wsi/CMU-1.svs`) ; WSI source: OpenSlides freely distributable test data (see appendix A.2.1)

4.4.4 RESTful API

The ASS provides a RESTful API. This was realized with Flasks route() decorators (compare subsection 4.4.1). Tab. 4.2 shows a list of corresponding functions. The listing below gives an overview of the URLs provided by the API:

(1) - **/wsi/[file path].dzi**
method: GET
Requests an ASV to view the requested DZI. [file path] must point to a valid DZI, otherwise a "404 Not Found" HTTP status code is returned.
Returns either a rendered ASV, which requests the DZI at [file path] or "404 Not Found"

(2) - **/wsi/[file path]**
method: GET

Works similar to (1), except that the requested image is a proprietary WSI instead of a DZI. Thus, a DZG is created to wrap the slide.
Returns a specifically generated URL (*[slug].dzi*).

(3) - **/[slug].dzi**
method: GET
Requests the DZI metadata of [slug] from the DZG. A corresponding DZG will only exist, if (2) was called beforehand. Never call this function manually, to ensure a safe execution of the related commands. The ASV will call this function automatically.
Returns the DZI metadata generated by the DZG or "404 Not Found" if no corresponding DZG was found.

(4) - **/[slug]_files/[level]_[col]_[row].[format]**
method: GET
Requests the Deep Zoom tile [col]_[row].[format] in [level] from the DZG. As with (3), do not call this function manually. The ASV will call it automatically to retrieve the image tiles needed.
Returns the image tile in the specified level at the requested position or "404 Not Found" if no image tile could be generated.

(5) - **/saveJson**
method: POST
Post request that saves the provided JSON data in the provided JSON file. Creates a new one, if it does not exist. The posted data must look like the following:

```
{
  "file":"[file name]",
  "content":"[json content]"
}

```

(6) - **/loadJson?src=[source]**
method: GET
Loads the JSON file specified in [source].
Returns the JSON data if a file was found, an empty JSON map ("[]") otherwise.

(7) - **/createDictionary?name=[name]&slide=[slide]**
method: GET
Requests the creation of a new, empty dictionary. The file will be called [name]. The ASV adds a ".json" if necessary. A manual call of this functions makes an added ".json" necessary. Otherwise, a text file will be created. The slide parameter provides the name of the WSI the dictionary was created at, so it can be changed in the slideDictionaries.json file.
Returns the name and path to the new dictionary as json:

```
{
  "name":"[name]",
  "path":"[dictionary path]"
}
```

or "error" if [name] is already taken by another dictionary.

(8) - **/static/dictionaries/[dictionary]**
method: GET
Loads the dictionary file located at [dictionary].
Returns a stringified version of the dictionary content if a dictionary was found at [dictionary], a "404 Not Found" HTTP status code otherwise.

(9) - **/getDictionaries**
method: GET
Requests a list of files contained in the *dictionaries* folder.
Returns a list with the file names, or "-1" if no dictionaries are present.

(10) - **/switchDictionary?name=[name]&slide=[slide]**
method: GET
Changes the key-value entry in the slideDictionaries.json file whose key matches the provided [slide] parameter. The value will be adjusted to [name].
Returns a "200 OK" HTTP status code.

(11) - **/runSegmentation?x=[x]&y=[y]**
method: GET
Requests the invocation of the segmentation script provided in the configuration file. The coordinates of the POI are passed as URL arguments and will be handed down to the segmentation script.
Returns a list of 2D coordinates, describing the contour around the POI, or a "404 Not Found" HTTP status code if no script was provided or the provided script was not found.

4.5 Annotation Service Viewer Implementation

The ASV is a browser application based on the MicroDraw open source project. It provides a client with a WSI viewer with annotation capabilities (see subsection 4.3.2). It is implemented using JavaScript, HTML5 and CSS. The following frameworks were used to add additional functionality:

- jQuery
- OSD
- Paper.js

The ASV consists of the following files:

- *as_viewer.html* (template/)
- *as_viewer.js* (static/lib/)
- *as_viewer.css* (static/css/)
- *style.css* (static/css/)

A documentation of the individual functions of the ASV JavaScript (as_server.js) can be found in appendix B.2.

4.5.1 Frameworks

The ASV uses jQuery, OSD and Paper.js for additional functionality.

jQuery is a common JavaScript library, that offers an API to handle HTML document traversal and manipulation, event handling, animation and Asynchronous JavaScript and XML (AJAX) requests [58]. AJAX is a group of technologies which enable a client to make asynchronous web requests [9]. jQuery is also supported by all common web browsers [57]. The ASV uses jQuery especially for its HTML document traversal and manipulation capabilities, as well as its AJAX support.

See the corresponding subsections for OSD and Paper.js.

OpenSeadragon

OSD is used by the ASV to show a WSI. It is a JavaScript based, open source web application to serve a viewer for *"high-resolution zoomable images"* [45]. It supports the following image formats:

- DZI
- IIIF
- OSM
- TMS

Furthermore, custom tile sources can be added to support other image formats as well. Since the ASS is capable of delivering every WSI as DZI, no custom tile source implementation is necessary for proprietary WSIs.

The ASV defines a placeholder block element in its HTML file, which will be used to hold the OSD viewer (OSDV):

```
1 <!-- OpenSeadragon viewer -->
2 <div id="openseadragon1" style="width:vh;height:hh"></div>
```

as_viewer.html

The OSDV is then created in the ASV's JavaScript file:

```
1  // create image viewer
2  viewer = OpenSeadragon({
3    id: "openseadragon1",
4    prefixUrl: staticPath + "/lib/openseadragon/images/",
5    showReferenceStrip: false,
6    showNavigator: true,
7    sequenceMode: false,
8    navigatorId:"myNavigator",
9    zoomPerClick: 1,
10 });
```

as_viewer.js

The relation between the element in the webpage and OSDV is created through the *id* parameter (line 3). A description of the other used parameters of the OpenSeadragon constructor is provided in Tab. 4.7 [6].

The OSDV provides a function called `zoomPerClick` that enhances the image section under the cursor when clicked. Since drawing paths and managing regions involves clicking onto the corresponding ROIs, this function must be disabled. If not disabled, each interaction with an ROI or region will lead to a call of `zoomPerClick`, which creates a highly disorienting experience.

Since DZI implements a version of the tiled image pyramid model (compare subsection 2.1.1), a number of images tiles must be opened to display a current view. For this purpose the OSDV provides an `open(tileSources)` function. The tileSources are provided as URL to the DZI's metadata file by the ASS (either "/static/wsi/[file].dzi" (natural DZI) or "/static/slide.dzi" (artificial DZI from the DZG (compare subsection 4.4.2). Based on the URL extension (always *".dzi"* in this AS' scope), the OSDV selects an appropriate *TileSource* interface to access the individual tiles and levels of the provided image format.

[6] See `https://openseadragon.github.io/docs/OpenSeadragon.html#.Options` for an in-depth documentation of every parameter available.

parameter	description
id	Id of the element to append the viewer's container element to.
prefixUrl	Prepends the provided prefixUrl to the path for the OSDVs internal images.
showReferenceStrip	If true, display a scrolling strip of image thumbnails for navigating through the images.
showNavigator	Makes the navigator minimap visible if true.
sequenceMode	Set to true to view a sequence of images.
navigatorId	The ID of a div to hold the navigator minimap.
zoomPerClick	The distance to zoom in on every mouse click. Setting it to 1.0 disables the feature.

Table 4.7: Overview of used options in OSDV constructor (source: [45])

A scalebar is added to the OSDV to support the assessment of ROIs:

```
1  var mpp = 0;
2  if(slide.mpp) {
3    ppm = slide.mpp > 0 ? (1e6 / slide.mpp) : 0
4  }
5
6  viewer.scalebar({
7    type: OpenSeadragon.ScalebarType.MICROSCOPE,
8    minWidth:'150px',
9    pixelsPerMeter:ppm,
10   color:'black',
11   fontColor:'black',
12   backgroundColor:"rgba(255,255,255,0.5)",
13   barThickness:4,
14   location: OpenSeadragon.ScalebarLocation.TOP_RIGHT,
15   xOffset:5,
16   yOffset:5
17 });
```

as_viewer.js

The scale is $pixels/\mu m$. Since the ASS only delivers MPP (which is $\mu m/pixel$) and the OSDV's scalebar expects $pixels/m$ (PPM), a conversion is necessary:

$$PPM = \frac{10^6}{MPP} \tag{4.1}$$

Eq. 4.1 can be seen in line 3 of the code snippet above. If no valid input is provided from the ASS (that is: when no value was specified for MPP in the WSI's metadata), PPM will be set to 0. If the pixelPerMeter parameter of the scalebar equals 0 it is automatically hidden [45]. The options in line 7 - 16 concern the styling of the scalebar (such as color, position and size).

When navigating through the tiles and layers of the OSDV's tile source, it will automatically load the tiles needed for the current view from the ASS via HTTP GET requests [45].

Paper.js

The ASV utilizes Paper.js to create a region's path. Paper.js is an *"open source vector graphics scripting framework"*, running on top of the HTML5 canvas [54]. It offers an API to create and manage vector graphics and bezier curves[7]. Additionally, it offers vector relevant entities, such as *point*, *size* and *rectangle* objects and enables the drawing of finely grained paths.

When the OSDV opens an image, a so called *annotation overlay* (AO) is created:

```
1 viewer.addHandler('open',function(){
2   initAnnotationOverlay();
3   [...]
4 }
```

Excerpt from `initAnnotationService()` in as_viewer.js

The AO is a canvas which is placed above the OSDV. On it, paths can be drawn, which will be used to enclose an ROI. The AO is initialized in `initAnnotationOverlay()` (see line 2 in the code snippet above). Once created, the AO is resized to fit the opened WSI in height and width. This only happens when initializing the canvas. Therefore the AO will have (and keep) the dimensions of the level the WSI was opened in.

This leads to issues when zooming in and out on the OSDV (and consequently changing the level of the tile source). To counteract this, the AO is stretched to fit the new dimensions of the WSI's currently viewed layer. This leads to the problem of having different pixel densities in AO and OSDV.

The supported granularity of the AO is very fine and, when added programmatically, can create path segments within the decimals of a single pixel [54]. In drawing mode however, segments are only added between whole pixels, making it impossible to draw regions in high zoom levels. Since both, AO and OSDV, calculate clicked pixel positions from the clicked pixel in the browser [45, 54], a conversion is possible. Two conversion functions haven been implemented:

(1) AO coordinate → image coordinate

(2) image coordinate → AO coordinate

The corresponding functions are `convertPathToImgCoordinates(point)` for (1) and `convertImgToPathCoordinates(point)` for (2). The following code snippet shows their implementation:

[7] A *bezier curve* is a parametric curve, frequently used in computer graphics [74].

```
1  function convertPathToImgCoordinates(point) {
2    // convert to screen coordinates
3    var screenCoords = paper.view.projectToView(point);
4    // convert to viewport coordinates
5    var viewportCoords = viewer.viewport.pointFromPixel(new
       OpenSeadragon.Point(screenCoords.x, screenCoords.y));
6    // convert to image coordinates
7    var imgCoords = viewer.viewport.viewportToImageCoordinates(
       viewportCoords);
8    return imgCoords;
9  }
10
11 function convertImgToPathCoordinates(point) {
12   // convert to viewport coordinates
13   var viewportCoords = viewer.viewport.imageToViewportCoordinates(
       point);
14   // convert to screen coordinates
15   var pixel = viewer.viewport.pixelFromPoint(viewportCoords);
16   // convert to project coordinates
17   var projectCoords = paper.view.viewToProject(pixel);
18   return projectCoords;
19 }
```

as_viewer.js

`convertPathToImgCoordinates(point)` receives a point that is in the AO coordinate system and turns it into a screen coordinate (line 3). The screen coordinate is converted into a viewport coordinate (line 5) from where it can be turned into an actual image coordinate (line 7). A viewport coordinate maps coordinates into values within the interval $[0, 100]$ for both dimensions x and y, instead of using pixel coordinates [45].

`convertImgToPathCoordinates(point)` receives a point coordinate of the baseline image, which is turned into a viewport coordinate (line 13). The viewport coordinate is turned into a screen coordinate (line 15), to be turned into an AO coordinate from there (line 17).

Both of those functions need to work with the viewport coordinates to create a relation between the dimensions of the image and the current level.

4.5.2 Definition: Region

The basic concept used to realize annotations in the ASV are regions. Each region has a *context, imgCoords, name, path, uid* and *zoom* property as shown in tab. 4.8. They contain the connecting information between an ROI (imgCoords, path) and its label (name), making them the interface between visual and textual information- Because of their central importance to the ASV, this section describes the `region` object and its properties.

:Region
context : Array
imgCoords : OpenSeadragon.Point[1..n]
name: String
path: Paper.Point[1..n]
uid: Integer
zoom: Float

Table 4.8: Class diagram of the region class

The **context** property is a list of unique label ids of other regions (compare subsection 4.3.2). If there is a region R_1 and a region R_2, then the label of R_2 is added to the context of R_1, if:

- R_2's path touches, crosses, surrounds or is surrounded by R_1's path, and
- R_2's label is not in the context list of R_1 already, and
- R_2's label $\neq R_1$'s label

The **imgCoords** (image coordinates) property represents the coordinates of the path's segments in the baseline image. They are represented as a list of 1 to n Openseadragon.Points[8]. This property is used to establish the relation between the ROIs of the AO and the baseline image.

The **name** property contains the title of the region's corresponding label. A region is created with the dictionary label entry currently active at the time of the regions creation.

The **path** property represents the coordinates of the path's segments in the AO. It is a list of 1 to n Paper.Points[9]. They are being used to manage the contour of an ROI in the AO. The number of points in imgCoords and path is always identical, as is their position inside the list.

The **uid** is a unique id (uid $\in \mathbb{Z}^+$) that is used to identify the region.

The **zoom** attribute describes the level of magnification at the point a region was created. It is the ratio between displayed and original image size (e.g. 1 = 1:1 ratio, 0.5 = 1:2 ratio). This information is added due to the fact that the annotation process seldom happens in the highest nor at a singular magnification. Instead, different magnifications are used (lower ones to speed up the process, higher ones to inspect ROIs in more detail). This leads to annotations that are rarely pixel precise and tend to contain numerous false positives and negatives [1].

[8] See `https://openseadragon.github.io/docs/OpenSeadragon.Viewport.html#viewportToImageCoordinates` for more information on OSD's Point class.

[9] See `http://paperjs.org/reference/point/` for more information on Paper.js' Point class.

4.5.3 Graphical User Interface

This subsections describes the GUI of the ASV and its individual elements (compare fig. 4.5 for reference):

[1] **Toolbar**
The toolbar menu contains the *Help button* ([2]), *toolbar toggle switch* ([3]), *navigation map* ([4]), *dictionary list* ([5]) and the *label list* ([6]).

[2] **Help button**
Hovering over the *help button* shows a tooltip containing the hotkey mapping of the ASV.

[3] **Toolbar toggle**
The *toolbar toggle switch* can be used to show and hide the toolbar.

[4] **Navigation map**
The *navigation map* shows a macro image of the currently opened WSI and highlights the viewed area.

[5] **Dictionary list**
The *dictionary list* shows the currently selected dictionary. A left click on it opens a list with all currently available dictionaries from which a new one can be selected. Once selected the old labels will be removed and the new labels loaded instead.

[6] **Label list**
The *label list* lists all available annotation labels from the loaded dictionary. The selected label is highlighted (see *Gefäß* (vessel) in fig. 4.5). Other labels can be selected by a mouse click. Additionally, the labels can be iterated via the *tab key*. A click on the eye symbol on top of the list toggles the visibility of all regions. A click on the eye symbol to the left of the label name toggles the visibility of all regions associated with that label. The color of a label can be changed with a click on the colored square. This will also change the color of all regions which are already drawn and associated to that label.

[7] **OSDV and AO**
The OSDV with its AO and exemplary regions.

[8] **Example of a region**
Example of a *"sonstige"* (miscellaneous) labeled region.

Figure 4.5: ASV's GUI with opened, annotated WSI

Chapter 5

Tessellation Service

5.1 Objective of the Tessellation Service

The objective of the TS is to provide a service that extracts image regions annotated by the AS to create samples for the training of a NN. In this context, "extraction" describes the process of creating sub images of the original image, which contain all of the annotated ROI and as little of anything else as possible. Additionally, the correspondence with the associated region label must be kept. The AS uses *regions* to describe ROIs (compare subsection 4.5.2). Those regions are persisted in a JSON file. Therefore, the TS has 2 objectives:

(1) parse a JSON file and acquire its region data

(2) extract ROIs based on the acquired region data

The handling of WSIs is complicated due to their size and file formats. To avoid unnecessary complication, the TS will use common file types (JPEG, PNG and TXT) for the extraction.

5.2 Methodology

The objective of the TS is to create usable training samples for NNs, as stated in section 5.1. Depending on the setup, chosen learning method (compare section 2.2) and purpose of the NN, the requirements imposed on the training samples may vary. Smith demonstrates in [72] exemplary how to train a NN to recognize letters in images of written text by training it with 10x10 pixel grayscale images (256 gray levels/pixel) of letters. Shereena and David introduce a novel content based image retrieval classification method in [77], based on color and texture features. Other approaches extract features through the use of mathematical models from the supplied images (such as edges or shapes) [32].

Because of those varying requirements, the TS will be capable of producing different output:

(1) Unaltered image of ROI

(2) Resize images to a specific width and height

(3) Approximate ROIs via tessellation

(4) Convert extracted images to grayscale

A user can draw a region's path without any restrictions concerning the pattern, resulting in ROIs that can be of arbitrary shape. Therefore, bounding boxes (BB) are used for ROI extraction in the cases (1) and (2) (see fig. 5.1 for an example). A BB is a rectangular body, fully enclosing a provided (two dimensional) object of arbitrary shape [76].

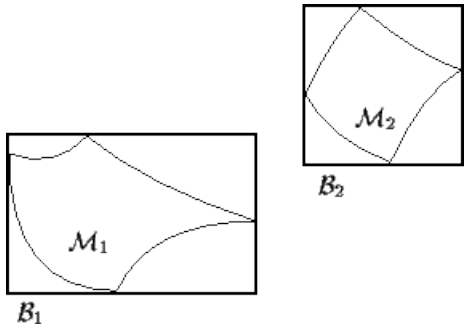


Figure 5.1: Examplary BBs: B_1 is BB of M_1, B_2 is BB of M_2 (source: `http://www.idav.ucdavis.edu/education/GraphicsNotes/Bounding-Box/Bounding-Box.html`)

In the case of (1), an ROI's BB is copied pixel by pixel into a new image. The resizing (scaling the output image up or down to the provided pixel values) in the case of (2) makes preprocessing of the BB necessary. This has 2 reasons:

- If the aspect ration of the provided width and height is different than the one of the BB, the resulting image will be distorted.
- When scaling images down, interpolation can be used, to reduce the information loss of the image [31]. This is partially possible when scaling images up as well, e.g. via fractal interpolation, but a non trivial task [24].

Therefore, the size of the BB will be adjusted to match the aspect ratio of the provided width and height. If the resulting BB is bigger then the provided width and height, the image will be scaled down and interpolated. If the BB is smaller, it will be scaled up instead of the image. This leads to a bigger area inside the BB that is not part of the ROI, but a pixel ratio of 1:1 between original and extracted image, resulting in no distortion or loss of quality.

Case (3) approximates a ROI by tessellating it into tiles of the provided width and height. *Tessellation* describes the tiling of a plane using one or more

geometric shapes with no overlaps or gaps (see fig. 5.2 for an example) [10]. Every tile inside the region's enclosing path (or touched by it) is targeted by the extraction. Each tile is extracted into an individual image.

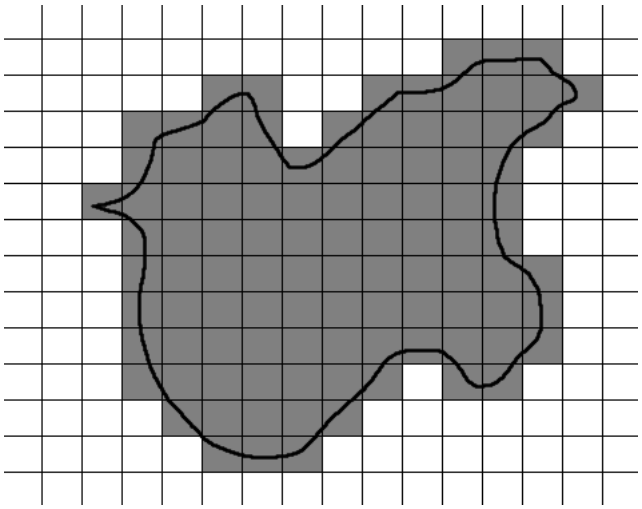


Figure 5.2: Example of an ROI approximated via tessellation (gray tiles belong to the ROI)

Cases (1) - (3) can be combined with (4) to convert the extracted ROI images into grayscale images.

A region contains additional information besides the label that can be utilized, especially the level of magnification (compare subsection 4.5.2). As Doyle has shown in [34], the level of magnification an annotation was made in is an important factor. This is due to the fact that manual annotations made in lower magnifications do not have the granularity needed for higher ones, which results in numerous false positives and negatives and therefore distorts the ground truth [1]. A region's zoom attribute informs at what level of magnification a region was drawn in, thus helping to assess the described issue.

To utilize the additional metadata, each extracted region will also have a corresponding metadata file. This metadata contains:

- the name of the extracted image file (case (1) and (2)), or a list of all image tiles which belong to the ROI (case (3))
- the region's label
- the level of magnification
- the region's context

Section 5.1 additionally stated the objective of keeping up correspondence between an ROI and its associated label. To achieve this, an extracted image

will be saved in a directory of the same name as its corresponding label. This way, all ROIs of a specific label can be collected in one location. Additionally, the corresponding metadata file will contain an association between image file and label name.

5.3 Implementation

The TS is implemented as a python script called ***TessellationService.py***[1]. As stated in sections 5.1 and 5.2, its purpose is to extract annotated ROIs from a WSI. This is done with the help of the following frameworks and libraries: *OpenSlide*, *NumPy*, *Pillow* and *OpenCV* (Open Source Computer Vision Library).

OpenSlide was used to access and read a provided WSI (compare subsection 4.5.1 - OpenSlide).

NumPy is a python library for efficient scientific computing, especially regarding operations involving multi-dimensional array objects [78]. Since a lot of image operations are based on the use of matrices, this library was used to increase efficiency.

OpenCV is an open source computer vision and machine learning software library. It was built to provide a common infrastructure for computer vision applications. It offers numerous functions regarding image processing and machine learning, with a strong focus on real-time computing and efficiency [43]. Its main purpose in the TS is to create a reference image for the tessellation function (see subsection 5.3.4).

Pillow is an imaging library, which offers methods to read from and write to images. It was used for reading, writing, scaling and converting the extracted images.

A detailed documentation of the TS' functions can be found in appendix C.

5.3.1 Execution

The TS can be called over the python interpreter:

```
1  $ python TessellationService [input] [dictionary] [params]
```

The arguments [input] and [dictionary] are mandatory. The files which are supposed to be targeted by the extraction process are handed to the TS via the *[input]* argument. This can be a list of WSIs, DZIs and directories. If a directory is handed to the TS, all subdirectories (except a DZI's *"_files"* directory) will be searched as well.

As mentioned in chapter 4, multiple dictionaries can be used for annotation. Since every WSI has an individual save file for each dictionary (compare subsection 4.4.3), it is necessary to provide the dictionary targeted for extraction via the [dictionary] parameter. Each input file must have its associated save file in the same directory.

[1] The TS is contained on the disc at the end of this thesis, see appendix A.1.

As mentioned in section 5.2, it might be necessary to create different output images for different purposes. Therefore, the TS has a list of optional parameters to manipulate the created output in order to be applicable to a wider variety of cases (see tab. 5.1). Those parameters are optional.

name	**description**	**default**
-h, –help	show help	-
-f, –force-overwrite	flag to overwrite images with the same name (if not supplied, a number will be added to the image name)	false
-g, –grayscale	flag to convert images to grayscale images before saving them	false
-i, –interpolation	choose interpolation method: nearest neighbor, bilinear, bicubic, lanczos	nearest neighbor interpolation
-o, –output [directory]	choose output directory (if not provided, the TS will save all extracted images in its current working directory)	-
-r, –resize [width] [height]	resize all output images to the provided [width] and [height] in pixel	-
-s, –show-tessellated	flag to create window and show stitched image resulting from the tessellation process (only for debugging purposes, see *"-t"*)	false
-t, –tessellate [width] [height]	tessellate regions into tiles of the provided [width] and [height] in pixels	-

Table 5.1: Optional parameters for the TS

5.3.2 Extraction process

The TS receives a list of files and folders to extract regions from and iterates over each entry. If the entry is a directory, each contained element will be checked for its type. If it is a WSI/DZI file, the extraction is started right away. Once the extraction is finished, the next element is examined. If the entry is a directory, each contained element is subject to the same evaluation. WSI/DZI files are extracted, while subdirectories are recursively traversed until the end of the directory tree. Each element found in this process that qualifies for region extraction is extracted. Fig. 5.3 visualizes the described process in an activity

diagram.

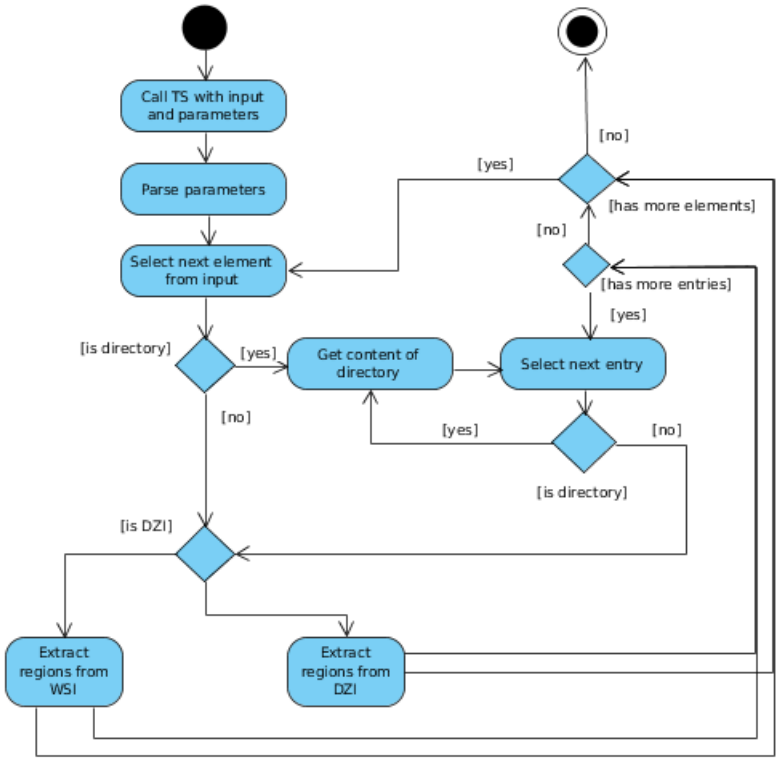


Figure 5.3: Activity diagram of TS' extraction procedure

OpenSlide is used to read a provided WSI. Since OpenSlide can not open a native DZI, but only wrap an OpenSlide object into a DZG(compare subsection 4.4.2), a different approach had to be chosen to access a provided DZI.

Both approaches share the following utility functions[2]:

- `read_json(path)` tries to read the JSON file at the provided path. If successful, it returns a list with the regions parsed from the file. If the file was not found or corrupt, a message will be printed on the terminal, informing the user about the exception.
- `save_image(image, region, slide_name, *tiles)` generates an appropriate name from the region information and the slide_name for the image and metadata file. The name differs between tessellated and non-tessellated output (see tab. 5.3).

[2] See appendix C for documentation of the individual function's source code.

If -r is provided, the image will be resized to the supplied height and width, with the interpolation method provided via -i (default: nearest neighbor interpolation). If -g is specified, the image will be converted to grayscale with the luma transform defined in the ITU-R 601-2 standard [49] (see eq. 5.1).

$$L = R * 299/1000 + G * 587/1000 + B * 114/1000 \tag{5.1}$$

Eq. 5.1 averages out the RGB values of every pixel under consideration of how a human eye perceives colors.

If -o was provided, the processed image will be saved in the supplied location inside a directory with the name of the extracted region's label.

- `save_metadata(name, region, *tiles)` writes the corresponding metadata to an extracted image into a file (see tab. 5.3 for the file name pattern). The metadata is a JSON map and contains the values listed in tab. 5.2. When the TS is run with -t, the *"image"* key is replaced by *"tiles"*, which contains a JSON list with the names of all tiles belonging to the extracted image (each individual image name is according to the pattern in tab. 5.3).

key	**value**	**type**
context	the image region's context	JSON list
image	name of the extracted image file (compare tab. 5.3)	string
label	the region's label	string
zoom	the region's zoom	float

Table 5.2: File name patterns of generated output

- `get_bounding_box(region)` iterates over every segment of a region's path and builds the BB by collecting the minimal and maximal values for x and y (compare fig. 5.1).
- `resize_bounding_box(bounding_box)` is used to adjust a region's BB to a different aspect ratio. This is to avoid image distortion when the -r parameter specifies a different image ratio than the BB's inherent one (see fig. 5.4 for an example).

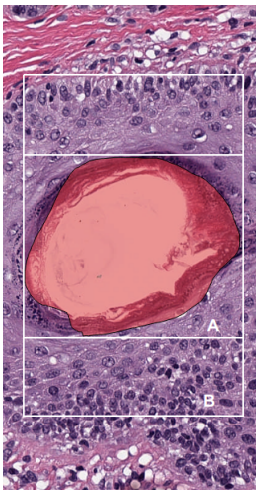


Figure 5.4: Example for adjusted BB (A - original BB with aspect ratio of ∼1:1, B - adjusted BB to aspect ratio of ∼1:2)

- `scale_bounding_box(bounding_box, scale)` scales a BB by a provided scale. This is to avoid image distortions due to scaling images up (compare section 5.2). While `get_bounding_box(region)` changes the aspect ratio of a BB, `scale_bounding_box(bounding_box, scale)` keeps the aspect ratio and changes the BB as a whole instead.

name	**image file**	**metadata file**
-t not provided	[slide_name]_[region.uid].jpeg	[slide_name]_[region.uid].metadata.json
-t provided	[slide_name]_[region.uid]([row]_[column]).jpeg	[slide_name]_[region.uid].metadata.tessellated.json

Table 5.3: File name patterns of generated output

5.3.3 Extraction without Tessellation

As mentioned above, the implementation of the conversion differs for WSI and DZI due to the role of OpenSlide in the extraction of WSIs. The following two sections describe the extraction process for both cases. A documentation of all mentioned functions can be found in appendix C.

WSI

The WSI extraction process is done in the `wsi(file)` function. It opens the provided file via OpenSlide, builds a name from its path and parses the regions

of the WSI via `read_json(path)`. The function iterates over every parsed region to extract it. The extraction process is visualized in fig. 5.5.

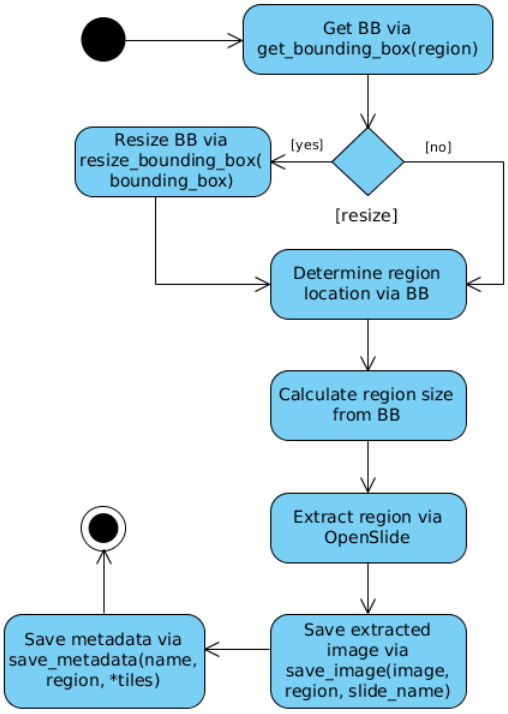


Figure 5.5: Activity diagram of TS' WSI extraction (without tessellation)

As shown in fig. 5.5, calculating the BB is the first step. To do so, the `get_bounding_box(region)` function is used, which returns a python dictionary with the minimum and maximum values of x and y in the associated region's path. If -r was provided, the BB's size will be adjusted to fit the supplied image ratio.

Next, the location of the region is determined. This is done by taking the minimum value for x and y from the BB and declaring it as upper, left corner of the region. The size is calculated by subtracting the minimum from the maximum value for x and y respectively (see eq. 5.2 and 5.3).

$$size_x = BB_{max(x)} - BB_{min(x)} \tag{5.2}$$

$$size_y = BB_{max(y)} - BB_{min(y)} \tag{5.3}$$

OpenSlide's `readRegion(location, level, size)` function is used to retrieve the region in a separate image from the original baseline image. This extracted image is then written into a file, together with its associated region's metadata.

DZI

Since it is not possible to open a DZI with OpenSlide, the extraction process for DZI differs from the one for WSI. It is implemented in the `dzi(file)` function, which starts and manages the extraction process. Since an image must be stitched together manually, the information about tile size and image dimensions must be parsed from the DZI metadata file, as well as the level containing the baseline image (since DZI uses n instead of 0 for the level with the highest resolution). Once the information is parsed, a file name is created from the file path and the corresponding regions are parsed from the associated JSON file. Fig. 5.6 visualizes the extraction process and its individual steps, which are executed for each region.

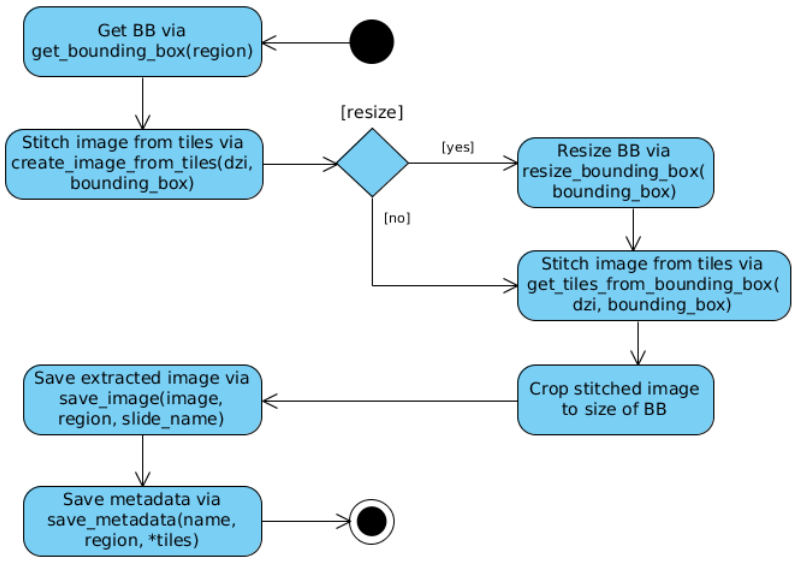


Figure 5.6: Activity diagram of TS' DZI extraction (without tessellation)

The BB is created via the `get_bounding_box(region)` function. Once the BB is created, the stitching process of the area covered by the baseline image begins. This is realized in the `create_image_from_tiles(dzi, bounding_box)` function. It first checks, if the -r parameter was provided and if so, if the aspect ratio of the BB matches the ratio of the supplied height and width. If necessary, the BB is adjusted via `resize_bounding_box(bounding_box)`.

The first step in stitching the baseline image area together, is determining the size of the BB (see eq. 5.2 and 5.3). To determine the needed tiles, $BB_{min(x)}$, $BB_{min(y)}$, $BB_{max(x)}$ and $BB_{max(y)}$ are divided by the DZI's tile size, resulting in $BB^t_{min(x)}$, $BB^t_{min(y)}$, $BB^t_{max(x)}$ and $BB^t_{max(y)}$. Those values are then iterated from the pair $(BB^t_{min(x)}, BB^t_{min(y)})$ to $(BB^t_{max(x)}, BB^t_{max(y)})$.

Each of the resulting pairs (BB_x^t, BB_y^t) refers to a tile from the baseline image. The baseline image area is stitched together by iterating over all pairs (BB_x^t, BB_y^t) and stitching the corresponding tiles together. This is implemented in the `get_image_from_bounding_box(dzi, bounding_box)` function. The stitched image is then cropped to fit the size and position of the BB and saved together with its corresponding metadata file (via `save_file(image, region, slide_name)` and `save_metadata(name, region, *tiles)`).

5.3.4 Extraction with Tessellation

If a region is extracted with the -t parameter (compare tab. 5.1), it will be tessellated into multiple image tiles of provided width and height (compare 5.2(3) and fig. 5.2). As in subsection 5.3.3, the main difference between the processing of a WSI and a DZI is the need to parse the DZI's associated metadata and stitch an area of the baseline image. With that exception, the tessellated extraction works identical in both cases.

The baseline image is tessellated into m*n virtual tiles (see eq. 5.4 and 5.5). All virtual tiles containing parts of the region's path are saved in a list L^{path}.

$$m = \text{baseline image height/tile height} \tag{5.4}$$

$$n = \text{baseline image width/tile width} \tag{5.5}$$

The virtual tiles are needed to create and work with the *reference image* (RI). The RI is a central component of the tessellated extraction process. It is a binary image, which is m pixels high and n pixels wide. Each pixel represents a virtual tile of the baseline image. If a pixel is white, the corresponding virtual tile is part of the region to extract.The RI is a matrix representation of which tile is needed from the baseline image (see fig. 5.7).

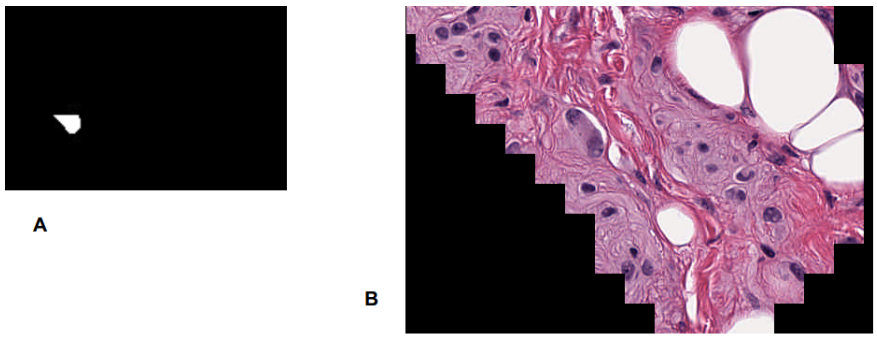


Figure 5.7: Example of RI (A) and resulting image tiles (B, stitched) with a tile size of 32x32 pixel (example was created using CMU-1.svs from Aperio, see appendix A.2)

When created, the RI has only black pixels. To color the pixels associated with the ROI white, OpenCV's `drawContours(image, contours, color,`

`fill)` function is used. Since the correspondence of pixels and tiles is bidirectional, L^{path} can be used to describe the contour. The drawn contour is then filled white.

The last step is to iterate each pixel of the RI. For every white pixel found, the corresponding tile is extracted from the baseline image and saved into an individual image file (compare tab. 5.3 for the naming convention).

5.4 Test

The TS was run on multiple WSIs from the test data set (see appendix A.2). The test case presented in the following subsections is based on *CMU-1.svs* of the Aperio test set[3] The WSI and corresponding JSON file used for the test are included on the disc at the end of this thesis (see appendix A.1). The annotations were made using the *example.json* dictionary.

5.4.1 Setup

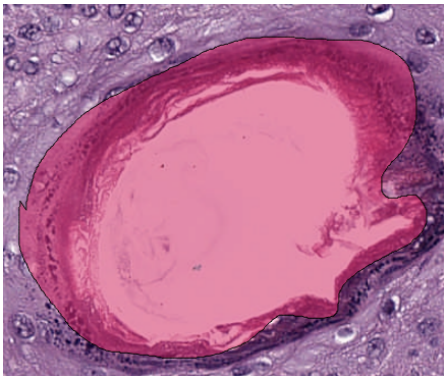
The WSI CMU-1.svs, its corresponding JSON file CMU-1.svs_example.json and TessellationService.py were placed in the same directory. To ensure that all parameters work correctly, the TS was provided with different parameters in multiple test cases (see tab. 5.4).

The provided JSON file has 3 saved annotations (see fig. 5.8), to test the behavior of small (BB of 39x41 pixel, see fig. 5.8b), medium (BB of 346x288 pixel, see fig. 5.8a) and large sized images (BB of 6085x1540 pixel, see fig. 5.8c). All test cases were executed with those 3 regions.

test #	**parameters**	**description**
1	-o noparams	no parameters provided, except -o
2	-o square -r 256 256	resize every image to 256x256 pixels (aspect ratio of 1:1)
3	-o rectangle -r 256 128	resize every image to 256x128 pixels (aspect ratio of 2:1)
4	-o tessellate -t 32 32	tessellation of ROIs with 32x32 pixel sized tiles
5	-o grayscale -r 256 256 -g	resize every image to 256x256 pixels and turn them to grayscale

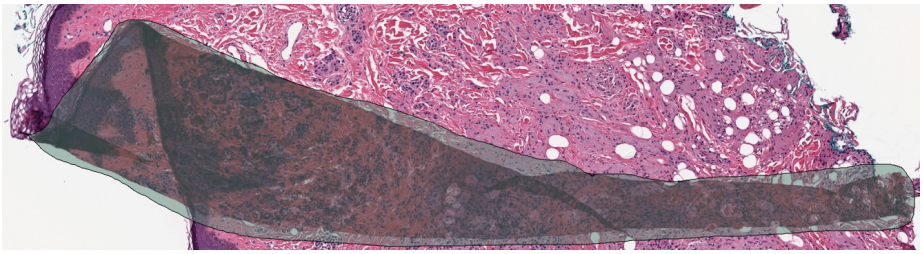
Table 5.4: Test cases for the TS

[3] Obtainable at `http://openslide.cs.cmu.edu/download/openslide-testdata/Aperio/`



(a) test region A (BB = 364x288 pixel)

(b) test region B (BB = 39x41 pixel)



(c) test region C (BB = 6085x1540 pixel)

Figure 5.8: Test case *Aperio - CMU-1.svs*

5.4.2 Result

Fig. 5.9 - 5.11 show the results of the test cases described in tab. 5.4. Tab. 5.5 shows how the results correspond to the tests.

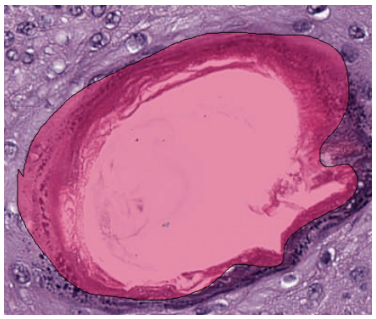
test #	**figures**
1	5.9b, 5.10b, 5.11b
2	5.9c, 5.10c, 5.11c
3	5.9d, 5.10d, 5.11d
4	5.9e, 5.10e, 5.11e
5	5.9f, 5.10f, 5.11f

Table 5.5: Test-result correspondence

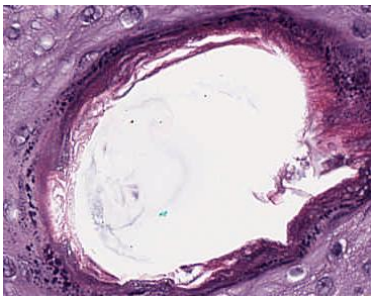
The extraction without parameters was successful in every case (see 5.9b, 5.10b, 5.11b).

The BBs were successfully adjusted and the images scaled to the provided size (5.9b, 5.10c, 5.11c and 5.9d, 5.10d, 5.11d).

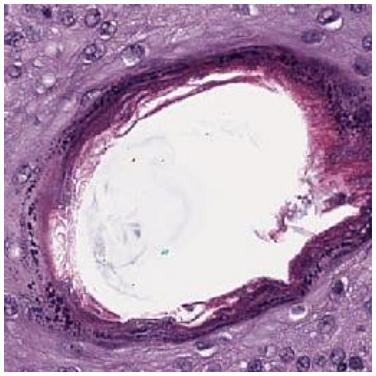
The tessellation of ROIs was successful for all provided regions (5.9e, 5.10e, 5.11e), as was the conversion to grayscale (5.9f, 5.10f, 5.11f).



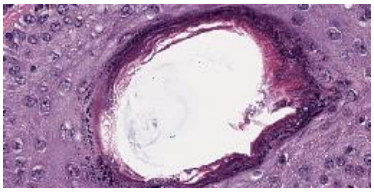
(a) Test region A in the ASV



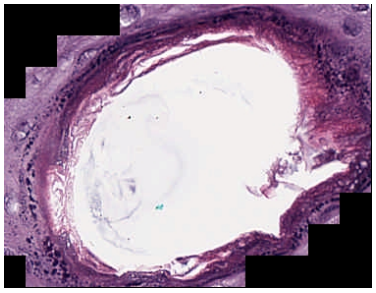
(b) Extracted without parameters



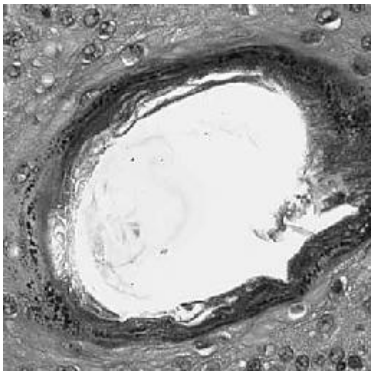
(c) Extracted and resized to 256x256 pixel



(d) Extracted and resized to 256x128 pixel



(e) Tessellated into 32x32 pixel sized tiles

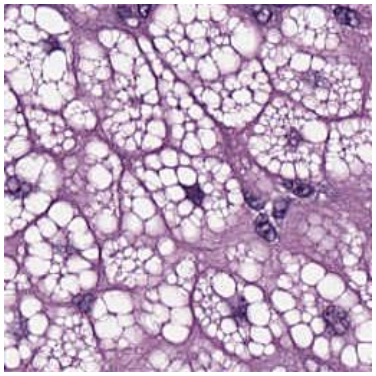


(f) Extracted, resized and converted to grayscale

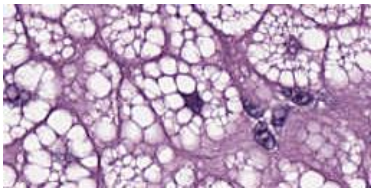
Figure 5.9: Results for test region A

(a) Test region B in ASV

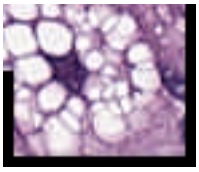
(b) Extracted without parameters



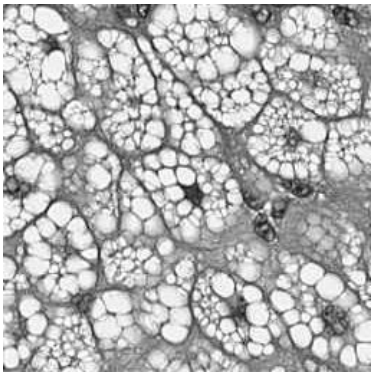
(c) Extracted and resized to 256x256 pixel



(d) Extracted and resized to 256x128 pixel

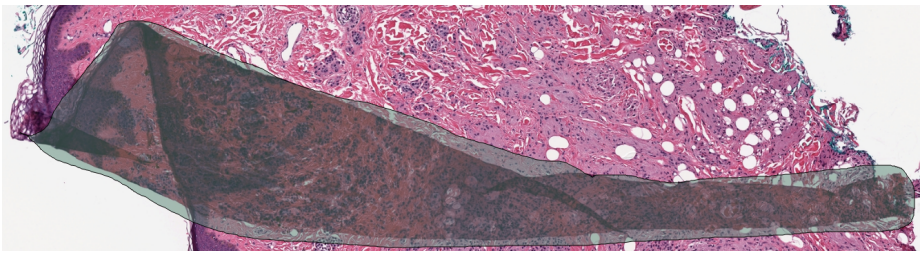


(e) Tessellated into 32x32 pixel sized tiles

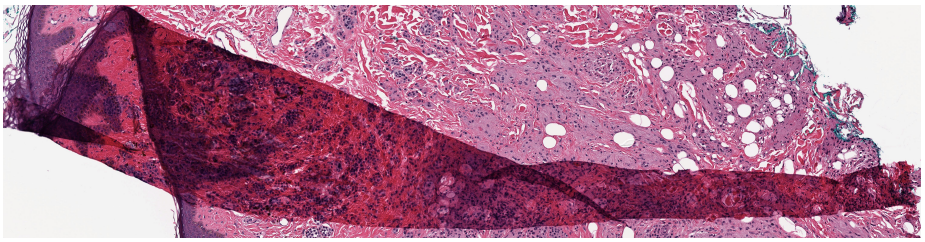


(f) Extracted, resized and converted to grayscale

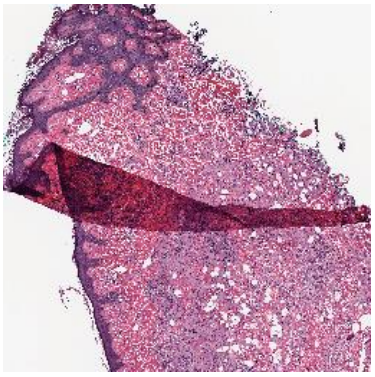
Figure 5.10: Results for test region B



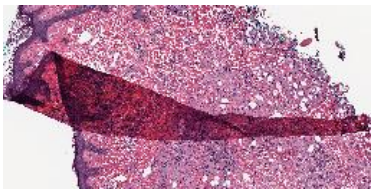
(a) Test region C in ASV



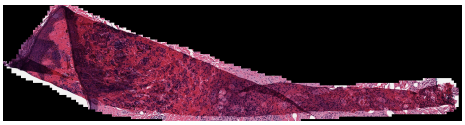
(b) Extracted without parameters



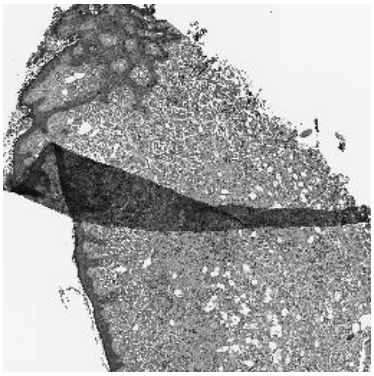
(c) Extracted and resized to 256x256 pixel



(d) Extracted and resized to 256x128 pixel



(e) Tessellated into 32x32 pixel sited tiles



(f) Extracted, resized and converted to grayscale

Figure 5.11: Results for test region C

Chapter 6

Conclusion

6.1 Results

The research goal of this thesis was to create a set of tools which enables deep learning and pathology experts to cooperate in creating a ground truth for NN by annotating WSIs. 3 objectives were defined to reach that goal (compare section 1.2):

(1) The introduction of a conversion tool, which turns proprietary WSI formats into an open one.

(2) The deployment of a WSI viewer tool with annotation capabilities and easy-to-understand GUI.

(3) The implementation of a tool that is capable of turning persisted annotations into a format usable as ground truth.

Each objective has a corresponding chapter, in which one or more tools were introduced. The tools, chapters and goals correspond as stated in tab. 6.1.

objective	**chapter**	**tool**
(1)	3	ConversionService.py
(2)	4	ASS & ASV
(3)	5	TessellationService.py

Table 6.1: Correspondence of objectives, chapters and tools

Chapter 3 introduced the CS, which is capable of converting a proprietary WSI to a DZI. A test was set up to ensure its functionality, which was successful (compare subsection 3.3).

Chapter 4 introduced the AS, which consists of the ASS and the ASV. It was developed in an iterative process with regular feedback after each iteration from

a pathologist concerning the GUI and UX. The ASS' use of OpenSlide enabled the AS to read and view a proprietary WSI, thus making the CS redundant.

Chapter 5 introduced the TS, which extracts annotations made and persisted by the AS into individual images with a corresponding metadata file, thus delivering a ground truth for the training of NN. To ensure, that the TS works as intended, a test case was set up (compare subsection 5.4), which was completed successfully.

6.2 Conclusion

Objective (1), the conversion of proprietary to open image formats, was accomplished by the CS. The need of converting a WSI to DZI prior to using it in the AS was inflexible and cumbersome. Thus, an on demand conversion was implemented into the AS, which eliminates the need of a prior conversion and enabled the AS to view a proprietary WSI directly.

Objective (2), developing a WSI viewer with annotation capabilities (compare subsection 4.3.2), was accomplished by the AS, which consists of 2 parts: ASS and ASV. The ASS delivers a local slide repository. Additionally, it offers a RESTful API, which can be used by the ASV to request and persist data. Since the ASV is running in a web browser, it can be used independent of an operating system. The ASV offers a simple GUI to make annotations to a WSI based on regions (compare subsection 4.5.2) and label dictionaries.

Objective (3), the extraction of the AS' annotations to create a ground truth, was accomplished by the TS. It offers a python script to extract an ROI out of a WSI into an individual image, while keeping correspondence to the region's label and metadata through the use of directory structure and metadata files. The extracted files are of common types (JPEG, PNG, TXT) to ensure uncomplicated further processing.

The tools presented in this thesis are capable of handling the complete process chain introduced in subsection 2.4. Thus, the goal of introducing tools for the cooperative annotation of WSIs by deep learning and pathology experts to create a ground truth was accomplished.

6.3 Future work

All tools introduced in this thesis are open source and available in their corresponding repositories on GitHub (see tab. 6.1). This makes their distribution convenient and enables third parties to add functionality they deem imperative. Additionally, new projects can be based on the existing code base.

Currently, the ASS is a local web server. In future iterations it could be turned into a dedicated RESTful web server with access to a slide repository, making an easier distribution of WSIs possible. Additionally, a functioning script for the automated segmentation has to be delivered yet.

Since the TS and the ASS are both written in python, the TS could be integrated into the ASS and be executable over the ASV's GUI. This would also make a visualization and manual manipulation of BBs possible.

For easier handling, a file browser could be integrated into the ASV, so that the user does not have to build and send URLs manually.

Appendices

Appendix A

Test Data

A.1 Files on disc

A disc is included at the end of this thesis. This disc contains the 3 introduced services from the chapters 3 (CS), 4 (AS) and 5 (TS). Alternatively, they are available at their corresponding GitHub repositories (see tab. A.1).

service	**disc path**	**GitHub repository**
CS	/services/ConversionService	`https://github.com/SasNaw/ConversionService`
AS	/services/AnnotationService	`https://github.com/SasNaw/AnnotationService`
TS	/services/TessellationService	`https://github.com/SasNaw/TessellationService`

Table A.1: Services with their disc paths and repository URLs

All tools presented in this thesis are open source and their use, further development and diversification are highly encouraged.

The disc also contains 2 exemplary WSI files (CMU-1.svs from Aperio, see A.2.1 and CMU-3.ndpi from Hamamatsu, A.2.3). Due to the limited storage space available on the medium and the size of most WSI files, only 2 could be provided on disc. They can be found under */testdata/wsi/* and are usable for all 3 services.

The JSON save file from 5.4 can also be found on the disc, at */testdata/json/CMU-1.svs_example.json*.

A.2 Free Whole Slide Images

The following test data was used for all tests involving CS, AS and TS. It can be found at OpenSlide's homepage, at the freely distributable test data section[1]. Various slides can be found there. The sections A.2.1 - A.2.8 give listings of all used WSIs, sorted by vendor and file format.

A.2.1 Aperio (.svs)

name	**size (MB)**	**description**
CMU-1-JP2K-33005.svs	126.42	Export of CMU-1.svs, brightfield, JPEG 2000, RGB
CMU-1-Small-Region.svs	1.85	Exported region from CMU-1.svs, brightfield, JPEG, small enough to have a single pyramid level
CMU-1.svs	169.33	Brightfield, JPEG
CMU-2.svs	372.65	Brightfield, JPEG
CMU-3.svs	242.06	Brightfield, JPEG
JP2K-33003-1.svs	60.89	Aorta tissue, brightfield, JPEG 2000, YCbCr
JP2K-33003-2.svs	275.85	Heart tissue, brightfield, JPEG 2000, YCbCr

Table A.2: Aperio data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Aperio/`)

A.2.2 Generic Tiled tiff (.tiff)

name	**size (MB)**	**description**
CMU-1.tiff	194.66	Conversion of CMU-1.svs to pyramidal tiled TIFF, brightfield

Table A.3: Generic Tiled tiff data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Generic-TIFF/`)

[1] See `http://openslide.cs.cmu.edu/download/openslide-testdata/` for more information.

A.2.3 Hamamatsu (.ndpi)

name	size (MB)	description
CMU-1.ndpi	188.86	Small scan with valid JPEG headers, brightfield, circa 2009
CMU-2.ndpi	382.14	Brightfield, circa 2009
CMU-3.ndpi	270.1	Brightfield, circa 2009
OS-1.ndpi	1,860	H&E stain, brightfield, circa 2012
OS-2.ndpi	931.42	Ki-67 stain, brightfield, circa 2012
OS-3.ndpi	1,370	PTEN stain, brightfield, circa 2012

Table A.4: Hamamatsu data set (.ndpi, source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Hamamatsu/`)

A.2.4 Hamamatsu (.vms)

name	size (GB)	description
CMU-1.zip	0.62	Brightfield
CMU-2.zip	1.13	Brightfield
CMU-3.zip	0.91	Brightfield

Table A.5: Hamamatsu data set (.vms, source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Hamamatsu-vms/`)

A.2.5 Leica (.scn)

name	size (GB)	description
Leica-1.scn	0.28	Brightfield, single ROI, 2010/10/01 schema
Leica-2.scn	2.1	Mouse kidney, H&E stain, brightfield, multiple ROIs with identical resolutions, 2010/10/01 schema
Leica-3.scn	2.79	Mouse kidney, H&E stain, brightfield, multiple ROIs with different resolutions, 2010/10/01 schema
Leica-Fluorescence-1.scn	0.02	Fluorescence, 3 channels, single ROI, 2010/10/01 schema

Table A.6: Leica data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Leica/`)

A.2.6 Trestle (.tiff)

name	size (MB)	description
CMU-1.zip	158.87	Brightfield
CMU-2.zip	304.22	Brightfield
CMU-3.zip	223.11	Brightfield

Table A.7: Trestle data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Trestle/`)

A.2.7 Ventana (.bif)

name	size (GB)	description
OS-1.bif	3.61	H&E stain, brightfield
OS-2.bif	2.53	Ki-67 stain, brightfield

Table A.8: Trestle data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Trestle/`)

A.2.8 Mirax (.mrxs)

name	size (GB)	description
CMU-1-Exported.zip	2.02	Export of CMU-1.mrxs with overlaps resolved, brightfield, JPEG, CURRENT_SLIDE_VERSION 2.3
CMU-1-Saved-1_16.zip	0.003	Quick save of CMU-1.mrxs at 1/16 resolution (multiple positions per image), brightfield, JPEG, CURRENT_SLIDE_VERSION 1.9
CMU-1-Saved-1_2.zip	0.14	Quick save of CMU-1.mrxs at 1/2 resolution (multiple images per position), brightfield, JPEG, CURRENT_SLIDE_VERSION 1.9
CMU-1.zip	0.54	Brightfield, JPEG, CURRENT_SLIDE_VERSION 1.9
CMU-2.zip	1.22	Brightfield, JPEG, CURRENT_SLIDE_VERSION 1.9
CMU-3.zip	0.65	Brightfield, JPEG, CURRENT_SLIDE_VERSION 1.9
Mirax2-Fluorescence-1.zip	0.06	Fluorescence, 3 channels, JPEG, CURRENT_SLIDE_VERSION 2
Mirax2-Fluorescence-2.zip	0.04	Fluorescence, 3 channels, JPEG, CURRENT_SLIDE_VERSION 2
Mirax2.2-1.zip	2.61	HPS stain, brightfield, JPEG, CURRENT_SLIDE_VERSION 2.2
Mirax2.2-2.zip	2.38	HPS stain, brightfield, JPEG, CURRENT_SLIDE_VERSION 2.2
Mirax2.2-3.zip	2.77	HPS stain, brightfield, JPEG, CURRENT_SLIDE_VERSION 2.2
Mirax2.2-4-BMP.zip	0.95	Brightfield, BMP, CURRENT_SLIDE_VERSION 2.2
Mirax2.2-4-PNG.zip	1.01	Brightfield, PNG, CURRENT_SLIDE_VERSION 2.2

Table A.9: Mirax data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Mirax/`)

Appendix B

Annotation Service Documentation

The following two sections document the implemented functions of the ASS (section B.1) and ASV (section B.2) in detail.

B.1 Annotation Service Server

`getDictionary()` is a utility function that fetches the dictionary associated with the current WSI/DZI from the slideDictionaries.json file[1].

It first opens slideDictionaries.json, creates a python dictionary from its content and tries to fetch the value corresponding to the WSI/DZI's name (line 2 - 4). If a value was found, it is returned (line 5 - 6), otherwise a new entry is created. The key is the WSI/DZI's file name, the value is the default dictionary from the configuration.json file, which is then returned (line 7 - 11).

getDictionary()

```
1  def getDictionary(file_path):
2    with open(SLIDE_DICTIONARIES, 'r') as file:
3      dictioary_map = json.loads(file.read())
4    dictionary = dictioary_map.get(file_path)
5    if dictionary:
6      return dictionary
7    else:
8      dictioary_map[file_path] = DEFAULT_DICTIONARY
9      with open(SLIDE_DICTIONARIES, 'w') as file:
10       file.write(json.dumps(dictioary_map))
11     return DEFAULT_DICTIONARY
```

[1] Compare subsection 4.4.3

index_dzi()

If the client requests a DZI (URL ends in *".dzi"*), `index_dzi()` renders an ASV and passes the necessary information (slide URL, file name, MPP and dictionary) to it.

It builds the file name and slide URL (line 3 and 4) for a requested DZI. A metadata.txt will be present in the [slide name]_files directory, if the DZI was created with the CS. If so, the function will try to fetch the metadata information about MPP and calculate the average size of a pixel (line 6 - 16). If the MPP metadata could not be fetched, it is set to 0 (line 17 - 18). File name, URL, MPP and dictionary are passed to the ASV, which is rendered with the given information (line 19).

```
1  @app.route('/wsi/<path:file_path>.dzi')
2  def index_dzi(file_path):
3    file_name = file_path + '.dzi'
4    slide_url = '/wsi/' + file_name
5    # read dzi file
6    try:
7      with open('static/wsi/' + file_path + '_files/metadata.txt') as
        file:
8        mpp_x = 0
9        mpp_y = 0
10       metadata = file.read().split('\n')
11       for property in metadata:
12         if openslide.PROPERTY_NAME_MPP_X in property:
13           mpp_x = property.split(': ')[1]
14         elif openslide.PROPERTY_NAME_MPP_Y in property:
15           mpp_y = property.split(': ')[1]
16       slide_mpp = (float(mpp_x) + float(mpp_y)) / 2
17   except IOError:
18     slide_mpp = 0
19   return render_template('as_viewer.html', slide_url=slide_url,
       slide_mpp=slide_mpp, file_name=file_name, dictionary=
       getDictionary(file_name))
```

index_wsi()

When the client requests a proprietary WSI (URL *does not* end in ".dzi"), `index_wsi()` renders an ASV and passes the necessary information (slide URL, file name, MPP and dictionary) to it. Furthermore, it wraps a DZG around the proprietary WSI and adds that to the WSGI object.

Line 1 - 8 create a map with the optional DZG parameters (compare tab. 4.3) and turn them into a dictionary. Line 9 reads the proprietary WSI. A DZG with the supplied parameters[2] is created, which wraps the proprietary slide object to add Deep Zoom support (line 9 - 12). The created DZG is added to the WSGI object (line 10). Line 13 - 18 fetch associated images, the metadata (line 14), wrap the associated images with a DZG of their own and add this, together with the metadata, to the WSGI object. Line 19 - 24 fetch the MPP

[2]Compare tab. 4.6

metadata and calculate the average MPP (or set it to 0, if not found). Line 25 creates a URL for the DZG object with Flasks `url_for(endpoint, **values)` function. This URL is passed, together with the MPP, file path and dictionary, to an ASV which then gets rendered (line 26).

```
1  @app.route('/wsi/<path:file_path>')
2  def index_wsi(file_path):
3    config_map = {
4      'DEEPZOOM_TILE_SIZE': 'tile_size',
5      'DEEPZOOM_OVERLAP': 'overlap',
6      'DEEPZOOM_LIMIT_BOUNDS': 'limit_bounds',
7    }
8    opts = dict((v, app.config[k]) for k, v in config_map.items())
9    slide = open_slide('static/wsi/' + file_path)
10   app.slides = {
11     SLIDE_NAME: DeepZoomGenerator(slide, **opts)
12   }
13   app.associated_images = []
14   app.slide_properties = slide.properties
15   for name, image in slide.associated_images.items():
16     app.associated_images.append(name)
17     slug = slugify(name)
18     app.slides[slug] = DeepZoomGenerator(ImageSlide(image), **opts)
19   try:
20     mpp_x = slide.properties[openslide.PROPERTY_NAME_MPP_X]
21     mpp_y = slide.properties[openslide.PROPERTY_NAME_MPP_Y]
22     slide_mpp = (float(mpp_x) + float(mpp_y)) / 2
23   except (KeyError, ValueError):
24     slide_mpp = 0
25   slide_url = url_for('dzi', slug=SLIDE_NAME)
26   return render_template('as_viewer.html', slide_url=slide_url,
       slide_mpp=slide_mpp, file_name=file_path, dictionary=
       getDictionary(file_name))
```

dzi(slug)

If `index_wsi()` was called before, a URL was generated for the WSI. This URL will be requested from the ASS by OpenSeadragon, which causes `slug(dzi)` to be called. `slug(dzi)` creates the DZI metadata and returns it to OpenSeadragon.

The *dzi* parameter is the slide URL generated in `index_wsi`.

Line 3 retrieves the format for the individual Deep Zoom tiles. Line 4 - 7 try to create a response with the requested DZI's metadata (via the DZGs `get_dzi(format)` function[3], line 5). If a response can not be created (because the requested DZG is unknown), a "404 Not Found" HTTP status code will be returned instead.

[3] Compare subsection 4.4.2

```
@app.route('/<slug>.dzi')
def dzi(slug):
  format = app.config['DEEPZOOM_FORMAT']
  try:
    resp = make_response(app.slides[slug].get_dzi(format))
    resp.mimetype = 'application/xml'
    return resp
  except KeyError:
    # Unknown slug
    abort(404)
```

tile(slug, level, col, row, format)

If a response for OpenSeadragon was created via `slug(dzi)`, OpenSeadragon will request the individual image tiles in such a way, that, through the use of the route() decorator, `tile(slug, level, col, row, format)` will be called.

As in `slug(dzi)`, the *slug* parameter is the slide URL generated in `index_wsi`. The parameters *level*, *col* and *row* describe the DZI level and address of the requested image tile. *format* is the image format of the tile.

If the format is not JPEG or PNG, the ASS return a "404 Not Found" HTTP status code (line 3 - 6).

If the format is either JPEG or PNG, the requested tile is generated through the use of the DZGs `get_tile(level, address)` function (line 8). If it was not possible to generate the tile, a "404 Not Found" HTTP status code will be returned.

The generated tile is then saved into a PIL image object[4], stored in either a JPEG or PNG image and returned as response to OpenSeadragon (line 15 - 19).

```
@app.route('/<slug>_files/<int:level>/<int:col>_<int:row>.<format>'
    )
def tile(slug, level, col, row, format):
  format = format.lower()
  if format != 'jpeg' and format != 'png':
    # Not supported by Deep Zoom
    abort(404)
  try:
    tile = app.slides[slug].get_tile(level, (col, row))
  except KeyError:
    # Unknown slug
    abort(404)
  except ValueError:
    # Invalid level or coordinates
    abort(404)
  buf = PILBytesIO()
  tile.save(buf, format, quality=app.config['DEEPZOOM_TILE_QUALITY'
    ])
  resp = make_response(buf.getvalue())
  resp.mimetype = 'image/%s' % format
  return resp
```

[4] See `http://pillow.readthedocs.io/en/3.3.x/reference/Image.html`

saveJson()

When the client sends JSON data to save, the `saveJson()` function is called.

The associated request is a POST request, thus the posted data must be extracted. This can be done via Flasks *request object* (line 3 - 5). The file path will be transmitted as *"source"*, the content to save as *"json"*.

If there is content to save (line 6), it will be written into the provided file. If the file does not exist yet, it will be created (line 7 - 8).

```
1 @app.route('/saveJson', methods=['POST'])
2 def saveJson():
3   dict = request.form
4   source = dict.get('source', default='')
5   json = dict.get('json', default='{}').encode('utf-8')
6   if len(source) > 0:
7     with open('static/' + source, 'w+') as file:
8       file.write(json)
9   return 'Ok'
```

loadJson()

When the client requests JSON data, `loadJson()` is called.

The source of the JSON data is passed in the URL as parameter (*"src=[path to source]"*). The src parameter can be extracted via Flasks request object[5] (line 3). If the provided source is a file, the content will be read and returned as JSON data (line 4 - 7). Otherwise an empty JSON list is returned (line 9).

```
1 @app.route('/loadJson')
2 def loadJson():
3   source = 'static/wsi/' + request.args.get('src', '')
4   if os.path.isfile(source):
5     with open(source, 'r') as file:
6       content = file.read()
7       return jsonify(content)
8   else:
9     return jsonify('[]')
```

createDictionary()

When the client requests the creation of a new dictionary, `createDictionary()` is called. The name of the new dictionary and the associated WSI is passed as URL parameter (*name= [name]* and slide=[WSI file name]).

Once the names of dictionary and WSI were extracted (line 3- 5), the function checks if a dictionary with the provided name already exists (6 - 8). If so, *"error"* is returned. Otherwise a new, empty dictionary is created (line 10 - 11). To associate the newly created dictionary to the current WSI, a key-value entry ([WSI name]:[dictionary name]) is added to the slideDictionaries.json file (13 - 17).

[5] Compare subsection 4.4.1

As response, the name and path of the newly created dictionary are returned (line 19 - 20).

```
1  @app.route('/createDictionary')
2  def createDictionary():
3    name = request.args.get('name', '')
4    slide = request.args.get('slide')
5    path = 'static/dictionaries/' + name
6    if os.path.isfile(path):
7      # dictionary already exists
8      return 'error'
9    else:
10     with open(path, 'w+') as dictionary:
11       dictionary.write("[]")
12
13   with open(SLIDE_DICTIONARIES, 'r') as file:
14     dictioary_map = json.loads(file.read())
15   dictioary_map[slide] = name
16   with open(SLIDE_DICTIONARIES, 'w') as file:
17     file.write(json.dumps(dictioary_map))
18
19   respone = '{"name":"' + name + '", "path":"/' + path + '"}';
20   return respone
```

loadDictionary(dictionary)

`loadDictionary(dictionary)` is called when the client requests to load a dictionary. This happens either after the creation of a new one or after switching dictionaries.

The function first builds the file path (line 3) and then checks, if a file can be found at the specified location (line 4). If not, a "404 Not Found" HTTP status code is returned (line 6). If a file could be found, its content is turned into a JSON compliant string and send to the client (line 8 - 10).

```
1  @app.route('/static/dictionaries/<dictionary>')
2  def loadDictionary(dictionary):
3    dictionary = 'static/dictionaries/' + dictionary
4    if os.path.isfile('/' + dictionary):
5      # no dictionary found
6      return '404'
7    else:
8      # return dictionary
9      with open(dictionary, 'r') as file:
10     return json.dumps(file.read())
```

getDictionaries()

The `getDictionaries()` function is called, when the client requests a list of all available dictionaries.

If no dictionaries could be found, "-1" will be returned, otherwise a JSON list of all available dictionaries.

```
@app.route('/getDictionaries')
def getDictionaries():
  dir = 'static/dictionaries/'
  if os.path.isfile(dir):
    # no dictionaries found
    return '-1'
  else:
    # return dictionaries
    return json.dumps(os.listdir(dir))
```

switchDictionary()

`switchDictionary()` is used to change the association between a dictionary and a WSI in the slideDictionary.json file.

The name of the WSI (slide) and dictionary (name) are extracted via Flask's request object (line 3 - 4). The slideDictionaries.json file is read and turned into a python dictionary. [slide] and [name] are then added to the dictionary (or updated, if the pair should already be in there), which is then written back into the file (line 6 - 10).

```
@app.route('/switchDictionary')
def switchDictionary():
  name = request.args.get('name', '')
  slide = request.args.get('slide')

  with open(SLIDE_DICTIONARIES, 'r') as file:
    dictioary_map = json.loads(file.read())
  dictioary_map[slide] = name
  with open(SLIDE_DICTIONARIES, 'w') as file:
    file.write(json.dumps(dictioary_map))

  return '200'
```

runSegmentation()

The `runSegmentation()` function is called, when the client tags a POI. It imports the python script provided in the configuration file[6] as module and calls the module's `run(x,y)` function.

The function uses Flasks *request object* to acquire the provided x and y coordinates from the provided URL (line 3 & 4). It then opens the configuration file and extracts the name of the segmentation script (line 5 - 7). If no name was provided for the script (string is empty) the server will print an error message and return with a "404 Not Found" HTTP status code (line 8 - 10).

If a script name was provided and successfully extracted, it is imported as python module (line 12). If the import was successful, the script's run method is called and the returned contour will be passed back to the server as JSON data (line 13 - 14).

[6] Compare tab. 4.5 in subsection 4.4.3.

If the script could not be imported, a error message will be printed by the server and a "404 Not Found" HTTP status code is returned (line 15 - 17).

```
1  @app.route("/runSegmentation")
2  def runSegmentation():
3    x = request.args.get('x', '0')
4    y = request.args.get('y', '0')
5    with open("static/configuration.json", 'r') as file:
6      config = json.loads(file.read())
7      module_name = config.get("segmentationScript")
8    if(len(module_name) == 0):
9      print("ERROR: no segmentation script provided in configuration
       file (configuration.json)!")
10     return "404"
11   try:
12     module = __import__("static.segmentation.%s" % (module_name),
       fromlist=["segmentation"])
13     contour = module.run(x,y)
14     return json.dumps(contour)
15   except ImportError:
16     print("ERROR: provided segmentation script (" + module_name)
       not found!")
17     return "404"
```

B.2 Annotation Service Viewer

Since the ASV has >2,000 lines of code, there will be no documentation of every single line of code. Instead, a brief description of every function is provided. The implementation of the ASV can be found on the disc at the end of this thesis (see appendix A.1).

B.2.1 Initialization functions

function init(file_name, url, mpp)

Parameters:
file_name: name of the requested WSI
url: URL to the DZI's metadata file
mpp: microns per pixel of the requested WSI

The `init(file_name, url, mpp)` is called by the *as_viewer.html* to start the initialization of the ASV JavaScript. The parameters are served by the ASS. *file_name* describes the name of the WSI, *url* contains the URL to the DZI metadata file and *mpp* equals the microns per pixel of the requested WSI (if that information could be retrieved from the metadata, 0 otherwise).

The function requests the content of the configuration file (via `loadConfiguration()`) from the ASS and creates the OSDV (via `initAnnotationService()`, see fig. B.1).

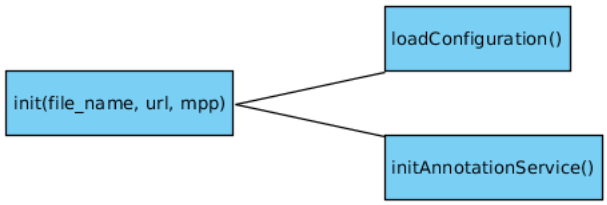


Figure B.1: Call hierarchy of `init(file_name, url, mpp)`

function initAnnotationService()

`initAnnotationService()` selects the navigation tool (via `selectTool()`), creates the OSDV and its scalebar. It then opens the tile source at the URL specified in the `init(file_path, url, mpp)` function. Additionally, event handlers to are set up to react to:

- mouse interaction
- opening of a WSI
- zooming

Furthermore, the toolbar is initialized.

Once a tile source was opened, the AO is initialized (via `initAnnotationOverlay()`) and the saved annotations are requested (via `loadJson()`, see fig. B.2).

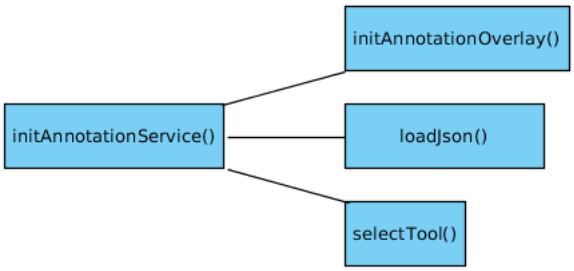


Figure B.2: Call hierarchy of `initAnnotationService()`

function initAnnotationOverlay()

`initAnnotationOverlay()` creates the ASV's AO. The AO is transformed to the size of the OSDV via the `transform()` function.

B.2.2 Data management functions

function saveConfig()

`saveConfig()` calls `saveJson(json, filePath)` to save the current configuration to the configuration file (see fig. B.3).

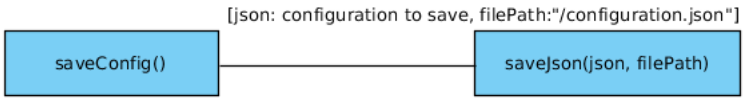


Figure B.3: Call hierarchy of `saveConfig()`

function loadConfiguration()

`loadConfiguration()` requests and parses the content of the configuration file from the ASS. Furthermore, it requests the list of available dictionaries (via `getDictionaryList()`) and loads the content of the dictionary specified in the configuration (via `loadDictionary(path)`, see fig. B.4).

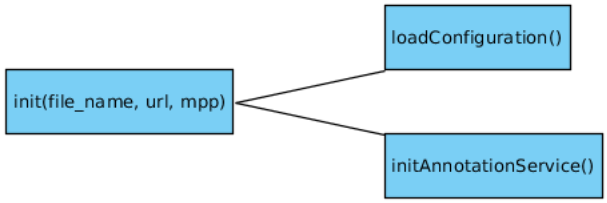


Figure B.4: Call hierarchy of `loadConfiguration()`

function getDictionaryList()

`getDictionaryList()` requests a list of all dictionaries from the ASS. The received list is then added as a clickable list to the toolbar.

function saveDictionary()

`saveDictionary()` calls `saveJson(json, filePath)` to save the current dictionary to the currently active dictionary's file (see fig. B.5).

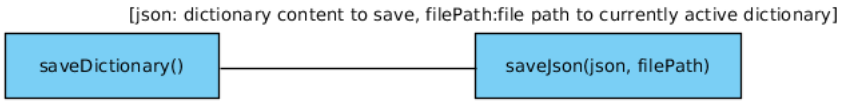


Figure B.5: Call hierarchy of `saveDictionary()`

function loadDictionary(path)

Parameters:
path: file path of the requested dictionary

`loadDictionary(path)` requests the content of the dictionary file at *path* from the ASS. The response is either a list of entries, which are then added to the list of available labels in the toolbar (via `appendLabelsToList()`), or a -1, if no dictionary was found. In this case, the ASV forces the user to create a new, empty dictionary (via `createNewDictionary(isCancelable)`, see fig. B.6).

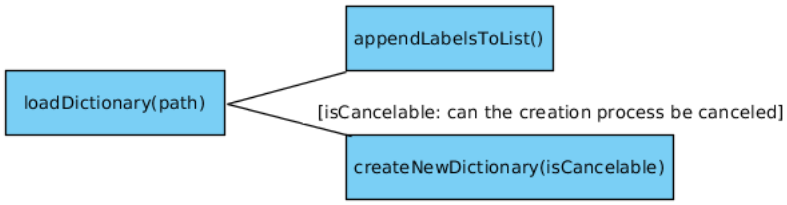


Figure B.6: Call hierarchy of `loadDictionary(path)`

function saveJson(json, filePath)

Parameters:
json: json data to save
filePath: path to the file where json should be saved

`saveJson(json, filePath)` sends a POST request to the ASS with the provided json data to save and the path of the file to save the data to.

function saveRegions()

`saveRegions()` calls `saveJson(json, filePath)` to save the ASV's annotations to a file (see fig. B.7).

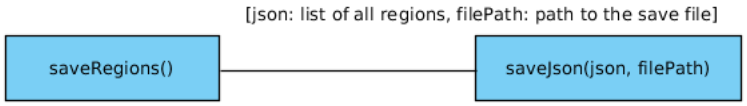


Figure B.7: Call hierarchy of `saveRegions()`

function loadJson()

`loadJson()` requests the saved annotations for the provided WSI. If an annotation file could be loaded by the ASS, a list of region data is returned. Each entry of the region data list is then turned into an actual region and added to the region list (via `newRegion(arg, imageNumber)`, see fig. B.8).

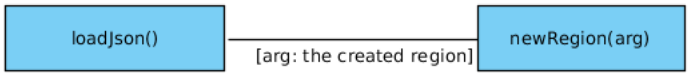


Figure B.8: Call hierarchy of `loadJson()`

function createNewDictionary(isCancelable)

Parameters:
isCancelable: 1 if the process can be canceled without providing a valid dictionary name, 0 otherwise

`createNewDictionary(isCancelable)` opens a prompt and asks the user to provide a name for the dictionary to create. If *isCancelable* is true (1), the prompt can be closed and the creation process is canceled. If it is false (0), the prompt will be shown until a valid name was provided.

After creation of the new dictionary it is selected as active one (and its empty content is loaded via `loadDictionary(path)`) and the list of available dictionaries is updated (via `getDictionaryList()`, see fig. B.9).

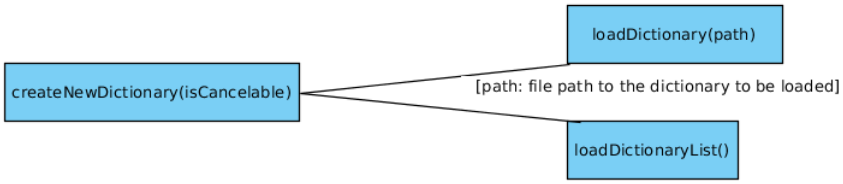


Figure B.9: Call hierarchy of `createNewDictionary(isCancelable)`

B.2.3 GUI functions

function selectTool()

`selectTool()` changes the mouse cursor to the icon of the currently selected tool.

function transform()

`transform()` resizes the AO to fit the view bounds of the OSDV.

function appendLabelsToList()

`appendLabelsToList()` first clears the currently shown list of labels. Then it iterates of the list of available labels (from the configuration) and creates a new, clickable label entry and adds it to the label list (via `appendLabelToList(label)`. Once all labels are created, the first entry is selected automatically (via `selectNextLabel()`, see fig. B.10).

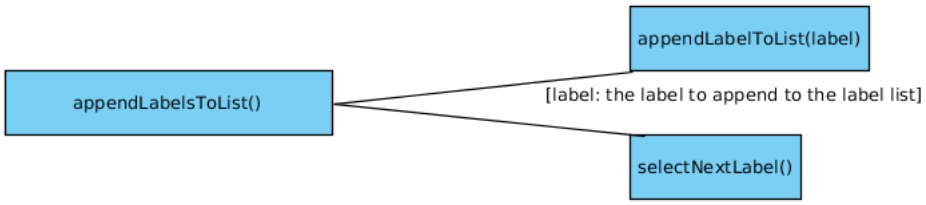


Figure B.10: Call hierarchy of `appendLabelsToList()`

function appendLabelToList(label)

Parameters:
label: name of the new label `appendLabelToList(label)` creates a new, clickable label for label list in the toolbar. It adds a click listener (via `singleClickOnLabel()`) to it and then selects it (via `selectLabel(el)`, see fig. B.11).

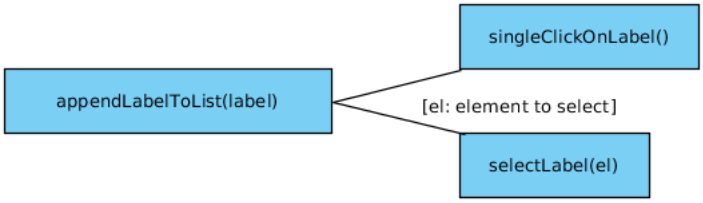


Figure B.11: Call hierarchy of `appendLabelToList(label)`

function selectNextLabel()

`selectNextLabel()` selects the next label in the toolbar list. If the currently selected label is the last one, the first entry is selected (via `selectLabel(el)`, see fig. B.12).

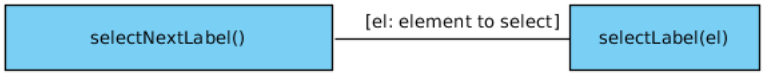


Figure B.12: Call hierarchy of `selectNextLabel(label)`

function selectLabel(el)

Parameters:
el: HTML element to select

`selectLabel(el)` selects the provided element *el* from the toolbar label list.

function newLabel()

`newLabel()` creates a new label for the toolbar's label list. The created label is also added to the dictionary and persisted there (via `saveDictionary()`). The function automatically generates a uid (via `uniqueID()`) and color (via `regionHashColor(label)`, see fig. B.13).

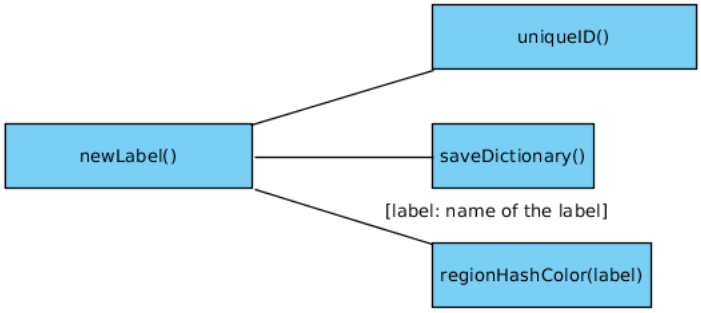


Figure B.13: Call hierarchy of `newLabel()`

function toggleDictPicker()

`toggleDictPicker()` toggles the visibility of the list of available dictionaries.

function dictListClick(index)

Parameters:
index: index of the selected dictionary

`dictListClick(index)` switches the currently active dictionary with the selected one, overwrites the currently active dictionary value in the configuration file (via `saveJson(json, filePath)`) and loads the entries from the new dictionary (via `loadDictionary(path)`, see fig. B.14).

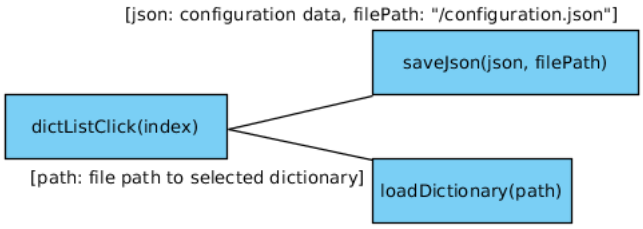


Figure B.14: Call hierarchy of `dictListClick(index)`

function annotationStyle(label)

Parameters:
label: label whose style to modify

`annotationStyle(label)` is used to manipulate the style values (stroke width and color, region color, alpha value) for the provided label.

function help(show)

Parameters:
show: 1 to show help tooltip, 0 to hide it

`help(show)` either shows or hides the help tooltip, depending on the provided value for show.

function toggleMenu()

`toggleMenu()` toggles visibility of the toolbar.

B.2.4 Region functions

function newRegion(arg)

Parameters:
arg: argument object (either a complete region or path and coordinate information)

`newRegion(arg)` creates a new region from the information provided in *arg* and adds it to the region list.

function selectRegion(reg)

Parameters:
reg: region to select

`selectRegion(reg)` selects the provided region.

function deselectRegion(reg)

Parameters:
reg: region to deselect

`deselectRegion(reg)` deselects the provided region.

function findContextRegion(region1)

Parameters:
region1: region to find the context to

`findContextRegion(region1)` finds the context of the provided region. The label of each region that crosses, touches, surrounds or is surrounded by *region1* is considered as context. Too avoid data redundancy, each label will only be added once to the context of region1 (via `isRegionAlreadyReferenced(region1, region2)`, see fig. B.15).

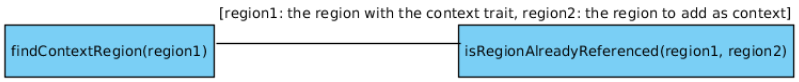


Figure B.15: Call hierarchy of `findContextRegion()`

function isRegionAlreadyReferenced(region1, region2)

Parameters:
region1: region with the context to check
region2: region whose label is checked for

`isRegionAlreadyReferenced(region1, region2)` checks if region1's context already contains region2's label.

function removeRegion(reg)

Parameter:
reg: the region to remove

`removeRegion(reg)` deletes the provided region.

function findRegionByUID(uid)

Parameter:
uid: unique id of the region to find

`findRegionByUID(uid)` finds a region with the provided id.

function toggleRegions(uid)

Parameters:
uid: the label uid of the regions to toggle in their visibility

`toggleRegions(uid)` turns all regions of the corresponding label (in-)visible. If one of the regions to be toggled is currently selected it is deselected (via `deselectRegion(reg)`, see fig. B.16).

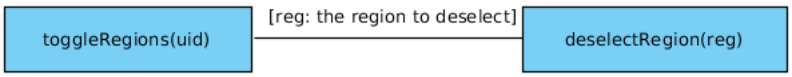


Figure B.16: Call hierarchy of `toggleRegions(uid)`

function toggleAllRegions()

`toggleAllRegions()` toggles the visibility of all regions.

B.2.5 Interaction functions

function singleClickOnLabel(event)

Parameters:
event: the click event

`singleClickOnLabel(event)` handles the click on a label in the toolbar's label list. Depending on what element inside the label is clicked, the following happens:

- the visibility of all regions with the corresponding label is toggled on/off (via `toggleRegions(uid)`)
- a window to open the annotation style (stroke width and color, alpha value, region color) is shown (via `changeRegionAnnotationStyle(uid)`)
- the clicked label is selected (via `selectLabel(el)`)

See fig. B.17 for the call hierarchy of `singleClickOnLabel(event)`.

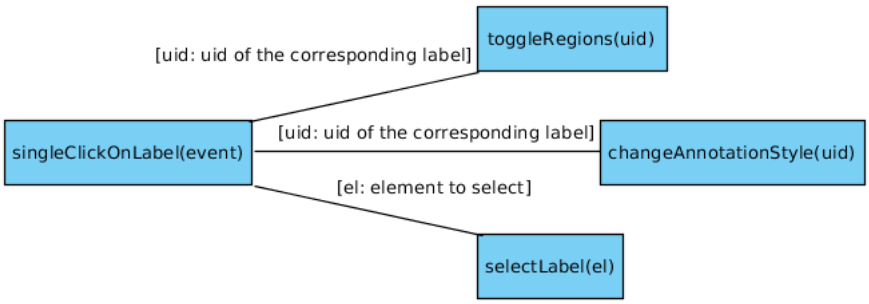


Figure B.17: Call hierarchy of `singleClickOnLabel(event)`

function changeRegionAnnotationStyle(uid)

Parameters:
uid: the uid of the corresponding label

`changeRegionAnnotationStyle(uid)` changes the annotation style (storke width and color, region color, region alpha value) for all regions corresponding to the label uid.

function addPoi(event)

Parameters:
event: the click event

`addPoi(event)` requests the server to run the segmentation script and provides it with the x and y coordinate the POI was placed at. Once the server returns with a valid answer (the image coordinates of the ROIs enclosing path), the coordinates are converted from image coordinates into AO coordinates (via `imgToPathCoordinates(point)`) and the ROI is enclosed by a path drawn along those AO coordinates. Then, a new region is created (via `newRegion(arg)`) and its checked for context (via `findContextRegion(region1)`).

If the ASS returns a "404 Not Found", an alert is shown and the currently selected POI tool is reset to the navigation tool (via `clearToolSelection()`).

Fig. B.18 shows the call hierarchy of `addPoi(event)`.

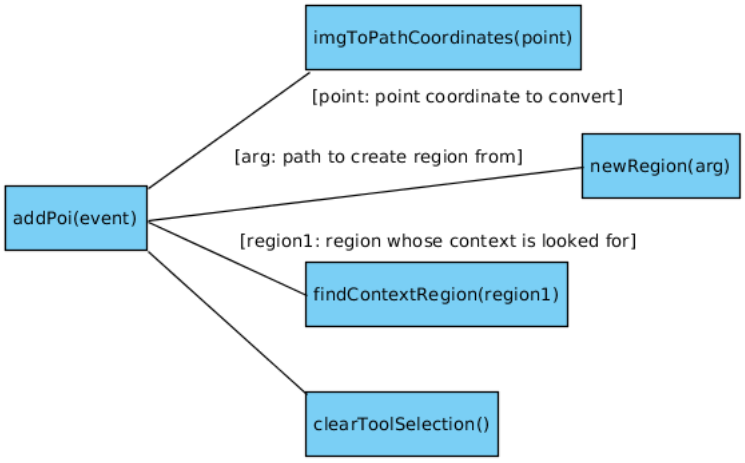


Figure B.18: Call hierarchy of `addPoi(event)`

B.2.6 Internal functions

UniqueID()

`UniqueID()` creates a unique id and returns it.

function regionHashColor(name)

Parameters:
name: name of a label

`regionHashColor(name)` generates a color (rgba, with a = default alpha from configuration) from the hash value of the provided name.

function convertPathToImgCoordinates(point)

Parameters:
point: point coordinate to convert

`convertPathToImgCoordinates(point)` converts a point coordinate from the AO to a point coordinate in the OSDV's image.

function convertImgToPathCoordinates(point)

Parameters:
point: point coordinate to convert

`convertImgToPathCoordinates(point)` converts a point coordinate from the OSDV's image to a point coordinate in the AO.

function mouseDown(x,y)

Parameters:
x: x coordinate of when left mouse button was pushed down
y: y coordinate of when left mouse button was pushed down

`mouseDown(x,y)` listens to mouse click events. It is called, once the mouse button is pushed down. The function handles every interaction the following interactions:

- create region in free hand and polygon drawing mode (via `newRegion(arg)`)
- add a new segment to a path in polygon drawing mode
- close a region's path in polygon mode, once a segment is clicked the second time (via `finishDrawingPolygon(closed)`)
- select a clicked region (via `selectRegion(reg)`)

See fig. B.19 for the call hierarchy of `mouseDown(x,y)`.

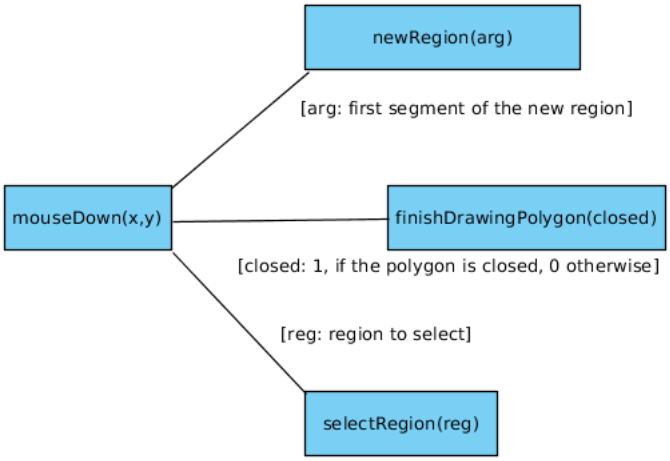


Figure B.19: Call hierarchy of `mouseDown(x,y)`

function mouseDrag(x,y,dx,dy)

Parameters:
x: x coordinate with current mouse position
y: y coordinate with current mouse position
dx: delta of current and former mouse position in x
dy: delta of current and former mouse position in y

`mouseDrag(x,y,dx,dy)` handles all mouse interaction that happens with a pushed down left mouse button:

- add new segment to path in free hand drawing mode (including conversion image coordinate → AO coordinate and AO coordinate → image coordinate[7], see fig. B.20)
- move region
- move segment in region

[7] See subsection 4.5.1 - Paper.js.

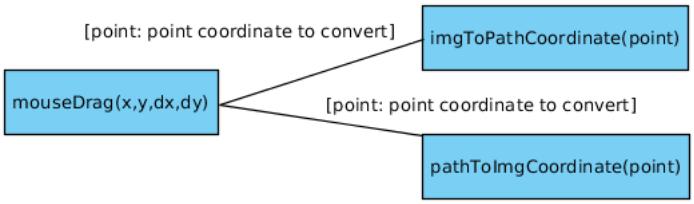


Figure B.20: Call hierarchy of `mouseDrag(x,y,dx,dy)`

function mouseUp()

When in free hand drawing mode, the `mouseUp()` function closes a region's path, once the left mouse button is released (including conversion as in the first item of `mouseDrag(x,y,dx,dy)`).

When in distance measurement mode, the distance tooltip is removed upon release of the left mouse button.

function finishDrawingPolygon(closed)

Parameters:
closed: 1 if path should be closed, 0 otherwise

`finishDrawingPolygon(closed)` finishes the drawing process of a region's path in polygon drawing mode. If closed is true (1), the first and last segment will be connected.

function getDistance()

`getDistance()` calculates the euclidian distance d between pixels p_a and p_b (see eq. B.1 [74]) of a special path called *ruler* that is exclusively used for the distance measurement tool.

$$d = \sqrt{(p_b.x - p_a.x)^2 + (p_b.y - p_a.y)^2} \tag{B.1}$$

function clearToolSelection()

`clearToolSelection()` resets the mouse cursor to the default icon.

B.2.7 Key listener

$(document).keydown(function(e))

Parameters:
e: key event

`$(document).keydown(function(e))` is used to handle the tool selection via hotkeys. A tool is selected depending on the keys pressed (via `selectToolOnKeyPress(id)`), see fig. B.21).

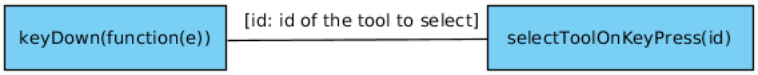


Figure B.21: Call hierarchy of `keydown(function(e))`

$(document).keyup(function(e))

`$(document).keyup(function(e))` resets the selected tool to the navigation tool upon release of a hotkey (via `clearToolSelection()`, see fig. B.22).

Figure B.22: Call hierarchy of `keyup(function(e))`

function selectToolOnKeyPress(id)

Parameters:
id: id of the tool to select

`selectToolOnKeyPress(id)` selects the tool corresponding to the provided id and changes the mouse cursor accordingly (via `selectTool()`, see fig. B.1).

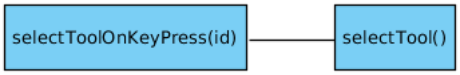


Figure B.23: Call hierarchy of `selectToolOnKeyPress(id`

Appendix C

Tessellation Service Documentation

The following sections document the functions of the TS in detail. The TS can be found on the disc included at the end of this thesis (see appendix A.1).

C.1 Main

run(input)

As stated in section 5.2, each input element can be a file or a dictionary. Therefore, the individual entries must be examined. The `run(input)` function does exactly that. Each element's type (file or directory) is checked. If it is a file, the extraction process is started (line 8). If it is a directory, its files and subdirectories are extracted (line 5).

```
1 def run(input):
2   for element in input:
3     # input is folder:
4     if(os.path.isdir(element)):
5       files_from_dir(element)
6     # input is file:
7     elif(os.path.isfile(element)):
8       regions_from_file(element)
```

files_from_dir(element)

`files_from_dir(element)` gets the content of the provided directory (line 4). For every subdirectory found, `files_from_dir(element)` is called recursively, until the end of each directory tree is reached. An exception from this are the DZI *"_files"* directories, since they only contain the corresponding DZI's tessellated image levels (line 6 - 8). If a file was found, the extraction process is started for it (line 9 - 10).

```
def files_from_dir(dir):
  if not dir.endswith('/'):
    dir = dir + '/'
  contents = os.listdir(dir)
  for content in contents:
    if os.path.isdir(dir + content):
      if not content.endswith('_files'):
        files_from_dir(dir + content)
    else:
      regions_from_file(dir + content)
```

regions_from_file(element)

Since the implementation for WSI and DZI differs, a distinction must be made, which of two approaches is chosen. `regions_from_file(element)` makes this distinction and starts the extraction process for the provided element. If the provided element is not of the DZI type, it is checked if it is a valid WSI (via `is_supported(file)`, see line 5) before the process is started. This way, only DZI and WSI will be targeted with extraction attempts.

```
def regions_from_file(file):
  if file.endswith('.dzi'):
    dzi(file)
  else:
    if(is_suppoted(file)):
      wsi(file)
```

C.2 WSI

wsi(file)

`wsi(file)` extracts the regions from a WSI. To do so, the provided file must have an associated JSON file which was annotated with the dictionary provided as argument when the TS was started[1].

OpenSlide is used to read the provided WSI (see line 2). The file name is extracted from the file path (see line 3). The saved regions are loaded and parsed via `read_json(path)` function (see line 4).

For each parsed region, an image is extracted together with a metadata file. If the tessellation parameter was provided when TessellationService.py was called, `tessellate_wsi(slide, slide_name, region)` is called. Otherwise, the BB is calculated via `get_bounding_box(region)`. If the -r parameter was provided, the BB will be adjusted to fit the supplied image ratio (via `resize_bounding_box(bounding_box)`, see line 10 - 12). Then, the BB's position inside the baseline image of the provided WSI is determined (see line 13). Afterwards, the size of the BB is calculated for both dimension (see line 14). OpenSlide's `read_region(location, level, size)` is used to access the ROI

[1] Compare subsection 5.3.1

located at the specified position (the baseline image is at level 0 [16], see line 15). The extracted image is then saved via the `save_image(image, region, slide_name)` function.

Once every region was extracted, the OpenSlide object is closed again.

```
1  def wsi(file):
2    slide = OpenSlide(file)
3    slide_name = file.split('/')[-1]
4    regions = read_json(file + '_' + DICTIONARY)
5
6    for region in regions:
7      if(TESSELLATE):
8        tessellate_wsi(slide, slide_name, region)
9      else:
10       bounding_box = get_bounding_box(region)
11       if(RESIZE):
12         bounding_box = resize_bounding_box(bounding_box)
13       location = (bounding_box['x_min'], bounding_box['y_min'])
14       size = (bounding_box['x_max'] - bounding_box['x_min'],
     bounding_box['y_max'] - bounding_box['y_min'])
15       image = slide.read_region(location, 0, size)
16       save_image(image, region, slide_name)
17
18   slide.close()
```

tessellate_wsi(slide, slide_name, region)

`tessellate_wsi(slide, slide_name, region)` realizes the approximation of a ROI through tessellation[2]. To do so, it first gets the slide dimension to calculate the amount of virtual tiles necessary to cover the whole WSI (see line 2 - 4). Afterwards, a contour is created by iteration over every segment of a region's path and associating it with a virtual tile (see line 10 - 18). The result is a contour of virtual tiles.

In the next step, the RI is created. Each pixel in the RI corresponds to a virtual tile. Because of this relation between tiles and pixels, the created tile contour can be used to fill in the ROI in the RI via OpenCV's `cv2.drawContours(...)` function (see line 23). The provided parameters are:

- **cv_ref_img**: the RI
- [**contour**]: the list of contours to draw
- **0**: which contour to draw from the provided array (0 equals all)
- **(255, 255, 255)**: a tupel with the color in which the contour is drawn in (white)
- **-1**: the stroke width of the contour (-1 equals fill)

[2] Compare subsection 5.3.4

The result is the RI where every pixel corresponding to a virtual tile in the ROI is white. In the next step, the function iterates over every single pixel of the RI and extracts a ROI into an individual image, whenever the corresponding pixel is white (see line 27 - 37).

After all tiles were extracted, an associated metadata file is created (see line 40).

```
1  def tessellate_wsi(slide, slide_name, region):
2    n,m = slide.dimensions
3    m = m / TESSELLATE[HEIGHT]
4    n = n / TESSELLATE[WIDTH]
5
6    if SHOW:
7      ox = 999999
8      oy = 999999
9
10   contour = []
11   for coords in region.get('imgCoords'):
12   if SHOW:
13     if(coords.get('y') < oy): oy = coords.get('y')
14     if(coords.get('x') < ox): ox = coords.get('x')
15   x = int(coords.get('x') / TESSELLATE[WIDTH])
16   y = int(coords.get('y') / TESSELLATE[HEIGHT])
17   if [x, y] not in contour:
18     contour.append([x, y])
19
20   contour = np.asarray(contour)
21   ref_img = Image.new('RGB', (n,m))
22   cv_ref_img = np.array(ref_img)
23   cv2.drawContours(cv_ref_img, [contour], 0, (255,255,255), -1)
24   if SHOW:
25     dbg_img = Image.new('RGB', (n,m))
26   tiles = []
27   for i in xrange(0, m):
28     for j in xrange(0, n):
29       px = cv_ref_img[i,j]
30       if (px == [255, 255, 255]).all():
31         location = ((j) * TESSELLATE[WIDTH], (i) * TESSELLATE[
     HEIGHT])
32         size = TESSELLATE
33         tile = slide.read_region(location, 0, size)
34         tile_name = save_image(tile, region, slide_name, i, j)
35         tiles.append(tile_name.split('/')[-1] + '.jpeg')
36         if SHOW:
37           dbg_img.paste(tile, (j * TESSELLATE[WIDTH] - int(ox), i *
      TESSELLATE[HEIGHT] - int(oy)))
38   if SHOW:
39   dbg_img.show()
40   save_metadata(generate_file_name(region, slide_name), region,
       tiles)
```

is_supported(file)

`is_supported(file)` checks if a provided file is a proprietary WSI (see subsection 2.1.2).

```
def is_suppoted(file):
  ext = (file.split('.'))[-1]
  if(
    'bif' in ext or
    'mrxs' in ext or
    'npdi' in ext or
    'scn' in ext or
    'svs' in ext or
    'svslide' in ext or
    'tif' in ext or
    'tiff' in ext or
    'vms' in ext or
    'vmu' in ext
  ):
    return 1
  else:
    return 0
```

C.3 DZI

dzi(file)

`dzi(file)` extracts the regions from a DZI. Since OpenSlide can not be used with DZIs, the corresponding ROI must be stitched manually. To do so, the provided file's DZI metadata must be parsed to access the information about width, height, format and location of the highest resolution layer (see line 2 - 6). The saved regions are parsed with the `read_json(path)` function (see line 7).

For each parsed region, an image is extracted together with a metadata file, like in the WSI scenario. If the tessellation parameter was provided when TessellationService.py was called, `tessellate_dzi(dzi, slide_name, region)` is called. Otherwise, the BB is calculated via `get_bounding_box(region)`. The tiles containing the ROI are stitched together and cropped to size in the `create_image_from_tiles(dzi, bounding_box)` function. The resulting image is saved via the `save_image(image, region, slide_name)` function (see line 10 - 15).

```
def dzi(file):
  slide_name = file.split('/')[-1]
  with open(file, 'r') as dzi_file:
    content = dzi_file.read()
  root = ET.fromstring(content)
  dzi = {'tile_size': int(root.get('TileSize')), 'width': int(root
    [0].get('Width')), 'height': int(root[0].get('Height')), '
    tile_source': get_tile_source(file), 'format': root.get('Format
    ')}
  regions = read_json(file + '_' + DICTIONARY)

  for region in regions:
    if TESSELLATE:
      tessellate_dzi(dzi, slide_name, region)
```

```
12      else:
13      bounding_box = get_bounding_box(region)
14      image = create_image_from_tiles(dzi, bounding_box)
15      save_image(image, region, slide_name)
```

create_image_from_tiles(dzi, bounding_box)

`create_image_from_tiles(dzi, bounding_box)` stitches the tiles containing the ROI to extract together by utilizing the ROI's BB.

If the -r parameter was provided at the TS' execution, the BB will be adjusted to fit the provided image ratio. This also has influence on what tiles will be needed (see line 2 - 3). The stitching of the baseline image area is realized in `get_tiles_from_bounding_box(dzi, bounding_box)` (see line 4). Once the tiles are stitched, the offset of the ROI's BB inside the stitched image is calculated (see line 6 - 7), as well as its corners (see line 9 - 12).

This information is used to crop the stitched image to the size of the ROI's BB.

```
1  def create_image_from_tiles(dzi, bounding_box):
2    if(RESIZE):
3      bounding_box = resize_bounding_box(bounding_box)
4    tile_image = get_tiles_from_bounding_box(dzi, bounding_box)
5
6    offset_x = bounding_box['x_min']
7    offset_y = bounding_box['y_min']
8
9    x_min = bounding_box['x_min'] - offset_x
10   x_max = bounding_box['x_max'] - offset_x
11   y_min = bounding_box['y_min'] - offset_y
12   y_max = bounding_box['y_max'] - offset_y
13
14   return tile_image.crop((x_min, y_min, x_max, y_max))
```

get_tiles_from_bounding_box(dzi, bounding_box)

`get_tiles_from_bounding_box(dzi, bounding_box)` stitches together a baseline image area based on the provided BB.

To do so, the corners of the BB are converted to tile positions (see line 2 - 5). Then, a new image is created which will hold the needed tiles (see line 7). All needed tiles are iterated in width and height and pasted into the corresponding postion of the holding image (see line 9 - 12).

The stitched image is then returned.

```
1  def get_tiles_from_bounding_box(dzi, bounding_box):
2    x_min = bounding_box['x_min'] / dzi['tile_size']
3    x_max = bounding_box['x_max'] / dzi['tile_size']
4    y_min = bounding_box['y_min'] / dzi['tile_size']
5    y_max = bounding_box['y_max'] / dzi['tile_size']
6
7    stitch = Image.new('RGB', ((x_max-x_min+1) * dzi['tile_size'], (
       y_max-y_min+1) * dzi['tile_size']))
```

```
8
9     for i in range(x_min, x_max+1):
10      for j in range(y_min, y_max+1):
11        tile = Image.open(dzi['tile_source'] + str(i) + '_' + str(j)
      + '.' + dzi['format'])
12        stitch.paste(tile, ((i - x_min) * dzi['tile_size'], (j -
      y_min) * dzi['tile_size']))
13    return stitch
```

get_tile_source(file)

`get_tile_source(file)` finds the level with the highest resolution in a DZI's *"_files"* directory and builds a file path based on that.

```
1  def get_tile_source(file):
2    files_dir = file.replace('.dzi', '_files/')
3    layers = os.listdir(files_dir)
4    layers.remove('metadata.txt')
5    layers = map(int, layers)
6    return files_dir + str(max(layers)) + '/'
```

tessellate_dzi(dzi, slide_name, region)

`tessellate_dzi(dzi, slide_name, region)` works in the same way as `tessellate_wsi(slide, slide_name, region)`, except for the tile extraction (see line 26 - 35).

As in `dzi(file)`, a baseline image area must be stitched (see line 3). To extract a virtual tile from that stitched image, `tessellate_dzi(dzi, slide_name, region)` crops that stitched image to the size of the current virtual tile and saves this in an individual image.

Once all tiles are extracted, a metadata file is created (see line 43).

```
1  def tessellate_dzi(dzi, slide_name, region):
2    bounding_box = get_bounding_box(region)
3    tile_image = get_tiles_from_bounding_box(dzi, bounding_box)
4
5    offset_x = bounding_box['x_min']
6    offset_y = bounding_box['y_min']
7
8    n,m = tile_image.size
9
10   m = m / TESSELLATE[HEIGHT]
11   n = n / TESSELLATE[WIDTH]
12
13   contour = []
14   for coords in region.get('imgCoords'):
15   x = int((coords.get('x') - offset_x) / TESSELLATE[WIDTH])
16   y = int((coords.get('y') - offset_y) / TESSELLATE[HEIGHT])
17   if [x, y] not in contour:
18     contour.append([x, y])
19
20   contour = np.asarray(contour)
21   ref_img = Image.new('RGB', (n,m))
```

```
22    cv_ref_img = np.array(ref_img)
23    cv2.drawContours(cv_ref_img, [contour], 0, (255,255,255), -1)
24    if SHOW:
25      dbg_img = Image.new('RGB', tile_image.size)
26    tiles = []
27    for i in xrange(0, m):
28      for j in xrange(0, n):
29        px = cv_ref_img[i,j]
30        if (px == [255, 255, 255]).all():
31          tile = tile_image.crop((j * TESSELLATE[WIDTH] + (
      bounding_box['x_min'] % dzi['tile_size']),
32          i * TESSELLATE[HEIGHT] + (bounding_box['y_min'] % dzi['
      tile_size']),
33          j * TESSELLATE[WIDTH] + (bounding_box['x_min'] % dzi['
      tile_size']) + TESSELLATE[WIDTH],
34          i * TESSELLATE[HEIGHT] + (bounding_box['y_min'] % dzi['
      tile_size']) + TESSELLATE[HEIGHT]))
35          tile_name = save_image(tile, region, slide_name, i, j)
36          tiles.append(tile_name.split('/')[-1] + '.jpeg')
37          if SHOW:
38            dbg_img.paste(tile, (j * TESSELLATE[WIDTH], i *
      TESSELLATE[HEIGHT]))
39    if SHOW:
40      dbg_img.show()
41    save_metadata(generate_file_name(region, slide_name), region,
      tiles)
```

C.4 Utility

read_json(path)

`read_json(path)` is utilized to read a provided JSON file and parse its contents into a python dictionary (see line 2 - 6). If an error occurred in the extraction process, the user will be informed (see line 7 - 8).

```
1 def read_json(path):
2   try:
3     with open(path, 'r') as file:
4     str = (file.read())
5     data = json.loads(str.decode('utf-8'))
6     return data
7   except IOError:
8     print('Could not load saved annotations from ' + path)
```

save_metadata(name, region, *tiles)

`save_metadata(name, region, *tiles)` is used to save the metadata to a corresponding extracted image.

The metadata for regions extracted with the -t parameters differs from the ones without. Because of this, *"_tessellated"* is added to the files name (see line 2 - 3). If the -f parameter was provided (compare tab. 5.1), files of the same

name will be overwritten. Otherwise, an upwards counting, positive integer is added in parenthesis to the file name (see line 4 - 10).

A file stream is opened and the associated region's metadata is written in a file (see line 12 - 19). The metadata file is encoded in UTF-8 to support special characters.

```
1  def save_metadata(name, region, *tiles):
2    if len(tiles) > 0:
3      name = name + '_tessellated.metadata.json'
4    if not FORCE:
5      cnt = 0
6      while os.path.isfile(name):
7        cnt+=1
8      name = name + '(' + str(cnt) +')'
9    else:
10     image_name = name
11   name = name + '.metadata.json'
12   with open(name, 'w+') as file:
13     data = {'label': region.get('name'), 'zoom': region.get('zoom')
       , 'context': region.get('context')}
14   if len(tiles) > 0:
15     data['tiles'] = tiles
16   else:
17     data['image'] = image_name.split('/')[-1] + '.jpeg'
18   content = json.dumps(data, ensure_ascii=False)
19   file.write(content.encode('utf-8'))
```

generate_file_name(region, slide_name, *tiles)

`generate_file_name(region, slide_name, *tiles)` creates a file name and output path for the provided information. The generated file name is used for the metadata as well as the image file.

If the -o parameter was provided (compare tab. 5.1), the supplied output directory will be added to the file path. If the path does not exist, it is created (see line 2 - 5). The region's uid is added to the file name (see line 8). If the image is tessellated into tiles (**tiles* will be a non-empty tupel with the x and y position of the tile), the position of the tile will be added to the file name (see line 9 - 13).

When the -f parameter was not specified, an upwards counting, positive integer will be added to the file name (see line 12 - 16). The created file name is then returned (see line 17).

```
1  def generate_file_name(region, slide_name, *tiles):
2    if(OUTPUT):
3      dest = OUTPUT + region['name']
4    else:
5      dest = region['name']
6    if not os.path.exists(dest):
7      os.makedirs(dest)
8    name = dest + '/' + slide_name + '_' + str(region['uid'])
9    if len(tiles) > 0:
10     for entry in tiles:
11       name += "_" + str(entry)
```

```
12    if not FORCE:
13      cnt = 0
14      while os.path.isfile(name):
15        cnt+=1
16      name = name + '(' + str(cnt) +')'
17    return name
```

save_image(image, region, slide_name, *tiles

`save_image(image, region, slide_name, *tiles` saves an extracted image.

The file name is generated, either for a tiled or non-tiled image (see line 2 - 5). If -r was specified (compare tab. 5.1), the image will be resized according to the specified values with the provided interpolation method, if applicable. Otherwise, nearest neighbor interpolation is chosen by default (see line 6 - 7).

If the grayscale conversion is activated (-g parameter, compare tab. 5.1), the image will be converted to grayscale via the luma transform method (see eq. 5.1 in subsection 5.3.2).

The image is then saved and the path returned (see line 11 - 14). If the image is not tessellated, the metadata is saved automatically as well (see line 12 - 13).

```
1  def save_image(image, region, slide_name, *tiles):
2    if len(tiles) == 0:
3      name = generate_file_name(region, slide_name)
4    else:
5      name = generate_file_name(region, slide_name, tiles)
6    if RESIZE:
7      image = image.resize(RESIZE, INTERPOLATION)
8    # L = R * 299/1000 + G * 587/1000 + B * 114/1000
9    if GRAYSCALE:
10     image = image.convert('L')
11   image.save(name + '.jpeg', 'jpeg')
12   if len(tiles) == 0:
13     save_metadata(name, region)
14   return name
```

get_bounding_box(region)

`get_bounding_box(region)` creates the BB for a provided region, by iteration over its path's segments and saving the smallest and largest values for the x and y coordinate.

```
1  def get_bounding_box(region):
2    x_min = sys.float_info.max
3    x_max = sys.float_info.min
4    y_min = x_min
5    y_max = x_max
6    for coordinate in region.get('imgCoords'):
7      x = coordinate.get('x')
8      y = coordinate.get('y')
9      if(x >= x_max):
10       x_max = x
```

```
11      if(x < x_min) :
12        x_min = x
13      if(y >= y_max):
14        y_max = y
15      if(y < y_min) :
16        y_min = y
17
18    return {'x_max': int(np.ceil(x_max)), 'x_min': int(np.floor(x_min
        )),
19      'y_max': int(np.ceil(y_max)), 'y_min': int(np.floor(y_min))}
```

resize_bounding_box(region)

`resize_bounding_box(region)` adjusts the width:height ratio of a provided BB to match the ratio of the provided width and height, if the TS was run with the -r parameter (see tab. 5.1).

In order to do so, the width:height ratios of the -r values (**r_ratio**) and the BB must be calculated (see line 2 - 5). If the ratios are identical, no further adjustments are necessary (line 6 - 7). Depending on the ration of the provided width and height of the -r parameter, the following adjustments are made:

- **r_ratio = 1**: the BB is adjusted to a square (see line 9 - 13)
- **r_ratio <1**: the BB must be adjusted to rectangle that is wider than high (see line 20 - 32)
- **r_ratio >1**: the BB must be adjusted to rectangle that is higher than wide (see line 33 - 45)

If the BB is smaller than the specified values after the adjustment, the BB is scaled up to match the specified values via `scale_bounding_box(bounding_, scale)` (see line 48 - 55). If the scaling process creates a negative value for either one of the minimum values, the BB is moved in such a way, that the negative value will turn to 0 (see line 57 - 64).

```
1  def resize_bounding_box(bounding_box):
2    r_ratio = RESIZE[WIDTH] / float(RESIZE[HEIGHT])
3    bb_width = float(bounding_box['x_max'] - bounding_box['x_min'])
4    bb_height = float(bounding_box['y_max'] - bounding_box['y_min'])
5    bb_ratio = bb_width / bb_height
6    if r_ratio == bb_ratio:
7      return bounding_box
8    else:
9      if r_ratio == 1:
10       # target is square
11       s1 = bb_height/bb_width
12       s2 = bb_width/bb_height
13       scaled = min(bb_width, bb_height) * max(s1, s2) - min(
       bb_width, bb_height)
14       if(bb_width > bb_height):
15         bounding_box['y_min'] -= int(np.floor(scaled/2))
16         bounding_box['y_max'] += int(np.ceil(scaled/2))
17       else:
```

```
            bounding_box['x_min'] -= int(np.floor(scaled/2))
            bounding_box['x_max'] += int(np.ceil(scaled/2))
        elif r_ratio < 1:
          # target is higher than wide
          h_s = 1 / r_ratio
          if bb_height > (bb_width * h_s):
            # adjust width:
            w_new = (bb_height / h_s) - bb_width
            bounding_box['x_min'] -= int(np.floor(w_new/2))
            bounding_box['x_max'] += int(np.ceil(w_new/2))
          else:
            # adjust height:
            h_new = h_s * bb_width - bb_height
            bounding_box['y_min'] -= int(np.floor(h_new/2))
            bounding_box['y_max'] += int(np.ceil(h_new/2))
        else:
          # target is wider than high
          w_s = r_ratio
          if bb_width > (bb_height * w_s):
            # adjust height
            h_new = (bb_width / w_s) - bb_height
            bounding_box['y_min'] -= int(np.floor(h_new/2))
            bounding_box['y_max'] += int(np.ceil(h_new/2))
          else:
            # adjust width:
            w_new = w_s * bb_height - bb_width
            bounding_box['x_min'] -= int(np.floor(w_new/2))
            bounding_box['x_max'] += int(np.ceil(w_new/2))

        # check if bb is big enough
        bb_width = float(bounding_box['x_max'] - bounding_box['x_min'])
        if bb_width < RESIZE[WIDTH]:
          s = RESIZE[WIDTH] / bb_width
          bounding_box = scale_bounding_box(bounding_box, s)
        bb_height = float(bounding_box['y_max'] - bounding_box['y_min']
        ])
        if bb_height < RESIZE[HEIGHT]:
          s = RESIZE[HEIGHT] / bb_height
          bounding_box = scale_bounding_box(bounding_box, s)

        if(bounding_box['y_min'] < 0):
          dif = bounding_box['y_min'] * (-1)
          bounding_box['y_min'] += dif
          bounding_box['y_max'] += dif
        if(bounding_box['x_min'] < 0):
          dif = bounding_box['x_min'] * (-1)
          bounding_box['x_min'] += dif
          bounding_box['x_max'] += dif

        return bounding_box
```

scale_bounding_box(bounding_, scale)

`scale_bounding_box(bounding_, scale)` scale a provided BB by the provided scale in height and width. The ROI stays in the center of the BB when scaling up.

```
def scale_bounding_box(bounding_box, scale):
  bb_width = float(bounding_box['x_max'] - bounding_box['x_min'])
  add_w = (bb_width * scale) - bb_width
  bounding_box['x_min'] -= int(np.floor(add_w/2))
  bounding_box['x_max'] += int(np.ceil(add_w/2))

  bb_height = float(bounding_box['y_max'] - bounding_box['y_min'])
  add_h = (bb_height * scale) - bb_height
  bounding_box['y_min'] -= int(np.floor(add_h/2))
  bounding_box['y_max'] += int(np.ceil(add_h/2))

  return bounding_box
```

Bibliography

[1] A. Madabhushi A. Janowczyk. Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases. *J Pathol Inform*, 7(29), July 2016.

[2] L. Fei-Fei A. Karpathy. *Deep Visual-Semantic Alignments for Generating Image Descriptions*. Department of Computer Science, Standford University, 2015. http://cs.stanford.edu/people/karpathy/cvpr2015.pdf.

[3] AgileLoad. Performance symptoms and issues. http://www.agileload.com/performance-testing/performance-testing-methodology/performance-symptoms-and-issues. Accessed: 24.08.2016.

[4] Neuroanatomy Applied and Theoretical (NAAT). Microdraw. http://microdraw.pasteur.fr/index.html. Accessed: 23.08.2016.

[5] R. Berta. deepzoom. http://search.cpan.org/~drrho/Graphics-DZI-0.05/script/deepzoom. Accessed: 22.08.2016.

[6] G. Brandl. *Python Web Server Gateway Interface v1.0*, June 2016. https://hg.python.org/peps/file/tip/pep-0333.txt.

[7] J. Bugwadia. Microservices: Five architectural constraints. http://www.nirmata.com/2015/02/microservices-five-architectural-constraints/, February 2015. Accessed: 12.08.2016.

[8] D. Siganos C. Stergiou. Neural networks. Technical report, Imperial College London, 1995. https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#Conclusion.

[9] L. Dykes C. Ullmann. *Beginning Ajax*. Wiley Publishing, Inc., 2007. http://crypto.cs.mcgill.ca/~simonpie/webdav/ipad/EBook/Programmation/Beginning%20Ajax%20(Programmer%20to%20Programmer).pdf.

[10] A. Clifford. *The Math Book: From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics*. Sterling, 2007.

[11] T. Cornish. An introduction to digital whole slide imaging and whole slide image analysis. http://www.hopkinsmedicine.org/mcp/PHENOCORE/CoursePDFs/2013/13%2019%20Cornish%20Digital%20Path.pdf, July 2013. Accessed: 12.04.2016.

[12] T. Cramer. The international image interoperability framework (iiif): Laying the foundation for common services, integrated resources and a marketplace of tools for scholars worldwide. https://www.cni.org/topics/information-access-retrieval/international-image-interoperability-framework, December 2011. Accessed: 18.08.2016.

[13] J. Cupitt. Vips. http://www.vips.ecs.soton.ac.uk/index.php?title=VIPS. Accessed: 25.05.2016.

[14] G. Seeman D. Bourg. *AI for Game Developers*. O'Reilly, 2004.

[15] M. Marcellin D. Taubmann. *JPEG 2000: Image compression fundamentals, standards and practice*. Kluwer Academic Publishers, 2001.

[16] Working Group 26. Pathology DICOM Standards Committee. Digital imaging and communications in medicine (dicom), supplement 145: Whole slide microscopic image iod and sop classes, August 2010.

[17] digitalpreservation. Bigtiff. http://www.digitalpreservation.gov/formats/fdd/fdd000328.shtml. Accessed: 20.08.2016.

[18] digitalpreservation. Tiff. http://www.digitalpreservation.gov/formats/fdd/fdd000237.shtml. Accessed: 20.08.2016.

[19] D. Doubrovkine. Dzt. https://github.com/dblock/dzt. Accessed: 22.08.2016.

[20] S. Eddins. Tiff, bigtiff, and blockproc. http://blogs.mathworks.com/steve/2013/08/07/tiff-bigtiff-and-blockproc/, August 2013.

[21] A. Goode et al. Openslide: A vendor-neutral software foundation for digital pathology. *J pathol inform*, 4(1), September 2013. http://download.openslide.org/docs/JPatholInform_2013_4_1_27_119005.pdf.

[22] D. Wilbur et al. Whole-slide imaging digital pathology as a platform for teleconsultation. *Arch Pathol Lab Med*, 133(12), December 2009. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3694269/pdf/nihms-486265.pdf.

[23] E. Tseytlin et al. Wsi zoomviewer. http://www.pathologyinformatics.com/sites/default/files/archives/2014/Day2/20140514%201120%20-%20WSI%20Zoom%20Viewer.pdf, 2014. Accessed: 22.08.2016.

[24] H. Guedri et al. Reconstruction of the human retinal blood vessels by fractal interpolation. *Journal of Theoretical and Applied Information Technology*, 83(2), January 2016. http://www.jatit.org/volumes/Vol83No2/9Vol83No2.pdf.

[25] H. Sharma et al. Robust segmentation of overlapping cells in histopathology specimens using parallel seed detection and repulsive level set. *IEEE Transactions on Biomedical Engineering*, 59(3), March 2012.

[26] H. Sharma et al. Deep convolutional neural networks for histological image analysis in gastric cancer whole slide images. *Computerized Medical Imaging and Graphics*, December 2016.

[27] M. Appleby et al. Iiif image api 2.0. http://iiif.io/api/image/2.0/. Accessed: 18.08.2016.

[28] M. Egmonst-Petersen et al. Image processing with neural networks - a review. *Pattern Regcognition*, 35, October 2002. https://www.researchgate.net/publication/220603536_Image_processing_with_neural_networks_-_a_review_Pattern_Recogn_352279C2301.

[29] M. Frosterus et al. Extending ontologies with free keywords in a collaborative annotation environment. Technical report, University of Helsinki, 2010. http://ceur-ws.org/Vol-809/paper-02.pdf.

[30] N. Farahanil et al. Whole slide imaging in pathology: advantages, limitations, and emerging perspectives. *Pathology and Laboratory Medicine International*, 7, June 2015. https://www.dovepress.com/whole-slide-imaging-in-pathology-advantages-limitations-and-emerging-p-peer-reviewed-fulltext-article-PLMI#ref10.

[31] P. Thévanez et al. Image interpolation and resampling. *Handbook of Medical Imaging, Processing and Analysis*, 2000. http://bigwww.epfl.ch/publications/thevenaz9901.pdf.

[32] R. Harvey et al. A neural network architecture for general image recognition. *The Lincoln Library Journal*, 4(2), 1991. https://www.ll.mit.edu/publications/journal/pdf/vol04_no2/4.2.5.neuralnetwork.pdf.

[33] R. Singh et al. Standardization in digital pathology: Supplement 145 of the dicom standards. *J pathol inform*, 2(23), March 2011. http://www.jpathinformatics.org/temp/JPatholInform2123-3144928_084409.pdf.

[34] S. Doyle et al. A boosted bayesian multi-resolution classifier for prostate cancer detection from digitized needle biopsies. *Transactions on Biomedical Engineering*, June 2010. https://www.researchgate.net/publication/44695140_A_Boosted_Bayesian_Multiresolution_Classifier_for_Prostate_Cancer_Detection_From_Digitized_Needle_Biopsies.

[35] S. Park et al. Digital imaging in pathology. *Clin Lab Med*, 4(32), December 2012.

[36] S. Wienert et al. Detection and segmentation of cell nuclei in virtual microscopy images: A minimum-model approach. *Scientific Reports*, 2(503), Juli 2012. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3694269/pdf/nihms-486265.pdf.

[37] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of Carolina, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[38] Flask. Flask user guide. http://flask.pocoo.org/docs/0.11/. Accessed: 24.08.2016.

[39] National Institute for Standards and Technology. Pyramidio. https://github.com/usnistgov/pyramidio. Accessed: 22.08.2016.

[40] Open Source Geospatial Foundation. Tile map service specification. http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification. Accessed: 26.04.2016.

[41] OpenStreetMap Foundation. Openstreetmap homepage. http://www.openstreetmap.org/about. Accessed: 18.08.2016.

[42] L. Fuller. sharp homepage. http://sharp.dimens.io/en/stable/. Accessed: 22.08.2016.

[43] A. Kaebler G. Bradski. *Learning OpenCV*. O'Reilly, 2008.

[44] D. Gasienica. Vips. https://github.com/zoomhub/zoomhub. Accessed: 22.08.2016.

[45] I. Gilman. Openseadragon. http://openseadragon.github.io/examples/creating-zooming-images/. Accessed: 26.04.2016.

[46] IIIF. Internation image interoperability framework homepage. http://iiif.io. Accessed: 18.08.2016.

[47] ImageMagick. Imagemagick: Formats. http://www.imagemagick.org/script/formats.php. Accessed: 22.08.2016.

[48] Optimus Information Inc. Open-source vs. proprietary software, pros and cons. http://www.optimusinfo.com/downloads/white-paper/open-source-vs-proprietary-software-pros-and-cons.pdf, 2015. Accessed: 16.08.2016.

[49] Radiocommunication Sector International Telecommunication Union. Recommendation itu-r bt.601-4 encoding parameters of digital television for studios. http://www-inst.eecs.berkeley.edu/~cs150/Documents/ITU601.PDF, 1998.

[50] intoPix. Everything you always wanted to know about jpeg 2000. http://www.intopix.com/pdf/JPEG%202000%20Handbook.pdf, 2008.

[51] K. Martinez J. Cupitt. Vips: an image processing system for large images. *Proc. SPIE*, 2663:19 – 28, 1996. http://eprints.soton.ac.uk/252227/1/vipsspie96a.pdf.

[52] K. Martinez J. Cupitt. Vips - a highly tuned image processing software architecture. In *IEEE International Conference on Image Processing*, pages 574 – 577, September 2005. http://eprints.soton.ac.uk/262371/.

[53] K. Martinez J. Cupitt. Nucleus modelling and segmentation in cell clusters. *Mathematics in Industry*, 15:217 – 222, May 2010.

[54] J. Puckey J. Lehni. Paper.js homepage. http://paperjs.org/about/. Accessed: 28.08.2016.

[55] M. Fowler J.Lewis. Microservices, a definition of this new architectural term. http://martinfowler.com/articles/microservices.html#footnote-etymology, March 2014. Accessed: 12.08.2016.

[56] JPEG. Overview of jpeg 2000. https://jpeg.org/jpeg2000/. Accessed: 18.08.2016.

[57] jQuery Foundation. jquery browser support. https://jquery.com/browser-support/. Accessed: 28.08.2016.

[58] jQuery Foundation. jquery homepage. https://jquery.com/. Accessed: 28.08.2016.

[59] A. Kesteren. *Cross-Origin Resource Sharing: W3C Recommendation 16 January 2014*, January 2014. https://www.w3.org/TR/cors/#cross-origin-request.

[60] D. Kriesel. *A Brief Introduction to Neural Networks*, 2007. http://www.dkriesel.com/en/science/neural_networks.

[61] D. Liu. A review of computer vision segmentation algorithms. Technical report, University of Washington, 2012. https://homes.cs.washington.edu/~bilge/remote.pdf.

[62] R. Joshi M. Rabbani. An overview of the jpeg2000 still image compression standard. *Signal Processing Image Communication*, 17(3), 2002. https://www.csd.uoc.gr/~hy471/bibliography/jpeg2000.pdf.

[63] Mozilla Developer Network (MDN). Same-origin policy for file: Uris. https://developer.mozilla.org/en-US/docs/Same-origin_policy_for_file:_URIs. Accessed: 22.08.2016.

[64] Microsoft. Deep zoom file format overview. https://msdn.microsoft.com/en-us/library/cc645077(v=vs.95).aspx. Accessed: 17.08.2016.

[65] K. Mulka. Gmap uploader tiler. https://github.com/mulka/tiler. Accessed: 22.08.2016.

[66] College of American Pathologists. Sample emplyment contract for a pathologist. http://www.cap.org/ShowProperty?nodePath=/UCMCon/Contribution%20Folders/WebContent/pdf/pm-sample-employment-contract.pdf. Accessed: 22.08.2016.

[67] OpenSlide. Openslide format documentation. http://openslide.org/. Accessed: 16.08.2016.

[68] J. Riggins. Microservices architecture: The good, the bad, and what you could be doing better. http://nordicapis.com/microservices-architecture-the-good-the-bad-and-what-you-could-be-doing-better/, April 2015. Accessed: 12.08.2016.

[69] R. Rojas. *Neural Networks - A Systematic Introduction*. Springer-Verlag, 1996. https://page.mi.fu-berlin.de/rojas/neural/.

[70] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), 1958. http://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf.

[71] D. Shiffman. *The Nature of Code*. D. Shiffman, 2012. http://natureofcode.com/book/chapter-10-neural-networks/.

[72] S. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997. http://www.dspguide.com/ch26/4.htm.

[73] The Internet Society. Hypertext transfer protocol – http/1.1. https://www.w3.org/Protocols/rfc2616/rfc2616.html, 1999. Accessed: 26.08.2016.

[74] G. Strang. *Introduction to Linear Algebra*. Springer, 2003.

[75] US San Diego Health System. Confidentiality agreement. https://health.ucsd.edu/about/volunteer/Documents/Confidentiality%20Agreement%20D214.pdf. Accessed: 22.08.2016.

[76] G. Toussaint. Solving geometric problems with the rotating calipers. *Proceedings of IEEE MELECON'83*, May 1983. https://www.cs.swarthmore.edu/~adanner/cs97/s08/pdf/calipers.pdf.

[77] J. David V. Shereena. Content based image retrieval: Classification using neural networks. *The International Journal of Multimedia & Its Applications*, 6(5), October 2014.

[78] S. van der Walt et al. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2), March 2011. http://ieeexplore.ieee.org/document/5725236/?arnumber=5725236.

[79] E. Wolff. *Microservices Primer*. innoQ, January 2016. https://leanpub.com/microservices-primer/read.

List of Figures

1.1 Example results of the in [2] introduced model (source: http://cs.stanford.edu/people/karpathy/deepimagesent/) 5

2.1 DICOMs image pyramid (source: [33]) 8
2.2 DZI pyramid model example (source: [64]) 12
2.3 Example of iiif request (source:http://www.slideshare.net/Tom-Cramer/iiif-international-image-interoperability-framework-dlf2012?ref=https://www.diglib.org/forums/2012forum/transcending-silos-leveraging-linked-data-and-open-image-apis-for-collaborative-access-to-digital-facsimiles/) 13
2.4 Results of IIIF request with different values for region parameter (source: [27]) 15
2.5 Results of IIIF request with different values for size parameter (source: [27]) 15
2.6 Neuron in a BNN (source: [71]) 18
2.7 Exemplary NN (source: [71]) 19
2.8 Perceptron by Rosenblatt (source: [71]) 20
2.9 Examples for linearly separable problems (source: [71]) 21
2.10 Examples for non-linearly separable problems (source: [71]) . . . 21
2.11 NN with multiple layers (source: http://docs.opencv.org/2.4/_images/mlp.png) 22
2.12 Activity diagram of the process chain 27
2.13 Visualization of the Conversion Service 28
2.14 Visualization of the Annotation Service 28
2.15 Visualization of the Tessellation Service 29

3.1 Content of input directory 38

4.1 Activity diagram of ASS and ASV 42
4.2 Example of context regions (B, C are context of A; A, C are context of B; A, B are context of C; D has no context region) . . 44
4.3 Microdraw GUI with opened WSI 45

4.4 Example URL to retrieve WSI (`http://127.0.0.1:5000/wsi/CMU-1.svs`) ; WSI source: OpenSlides freely distributable test data (see appendix A.2.1) 51
4.5 ASV's GUI with opened, annotated WSI 61

5.1 Examplary BBs: B_1 is BB of M_1, B_2 is BB of M_2 (source: `http://www.idav.ucdavis.edu/education/GraphicsNotes/Bounding-Box/Bounding-Box.html`) 63
5.2 Example of an ROI approximated via tessellation (gray tiles belong to the ROI) 64
5.3 Activity diagram of TS' extraction procedure 67
5.4 Example for adjusted BB (A - original BB with aspect ratio of ∼1:1, B - adjusted BB to aspect ratio of ∼1:2) 69
5.5 Activity diagram of TS' WSI extraction (without tessellation) . . 70
5.6 Activity diagram of TS' DZI extraction (without tessellation) . . 71
5.7 Example of RI (A) and resulting image tiles (B, stitched) with a tile size of 32x32 pixel (example was created using CMU-1.svs from Aperio, see appendix A.2) 72
5.8 Test case *Aperio - CMU-1.svs* 74
5.9 Results for test region A 76
5.10 Results for test region B 77
5.11 Results for test region C 78

B.1 Call hierarchy of `init(file_name, url, mpp)` 96
B.2 Call hierarchy of `initAnnotationService()` 96
B.3 Call hierarchy of `saveConfig()` 97
B.4 Call hierarchy of `loadConfiguration()` 97
B.5 Call hierarchy of `saveDictionary()` 98
B.6 Call hierarchy of `loadDictionary(path)` 98
B.7 Call hierarchy of `saveRegions()` 99
B.8 Call hierarchy of `loadJson()` 99
B.9 Call hierarchy of `createNewDictionary(isCancelable)` 99
B.10 Call hierarchy of `appendLabelsToList()` 100
B.11 Call hierarchy of `appendLabelToList(label)` 101
B.12 Call hierarchy of `selectNextLabel(label)` 101
B.13 Call hierarchy of `newLabel()` 102
B.14 Call hierarchy of `dictListClick(index)` 102
B.15 Call hierarchy of `findContextRegion()` 104
B.16 Call hierarchy of `toggleRegions(uid)` 104
B.17 Call hierarchy of `singleClickOnLabel(event)` 105
B.18 Call hierarchy of `addPoi(event)` 106
B.19 Call hierarchy of `mouseDown(x,y)` 108
B.20 Call hierarchy of `mouseDrag(x,y,dx,dy)` 109
B.21 Call hierarchy of `keydown(function(e))` 110
B.22 Call hierarchy of `keyup(function(e))` 110
B.23 Call hierarchy of `selectToolOnKeyPress(id` 110

List of Tables

2.1 File formats by vendor 9
2.2 Valid values for *region* parameter (source: [27]) 14
2.3 Valid values for *size* parameter (source: [27]) 16

3.1 File formats by vendor 30
3.2 Overview of conversion options for zooming image formats (source: [45]) 32
3.3 Options for deepzoom.py 33
3.4 Options for VIPS 35
3.5 Results of Conversion Service Test 38

4.1 Available converters in Flask (source: [38]) 47
4.2 ASS' URL-function binding overview 47
4.3 DZG parameters (source: [67]) 48
4.4 Description of `get_tile(level, address)` parameters (source: [67]) 49
4.5 Configurable parameters in configuration.json 50
4.6 Parameters for as_server.py 51
4.7 Overview of used options in OSDV constructor (source: [45]) . . 56
4.8 Class diagram of the region class 59

5.1 Optional parameters for the TS 66
5.2 File name patterns of generated output 68
5.3 File name patterns of generated output 69
5.4 Test cases for the TS 73
5.5 Test-result correspondence 75

6.1 Correspondence of objectives, chapters and tools 79

A.1 Services with their disc paths and repository URLs 83
A.2 Aperio data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Aperio/`) 84
A.3 Generic Tiled tiff data set (source: `http://openslide.cs.cmu.edu/download/openslide-testdata/Generic-TIFF/`) 84

A.4 Hamamatsu data set (.ndpi, source: http://openslide.cs.cmu.edu/download/openslide-testdata/Hamamatsu/) 85
A.5 Hamamatsu data set (.vms, source: http://openslide.cs.cmu.edu/download/openslide-testdata/Hamamatsu-vms/) 85
A.6 Leica data set (source: http://openslide.cs.cmu.edu/download/openslide-testdata/Leica/) 86
A.7 Trestle data set (source: http://openslide.cs.cmu.edu/download/openslide-testdata/Trestle/) 86
A.8 Trestle data set (source: http://openslide.cs.cmu.edu/download/openslide-testdata/Trestle/) 86
A.9 Mirax data set (source: http://openslide.cs.cmu.edu/download/openslide-testdata/Mirax/) 87

Nomenclature

AJAX	Asynchronous JavaScript and XML
AO	Annotation Overlay
AS	Annotation Service
ASS	Annoation Service Server
ASV	Annotation Service Viewer
BB	Bounding Box
BNN	Biological Neural Network
CORS	Cross Origin Resource Sharing
CS	Conversion Service
DICOM	Digital Imaging and Communications in Medicine
DZG	DeepZoomGenerator
GUI	Graphical User Interface
IIIF	International Image Interoperability Framework
MPP	Microns Per Pixel
MS	Microservice
NN	Neural Network
OpenCV	Open Source Computer Vision Library
OSD	OpenSeadragon
OSDV	OpenSeadragon Viewer
OSGF	Open Source Geospatial Foundation
OSM	OpenStreetMap
POI	Point of Interest
REST	Representational State Transfer
RI	Reference Image
ROI	Region of Interest
SOP	Same-Origin Policy
TIFF	Tagged Image File Format
TMS	Tiled Map Service
TS	Tessellation Service
VIPS	VASARI Image Processing System
WSGI	Web Server Gateway Interface
WSI	Whole Slide Image

Statutory declaration

I declare that I have developed and written the enclosed Master Thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. The Master Thesis was not used in the same or in a similar version to achieve an academic grading or is being published elsewhere.

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 23.09.2016 ***Sascha Nawrot***