

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

M. MATTHIEU PIZENBERG

le vendredi 28 février 2020

Titre :

Interactive Computer Vision through the Web

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

M. VINCENT CHARVILLAT

M. AXEL CARLIER

Rapporteurs :

M. MATHIAS LUX, ALPEN ADRIA UNIVERSITAT

Mme VERONIQUE EGLIN, INSA LYON

Membre(s) du jury :

Mme GÉRALDINE MORIN, TOULOUSE INP, Président

M. AXEL CARLIER, TOULOUSE INP, Membre

M. CHRISTOPHE DEHAIS, ENTREPRISE FITTINGBOX, Membre

M. OGE MARQUES, FLORIDA ATLANTIC UNIVERSITY, Membre

M. VINCENT CHARVILLAT, TOULOUSE INP, Membre

Acknowledgments

First I'd like to thank my advisors Vincent and Axel without whom that PhD would not have been possible. I would also like to thank Véronique and Mathias for reviewing this manuscript, as well as the other members of the jury, Oge, Géraldine and Christophe for your attention, remarks and interesting discussions during the defense. Again, a special thank you Axel for all that you've done, throughout this long period and even before it started. I haven't been on the easiest path toward completion of this PhD but you've always been there to help me continue being motivated and that's what mattered most!

Wanting to begin a PhD certainly isn't a one-time moment, but for me, the feeling probably started during my M1 internship. I was working in the VORTEX research team (now REVA) on a project with Yvain and Jean-Denis and it was great! Yet "I don't think so" was more or less what I kept answering to my teachers when they would ask if I wished to start a PhD at that time. And it lasted until almost the end of my M2 internship in Singapore, when working on a project to transfer research to a product. The internship and my first experience in Singapore was great and I'd like to especially thank Axel, Vincent, Thanh, Nizar and Igor for that. It was at that period I realized this is what I wanted!

As a consequence, a significant time of this PhD was spent in Singapore. It is at the origin of all the visual odometry part of this thesis. I'd like to thank Mounir, Xiong Wei and Janice in addition to my supervisors who made that possible. I also met amazing friends there without whom this would have been a much different experience. Justin, Yolanda, Thomas, Ariane, Flo, Bastien, Clovis, Martin, Ana and Joaquim thank you!

The major part of my PhD was spent in Toulouse, surrounded by wonderful colleagues and friends. Some of them became futsal mates, running partners, Tarot and Belotte players, oxidizers (increasing my amount of Rust), temporary flatmates, gaming friends, and night watchers (on TV as well as in bars!). In the lab, it's all the little things, from Super AdMinistrative powers to team workshops or even Thursday burgers, that add up to form a very welcoming and enjoying working environment. Outside the lab, I've been lucky to meet all my friends in Toulouse and I hope that I'll be able to keep in touch. Jean-Denis, Yvain, Simone, Géraldine, Sylvie, Pierre, Charlie, Sam, Axel, Vincent C, Vincent A, Thibault, Bastien, Chafik, Arthur, Julien, Paul, Thomas, Matthieu, Thierry, Damien, Sonia, Jean, Richard, Simon, Patrick, Alison, Etienne, Antoine, Matthias, Nicolas, Korantin thank you

all! A dedication also to the friends that helped me becoming who I am before joining the pink city, Yanis, Alexandre, Alain, Baptiste, Bastien, Océane and Alice thank you!

Finally, I would like to thank my family for always encouraging me and giving me the means to pursue that science quest of mine. En particulier, merci Papa, Maman.

Abstract

Computer vision is the computational science aiming at reproducing and improving the ability of human vision to understand its environment. In this thesis, we focus on two fields of computer vision, namely image segmentation and visual odometry and we show the positive impact that interactive Web applications provide on each.

The first part of this thesis focuses on image annotation and segmentation. We introduce the image annotation problem and challenges it brings for large, crowdsourced datasets. Many interactions have been explored in the literature to help segmentation algorithms. The most common consist in designating contours, bounding boxes around objects, or interior and exterior scribbles. When crowdsourcing, annotation tasks are delegated to a non-expert public, sometimes on cheaper devices such as tablets. In this context, we conducted a user study showing the advantages of the outlining interaction over scribbles and bounding boxes. Another challenge of crowdsourcing is the distribution medium. While evaluating an interaction in a small user study does not require complex setup, distributing an annotation campaign to thousands of potential users might differ. Thus we describe how the Elm programming language helped us build a reliable image annotation Web application. A highlights tour of its functionalities and architecture is provided, as well as a guide on how to deploy it to crowdsourcing services such as Amazon Mechanical Turk. The application is completely open-source and available online.

In the second part of this thesis we present our open-source direct visual odometry library. In that endeavor, we provide an evaluation of other open-source RGB-D camera tracking algorithms and show that our approach performs as well as the currently available alternatives. The visual odometry problem relies on geometry tools and optimization techniques traditionally requiring much processing power to perform at realtime framerates. Since we aspire to run those algorithms directly in the browser, we review past and present technologies enabling high performance computations on the Web. In particular, we detail how to target a new standard called WebAssembly from the C++ and Rust programming languages. Our library has been started from scratch in the Rust programming language, which then allowed us to easily port it to WebAssembly. Thanks to this property, we are able to showcase a visual odometry Web application with multiple types of interactions available. A timeline enables one-dimensional navigation along the video sequence. Pairs of image points can be picked on two 2D thumbnails of the image sequence to realign cameras and correct drifts. Colors are also used to identify parts of the 3D point cloud, selectable to reinitialize camera positions. Combining those interactions enables improvements on the tracking and 3D point reconstruction results.

Contents

Introduction **1**

I Image Annotation **5**

1 The Image Annotation Problem **7**

1.1 Computer vision problems that require annotation 9

1.1.1 Image classification 9

1.1.2 Image captioning 10

1.1.3 Object detection 11

1.1.4 Object segmentation 13

1.2 Discussion on dataset gathering 14

1.2.1 Explicit vs. implicit annotation process 14

1.2.2 Expert vs. non-expert annotators 16

1.2.3 Quality check of annotations 17

1.2.4 Annotation interaction usability 18

1.3 Existing interactions for user-assisted segmentation 19

2 Outlining for Segmentation **23**

2.1 Introduction 24

2.2 Outlining objects for interactive segmentation 26

2.2.1 Outline erosion 27

2.2.2 Blum medial axis algorithm 28

2.2.3 Enhancing foreground with superpixels 29

2.3 Experiments 29

2.3.1 Experimental setup 29

2.3.2 Usability metrics 32

2.3.3 Interaction informativeness 34

2.3.4 Segmentation quality 36

2.3.5 Discussion 37

2.4 Conclusion . . . 38

3 Reliable Web Applications 39

3.1 What is the Web? . . . 40
3.1.1 What is a Web application? . . . 40
3.1.2 Rich Web Application . . . 42
3.2 JavaScript, formally known as ECMAScript . . . 42
3.2.1 Genesis of JavaScript . . . 42
3.2.2 Browser performance . . . 42
3.2.3 Explosion of JavaScript . . . 44
3.2.4 JavaScript issues . . . 45
3.2.5 JavaScript as a compilation target . . . 50
3.3 Frontend Web programming . . . 53
3.3.1 Single Page Application (SPA) . . . 53
3.3.2 Reactive programming . . . 54
3.3.3 Virtual DOM . . . 55
3.3.4 How to choose? . . . 57
3.4 Elm . . . 59
3.4.1 Pure functions . . . 59
3.4.2 Algebraic Data Types (ADT) . . . 60
3.4.3 Total functions . . . 62
3.4.4 The Elm Architecture (TEA) . . . 64
3.4.5 Elm-UI, an alternative layout strategy . . . 65
3.4.6 Reliable packages . . . 67

4 Interactive Annotation on the Web 69

4.1 Introduction . . . 70
4.2 Presentation of the application . . . 71
4.3 Technical choices . . . 73
4.3.1 The model states . . . 73
4.3.2 The messages . . . 73
4.3.3 The view . . . 75
4.3.4 Library and application duality . . . 75
4.4 Crowdsourcing annotations . . . 76
4.5 Acknowledgments . . . 76
4.6 Conclusion . . . 77

II RGB-D Visual Odometry 79

5 Introduction to the RGB-D Visual Odometry Problem 81
5.1 Modeling Image Capture in a Camera . . . 82
5.1.1 Historic Remarks . . . 82
5.1.2 Projective Geometry . . . 82
5.1.3 Pinhole Camera Model . . . 83
5.1.4 Intrinsic Parameters . . . 84
5.1.5 Radial Distortion . . . 85
5.2 Modeling Camera Movements . . . 86
5.2.1 Origins of Visual Odometry . . . 86
5.2.2 3D Space & Rigid Body Motion . . . 86
5.2.3 The Lie Group $SO(3)$ and Lie Algebra $\mathfrak{so}(3)$ . . . 88
5.2.4 The Lie Group $SE(3)$ and Lie Algebra $\mathfrak{se}(3)$ . . . 90
5.3 Visual Odometry Approaches . . . 90
5.3.1 Capturing Device . . . 91
5.3.2 Relation to Visual SLAM . . . 92
5.3.3 Reducing Drift . . . 93
5.4 Motion Estimation . . . 94
5.4.1 Feature-Based Motion Estimation . . . 94
5.4.2 Appearance-Based Motion Estimation . . . 96

6 Performant Web Applications 101
6.1 A Brief History of Native Code in the Client . . . 102
6.1.1 Java Applets . . . 102
6.1.2 Flash . . . 104
6.1.3 Google Native Client (NaCl) . . . 104
6.1.4 Emscripten and asm.js . . . 105
6.2 WebAssembly . . . 106
6.2.1 Relation to Previous Technologies . . . 107
6.2.2 Compilation to WebAssembly . . . 108
6.2.3 WebAssembly Minimum Viable Product (MVP) . . . 108
6.2.4 WebAssembly Bright Future . . . 108
6.2.5 Why this Matters for Research . . . 109
6.3 C++ Portability Pitfalls . . . 110
6.3.1 Web Limitations . . . 110
6.3.2 Low Level Native or Architecture Specific Code . . . 110
6.3.3 No Dynamic Linking to OS Libraries . . . 110
6.4 Rust and WebAssembly . . . 111
6.4.1 The Rust Programming Language . . . 111

6.4.2 WebAssembly in Rust . . . 112
6.5 Conclusion . . . 112

7 Interactive Visual Odometry on the Web 115
7.1 Visual Odometry in Rust (VORS) . . . 116
7.1.1 Overview of VORS . . . 116
7.1.2 Intuition on Direct Image Alignment . . . 116
7.1.3 Sparse Points Selection . . . 118
7.1.4 Multi-Resolution Direct Image Alignment . . . 119
7.1.5 Limits of the Implementation . . . 122
7.2 RGB-D Visual Odometry Evaluation . . . 122
7.2.1 Dataset Creation / Acquisition . . . 123
7.2.2 Evaluation Metrics . . . 126
7.2.3 Setup and Algorithms Evaluation . . . 128
7.3 Interactive VORS on the Web . . . 133
7.3.1 Port of VORS to WebAssembly . . . 133
7.3.2 Interactive VORS Web Interface . . . 135
7.3.3 Human in the Loop Closure . . . 137
7.4 Conclusion . . . 140

Conclusion 141

A Gradients of Reprojection Function 145
A.1 Notation . . . 145
A.2 Reprojection Using Twist Coordinates . . . 146
A.3 Jacobian Expression . . . 146
A.4 Partial Derivatives Relative to Linear Velocity Terms . . . 147
A.5 Partial Derivatives Relative to Angular Velocity Terms . . . 148
A.6 Partial Derivatives with Normalized Coordinates . . . 149

Bibliography 151

Abstracts 171

Introduction

Computer vision is the computational science aiming at reproducing and improving the ability of human vision to understand its environment from light sensors. Throughout a somewhat unconventional, multidisciplinary journey, this document aims at answering the following question. How can we leverage user interactions and the Web platform to improve fields of computer vision such as image segmentation and visual odometry.

On the one hand, image segmentation (cf Figure 1) is the task of identifying precise regions in an image that are structurally or semantically different. For medical images, it could be localizing cancer cells while for urban images, differenciating people, vehicles and traffic signs. On the other hand, visual odometry consists in analyzing the video stream of a sensor such as a camera to locate and track its trajectory with regard to its environment (cf Figure 2).

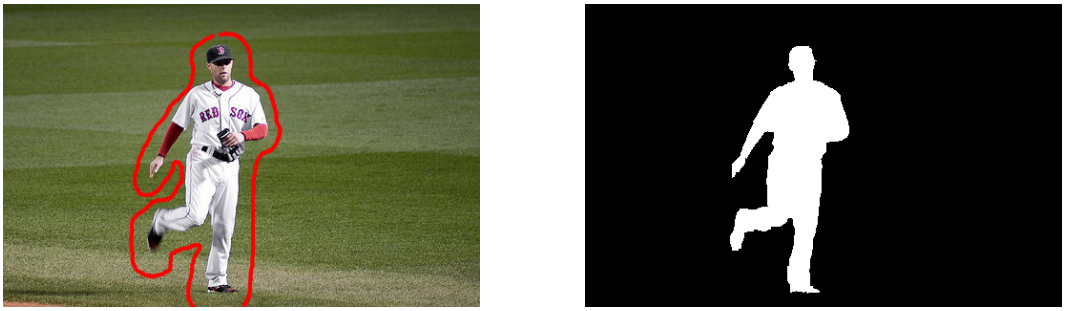


Figure 1: Outlining interaction in red and resulting segmentation mask.

Thanks to improvements in imaging and algorithms, we are now able to automate tasks that were considered science-fiction until recently. For instance, some companies [142] claim to have reached level 3 of SAE classification [110], meaning that a self-driving vehicle provides autonomy at limited speed, conditionned by locality and weather among other restrictions. Nethertheless, we are still far from reaching level 5 of SAE classification which requires full autonomy in any driving condition. One non-technical reason is that owners of autonomous vehicles would be reluctant to accept liability for potential accidents like the incident of March 18, 2018, that killed Elaine Herzberg [67]. I believe that such vehicles

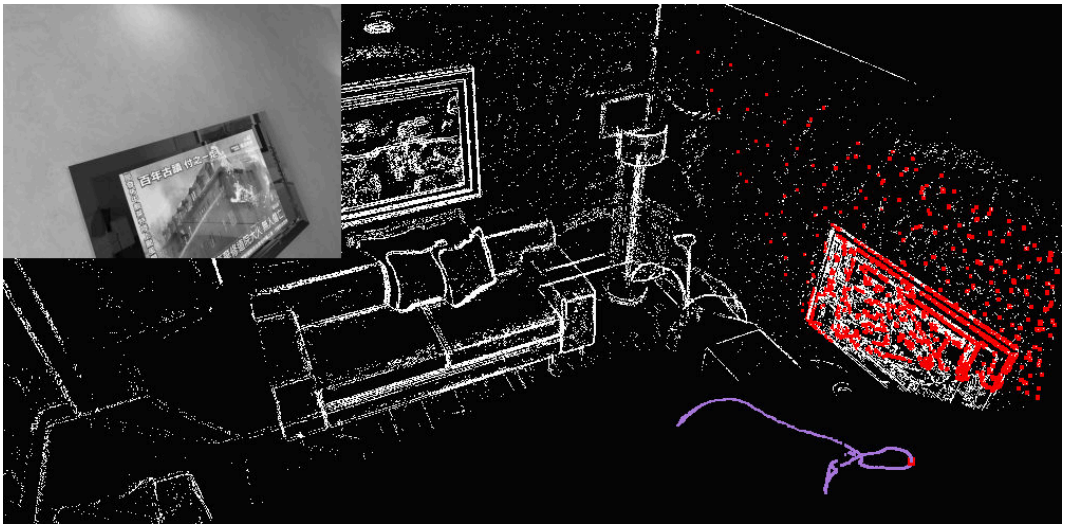


Figure 2: Camera trajectory (purple) and sparse 3D reconstruction generated by visual odometry.

will only widespread when they are safe enough for manufacturers to bear responsability of accidents. Those capabilities depend on many research fields including object detection and segmentation of urban images, as well as visual odometry. The former is required to detect the road, understand traffic signs, avoid people, while the latter is needed to precisely record the vehicle trajectory, especially in situations where other sensors are not available or sufficiently precise, such as GPS in covered areas.

There exist many approaches for object detection and image segmentation. Today, the ones performing best rely on a field of research named machine learning. It consists in building prediction models by aggregating knowledge from databases of pairs of inputs and outputs called learning datasets. There are subtleties within the field and not all algorithms perform equal but a general rule is that the bigger and most accurate the learning dataset is, the best will be the detection and segmentation results. I will thus not focus on machine learning algorithms but rather on the creation of those datasets. That process is known as image annotation. Annotating an image may take different forms depending on the task, whether it is classification, object detection or segmentation. In general, it consists in people using image manipulation tools to draw rectangles, lines, polygons, and other geometric shapes to identify regions of an image and assign it a label. The Microsoft COCO dataset [136] for example contains 2.5 million labeled instances accross 328k images annotated by humans, and required over 22 hours per thousand segmentations. For a French worker, 35h a week, 228 day a year, this represents approximately 35 years, almost a full career devoted to that single task. It is thus understandable that building such datasets must be carefully thought of.

In the first part of this document I will focus on image annotation and segmentation.

Chapter 1 introduces in more details the image annotation problem and challenges it brings for large, crowdsourced datasets. In Chapter 2, we focus on the interactive segmentation task. Many interactions have been explored in the literature to help segmentation algorithms. The most common consist in designating contours [185], bounding boxes around objects [183], or interior and exterior scribbles [143]. Crowdsourcing such tasks however implies that non-expert users have to perform those interactions and the distinction between expert and non-expert users is rarely touched. Inspired by the work of Korinke et al. [123, 124], we present a user study showing the advantages of the outlining interaction for crowdsourcing annotations to a non-expert public. This work has been published at ACM Multimedia 2017 [172]. Another challenge of crowdsourcing is the distribution medium. While evaluating an interaction in a small user study does not require complex setup, distributing an annotation campaign to thousands of potential users might differ. The best way to proceed is to build a Web application; and since online annotators are paid for the task, we need the Web application to be as reliable as possible. Therefore, in Chapter 3 we review evolutions of the Web since its creation in 1991, especially regarding the development of reliable frontend applications. In particular, we describe how the Elm programming language can help us build a bug-free annotation Web application. Finally in Chapter 4, we present the open-source Web application we built for the image annotation task. A highlights tour of the functionalities and the application architecture is provided, as well as a guide on how to deploy it to crowdsourcing services such as Amazon Mechanical Turk. The presentation of this application was published in the open-source competition track of ACM Multimedia 2018 [173].

Being a computational science, progress in visual odometry tends to bring larger, more complex and computationally intensive algorithms over time. Although beeing a poor unit of measure, number of lines of code provide an approximation of the relative algorithmic complexity of similar projects. Let's examine SLAM, which is an extension to visual odometry. Figure 3 illustrates the growth of open-source SLAM libraries. As visible in that figure, projects code bases are growing to unreasonable sizes for research purposes. This observation is even worse when considering complete structure from motion libraries such as OpenMVG, reaching 461k lines of code.

Most of SLAM projects are developed using the C++ programming language for performance reasons. I will argue however, that by continuing to do so, we are hindering mid and long-term research in the field. C++ projects are difficult to build, mainly because of assumptions on requirements, dependency conflicts and usage of Linux, Mac, Windows or architecture specific libraries. To mitigate those issues, projects tend to include within the source code all of their dependencies. From the 461k lines of code in OpenMVG, 390k are coming from the `src/nonFree/` and `src/third_party/` directories. Although seemingly matters of engineering, those characteristics actually influence research by putting a very high barrier to entry for new approaches to be able to reproduce already available results and

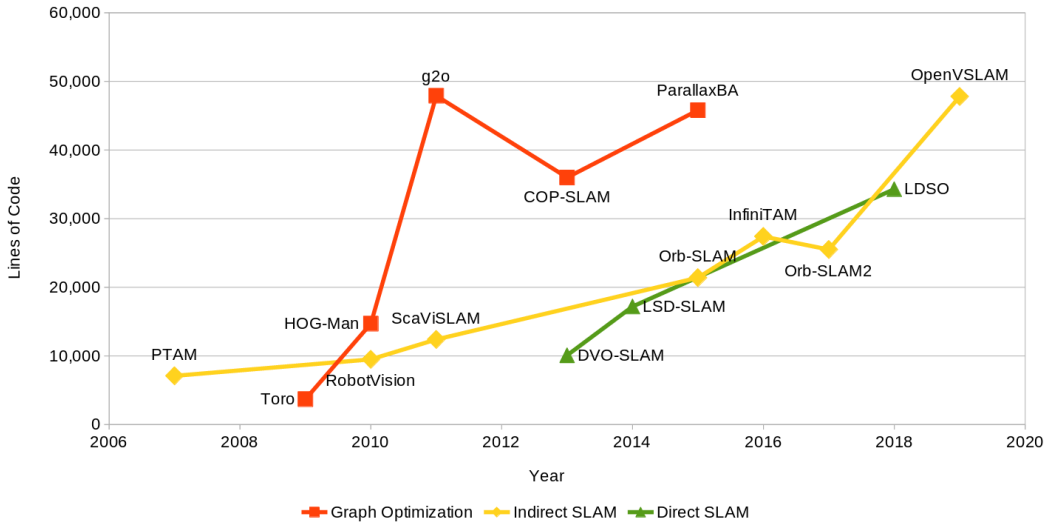


Figure 3: Growth of SLAM libraries over time.

compare with them. One should also note that frozing dependency versions brings security concerns, since upstream security patches requires manual actions to be replicated. This is especially true in the context of open-source, where we have less control over contributions. Even careful companies like Microsoft suffer from C++ memory safety bugs for 70% of their critical security issues [132]. Research code will eventually reach critical software, such as autonomous vehicles. With great research comes great responsability!

In the second part of this document, we will focus on visual odometry. Chapter 5 introduces the visual odometry problem and the fundamental geometry tools required to modelize it. Since we aspire to run those algorithms directly in the browser, Chapter 6 reviews past and present technologies enabling high performance computations on the Web. In particular, we detail how to target a new standard called WebAssembly from C++ and Rust. In Chapter 7 we present our open-source visual odometry library, which features a new points selection algorithm for the camera tracking. We started it from scratch in the Rust programming language, which allowed us to easily port it to WebAssembly. Finally, we showcase an interactive visual odometry Web application, enabling improvements on the tracking and 3D geometry results thanks to user interactions. A paper describing the open-source library and the interactive application is intended to be published.

Part I

Image Annotation

Chapter 1

The Image Annotation Problem

Contents

1.1 Computer vision problems that require annotation 9

1.1.1 Image classification 9

1.1.2 Image captioning 10

1.1.3 Object detection 11

1.1.4 Object segmentation 13

1.2 Discussion on dataset gathering 14

1.2.1 Explicit vs. implicit annotation process 14

1.2.2 Expert vs. non-expert annotators 16

1.2.3 Quality check of annotations 17

1.2.4 Annotation interaction usability 18

1.3 Existing interactions for user-assisted segmentation 19

In this chapter we will discuss the concept of image annotation, and review the body of work that have been researched in this domain. But first, what is image annotation? Fundamentally, it is the process of augmenting an image with information. This information can be of various nature, typically provided by a human operator, also called annotator.

We could consider image captioning as the first historical example of image annotation, simply consisting in adding a caption to an image. We could also consider photogrammetry as a form of image annotation which, long before digital images even existed, is the process of measuring distances and lengths of the real world from 2D images. It requires annotating these distances and lengths in the image space, before inferring the values in the real world. Early techniques in the old cinema also involved manually editing the filmstrip to create special effects, which is a form of annotation.

Digital imaging has progressively brought new needs for image annotation. The first digital image was scanned from a photograph in 1957 by Russell Kirsch, and the first digital camera was built in 1975 by Kodak engineer Steve Sasson. Commercial models of digital cameras became really available in the 1990s, and from then the volume of digital images produced grew exponentially every year. Meanwhile the field of computer vision, aiming at understanding those images, also developed its own research community. The highly influencial journal *IEEE Transactions on Pattern Analysis and Machine Intelligence* (TPAMI) was for example created in 1979. Computer vision is also related to the field of machine learning, which designates a class of algorithms in which a model learns from experience, materialized by data samples. One sub-domain of machine learning, called *supervised* machine learning, requires in particular annotated samples, meaning that a label should be assigned to each piece of data before an algorithm can be trained to predict these labels. Supervised machine learning gained traction in the 1990s during which some applications reached high enough maturity to be exploited commercially. A famous example of this is the digit recognition algorithm from Lecun et al. [131] which was used by AT&T to automatically process cheques in ATM (see Figure 1.1). A nowadays popular dataset, called MNIST (Mixed National Institute of Standards and Technology) was created for this work; this dataset associates labels (digits, from 0 to 9) to 28×28 pixel images of handwritten figures.

This dataset illustrates how image annotation could be used to produce desirable applications, and is only a small example of what has now become a classic pipeline to solve problems in the computer vision community. Since image annotation has become key in this community, this chapter focuses on computer vision but the machine learning pipeline we mention is also used in many other problems such as audio or natural language processing.

Theoretical results in machine learning postulate that problems of great complexity could be adressed with this technique, provided that (i) there exists a model of sufficient capacity to cope with the problem complexity, and (ii) a sufficiently large sample of annotated data is available. Some thresholds have been established by the community to estimate what

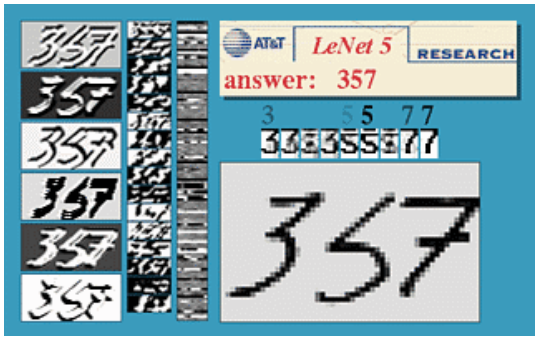


Figure 1.1: Illustration of the digit recognition algorithm from Lecun et al. [131]

"sufficiently large" means [177, 111], but computer vision problems typically requires millions of annotated images to be solved with an acceptable performance. Models relying on deep neural networks are nowadays the most popular techniques in machine learning, but other models, such as deep random forests, used in the human body pose estimation embedded in the Kinect [196], may still be considered depending on the application. Note that while gathering more and more data is the current trend in computer vision, an important field of research conversely focuses on learning on few samples; this field regroups the notions of semi-supervised learning, weakly-supervised learning, one-shot and few-shots learning, etc.

In what follows, we will motivate the study of image annotation techniques by reviewing the computer vision problems for which large datasets, often combined with (deep) machine learning techniques, have recently significantly improved the state-of-the-art. We will then discuss the process of gathering these annotations, focusing on some key aspects such as expert vs. non-expert annotations, quality control, etc. Finally, we will end this chapter with a focus on image segmentation, reviewing the possible interactions that can be used to provide such type of annotation. This last part will naturally lead to the next chapter that present our contribution on interactive segmentation.

1.1 Computer vision problems that require annotation

1.1.1 Image classification

Image understanding forms a category of hard problems, but among them, what could be considered the simplest one is *image classification*. Classifying an image consists in assigning it labels describing either the type of scene it depicts such as interior, exterior, beach, mountain, forest, city, or the objects that are displayed, sorted by importance. This task requires an advanced understanding of the images.

In fact, two of the most important challenges in computer vision have highlighted this task as one of the main problem to be solved. The first one, Pattern Analysis, Statistical

Modelling and Computational Learning Visual Objects Challenge, often called PASCAL VOC [72], has run from 2005 to 2012 and figured at its peak 11,530 images depicting 20 classes. It is interesting to note that this challenge coincides in time with the rise of machine learning popularity in the computer vision community. In a way, PASCAL VOC has been both a marker and a catalyzer of the importance of machine learning in image processing problems. PASCAL VOC stopped in 2012 sadly due to the passing of one of its most invested organizers, Mark Everingham, as well as due to the growing importance of a much larger challenge: the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

ILSVRC [184] started in 2010, motivated by the goal to solve larger scale problems. ILSVRC relies on a gigantic dataset called ImageNet, that originally intended to match a natural language dataset called WordNet [146]. WordNet is a database of english words, grouped into sets of synonyms called synsets. The goal of ImageNet is to provide a set of images to describe each of these synsets. As of December 4th 2019, ImageNet displays 14,197,122 images that depict 21,841 different synsets.

We should also mention a parallel effort funded by the Canadian Institute for Advanced Research, which led to the creation of CIFAR-10 and CIFAR-100. Those datasets contain images of size 32×32 collected over various Web images searching tools, and classified under 10 and 100 classes respectively. ImageNet and CIFAR-100 are often both used to assess the performance of image classifiers.

While many subsequent work have further improved state-of-the-art classification results, we could synthesize the progress in classification by citing two papers. The first one from Krizhevsky et al. in 2012 [127], nicknamed AlexNet, was probably key in the rise of deep learning that followed. AlexNet won the ILSVRC 2012 challenge by a large margin, starting the trend of using deep neural networks to solve computer vision problems. The second paper from He et al. in 2016 and often called ResNet [102], introduced residual blocks through skip connections to ease the training of very deep neural networks, up to 1000 layers! The 152 layers version of ResNet won the ILSVRC 2015 challenge by reaching an error rate so low that it could be considered below the average human performance. The general trend in subsequent work has been to reach comparable or higher performance than ResNet while reducing the number of parameters and operations to a minimum.

1.1.2 Image captioning

Another important topic, that extends in a sense the image classification problem, is the one of *image captioning*. It consists in describing an image with a set of sentences. Image captioning is a much harder problem than classification, because captions require a higher level of image understanding as well as natural language capabilities to generate valid sentences. In terms of annotations, it is also much longer to caption an image than just assigning it a class. Another difficulty for data gathering is the quality check of the annotations, since two sentences from two different users may be completely different but still convey the same

Table 1.1: Datasets for image classification and their characteristics.

Dataset	Year	# classes	# images	annotation process
PASCAL VOC [72]	2005 – 2012	20	11.5k	In-house
ESP Game [212]	unreported	any	100k	ESP Game players
CIFAR-10 [126]	2009	10	60k	Recruited students
CIFAR-100 [126]	2009	100	60k	Recruited students
SUN397 [227]	2010	397	130k	
ImageNet [184]	2010 – now	21k	14M	Mechanical Turk
Open Images [128, 125]	2016 – now	8.5k	9.2M	In-house and Crowdsource app

Table 1.2: Datasets for image captioning and their characteristics.

Dataset	Year	# captions	# images	annotation process
Flickr30k [231]	2014	150k	30k	Mechanical Turk
MS COCO [41]	2015	1M	164k	Mechanical Turk
Conceptual Captions [194]	2018	3.3M	3.3M	Web crawling
`nocaps` [3]	2019	166k	15k	Mechanical Turk

semantic meaning.

The datasets introduced in Table 1.2 have brought large enough sets of examples to efficiently train deep neural networks. Image captioning requires more advanced architectures, as it involves performing two difficult tasks at the same time: (i) image understanding (computer vision) and (ii) sentence generation (natural language processing). The first task has become fairly standard, provided that large datasets are available, and relies on convolutional neural networks. The latter is a well-known task as well, and can be solved using recurrent neural networks, which are useful to handle sequential data as well as generating sequences (such as sentences) of variable length. One of the first and most popular papers to build a system that brought together these two components was published by Xu et al. [228]. This work, named *Show, Attend and Tell*, uses an attention model to focus on different regions of an image while guiding the sentence generation. Attention models have been later extended to Transformers models [210], and this extension has been adapted to image captioning by the authors of the Conceptual Captions dataset [194].

1.1.3 Object detection

On top of naming or precisely describing the objects in an image, many applications also require to locate the objects. There are several levels of precision to which this problem can be achieved. The coarser grain application is often coined *object detection* and consists in drawing a bounding box around objects in an image. Object detection is a generalization of the object localization problem, for which there can be at most a single instance of each object in an image. Object detection is a much more difficult problem, since there could be

Table 1.3: Datasets for object detection and their characteristics.

Dataset	Year	# classes	# images	# instances	annotation process
PASCAL VOC [72]	2005 – 2012	20	11.5k	27.4k	In-house
ImageNet [184]	2010 – now	200	450k	500k	Mechanical Turk
MS COCO [136]	2015 – now	91	328k	2.5M	Mechanical Turk
Open Images [128, 125]	2016 – now	600	1.9M	15.8M	In-house

hundreds of instances of the same object in a scene (such as humans in a crowd picture, or cars in a parking lot for example).

Object detection datasets mostly originate from the image classification datasets inroduced in Table 1.1, which they are often extending. Table 1.3 sums up the main characteristics of four of the most prominent ones. The PASCAL Visual Objects Challenge [72] for example, has included a detection challenge ever since it first ran in 2005 but on only a few thousands image. The dataset increased in size over time and reached almost 30k annotated bounding boxes in the end in 2012. The ImageNet challenge, which originally started in 2010, later added a detection task (in 2013 and 2014) with a large dataset of more than 500k annotated bounding boxes on 200 classes (which include for the most part the 20 classes of PASCAL). Note that ImageNet also ran a localization challenge for which more than 500k images of the 1000 classes that were used in the classification challenge were annotated with one, and sometimes more, bounding boxes per image. The difference between localization and detection is that there is only one instance of object in an image of a localization dataset. In total, there are more than a million images including one or more bounding box annotations in the ImageNet dataset. The third dataset, called Microsoft Common Objects in Context (MS COCO) was released in 2014 [136]. It figures annotations that are in fact object segmentations, but that are used to generate bounding boxes valid for an object detection task. MS COCO was designed to provide a larger number of annotations per class than ImageNet and PASCAL, but on a smaller number of 91 classes. Finally, the most recent and large dataset is called Open Images [128, 125] and features a tremendous amount of more than 15 million bounding boxes of 600 classes on almost 2 million images.

These datasets have largely contributed to the performance improvements observed in the literature from 2014 to 2016. Such improvements are mainly due to two body of works that have driven the research in object detection forward. The first line of work directly derives from a trend that emerged at the end of the 2010s in computer vision. A category of segmentation algorithms called superpixels became quite popular and many influential papers [76, 1, 133] proposed solutions for computing oversegmentations that could be used as a building block of more complex methods. In particular, some object detection algorithms started using a set of object proposals [209], computed from a superpixel segmentation, that would be later classified as objects or not. This approach constitutes the core idea of the Mask-RCNN paper [87], the classification step being performed by a standard convolutional

Table 1.4: Datasets for object segmentation and their characteristics.

Dataset	Year	# classes	# images	# instances	annotation process
PASCAL VOC [72]	2005 – 2012	20	11k	7k	In-house
SUN2012 [227]	2012	4479	131k	313k	LabelMe [185, 19]
MS COCO [136]	2015 – now	91	328k	2.5M	Mechanical Turk
Open Images [128, 125]	2016 – now	350	1M	2.8M	In-house

neural network. This approach was later optimized by the same author [86] (Fast-RCNN), until the whole process was merged into a single end-to-end neural network that jointly performs object proposals and classification [181] (Faster-RCNN). A second line of work adopts a different type of neural network architectures. It focuses on the task of predicting bounding box coordinates for all object classes, splitting an image into a grid and being able to predict a bounding box centered in each cell of the grid. This approach, called YOLO (You Only Look Once) [178], was later optimized to be able to handle very large scale problems [179], up to 9000 classes, and to improve performances [180].

1.1.4 Object segmentation

The finer grain at which localization can be achieved is at pixel level: this is called *image segmentation*, or *image parsing*. This problem can also be stated as classifying each pixel in an image. A variant of this problem is called *object segmentation*, or instance segmentation, in which only some objects of an image are segmented. Annotations for the segmentation are much more difficult to gather, due to the need for a pixel-wise precision and the potential complexity of objects contours. The problem of automatic image segmentation is also a very complicated one, and requires a large number of annotations to be efficiently solved. The first dataset to offer an important number of annotations is once again PASCAL, with roughly 7,000 annotated instances of the same 20 classes as the one used for the classification and detection tasks. A parallel effort was developed in the framework of the SUN database, thanks to a popular labelling tool called LabelMe [185], that produced more than 300k labelled instances. LabelMe is a tool that allows drawing polygons around objects of interests. The segmentations obtained with this method are often quite coarse, but the authors of [136] nonetheless reported that it takes 22 hours of human annotations to segment 1000 object instances. In their dataset, which was described in an earlier paragraph, more than 2 millions instances were annotated. The latest dataset is again Open Images, with a million images depicting 350 classes and 2.8 million segmented instances.

The first paper to implement an end-to-end neural network for image segmentation was published in 2015 [138]. J. Long et al. present a fully convolutional architecture which combines predictions at different levels of resolution to produce a detailed segmentation map. This architecture was improved by two papers, who systematize the combination of predictions by introducing an encoder-decoder architecture, with skip connections that allows

retrieving fine-grained details. U-Net [182] is one of these two papers, and originated from the medical imaging community. SegNet [11] on the other hand specifically targets urban scenes segmentation, with autonomous driving as a direct application. These two papers share the same neural network architecture, with slight specificities in the skip connection implementation. Another very popular paper in the field of segmentation is DeepLab [40]. Previous architectures use an autoencoder structure, which allowed to derive a global understanding of the image in the bottleneck region of the network before retrieving a more local clasification of the pixels. Instead, DeepLab uses a spatial pyramid of different filter sizes to perform a multi-scale analysis of the image and make a local prediction that takes into account a larger area.

1.2 Discussion on dataset gathering

Instead of specifically commenting on each dataset methodology for gathering annotations, we will discuss in this section particular points that are of interest when one wants to create its own dataset.

1.2.1 Explicit vs. implicit annotation process

As visible in the tables of Section 1.1 the annotation process was predominantly explicit to the human annotators. By explicit, we mean that the humans involved in the task were fully conscious of the tasks they were performing, and that their goal was to create an annotated dataset. This is mostly due to the fact that image annotation takes time, and requires an incentive, typically money on crowdsourcing platforms. There are however a few exceptions, some of which have been mentioned before.

First, the Conceptual Captions dataset [194] has been obtained through a mostly automated process, looking for sentences on Web pages that accompany the images. One could say the original authors of the sentences implicitly annotated the images for this dataset.

A more interesting example is the ESP Game dataset [212]. The ESP Game has been created by Luis Von Ahn in 2005 and figures two humans playing collaboratively over the same image, as depicted in Figure 1.2. They score points whenever they manage to write matching words to describe the image. From an annotation point of view, whenever the two players agree on a word one can safely assume this word describes an object present on the image, or an action happening on the image. In order for the game to provide a good annotation coverage, words can be ruled out of the game, called *taboo*, which means the players can see these words and know they have to specify a different one. The ESP game started a trend of Games With A Purpose (GWAP) [213] including some games designed to perform image annotation such as PeekaBoom [214], KissKissBan [106] and Click'n'Cut [36], but ESP remains the only game that gathered enough data to create a dataset.



Figure 1.2: Screenshot of the ESP game, by Luis Von Ahn [212]

There exists another well-known mean to gather data in an implicit way: CAPTCHAs. The term has been originally coined by Luis Von Ahn (again) et al. [211] and stands for Completely Automated Public Turing Test to Tell Computer and Humans Apart. CAPTCHAs have been created to stop automated attacks on websites, to prevent automated creation of millions of malicious email accounts. The idea is to create a Turing test [208], i.e. a test that a human should be able to complete effortlessly while a machine would be unable to perform it. Original versions of CAPTCHAs displayed distorted, geometrically transformed words which would make it unrecognizable by standard optical character recognition (OCR) softwares. The task remained fairly easy to humans, and required in average 13s [215]. At the time, ambitious projects were ran in parallel to digitize tremendous collections of books, like *Google Books*, which makes available searching through millions of books. This process of digitizing books was automated using OCR softwares, but failed for 20% of the words in older books due to faded ink for example. The reCAPTCHA system offers a clever way to match the two very different problems of securing websites and digitizing old books by proposing words from old books, that could not be recognized automatically, to humans who wish to use an online service and are perfectly able to recognize them. The authors report that in 2008, after one year of deployment, reCAPTCHA has helped decipher 440M words, which amounts for more than 17,000 books.

While there are no further publications on reCAPTCHA, one has been able to observe its evolution through the years (see figure 1.3). Driven by the need to come up with problems that still resist computers, it went from text to image recognition, from digit recognition in pictures probably extracted from *Google Street View* to object detection in urban scenes pictures, very likely to target autonomous vehicles applications.

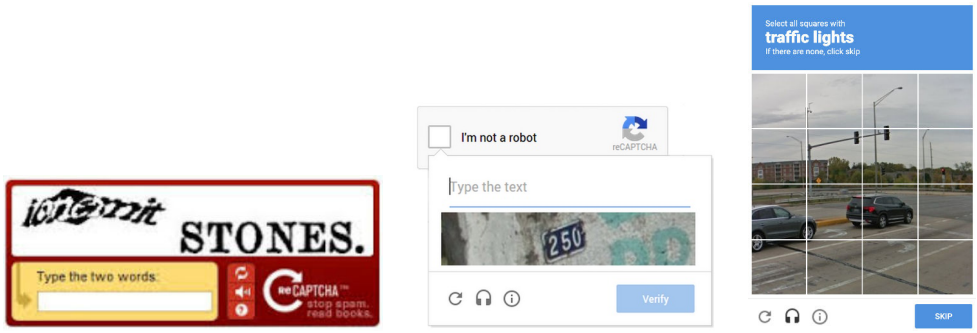


Figure 1.3: Screenshots illustrating the evolution of reCAPTCHA [215] tasks over the years.

1.2.2 Expert vs. non-expert annotators

Whether they are implicit or explicit, annotations need to be performed by human annotators and there have been several trends throughout the years. In essence, we could split annotators into two sets of users: experts, from whom we can expect high quality annotations but are rare and expensive, and non-expert users who tend to make more mistakes but provide much cheaper annotations.

Historically, the first datasets of reasonable scale which are PASCAL and CIFAR, did not require too many annotators. In fact, the authors of PASCAL [72] reported that a "party" of users annotated the images after an initial training. They were probably students, but no details are provided as to how many of them. These users were also regularly observed during annotations to ensure the quality of their work. Finally, one of the organizers of the PASCAL challenge checked all the annotations. Similarly, the CIFAR dataset [126] was labelled by a group of students paid for the task.

With time, there was a demand for larger datasets. They aimed at expanding the number of classes, as well as the number of annotated images. With this new goal in mind, the dataset authors started *crowdsourcing* the annotations. Crowdsourcing is a process in which a task that should usually be performed by an expert is outsourced to a crowd of non-expert users. The term was first coined by Jeff Howe in 2006 [107]. The creation of specific online platforms such as Amazon Mechanical Turk or Crowdflower considerably eased the process of crowdsourcing image annotation. ImageNet and MS COCO have both been annotated by Turkers, i.e. humans recruited and paid through Amazon Mechanical Turk. Since the difficulty to clearly instruct remote users is increased, the annotations interfaces need to be carefully designed, and the annotation quality need to be properly ensured; this will be described in Section 1.2.3.

While crowdsourcing is widely used, it may not be relevant to all setups. In cases when images are sensitive for example, it is not possible to outsource the annotation process outside of a company. In addition, the need for redundancy to ensure annotation quality limits the positive impact on cost that crowdsourcing is supposed to provide. Although being

one of the largest annotated dataset, Open Images has been for the most part annotated by in-house employees at Google. To achieve this, interactions have been specially designed to lead to good annotations in a faster way. For example, extreme clicking for bounding box annotations [165] introduces a clever interaction that allows to draw bounding boxes more quickly, avoiding the errors that usually force users to start over. Alternatively, the authors of [21] propose an interactive segmentation algorithm in which user clicks guide a deep neural network and refine the segmentation mask. The images can be annotated 3 times faster than when using a standard polygon drawing tool such as with LabelMe [185], and the segmentation boundaries are much more precise. We present a similar contribution in the next chapter.

1.2.3 Quality check of annotations

As stated in the previous section, crowdsourcing is currently one of the leading methods to gather annotations for creating image datasets. However, and while the annotations come at a relative cheap cost, the quality of the collected data is often questionable. Oleson et al. [162] have classified three categories of errors that non-expert users are likely to commit when performing Human Intelligence Tasks (HIT), the term coined in Amazon Mechanical turk to designate the micro-jobs offered to the users. The first source of errors, called *unsufficient attention* by Oleson, simply occurs when the humans enrolled to perfom HITs, called workers, make occasional mistakes, due to the task complexity or a lack of attention for example. Some other workers, called *incompetent* by Oleson, may not understand the task and behave unpredictably. The data they provide is often unusable. Finally, a last class of workers, called *scammers*, designates users who try to trick the system to collect the reward. A good quality checking process should account for these three types of possible errors. Oleson et al. point out the advantages of adding gold standard images i.e. images for which a ground-truth annotation is known, among the data to be able to estimate workers reliability. The authors of ImageNet [184] also reported using this technique. Using gold standard is a good practice to detect scammers and incompetent workers.

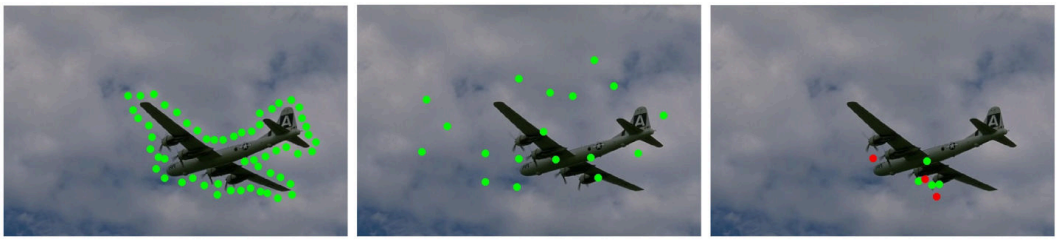


Figure 1.4: Illustration of possible mistakes that happen during a crowdsourcing campaign [38]: on the left, annotations from a user who misunderstood the task, to be compared with the expected results on the right (red points on the background, green points on the foreground). In the middle, annotation of a *scammer*.

The most obvious way to detect workers mistakes is to introduce redundancy in the annotations. Multiple workers are tasked to label the same image, and the annotations are validated if they are similar enough. This straightforward strategy has been formalized by Luis Von Ahn [213] under the term *output agreement*: multiple users agree on the same output annotation for a given image. Authors of MS COCO [136] report asking up to 8 workers to perform the same task, for example to increase the recall in an instance spotting task. Redundancy helps reducing the impact of unsufficient attention.

Another technique to prevent having too many incompetent workers is to go through a tutorial before starting the task. Gottlieb et al. [92] have shown that the performance of users who complete a tutorial beforehand is significantly higher than users who did not. In [136], users are filtered based on their performance in an initial training task and are periodically verified during the whole annotation process.

The error rate can also be limited by a clever labor division. A Human Intelligence Task should always be an atomic operation, simple to explain and quick to perform. For example, the authors of [204] introduce a Find-Fix-Verify pattern for object detection annotations. Output agreement is difficult to implement for bounding boxes due to the the presence of thresholds to measure similarity. Therefore, the authors introduce a series of micro-tasks ensuring the quality of the final bounding boxes. A first group of workers is tasked to draw bounding boxes (*Find*), and a second group of different workers is asked to validate each of the bounding boxes (*Fix*). Being a binary decision, it is easier to implement output agreement on this second task. Finally a last group of users is in charge of checking whether some objects have been omitted by the first pool of workers (*Verify*). Another example of work division is described in [41] for instance segmentation. Four different tasks are defined and sequentially operated by different workers: image labelling (*Which objects appear in the image?*), instance spotting (users should click on each instance of a particular object), instance segmentation, and segmentation verification. Redundancy is introduced at every step except for the third one, which is the more time-consuming and is especially verified by the fourth task.

A final control of the annotations quality can be done at the end of the study. This is an approach adopted in PASCAL [72] for example, but also in Open Images [128] in which the expert annotators verify automatically derived labels, obtained through deep learning.

1.2.4 Annotation interaction usability

A common feature of all the points we have discussed in the three previous sections is that the goal of the annotation process is to obtain the largest possible number of annotations, of the highest possible quality, and at the lowest possible cost. Implicit annotation is cheaper but often difficult to put into place in practice. Expert annotators provide reliable annotations but are expensive, whereas a large number of less precise annotations can come at a cheaper cost when using crowdsourcing. Quality check helps ensuring annotations reliability but

introduce new expenses.

With that goal in mind, there also exists another variable which is rarely taken into account: the usability of the annotation interaction. Usability is a computer-human interaction concept that has been defined by Nielsen [154]. In essence, it describes the extent to which an interface can be mastered and used to efficiently perform the task it has been designed for. Nielsen also introduces five criteria that help measuring an interface usability: subjective satisfaction, easiness to learn and to remember, efficiency with respect to a certain task, and error robustness. It is really interesting to note that all of these criteria are relevant to image annotation; we want the users to be efficient, i.e. to provide good annotations in the minimum amount of time. We want the users to make as few errors as possible, and we also want them to easily learn and remember their task, once again, to be cost-effective. While usability is a core concept of computer human interaction, very few of the annotation tools that we described previously mention they want to optimize it.

Korinke et al. [123, 124] have studied how touch devices should be used to perform image segmentation. They compare several types of interactions and conduct two user studies to evaluate and compare these interactions. More recently, and while usability is not explicity mentioned, the work of Papadopoulos et al. [165] on extreme clicking is particularly interesting. The goal of the work is two-fold: reducing the annotator cognitive load and improving the quality of the annotations. The authors propose replacing the traditional bounding box drawing by clicking four extreme points (top, down, left and right) of the object and deriving the bounding box from these points. This interaction offers an interesting advantage: it reduces the time needed to appropriately draw a bounding box by a factor of 5. Users otherwise tend to start over multiple times due to the non-convexity of objects shape. It also provides richer annotations as four points lying on the object boundary are provided; the authors take advantage of this property to generate reasonable quality segmentations from this very weak information of only four points. Some subsequent work of the same group adopt similar approaches for image segmentation, providing users with the ability to correct segmentations by scribbling [4] or clicking [21].

1.3 Existing interactions for user-assisted segmentation

As image segmentation involves annotating images with a very rich amount of information, many interactions have been explored in the literature to provide users a way to bring semantic information to help existing segmentation algorithms. We review the interactions in this section and present briefly the algorithms that are attached to them.

The most intuitive methods are the ones that require the user to manually designate the contours of the object. The LabelMe tool [185] (Figure 1.5) is the most famous example of such an interface. The Web-based interface developed by the authors allows users to draw a polygon around an object. The segmentation obtained with this technique is not

necessarily precise at the pixel level, but is sufficient in many cases and has the advantage of being easily understood by users. In a variant of this technique called the Intelligent Scissors [148], the users click points on the contour of the object and a dynamic programming algorithm searches the optimal path that ties those points. There exists another variation of contour drawing called Soft Scissors [218]. One has to follow the contour using a soft constrained, size-adaptable thick contour brush, requiring less precision than exact polygon contour drawing.

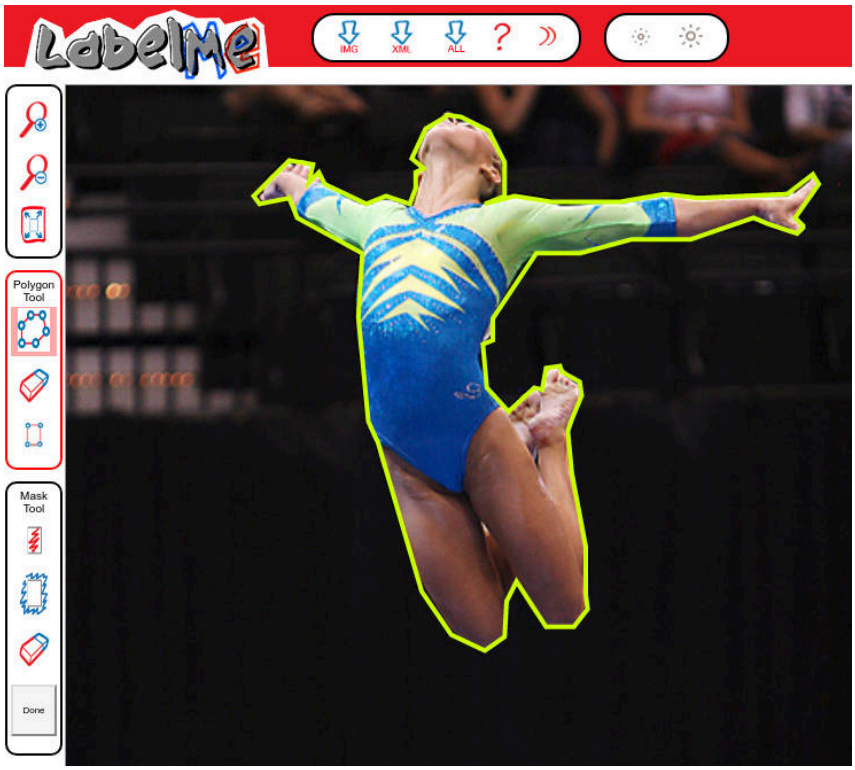


Figure 1.5: Visualization of an image annotated with the LabelMe tool [185].

A second possibility for interactive segmentation has been proposed by Rother et al. [183]. The user is only required to draw a bounding box around the object (Figure 1.6), which is used to learn a background model. The foreground is then obtained using iterative graph-cut and alpha matting. This method works very well for objects that distinctly emerge from a repetitive background. However in the case of complex scenes, the authors allow users to perform an additional refining step based on scribbles.

Scribbles form another category of interactions for segmentation, and are undoubtedly the most widely used (Figure 1.6). Users can typically draw foreground and background scribbles on the image, and receive a feedback on the current state of the resulting segmentation mask. Boykov and Jolly [27] use this input to build a trimap, i.e. a partition of the

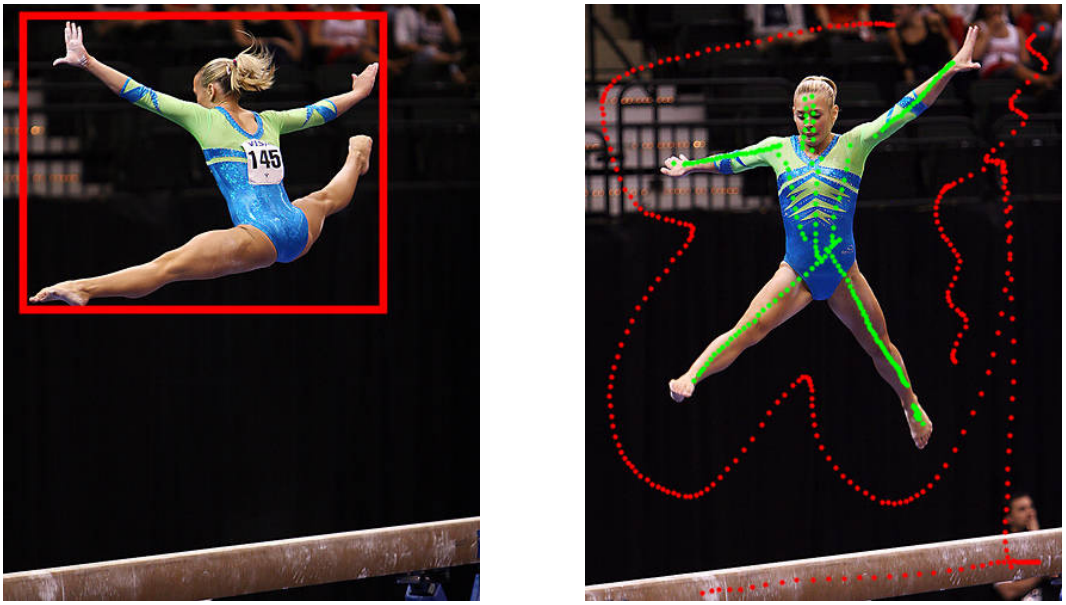


Figure 1.6: Example bounding box and scribbles interactions. On the left image, a user drew a bounding box around the gymnast. On the right image, a user drew green foreground scribbles on the gymnast and red background scribbles outside.

image into hardly constrained foreground and background regions, and a softly constrained in-between region. They run a graph-cut algorithm to find the optimal object boundary on the softly constrained region. McGuinness and O'Connor [143] describe how to use scribbles to segment an image using a Binary Partition Tree (BPT) [187]. The BPT is a hierarchy of image segments that can be used to propagate the foreground and background inputs between similar regions. Scribbles have also been used in the context of image co-segmentation [20], to provide foreground and background information across a set of images depicting the same object. As an alternative to scribbles, single foreground and background points have been used as input to select the best masks among a set of object candidates [37], or to guide the prediction of deep neural networks [21].

The mouse is used in most of these work as interaction device, which probably explains why outlines are rarely studied in the literature. Outlines are indeed tedious to perform with a mouse. However, most of the literature algorithms can take outlines as an input; in the work we present in the next chapter, we choose to use GrabCut to obtain a segmentation from the outlines.

Chapter 2

Outlining for Segmentation

Contents

2.1 Introduction 24
2.2 Outlining objects for interactive segmentation 26
2.2.1 Outline erosion 27
2.2.2 Blum medial axis algorithm 28
2.2.3 Enhancing foreground with superpixels 29
2.3 Experiments 29
2.3.1 Experimental setup 29
2.3.2 Usability metrics 32
2.3.3 Interaction informativeness 34
2.3.4 Segmentation quality 36
2.3.5 Discussion 37
2.4 Conclusion 38

Interactive segmentation consists in building a pixel-wise partition of an image, into foreground and background regions, with the help of user inputs. Most state-of-the-art algorithms use scribble-based interactions to build foreground and background models, and very few of these work focus on the usability of the scribbling interaction. In this chapter, we study a very intuitive interaction to non-expert users on touch devices, named outlining. We present an algorithm, built upon the existing GrabCut algorithm, which infers both foreground and background models from a single outline. We conducted a user study on 20 participants to demonstrate the usability of this interaction, and its performance for the task of interactive segmentation.

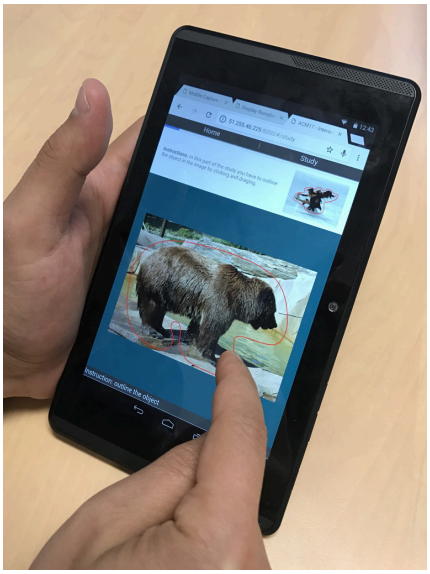
2.1 Introduction

The number of pictures that are captured, stored and shared online is growing everyday. In march 2017, Facebook reported that 300 million pictures were uploaded each day on their website. These pictures are increasingly used by companies and individual users, enabling new applications trying to improve everyday life. Object segmentation serves as an important step toward automatic image understanding which is key to those smart applications.

Object segmentation in an image remains a challenging task. This process of assigning a label to each pixel is very sensitive to the classical difficulties encountered in computer vision such as lighting conditions or occlusions. Recent advances in deep learning have enabled researchers to obtain state-of-the-art results [138] by training on the PASCAL segmentation dataset [73]. Some other techniques learn to infer a pixel-wise segmentation from weak annotations, i.e. bounding boxes around objects [166]. These methods are very promising but need huge amount of human labeled samples in order to train deep neural networks. Recent approaches have tried to overcome this issue, introducing active learning to train deep neural networks using a limited amount of selected samples [137] on the problem of image classification, but none of these methods have yet been applied on semantic segmentation.

Since fully automatic segmentation is still in many cases out of algorithms' reach, researchers have introduced the concept of interactive segmentation. This problem has often been approached with a task-driven point of view: what type of interaction may bring the necessary information to significantly help an algorithm achieve an acceptable segmentation? The users providing the interactions are often supposed to have a fair understanding of what segmentation is. This assumption is problematic, especially when putting into perspective the extraordinary amount of images to be annotated. That is why our target audience is composed of non-expert users who are not knowledgeable about image processing and segmentation. As a consequence, most of the existing work are not suitable to our problem. They rely on foreground and background scribbles requiring high cognitive load from the users.

Instead, we propose to use an intuitive interaction, outlining (Figure 2.1), that can be



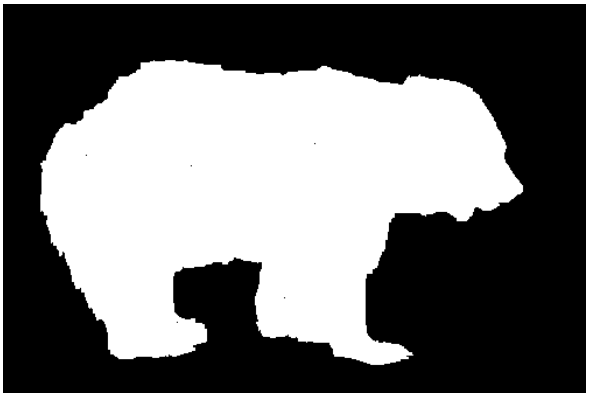


Figure 2.1: A user outlining an object on a touch device, and the resulting segmentation mask obtained with our method.

performed quickly and lead to good segmentation results while keeping users from entering a process of iterative segmentation refinement. This outlining interaction is particularly well suited for touch devices, which is appropriate considering the growing usage of tablets and smartphones compared to computers. All these properties make the outlining interaction very interesting for crowdsourcing segmentation annotations on thousands of images, with non-expert users.

We present two main contributions in this work: first, a modification of the GrabCut algorithm that takes as input an outlining interaction, instead of a bounding box. We take advantage of the free-form shape drawn by the users to extract information about foreground (using the Blum Medial Axis computation) from a background annotation (the outline). The second contribution of this work is the usability comparison of various interactions used in interactive segmentation. We argue that the outline offers the advantage of being a quick, easy-to-understand and usable interaction while providing a high amount of supervision to obtain a good segmentation.

The rest of the chapter is organized as follows. Since we have already extensively described the state-of-the-art in the previous chapter, we first introduce the outlining interaction along with our method to compute segmentation masks in Section 2.2. We then present in Section 2.3 our experiments and the results showing that our simple interaction leads to segmentations of good quality.

2.2 Outlining objects for interactive segmentation

In this section we detail why we use outlining interactions, and our method to compute segmentation masks from those.

As stated in the previous section, most of prior crowdsourcing campaigns in image segmentation have asked users to draw a polygon around the object of interest. This interaction has some merit in terms of usability: it is straightforward to understand, and does not require iterative refinement from the user. In addition, the user does not have to evaluate the quality of the produced segmentation mask to know when to stop interacting. When the polygon is drawn, the segmentation is over.

However, we have two main concerns with this interaction. First, it is tedious and time consuming. It requires users' full attention, in order to precisely click on the object boundary. It also requires users to implicitly determine the number of edges of the polygon they should draw. A second limitation of this interaction is the pixel-wise quality of the segmentation mask obtained. Shape details and curved boundaries can only be approximated by a polygon, and their quality is correlated with the time the human annotator is willing to spend annotating.

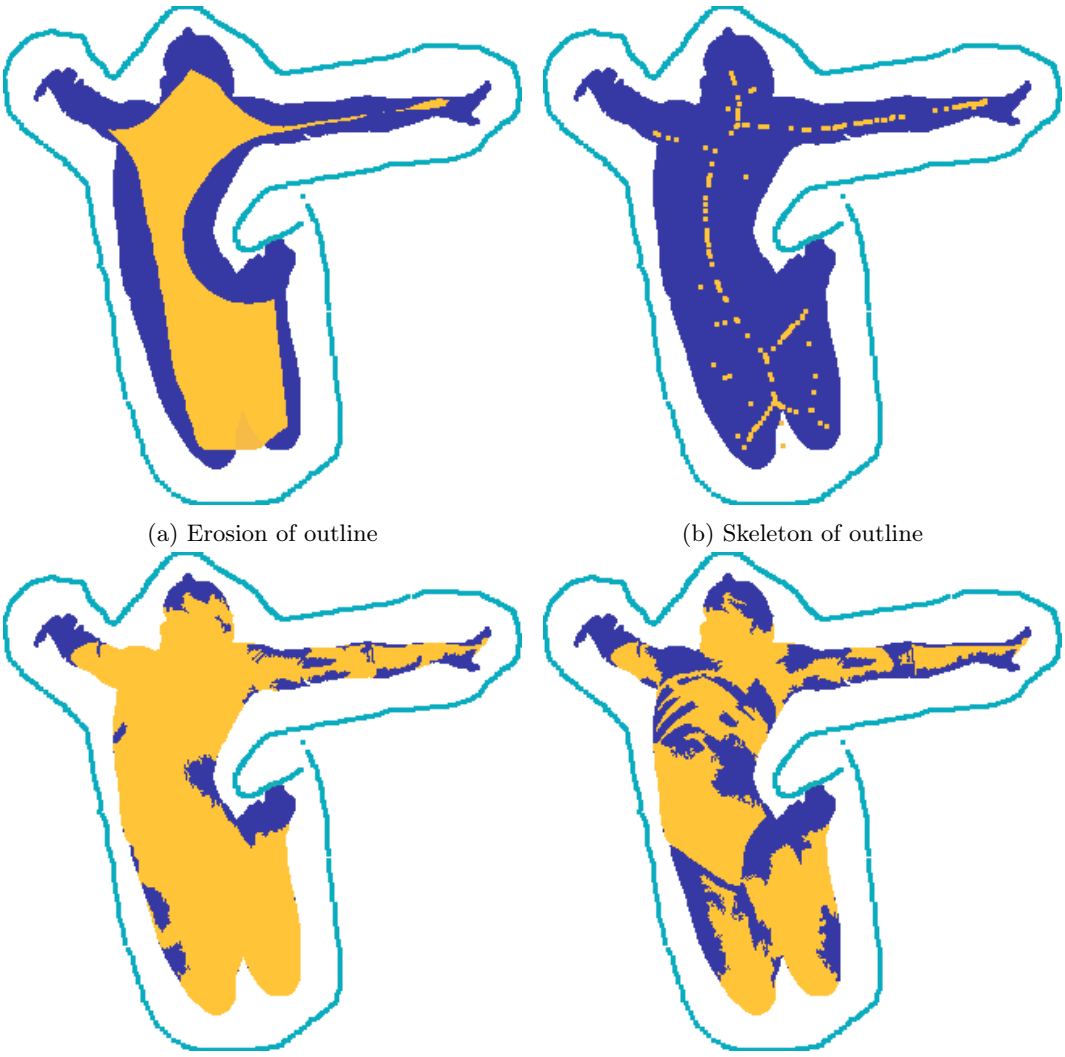
Outlining an object has the same merits than drawing a polygonal shape around the object: the task is easily defined, and it is easy for a user to assess the quality of an outline. It also adresses the first limitation of the polygons: since it requires less precision in following the object boundaries, it is less tedious and time consuming. It has however an important drawback: it does not provide an accurate segmentation.

In order to address this problem, we choose to rely on the popular GrabCut algorithm [183]. The original GrabCut takes a bounding box as an input. It considers every pixel outside of the bounding box as fixed background, and aims at separating foreground from background inside the bounding box. To this end, a background model is estimated from the fixed background, and a foreground model is estimated from the pixels inside the bounding box. The likelihood of each pixel inside the bounding box to be foreground or background is then estimated, and graph-cut is applied to obtain a temporary segmentation mask. This mask is then used to update the foreground and background models, and the process is iterated until convergence.

In our implementation, we slightly alter the GrabCut algorithm to take into account a major difference between outlines and bounding boxes: we can make stronger assumptions on the foreground positions from an outline than from a bounding box by looking at the general shape of the outline. We restrict the initial foreground model computation to the pixels that are most likely to be foreground, which decreases the number of iterations needed for convergence and improves the segmentation quality.

In the rest of the section, we explain two different methods to infer foreground from the ouline shape: the first method consists in eroding the outline, and the second is based on the Blum medial axis computation. We then post-process the foreground pixels using

superpixels.



(a) Erosion of outline

(b) Skeleton of outline

(c) Erosion of outline extended with superpixels

(d) Skeleton of outline extended with superpixels

Figure 2.2: Different foreground inferring methods from a user outline. The ground truth mask is in dark blue. The user outline is in cyan. The inferred foreground is in yellow.

2.2.1 Outline erosion

The simplest method to obtain points that are likely to be foreground from an outline is to apply morphological erosion of a mask representing the inside points of the outline. We use a disk as a structuring element for the erosion, and the only parameter of this method is the radius of the disk.

In our implementation, the disk radius is specific to each user and computed by studying

the outline performed by the user on a reference image. We compute the mean m_d and standard deviation s_d of the distance d from each outline point to the ground truth mask. Assuming the user consistently outlines all images, i.e. the mean distance of the user outline to an object is more or less constant across all images, a disk radius equal to $m_d + 2 \cdot s_d$ should produce an eroded outline that is almost certainly completely foreground.

An example of this process can be visualized on Figure 2.2a. The eroded outline (yellow) is almost entirely contained in the ground truth mask (dark blue).

2.2.2 Blum medial axis algorithm

In shape analysis and model animation, the Blum medial axis transform [25] is one of the most popular tools. The Blum medial axis of a shape is composed of the centers of the circles that are tangent to the shape in at least two points. It is especially appropriate to compute skeletons, composed of the medial axis points inside the shape.

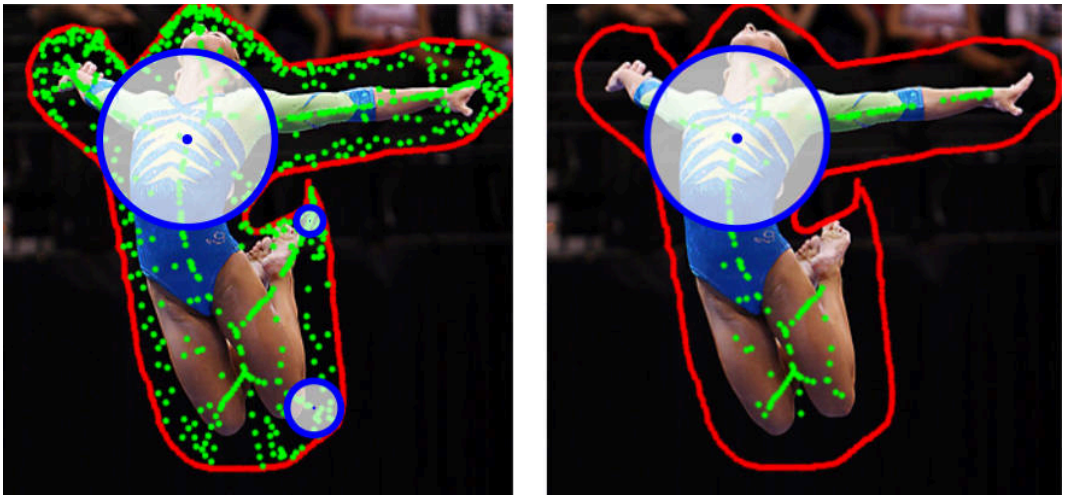


Figure 2.3: Skeleton (in green) computed using the Blum medial axis algorithm from an outline (in red). Few example disks are shown in blue. In the image on the left, all disks centers (green points) are kept, generating a very noisy skeleton. In the image on the right the skeleton is pruned, by filtering out centers of small disks.

One of the problems of the medial axis algorithm is its stability when the shape frontier is noisy. It tends to create a high number of branches (Figure 2.3), which deteriorates the simplicity of the skeleton, and incidentally the comprehension of the shape. In our case, this is rather an advantage. Indeed more ramifications lead to a higher number of points inside the shape for our foreground scribbles. However, we need to filter the inside points, since those close to the outline have a high probability of being outside of the object to segment. Radius of the inside circles of medial axis points constitute a good filter option because the medial axis points with the smaller radius are typically close to the outline. In

our implementation, we choose to keep only centers with a radius higher than half the larger radius. Figure 2.2b depicts a ground truth mask in dark blue, a user outline in cyan and the filtered medial axis points in yellow. Most of the yellow points fall inside the ground truth mask, thus making it a good starting point to learn the foreground model.

2.2.3 Enhancing foreground with superpixels

These two methods, Blum medial axis and outline erosion, allow to select foreground points that make a valuable input to the GrabCut algorithm. However, we add a post-processing step to (i) extend this foreground information and (ii) filter as much false foreground points as possible.

To do so, we compute a superpixels segmentation of the image, i.e. an oversegmentation that groups neighbouring pixels with similar colorimetric properties. We (i) extend the foreground labels from pixels to the superpixels they belong to. This considerably increases the surface of the foreground region. In addition, we (ii) handle conflicting superpixels, which contain both pixels denoted as foreground and a piece of the outline, by removing them from the foreground mask. An example of the result can be seen on Figure 2.2c and Figure 2.2d. Note that the errors arising from the first step (between the knees in Figure 2.2a and Figure 2.2b) have successfully been removed in the post-processed inferred foreground mask.

We choose to use the Mean-Shift superpixels [51] because no compacity constraint is used in their computation. As a consequence, a superpixel can cover a large area (especially in the case of similar background regions, such as an homogeneous sky) and will more likely correct wrongly inferred foreground points.

2.3 Experiments

In this section we describe the setup of our experiments and analyze the outcome of the study.

2.3.1 Experimental setup

Interactions Since the subject of the study is interactive segmentation on touch devices, we choose to compare only three annotations: outlines, scribbles, and bounding boxes. We do not include polygon drawing since it is clearly not adapted to a touch device. Indeed, fingers are too big to precisely touch the boundary of an object, they would hide the area where the user should try to place the vertex on.

The interfaces are kept as simple as possible. The user is shown an image and has to provide a valid input to be allowed to move on to the next image.

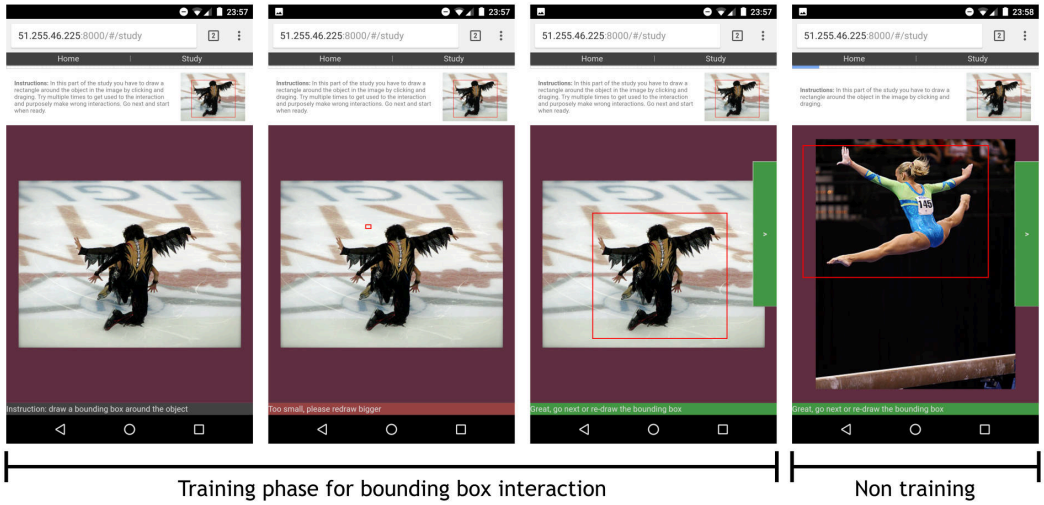


Figure 2.4: Several screenshots of the bounding box interface during training (left) and study (right) phases. Some minimal input validation is performed, such as checking that the surface of the annotation is above a minimal threshold. If not, a message with a red banner is displayed at the bottom of the screen as in the second image above.

The bounding box interface allows the user to draw a rectangle over the image using a touch and drag interaction (Figure 2.4). If the user is not satisfied with their previous attempt, they can start over, which will replace the former rectangle with a new one. The user can only move on to the next image when the current rectangle is of sufficient size (we discard rectangles that are too small to avoid common mistouch issues).

The outlining interface is very similar to the bounding box interface. The user can draw the outline using a touch and drag interaction; the system automatically draws the closing segment between the ending and starting points when the user releases their finger. The user can also start over if not satisfied with the current outline. The system allows the user to move on to the next image if the outline is of sufficient area. In addition, for the training image only, the system checks the absence of loops in the outline path (Figure 2.5), for they may reveal incorrect usage. This loop detection feature is deactivated for the other images to limit its impact on the interaction and user frustration.

The scribbling interface displays three buttons: one to select the foreground scribbles, which are drawn in green, one to select the background scribbles, which are drawn in red, and one to remove the last drawn scribble. Users are required to provide at least a minimum scribble length to be allowed to move on to the next image.

Device and software We use a regular 8” android tablet, for which the buttons appear large enough to be easily clickable. The user study is conducted on a Web application in the Chrome browser for android. The code for this study (Web client and server), as well as the results presented here are all available online (github.com/mpizenberg/otis).

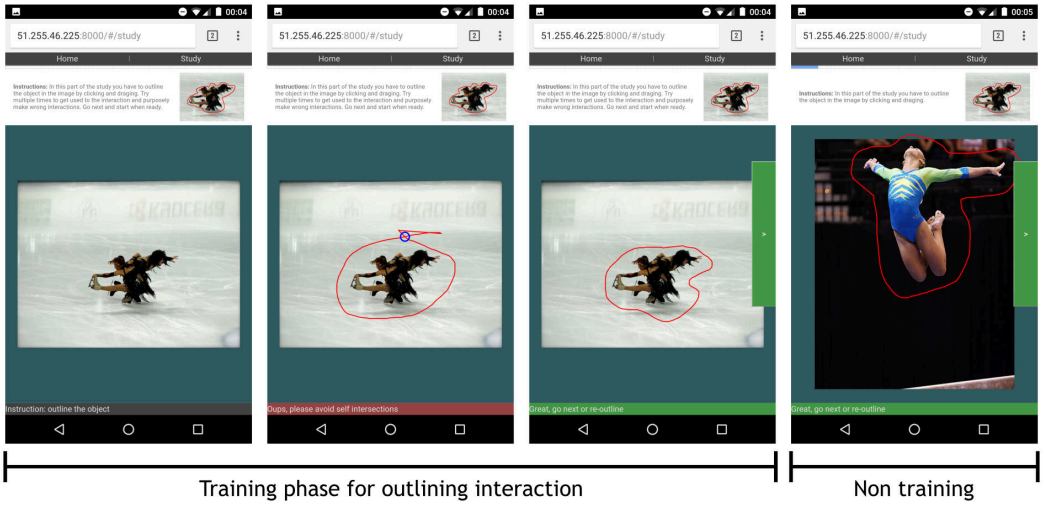


Figure 2.5: Several screenshots of the outlining interface during training (left), and study (right) phases.

Images We select 36 images from the iCoseg dataset [20], which we divide into 3 groups of 12 images. We want the segmentation results to be comparable between different interactions, but since each user tests the three interfaces, we do not want the same images for every phase of the study. This would risk biasing the results since users might get annoyed of annotating three times the same images, affecting the quality of their annotations. The iCoseg dataset provides multiple images depicting the same object in different situations so we use similar images in the 3 groups. Examples of these images can be seen on Figure 2.7.

Methodology The protocol of the study is as follows.

The users are not explained the concept of segmentation, we tell them that we require annotations on images, and that we wish to compare three interactions to provide those annotations.

The study is composed of three steps, one step per interaction. For each step, the evaluator first explains the user how the interaction works, and demonstrates it on a training image. The evaluator demonstrates good and bad examples of interactions. Then the user tests the interaction on the same training image. The evaluator can correct the user and criticize or validate the users interactions. Once the user understands the tool, the eleven other images are proposed for interaction, without any help or guidance from the evaluator. Finally, at the end of each step, the user answers two questions about the interaction. In order to limit bias, the order of the interactions is randomized, as well as the order of appearance of each image during each step.

Among the eleven images annotated by the user, one is considered the reference. It is introduced to (i) check whether the user is performing the task correctly (this is particularly useful in a crowdsourcing context), and (ii) to learn the radius of the erosion disk for this

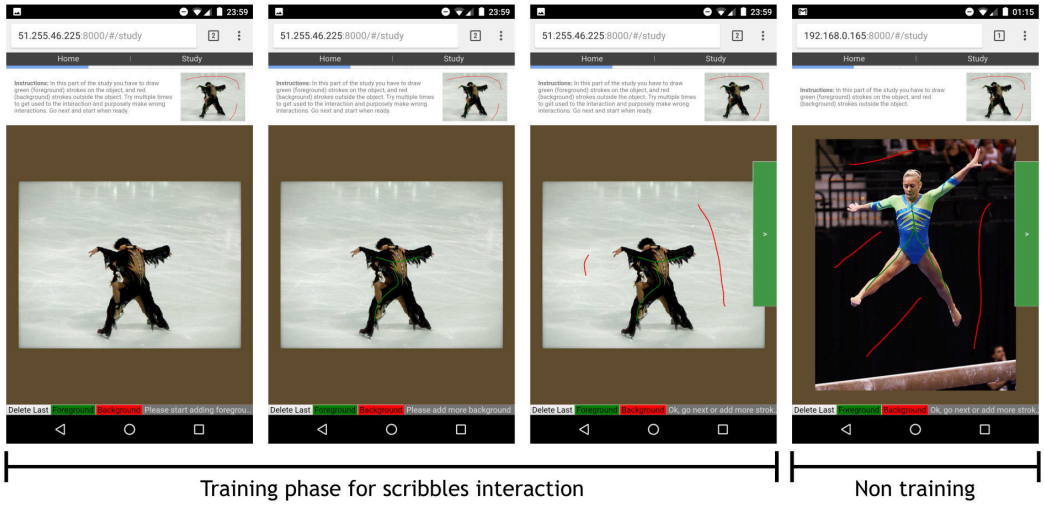


Figure 2.6: Several screenshots of the scribbling interface during training (left), and study (right) phases.

specific user (see Section 2.2.1).

The following two questions are asked at the end of each step of the study.

- Overall, I am satisfied with the ease of completing the tasks in this scenario.
- Overall, I am satisfied with the amount of time it took to complete the tasks in this scenario.

Users can answer on a scale from 1 (strongly agree) to 7 (strongly disagree). We choose to ask only these two questions since we are not trying to assess the usability of a whole system, but only of an interaction. A standard usability questionnaire, such as SUS (used in [124]), was not really adapted to our use case and instead we extracted these two questions from a popular post-task questionnaire (ASQ, After Scenario Questionnaire).

Finally at the end of the study, we ask users to rank the three interactions in their order of preference (see "Rank" in Table 2.1).

Participants Twenty users (10 Male, 10 Female) participated to this study, with ages ranging from 25 to 55 years old. Most users have no experience in image segmentation, some of them are familiar with the concept.

2.3.2 Usability metrics

Among the criteria stated by Nielsen [154] as defining the usability of a system, we evaluate efficiency, errors, and user satisfaction. Efficiency designates the swiftness with which users are able to complete the tasks once they learn how to interact with the system. We evaluate this criterion both subjectively, by asking users about their perception of the time they spent on the task (table 2.1), and objectively by measuring the time it takes to complete their



Figure 2.7: Some images from the iCoseg dataset.

interactions on each image (Figure 2.8). User satisfaction is measured through our questionnaire, both by the question on the perceived task easiness and the interaction ranking. Finally, errors are measured by counting the number of times users repeat interactions. We record how many times bounding boxes and outlines are re-drawn, and the number of clicks on the *Undo last scribble* button for the scribbling interaction (Figure 2.9).

Method	Bounding box	Outline	Scribble
Ease	2.1 ± 0.62	2.65 ± 0.74	2.1 ± 0.61
Time	2.35 ± 0.69	2.5 ± 0.67	2.6 ± 0.70
Rank	1.95 ± 0.43	1.90 ± 0.32	2.15 ± 0.37

Table 2.1: Results of the questionnaire with a 95% confidence interval.

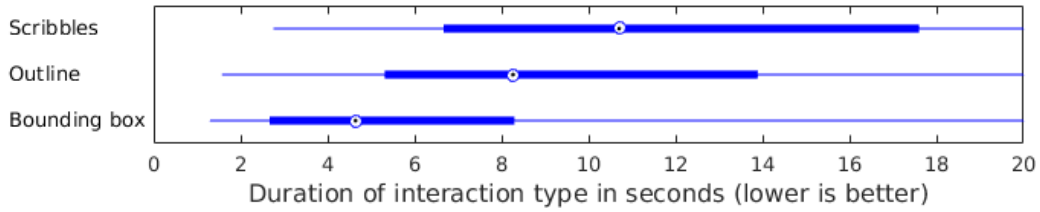


Figure 2.8: Duration of interactions on all images and all users. The dots are the median durations, and the thick blue line delimits the first and third quartiles.

Overall, the questionnaire results can not allow us to conclude on the superiority of one interaction method over the others. Although slightly in favor of the bounding box interaction, the perceived ease and time are not statistically better for any of the three interactions. However, the results are all between 2 and 3 (on a scale from 1 to 7), which means users were mostly satisfied with all three interactions. We can note that the time perception results

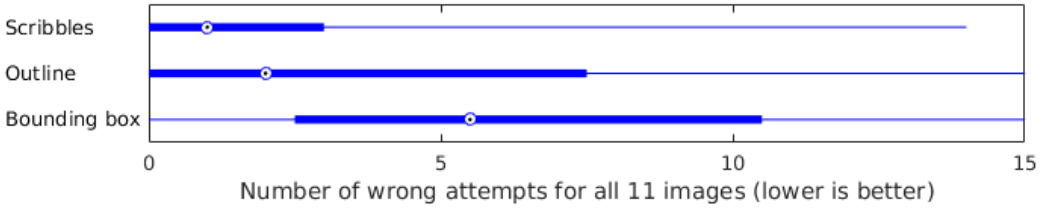


Figure 2.9: Number of errors per interaction and per user on all images. The dots are the median number of errors, and the thick blue line delimits the first and third quartiles.

(table 2.1) are correlated with the objective duration of interaction (Figure 2.8), measured during the experiments. The bounding box is the quickest interaction, while the scribbles suffer from the time needed to switch between foreground and background scribbling.

Surprisingly, the outline ranks first in the users preference (although not significantly), ahead of the bounding box interaction. The reason of this observation, as explained by many of the participants during the experiment, is due to the frustration that can arise when trying to draw a bounding box around a non-convex object. Users trying to draw the bounding box close to the object boundary often need several attempts, because of the difficulty to position the first bounding box corner. This issue is visible on Figure 2.9, which shows the high number of errors for bounding boxes. Errors occurring with the outline interaction are mostly due to high speed interactions, or due to masking the object with their hand during the interaction for users less familiar with touch devices.

2.3.3 Interaction informativeness

We define the background area of user inputs as follows. For a bounding box (resp. outline), the background area is composed of all pixels outside of the bounding box (resp. outline). For scribbles, the background area is the union of the superpixels annotated as background (containing part of a background scribble).

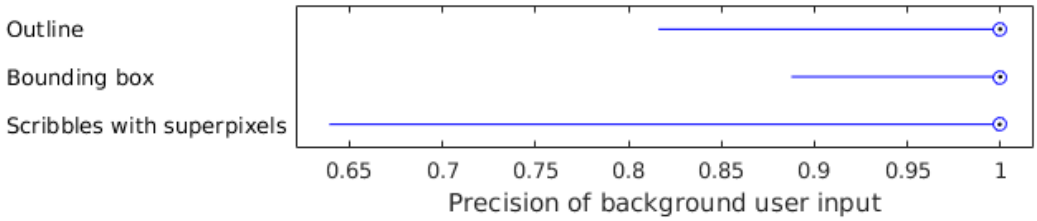


Figure 2.10: Precision of background user input.

Looking at the precision of background user inputs (Figure 2.10) we see that more than 75% of user annotations are perfect (a precision score of 1). This means that 75% of user inputs do not intersect at all the object of interest. We can conclude that users understand well the tasks they are given.

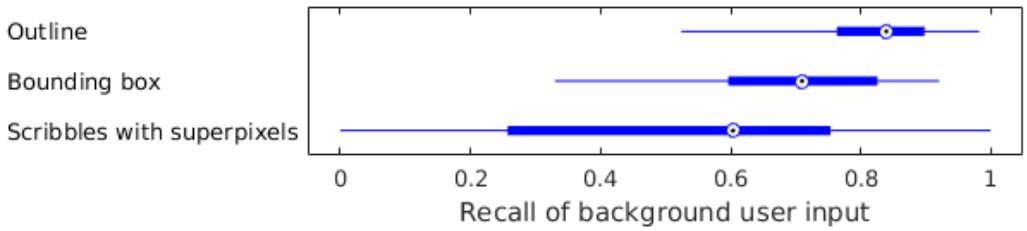


Figure 2.11: Recall of background user input.

In order to estimate the informativeness of an interaction, we also measure the recall index (Figure 2.11). It indicates the percentage area of all background that is annotated by an interaction. With no surprise, outlining is the more informative since it is often very close to the boundary of the object (Figure 2.16) and thus, the outside of the outline covers most of the image background. Background (red) scribbles are the least informative here since only superpixels that are scribbled over count as background information.

Except for the foreground (green) scribble interaction, we do not have raw foreground annotations. We thus define the foreground input area as the inferred foreground (through erosion or medial axis computation, extended by superpixels as explained previously).

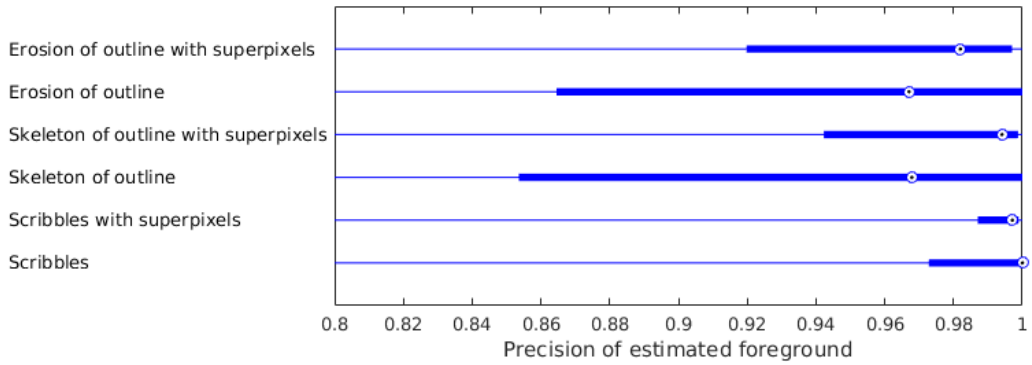


Figure 2.12: Precision of foreground user input.

The precision of foreground area is given in Figure 2.12, relatively to the ground truth masks. We can observe that more than 75% of foreground (green) scribble inputs are over the 0.97 index. This means that the task of scribbling inside the objects is globally well performed but still slightly harder than background (red) scribbles. It is explained by the fact that objects can have thin shapes and thus not precisely locatable under the finger during the touch interaction.

Using the superpixels extension of the scribbles, we observe that the smart background correction mentioned in Section 2.2.3, enhances the 75% index to a precision of 0.99. With the two foreground inference techniques (erosion and skeleton), the improvement provided by the superpixels extension is obvious.

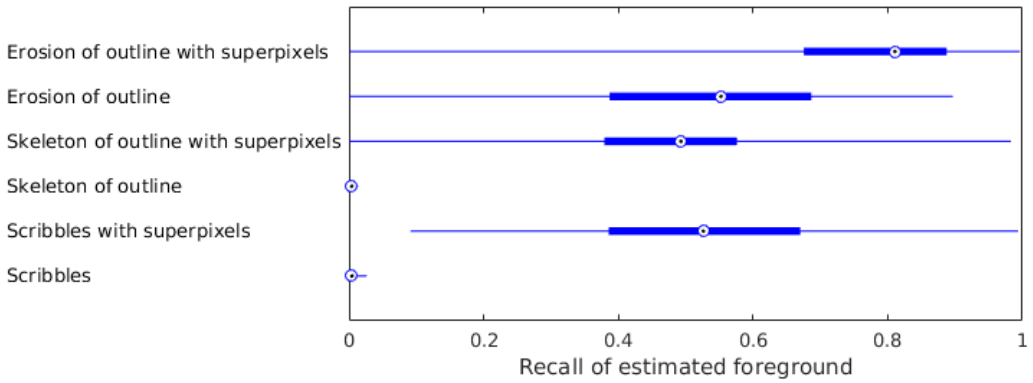


Figure 2.13: Recall of foreground user input.

The recall of foreground area (Figure 2.13) provided by these interactions, extended through superpixels is also coherent with what we observe in Figure 2.2. Skeleton and scribbles recall values are almost 0 since they are of dimension 0/1 (points/lines) for a measure of surfaces (dimension 2). Erosion provides the most foreground information, but has the lower precision rate (Figure 2.13). We will show in the next section that this trade-off is worth exploring.

2.3.4 Segmentation quality

We compute the resulting segmentation of images using five different methods. As a reference method, the mean Jaccard index obtained with foreground and background scribbles is 0.79 (Table 2.2). When using bounding boxes, that provide a more complete background model input for the GrabCut algorithm, the mean Jaccard index increases to 0.82. As expected, it increases even more when using outlining interaction inputs, providing richer inferred initial foreground models to the GrabCut algorithm. The higher scores (0.88 and 0.89) are respectively obtained when using the erosion and skeleton processing of the outline. The best performance is achieved using the skeleton processing, which tends to show that for the results presented in the previous section, the precision of the foreground user input is more relevant than its recall.

Method	Scrib.	B. Box	Outl.	Outl. + er.	Outl. + BMA
Mean Jaccard	0.79	0.82	0.86	0.88	0.89

Table 2.2: Mean Jaccard index obtained on all images for all users for each interaction.

Perhaps more importantly, the outlining interaction enables reaching consistently higher Jaccard index than the other techniques. In Figure 2.14, we observe that the first quartile is always higher than 0.8 with variants of the outlining interaction. Some final segmentation results are visible in Figure 2.15 and show the clear improvement brought by an outline over

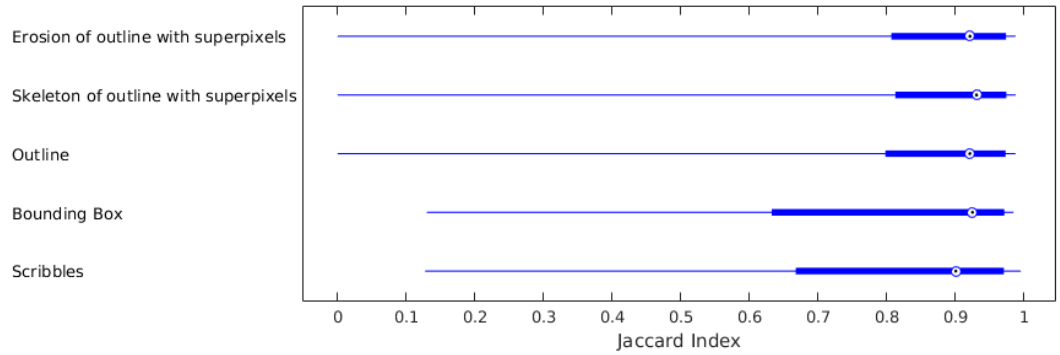


Figure 2.14: Jaccard index obtained on all images for all users for each interaction type.

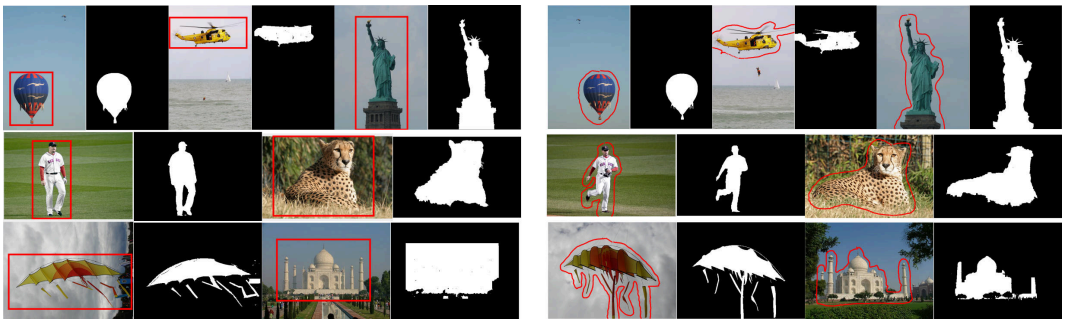


Figure 2.15: Segmentation results for bounding box and outlining interactions from a user.

a bounding box.

2.3.5 Discussion

All the results we obtained confirm the good properties of the outlining interaction in the perspective of being used in a segmentation crowdsourcing campaign.

First, it is a very straightforward interaction. One of the users explained it in these terms: *outlining is easier since you do not need to think, just trace the object. Bounding boxes are tougher, particularly in determining a correct size, and scribbles is too much thinking and a bit more time consuming.* Another user said: *It's actually more fun to draw around object and would seem to me less tiring than the other methods.* The usability criterion points out that outlining might be slightly less usable than drawing a bounding box or scribbling, but remains a very usable interaction.

Another interesting property of the outlining interaction is the speed at which it can be performed. Figure 2.8 shows that most of the outlines were produced in less than 10 seconds, which is very reasonable considering some of the images we chose have complex shapes (Figure 2.16).

The quantity of information brought by outlines is also very good, as discussed in the



Figure 2.16: Outlines drawn by the third user on three images with complex shapes.

previous section, especially when balanced with the interaction usability. This information is of course less complete than a polygon drawn on the boundary of the object (such as in LabelMe), but can be augmented using computer vision techniques (Blum medial axis, superpixels, GrabCut, etc.) and lead to very good segmentation masks. The average Jaccard index of 0.89 obtained with the outlines is particularly impressive considering there was no interactive refinement step, and it was performed in less than 10 seconds in average (see for example the comparison with Jaccard index vs. time curves described in [37]).

2.4 Conclusion

In this chapter, we evaluated the outlining interaction on touch devices for interactive segmentation. We found that outlining is a simple and natural interaction, allowing to quickly obtain accurate information on the location of an object. This information can be augmented with foreground inference, and then used to compute a segmentation mask. The segmentation masks obtained with this method reach an average Jaccard index of 0.89, which is a very good result considering the interaction does not require any knowledge on image processing or computer vision from the user.

Thanks to all these good properties (simplicity, swiftness, accuracy), outlining appears to be an interesting avenue to explore for the gathering of large datasets of image segmentation masks. Those datasets are crucial to bring automatic image segmentation algorithms, today mostly based on deep learning techniques, to a new level of effectiveness. It is our intention to pursue this goal so we will next introduce an annotation Web application built to easily start a crowdsourcing campaign on Amazon Mechanical Turk.

Chapter 3

Reliable Web Applications

Contents

3.1 What is the Web? 40
3.1.1 What is a Web application? 40
3.1.2 Rich Web Application 42
3.2 JavaScript, formally known as ECMAScript 42
3.2.1 Genesis of JavaScript 42
3.2.2 Browser performance 42
3.2.3 Explosion of JavaScript 44
3.2.4 JavaScript issues 45
3.2.5 JavaScript as a compilation target 50
3.3 Frontend Web programming 53
3.3.1 Single Page Application (SPA) 53
3.3.2 Reactive programming 54
3.3.3 Virtual DOM 55
3.3.4 How to choose? 57
3.4 Elm 59
3.4.1 Pure functions 59
3.4.2 Algebraic Data Types (ADT) 60
3.4.3 Total functions 62
3.4.4 The Elm Architecture (TEA) 64
3.4.5 Elm-UI, an alternative layout strategy 65
3.4.6 Reliable packages 67

The best way to crowdsource an annotation campaign is to provide a Web application. Since online annotators are paid for the task, we need the Web application to be as reliable as possible. Therefore, in this chapter we review evolutions of the Web since its creation in 1991, especially regarding the development of reliable frontend applications. In particular, we describe how the Elm programming language can help us build a bug-free annotation Web application.

3.1 What is the Web?

The Internet and the Web are ubiquitous technologies of our everyday lives, that flourished around the 80's. Inter-connected networked appeared as early as in the 60's. ARPANET, founded by the Advanced Research Projects Agency (ARPA) in 1969, standardized the communication protocols named TCP/IP in 1982 for its network. These are the protocols still in use on the Internet today. In August 1991, Tim Berners-Lee who had been working at CERN for the previous seven years, shared a summary of his World Wide Web project, including the HyperText Transfer Protocol (HTTP), the HyperText Markup Language (HTML), and the first Web browser. Social media, communication, search, news, entertainment, mapping, shopping, learning, virtually any activity is now digital and online. Simply put, the Web, also called World Wide Web (WWW), consists of the sum of all resources, available through unique identifiers (URI), that we share on the Internet, the global network carrying them.

In this chapter, we will recap the Web main evolutions, from static content to dynamic applications, and explain the choices we made to build reliable annotation Web applications.

3.1.1 What is a Web application?

An application, in the context of programming (/computers), is a piece of software presenting information to a user, usually in an actionable manner. This includes programs like email clients, image editors, video games, word processors, automatic translators, and virtually any functionality available on a regular computing device.

Web resources are commonly accessible through a Web browser. Thus, we can define a Web application as a user-facing software, accessed through a Web browser. As of May 2019 according to statcounter [28], the most used Web browsers are Google Chrome (62.7% of global market share), Apple Safari (15.9%) and Mozilla Firefox (5.1%).

The three pillars of Web applications are HTML, CSS and JavaScript. HTML, for "HyperText Markup Language" is a description language organizing a page information as a hierarchy of tagged content. In Listing 3.1, a `body` tag contains four other tags, a title `h1` (h for header), a paragraph `p`, an image `img` and a button not yet linked to any action. This hierarchical organization of an HTML page is call the DOM, for "Document Object Model". CSS, for "Cascading Style Sheet", complements HTML by styling the content of associated HTML documents. Listing 3.2 shows how one would add a left margin of 20 pixels on all

the document body, and make the h1 title red and bold. Finally, JavaScript is a scripting language, not affiliated in any form to the Java programming language. It is run inside the browser to add dynamic behavior to a Web page. In Listing 3.3 we show how one could count and display the number of times a user clicked on the button in the page.

```
<body>
   <h1>Example Title</h1>
   <p>Followed by a paragraph of text and an image.</p>
   <img src="/image/file.jpg" alt="Image description"/>
   <button id="the-button" type="button">Click me!</button>
</body>
```

Listing 3.1: Example HTML code.

```
body {
  margin-left: 20px;
}
h1 {
  color: red;
  font-weight: bold;
}
```

Listing 3.2: Example CSS code.

```
// Function creating a paragraph element containing
// only the number given in parameter.
function makeParagraph(n) {
  let node = document.createElement("p");
  let text = document.createTextNode(n.toString());
  node.appendChild(text);
  return node;
}

// Create a reference to the button in the HTML document.
let theButton = document.getElementById("the-button");

// Global counter to keep track of the clicks.
let count = 0;

// Attach an event triggering on clicks on the button.
// When clicking we add a paragraph containing
// the number of times we clicked on the button.
theButton.addEventListener("click", function() {
  count = count + 1;
  let newParagraph = makeParagraph(count);
  document.body.appendChild(newParagraph);
});
```

Listing 3.3: Example JavaScript code.

3.1.2 Rich Web Application

Traditionally, websites used to present their resources in the form of a collection of static documents, known as Web pages, linked together with hyperlinks. The nature of Web pages would mostly be informative, visual or textual, with very few other interactions than navigation through the site by clicking on the links.

Today, thanks to evolutions of Web technologies that we will detail later, Web applications have become full-fledged applications with almost the same capabilities as desktop ones. They feature functionalities like 3D graphics, audio processing or interactive elements, and are sometimes called rich Web applications. Associated concepts such as "single page applications" (SPA) are also explained in the following sections. In the next section, we will dive into the cornerstone of Web pages dynamism, JavaScript.

3.2 JavaScript, formally known as ECMAScript

3.2.1 Genesis of JavaScript

In 1995, the dominating Web browser was the Netscape Navigator. Realizing that pages dynamism was key in the competition against Microsoft's own Web technologies, Netscape Communications recruited Brendan Eich, with the aim of integrating a scripting language into their browser. A first prototype was thus developed in 10 days (May 1995). Assumably for marketing reasons, it was officially named JavaScript when released in Netscape Navigator 2.0 beta 3.

Two years later, in June 1997, the European Computer Manufacturers Association (ECMA) standardized the first version of "ECMAScript" as ECMA-262, JavaScript being its most well-known implementation. The ECMAScript (ES) standard has been evolving ever since. Today, all browsers fully implement ES5, released in 2009, and partially implement the most recent versions, ES2015, ES2016, ES2017 and ES2018.

3.2.2 Browser performance

In this section, we are particularly interested in the wide performance improvements of the JavaScript engines, starting around 2008 when Google released its Chrome browser. On September 2, 2008, Google announced a new Web browser called Chrome [91]. Its main selling feature was a new JavaScript engine called V8, greatly improving the browser performances on Web applications making heavy use of JavaScript like their email client Gmail. Note that performance in a browser depends on many factors such that network latency, DOM computation, page rendering or JavaScript processing. In this section, we will specifically focus on JavaScript execution performances.

Dynamic interpretation

JavaScript was originally an interpreted language. For each line of code, the engine would translate it into machine code, and immediately execute it. This means that for a loop, the same transformation from JavaScript to machine code is repeated over and over again. In addition, JavaScript is a dynamic language, which is both one of its strongest points and a huge drag on execution. Let's take the function adding two numbers depicted in Listing 3.4 as an example.

```
1 function add(x, y) {
2   return x + y;
3 }
```

Listing 3.4: Adding two values.

The following text until "NOTE 2" is a quote from the ECMAScript specification [65] detailing the complicated process evaluating an addition.

> the addition operator either performs string concatenation or numeric addition. The production "AdditiveExpression : AdditiveExpression + MultiplicativeExpression" is evaluated as follows:
>
> 1. Let lref be the result of evaluating AdditiveExpression.
> 2. Let lval be GetValue(lref).
> 3. Let rref be the result of evaluating MultiplicativeExpression.
> 4. Let rval be GetValue(rref).
> 5. Let lprim be ToPrimitive(lval).
> 6. Let rprim be ToPrimitive(rval).
> 7. If Type(lprim) is String or Type(rprim) is String, then
>   (a) Return the String that is the result of concatenating ToString(lprim) followed by ToString(rprim)
> 8. Return the result of applying the addition operation to ToNumber(lprim) and ToNumber(rprim). See the Note below 11.6.3.
>
> **NOTE 1**. No hint is provided in the calls to ToPrimitive in steps 5 and 6. All native ECMAScript objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Host objects may handle the absence of a hint in some other manner.
>
> **NOTE 2**. Step 7 differs from step 3 of the comparison algorithm for the relational operators (11.8.5), by using the logical-or operation instead of the logical-and operation.

In theory, if we know that we will only use this function to sum two numbers, it should compile to a single instruction. However, due to the dynamic nature of JavaScript, as specified in the standard, the code has to check if the arguments are strings, objects, and proceed first with conversions before eventually reaching the instruction that actually computes the addition. This process results in one or two orders of magnitude slower code, compared to statically typed languages like C or Java.

Just-in-time (JIT) compilation

Statically typed languages usually compile code ahead-of-time (AOT), while dynamically typed languages interpret code at runtime. Starting with Chrome in 2008, all browser vendors began implementing just-in-time (JIT) compilers.

The key ingredient is a "monitor" sometimes called "profiler". The monitor watches the code while it is run by the interpreter, and keeps track of how often a piece of code is executed. Once a path of code is found to be repeatedly executed, it becomes "hot", which triggers an optimizing compiler. According to the types previously used in the hot path, the optimizing compiler will make assumptions enabling extremely efficient machine code. If the same code is used once with different types however, it gets de-optimized back to the baseline compiler. Multiple optimization and de-optimization round trips hinders the performances, and consequently will permanently mark the section as not to be optimized anymore. For more information on JIT compilation, Lin Clark [44] wrote an enlightning introductory blog post.

3.2.3 Explosion of JavaScript

Node.js

Not long after the release of the V8 engine from Google, Ryan Dahl announced at the European JSConf of 2009 a new project named Node.js [158]. As he explains in his talk [57], Node is a cross-platform JavaScript runtime environment based on V8. It features an event-driven architecture, with non-blocking input/output (I/O) APIs. The project matured from the observation that blocking I/O is extremely non-efficient, since it requires many threads and a large memory to scale with connections. Being event-driven by nature in the browser, JavaScript was a perfect fit for the Node project.

In order to provide non-blocking asynchronous I/O, Node is composed of an event loop managing callbacks in queued fashion, and of a thread pool, executing all blocking I/O calls like file reading. Both are abstracted away by the system, and so a user simply has to provide callbacks that will automatically be run upon completion of I/O tasks. An example of reading a file is presented in Listing 3.5.

```
1 let fs = require('fs');
2 fs.readFile('file.txt', 'utf8', function(err, contents) {
```

```
3     console.log(contents);
4 });
```

Listing 3.5: Read a file with Node.js. Notice the event-driven architecture with an anonymous callback function passed as argument.

Node package manager (npm)

> *"To increase speed, you can either push harder or reduce friction."* — Isaac Z. Schlueter, node.conf, Portland, OR, May 5th, 2011

With the rise of Node for server-side JavaScript, another highly influencial project was born late 2009, the Node package manager (npm). Isaac Z. Schlueter, while working at Yahoo, wanted to increase usage of JavaScript for full stack Web development. According to him, many people were already pushing hard on Node.js, so he attempted at lowering friction by creating the Node package manager (npm). The core design decisions of npm are rooted in the principle of reducing most sources of friction, including the following:

- **Conflicting dependencies.** When transitive dependencies require different versions of the same package. As a consequence, npm retrieves every version needed by dependencies.
- **Inconsistent package installation.** Typically, one would need to clone, make, copy, rename files, etc. With `npm install`, dependencies are all installed locally, under the `node_modules/` directory and usable by invoking `require('the-module')`.
- **Publishing difficulties.** Usually, package registries require a lot of metadata. Npm only requires two fields, name and version.

As a result, npm grew exponentially, to become the world's largest package registry ever, by a large amount, with over a million packages since June 2019. At NodeConf 2011 [189], when Isaac Schlueter announced npm 1.0, the registry contained 1900 packages and almost 800 active package authors. This roughly corresponds to doubling the registry size every year!

Unfortunately, reduced friction and a policy favoring package creators over users brought a few security issues. The most notable one is probably the event-stream incident late 2018 [159] where a new maintainer of the event-stream package added a dependency to a malicious package, harvesting bitcoin from visitors of a targeted application.

3.2.4 JavaScript issues

Organic growth and backward compatibility

Most programming languages tend to grow in complexity with time. New features are regularly added, and backward compatibility requires that outdated practices are kept in

the language. JavaScript is a good illustration of this kind of organic growth. As an example, the language specification of JavaScript is 805 pages [66]. This is roughly the same size as the Java specification with 772 pages [113] or the C specification with 571 pages [32] To compare, the specification for the Go progamming language by Google [89], contains approximately 100 pages.

The most salient evolutions occurred with ES2015 (previously known as ES6). The addition of the `const` and `let` keywords for example are confusing for beginners. They introduce two new ways of declaring variables, bringing it to a total of four, along with the `var` keyword and no keyword. Differences between those are presented in Listing 3.6.

```
1  // x and y are global variables
2  x = 42;
3  y = 14;
4
5  // Let's open an inner block scope
6  {
7    // The "var" keyword has a function scope
8    // so it will erase the global value 42.
9    var x = 14;
10
11   // The "let" keyword has a block scope
12   // so it will not replace the global y.
13   let y = 0;
14
15   // The "const" keyword has a block scope.
16   // It prevents reassignment of the variable.
17   const z = 3;
18   // z = 4; TypeError: Assignment to a constant variable
19 }
20
21 // x: 14
22 console.log("x: " + x.toString());
23
24 // y: 14
25 console.log("y: " + y.toString());
```

Listing 3.6: Variable scope in JavaScript.

Callback hell

As mentioned when introducing Node.js, JavaScript event-driven APIs rely on callback functions. Let's consider a simple case where we want to retrieve information from a database. Listing 3.7 outlines how a blocking synchronous API would look like. The control flow of the program is easy to follow, but blocking at `getDatabase` and `db.get` calls means the server (or the graphical interface) is not responding during this time.

```
1  // With theoretical blocking and synchronous APIs.
```

```
function getDataSync(url, id) {
  db = getDatabase(url);
  return db.get(id);
}

// How calling the function would look like.
try {
  data = getDataSync("some_url", 42);
  doSomethingWith(data);
} catch (error) {
  console.log(error);
}
```

Listing 3.7: Hypothetical blocking and synchronous API.

In contrast, the asynchronous callback version in Listing 3.8, is efficiently giving back control while waiting for the database to connect and respond. The main drawback resides in the complexity of the control flow, and the verbosity of the code. By a convention that emerged with time, callback functions are supposed to handle a potential error as first argument, and successful result as second argument. This model tends to produce extremely nested code because of function callbacks and if statements for error handling, and thus has been coined in the community the "callback hell" [161].

```
// With callback asynchronous APIs.
function getDataAsync(url, id, callback) {
  getDatabase(url, function(error, db) {
    if (error) {
      callback(error, null);
    } else {
      db.get(id, function(error, data) {
        if (error) {
          callback(error, null);
        } else {
          callback(null, data);
        }
      });
    }
  });
}

// How calling the function would look like.
getDataAsync("some_url", 42, function(error, data) {
  if (error) {
    console.log(error);
  } else {
    doSomethingWith(data);
  }
});
```

Listing 3.8: Typical asynchronous API based on callbacks.

We should mention that recent JavaScript standards provide new syntax making use of `async` and `await` keywords to simplify the control flow, while preserving the performances of the callback model. Listing 3.9 shows how the same code can take advantage of the new syntax. Unfortunately, this is to the detriment of language simplicity, as explained previously.

```
1  // With new async/await keywords and APIs.
2  async function getDataSync(url, id) {
3    db = await getDatabase(url);
4    data = await db.get(id);
5    return data;
6  }
7
8  // How calling the async function looks like.
9  try {
10   data = await getDataSync("some_url", 42);
11   doSomethingWith(data);
12 } catch (error) {
13   console.log(error);
14 }
```

Listing 3.9: Asynchronous version with the new async/await syntax.

Context of `this`

In other object-oriented languages, `this` (or `self`) usually refers to the currently used instance of a class. According to the specification, *The this keyword evaluates to the value of the ThisBinding of the current execution context* which is a little cryptic, so we will explain with a concrete example how `this` behaves. JavaScript not being a typical object oriented language, `this` can take many shapes, depending on the current execution context. The execution contexts are in a stack in which new contexts are created and pushed whenever code not associated with the current context starts running, which typically happens for function calls. Let's take Listing 3.10 as an example to exhibit some oddities of the `this` value. In that example, we first define different contexts for a log function whose purpose is to display the content of `this.x`. Then, starting at line 17, we actually call those different functions to show how complex it is to predict what `this` is going to refer to.

```
1  function log() {
2    console.log(this.x);
3  }
4  function Who(x) {
5    this.x = x;
6  }
```

```
7  Who.prototype.log = log;
8  Who.prototype.logF = function(){console.log(this.x);};
9  Who.prototype.logF2 = function(){log();};
10 Who.prototype.logCall = function(who){log.call(who);};
11 x = 1;
12 me = new Who(2);
13 logMe = me.log;
14 you = new Who(3);
15
16 // Try guessing what will be printed
17 console.log(this.x);            // 1
18 log();                         // 1
19 logMe();                       // 1
20 me.log();                      // 2
21 me.logF();                     // 2
22 me.logF2();                    // 1
23 me.logCall(you);               // 3
```

Listing 3.10: Value of `this` in JavaScript.

By default, if `this` is undefined, as in lines 17 and 18, it is binded to the global object. At line 11, we define `x = 1` with no keyword, so `x` is a global variable. As a result, line 17 and 18 print 1. The definitions of the `log` and `logF` methods on the `Who` class lines 7 and 8 are equivalent. The behavior of `this` in that context, is what we expect to see for methods call on objects and thus, lines 20 and 21 both print 2. The `call` JavaScript function (and some others), used for the definition of the `logCall` method, enables binding of the `this` value to a specific object given as first argument. That is why line 23 prints 3.

Now the most surprising results are lines 19 and 22, both printing 1 instead of 2. At line 13, `logMe` is defined as the same function than `me.log` which actually is the original `log` function. As a consequence, line 19 is strictly equivalent to line 18, and they both print 1. Finally, the `logF2` method also prints 1 because it's definition isn't the log function (as defined for the `log` method) but rather calls the `log` function which generates another context in which `this` is not defined anymore. The behavior is thus the same than for lines 17 and 18, which binds `this` to the global object, and prints the global variable `x = 1`.

Dynamic typing and implicit conversions

JavaScript is a dynamically typed language. This means that types of values are only known at runtime, and that they can change during the execution of the program as shown in Listing 3.11.

```
1 // Define x as an empty object.
2 x = {};
3
4 // Type of x dynamically changed
5 // to an object with a 'hello' field.
```

```
6 x.hello = "world";
```

Listing 3.11: Dynamic typing in JavaScript.

In addition, JavaScript performs implicit conversions between type, depending on the operators and functions being used. In Listing 3.12, the number 42 gets converted into the string "42" before concatenation, and the string "6" is converted to the number 6 before multiplication with the number 7.

```
1 "hello " + 42; // -> "hello 42"
2 "6" * 7;       // -> 42
3 "6" + 7;       // -> "67"
```

Listing 3.12: Weak typing in JavaScript (implicit conversion).

By combining dynamic types, and implicit conversions, JavaScript often generates extremely surprising situations, resulting in unexpected behaviors. It can also lead to very original use cases. In 2010, an informal code obfuscation competition resulted in the creation of a subset of JavaScript containing only six characters [116], `[`, `]`, `(`, `)`, `!` and `+`, able to represent any valid JavaScript code. The value `false` would be obtained with `![]`, since negation of an empty array returns `false` according to JavaScript specification. Numbers, characters, and other language constructs are obtained through similar implicit conversion tricks.

Undefined is not a function

For JavaScript developers previous to 2015, the error "undefined is not a function" had become a meme in the online community. This error would very often rise from a typo somewhere in the code, generating an `undefined` value instead of a function. Due to the dynamic nature of JavaScript, the error can be reported late in the call stack. Indeed, even if an error in the code might create an `undefined` value, it is only later when called as a function that an uncaught error would trigger. As shown in Figure 3.1, browsers are more helpful now, at the cost of loosing an iconic error for all JavaScript developers. Yet, not having a compile step prevents advanced static analysis of the code, and at the same time lengthen the feedback loop to fix errors.

3.2.5 JavaScript as a compilation target

As we now know, JavaScript exists since 1995, and from 1997 onward, has mostly been the only way to run code dynamically in the browser. For this reason, many alternative languages started treating JavaScript as a compilation target to run code in a browser.

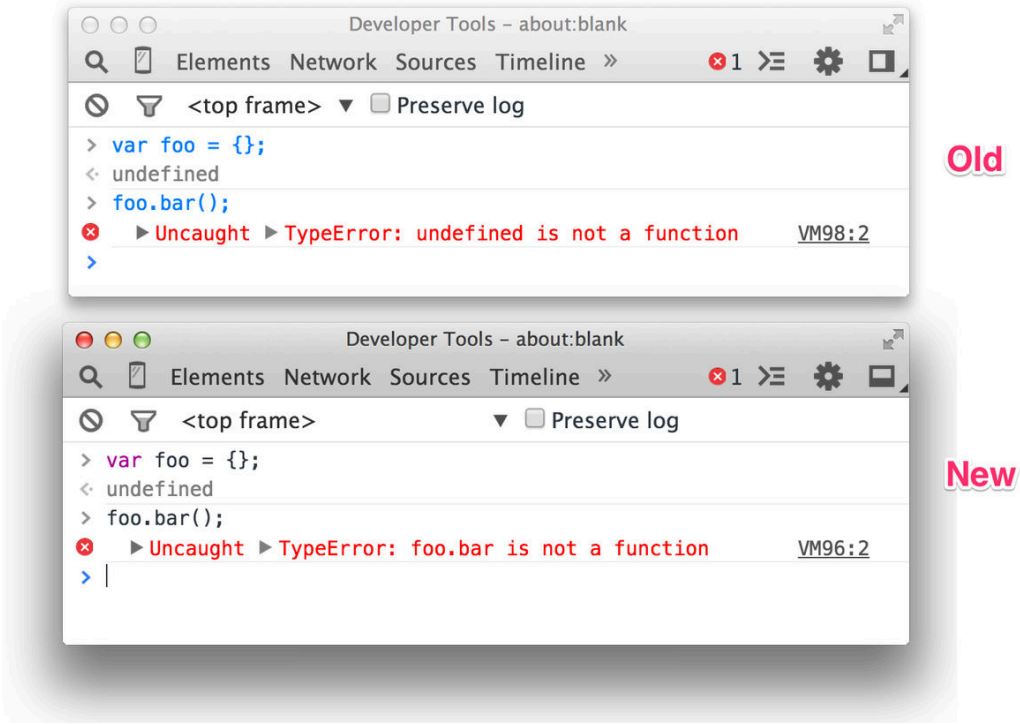


Figure 3.1: Tweet by Addy Osmani (21-Feb-2015) announcing error report improvements in Chrome.

Multi-tier programming

Haxe was probably the first production ready language to target JavaScript, in 2006. At this time, there was no JIT, and JavaScript performances were fairly limited. Some people were nonetheless trying to make the Web a video and gaming platform. Flash, a multimedia platform running the ActionScript language was the most popular solution at the time. In 2005, YouTube was for example relying on a Flash player to distribute videos. Haxe was created by Nicolas Cannasse [34] with the clear purpose of removing the overhead of composing heterogeneous components like a Flash client, a Web server, and additional JavaScript for Web games design.

Many other languages later followed that path of using the same language for server and client code, sometimes called "isomorphic" frameworks, or multi-tier programming environments. Google announced their Google Web Toolkit (GWT) in May 2006 [96], enabling Java developers to build client applications. The Ocsigen framework by V. Balat et al. [17] in 2006 allowed building Web applications in the OCaml programming language. The Opal compiler [163] translates Ruby code into JavaScript, enabling full stack Ruby Web applications. Today, most programming languages can target JavaScript, including Python, C,

Erlang, Haskell, etc.

Around the same period, academic research has also been trying to solve multi-tier Web programming with unique new languages like Links by E. Cooper et al. [52], or Hop by Serrano et al. [193] in 2006. Those efforts are continuing with for example A. Chlipala et al. [42] who created the Ur/Web variant of the Ur modeling language, or Sinha et al. [197] with the WebNat programming language in 2015. According to Sinha et al. [197] however, experienced Web developers require fine grained control of the generate code, debugging tools, deployment and configurations features for designing complex real-world Web applications. Unfortunately, those research attempts at novel ways of programming the Web are not mature enough yet to be adopted by developers.

JavaScript as the main target

Instead of trying to tackle both server and client-side programming, a new category of languages later emerged, focusing on the client side, and with JavaScript being the only or main compilation target. The most notable ones are CoffeeScript [8] released in 2009 by Jeremy Ashkenas, Dart [13] designed by Lars Bak (creator of the V8 engine) and Kasper Lund for Google in 2010, Elm [56] the product of Evan Czaplicki senior thesis on functional reactive programming in 2012, and Reason [217] (also known as ReasonML) in 2016 by Jordan Walke (who is also the original designer of the React framework we will discuss later).

All this excitement around new languages targetting the Web by considering JavaScript as a compilation target confirms that people are actively trying to solve JavaScript shortcomings with completely different designs. One can also notice that appart from Dart, which is heavily object-oriented, those new languages follow the functional paradigm. It may be related to guaranties brought by functional programming that we will develop when exploring the Elm programming language.

Gradually typed JavaScript

Coding with a completely different language is a rather extreme approach which can be disturbing for developers. From this observation, both Microsoft and Facebook decided to bring new contributions to the JavaScript ecosystem under the form of gradual typing. Gradual typing is a type system where values are partially typed. Some may be typed, and consequently static typing rules are verified, and some may be untyped, left for runtime verifications.

In October 2012, Microsoft released TypeScript [24], a superset of JavaScript, introducing optional type annotations. As a consequence, any valid JavaScript program is also a valid TypeScript program. This property was most certainly the major success factor of TypeScript. Programs can be ported progressively to benefit from static analysis. Listing 3.13 exhibits the core type annotation feature of TypeScript.

```
// JavaScript version
function add(x, y) {
  return x + y;
}

// TypeScript version
function add(x: number, y: number): number {
  return x + y;
}
```

Listing 3.13: JavaScript and TypeScript version of an add function.

Another benefit of static typing that JavaScript developers are discovering when switching to TypeScript is the improved IDE support, which includes for example better autocompletion tools, jumping to definitions, etc.

In 2014, another tool named Flow and led by Facebook [39] enabled gradual typing of JavaScript. Ultimately, TypeScript seems to be the most popular one, but choosing between the two will most likely depend on how well they integrate with the JavaScript framework and tools used in the corresponding application.

JavaScript transpilation

Despite increasing language complexity as explained in Section 3.2.4, ES2015 and later specifications brought very appreciated new features, often influenced by other languages like CoffeeScript. The `async / await` pair of keywords is such example of syntax reducing complexity of the code control flow. New specifications, however, are not always immediately available in all browsers, especially mobile versions. But there exists one version of JavaScript fully supported on all browsers, ES5. Inspired by the Traceur compiler created by Google engineers [98], Sebastian McKenzie started writing `6to5` [10] on September 2014, at the age of 17. His `6to5` project, now renamed Babel, is known as a JavaScript "transpiler", i.e. a program converting recent JavaScript source code into another (older) version of JavaScript source code. Today, Babel has become one of the most popular tools with 7 million weekly downloads on `npm`.

3.3 Frontend Web programming

3.3.1 Single Page Application (SPA)

In a desire to improve user experience in Web applications, code location has progressively been shifting from server to client. Since 2009, a Web framework named AngularJS [104] strongly pushed the Web actors toward writing "Single Page Applications" (SPA). A Single Page Application gets its name from the fact that only one HTML page is sent to the client

browser. This page however, contains JavaScript code taking control of the application and rendering it for the rest of the user navigation. When new data is required, the application can send requests with the XMLHttpRequest (XHR) object, or a WebSocket provided by the browser, then process the answer and re-render the HTML page accordingly. Since February 2005, this technique was popularized under the name Ajax [82] by Jesse James Garret and is represented in Figure 3.2. Ajax stands for asynchronous JavaScript and XML, though today data is mostly exchanged in the JSON format (JavaScript Object Notation) instead of XML, and occasionally just raw bytes depending on use cases and protocols.

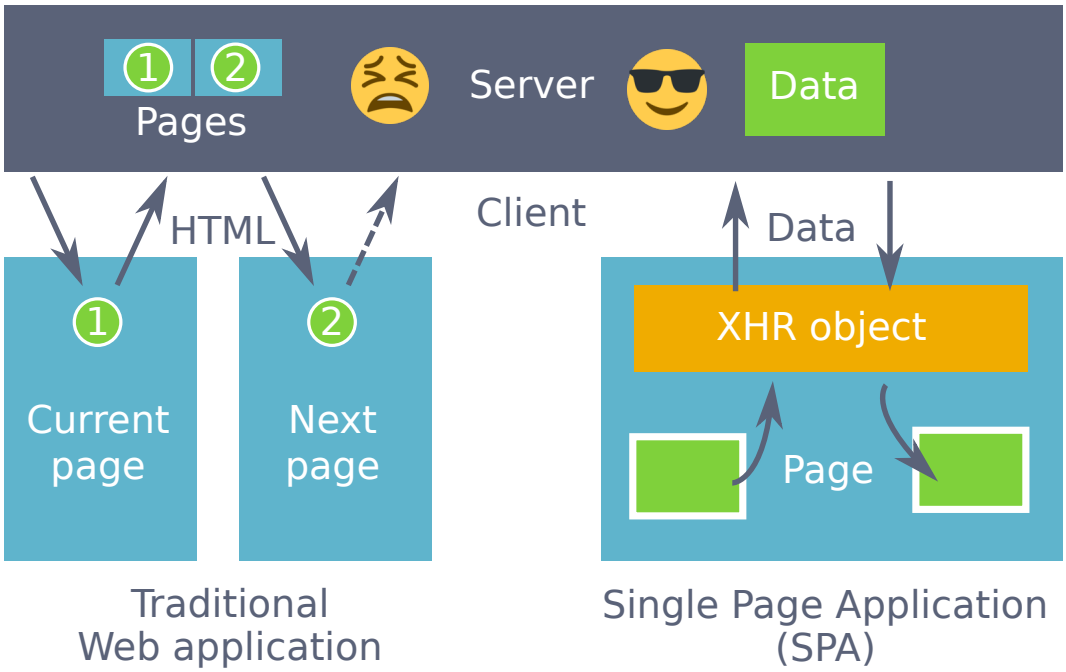


Figure 3.2: Difference between traditional Web applications and Single Page Application (SPA). A traditional application will ask the server to generate a new HTML page to access the required content. A SPA will just ask for the required data and render it in the client directly.

3.3.2 Reactive programming

In 2007, Sean Parent gave a talk at Google [167, 168] on declarative user interface logic and building the "Property Model Library" at Adobe [112]. An analysis of Adobe's desktop applications code highlighted that a third of it is devoted to event handling logic, and that half of the bugs exist in this portion of the code.

Reactive programming is a paradigm focusing on manipulation of time-varying values. It is especially suited for event-driven applications such as graphical user interfaces (GUI), and tries to solve many of the issues brought by asynchronous callbacks. Spreadsheet softwares

for example, are usually implemented as reactive systems in which modification of a cell will propagate to all computed cells depending on it. In a survey on reactive programming [12], E. Bainomugisha et al. classify approaches along six axes: representation of time-varying values, evaluation model, lifting operations, multidirectionality, glitch avoidance, and support for distribution.

Representation of time-varying values and lifting operations mostly depend on the underlying programming language. Statically typed languages, will require a differentiation in the type between a normal value and a time-varying one, thus also needing "lifting" operations i.e. ways of transforming functions working on regular values (like sum of two numbers) into functions that operate on their time-varying version. Dynamically type languages may figure this out at runtime, and consequently avoid usage of lifting operators.

The evaluation model can be of two kinds, push-based, pull-based, and sometimes a mix of the two. The most common evaluation mechanism is push-based, meaning once a time-varying value changes, it pushes the change to other values depending on it. A pull mechanism however usually relies of lazy evaluation languages such as Haskell. The major issue of pulling is that the system may suddenly require many depending past values and that will often result high memory consumption and program pauses due to burst of computations. Push-based evaluation has the advantage of lower memory usage and quicker response, but may introduce temporary inconsistent states called glitches if propagation of changes are pushed in a wrong order.

Most research on the topic have occurred in the context of functional reactive programming (FRP), following Fran [68] in 1997. Almost none of those research projects however are being picked up for use in production. Even Elm [56] that got some traction in the Web industry, decided to move away from FRP [55] with its 0.17 release in 2016. The reason for the move in Elm and low adoption of reactive programming in general is probably due to the big learning curve for the concepts. A recent JavaScript "compiler" project named Svelte [100] is trying to reintroduce reactive foundations with minimal alterations to the JavaScript language. It is too soon to predict if this approach will be successful, even though it is getting traction.

One important aspect of reactive programming that has been picked up however, is the declarative nature of binding graphical user interfaces to the underlying data such that when the data changes, the interface is automatically updated. This is often called one-way or two-way data binding depending on if modifications of the user interface also immediately pushes changes to the associated data.

3.3.3 Virtual DOM

The document object model (DOM) is the hierarchical structure of elements composing an HTML page. In order to understand what a virtual DOM is, and why it is useful, we first need to understand how the browser renders the DOM.



Figure 3.3: Rendering process of a Web page.

The rendering process is depicted in Figure 3.3. First, the browser needs to build the DOM and the CSSOM trees. The CSSOM is the equivalent of the DOM but for the hierarchy of CSS (styling) properties. When combining the DOM and the CSSOM, the browser generates a render tree, which contains all the visible nodes with their computed styles. From the render tree, the browser can compute the layout of the page, i.e. the exact position and sizes of all visible elements in the Web page. Finally, the browser can "paint" the actual pixels on screen. This used to be done in two phases, first paint different virtual layers of related elements, then compose and render those different layers on screen. Recently, new rendering engines tend to combine those steps into a process similar to those of game engines. For more information on browsers rendering engine, Lin Clark [45, 46] also wrote two enlightning blog posts on that subject.

In theory, everytime JavaScript code modifies the DOM or CSS properties, the whole process should be called to rerender the page. In practice, the browser framerate approximates the screen's one. Therefore, the "paint" step only occurs roughly 60 times per second. The layout however, may need to be recomputed at a higher framerate. For example, reading the `offsetX` and `offsetY` properties of a mouse event to get the current coordinates of the mouse while annotating images, will require a reflow (recomputation of layout) if the DOM or CSSOM has changed. Unfortunately that is exactly what we do when annotating images. We retrieve mouse coordinates in the image, and then draw the associated annotation on top. This alternation of modifying the DOM and reading properties needing reflow is the worst source of performance drops when happening at high frequencies. Listing 3.14 is an example of code showcasing this issue. Computing this loop with `nbiter = 1000` takes 200ms on Chrome while splitting the loop in two loops takes 3ms. This pattern is known under the name "layout thrashing".

```
1 for (let i=0; i < nbiter; i++) {
2     document.body.offsetHeight;
3     document.body.style.height = i + "px";
4 }
```

Listing 3.14: Example code forcing layout recomputation by intertwining layout read and CSSOM write.

A virtual DOM is the combination of a data structure and an update mechanism circumventing this kind of issues by batching all DOM modifications once per frame. Instead of directly modifying the DOM, one should modify the virtual DOM data structure. Usually, the library providing the virtual DOM implementation will update the DOM at each frame thanks to a diffing algorithm between the versions of the virtual DOM data structure at last frame and at the new frame. Representing the DOM as an intermediate data structure also have many benefits regarding testing and enable writing pure visualization functions (not performing any side effect).

3.3.4 How to choose?

Angular

AngularJS [104] dates back to 2009, when Miško Hevery and Adam Abrons were trying to sell online storage services through software at getangular.com. The project wasn't successful enough, and so they made <angular/> open-source. Miško Hevery was later recruited at Google to work on a new project. The story, as told by Miško Hevery and Brad Green at Google I/O 2013 [105], says that it took Miško three weeks (though he had bet two) to rewrite a six-month work with 17000 lines of code into 1500 lines of code with <angular/>. Impressed, Brad decided to embrace the <angular/> project under Google's wing and it got rebranded AngularJS with a new logo. In 2016, Google released its successor, renamed Angular (without the JS part). An important difference is that Angular is using TypeScript instead of JavaScript. The key feature of Angular/AngularJS is declarative two-way data binding between the application state and the view, showcased in Listing 3.15 with the `myCtrl` Angular controller. Another important design decision is that Angular is trying to provide a framework handling the totallity of frontend developer needs, while other frameworks target specific, restricted subjects.

```
<div ng-app="myApp" ng-controller="myCtrl">
  Firstname: <input ng-model="firstname" />
  Lastname: <input ng-model="lastname" />
  <h1>{{firstname}} {{lastname}}</h1>
</div>

<script>
  var app = angular.module("myApp", []);
  app.controller("myCtrl", function($scope) {
    $scope.firstname = "John";
    $scope.lastname = "Doe";
  });
</script>
```

Listing 3.15: Two-way data binding in AngularJS.

React

Contrary to Angular, React is designed to solve a very specific use case, which is how to build user interfaces. As such it doesn't care about how you store data, or manage routing of the SPA with the url. The core feature of React is its virtual DOM. It avoids layout thrashing and provides one-way data binding between the state of a React component and its rendering in the DOM. The syntax used for the binding is showcased in Listing 3.16.

```
// Leaving out details in ...
class Timer extends React.Component {
    constructor() {
        this.state = { seconds: 0 }
    }
    ...
    render() {
        return (<div>Seconds: {this.state.seconds}</div>);
    }
    ...
}
```

Listing 3.16: React example showing the state and render function of a component.

Vue

Vue describes itself as a progressive framework, meaning it provides core features targetting a small scope, and other opt-in layers bringing more functionalities. In a sense, it shares advantages and inconvenients of both React and Angular, with a different balance point. It provides inbuilt conveniences to simulate two-way data binding through automatically attaching event listeners when using the `v-model` property. An example is given in Listing 3.17.

```
<div id="myApp">
  Firstname: <input v-model="firstname" />
  Lastname: <input v-model="lastname" />
  <h1>{{firstname}} {{lastname}}</h1>
</div>

<script>
  new Vue({
    el: "#myApp",
    data: {
      firstname: "John",
      lastname: "Doe"
    }
  });
</script>
```

Listing 3.17: Simulated two-way data binding in Vue using the `v-model` property.

Summary

All three frameworks provide data binding between the state of the application and its rendering, making the user interface declarative and reactive. Angular and React are the more mature projects with the biggest community. Angular provides a solution covering most aspects needed for a SPA, while React is focused on the user interface and will often be paired with libraries to manage state efficiently like Redux. Thanks to its opt-in layered architecture, Vue like React has a lower barrier to entry if we only use its core components. But if needed, it also offers a coherent set of functionalities making it an all-in-one solution similar to Angular. In order to produce small application code compatible with most browsers, one also should add transpilation, minification, and other preprocessing tasks readying the code and assets for serving them on the Web. We effectively end up making Web development similar to static and compiled development environments. Knowing all this, I will argue that we should instead use Elm, a functional programming language compiling to JavaScript. In the next section I will detail how it can bring all the advantages of other Web frameworks, but with an improved developer experience, and a more reliable application at the end.

3.4 Elm

> *"Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher"* — Antoine de Saint-Exupéry, Terre des Hommes, chapitre III, L'avion, 1939.

Elm is a statically typed functional programming language for building Web applications. Its syntax comes from the Meta Language (ML) family of languages, similar to Haskell and OCaml. It strives for simplicity by removing non essential features like custom operators in version 0.19, and by avoiding functional programming jargon such as monads, functors, etc. The home page of the language claims that Elm generates JavaScript with great performances and no runtime exceptions. In the following sections, we will see how its properties enable such a claim.

3.4.1 Pure functions

All functions in Elm, except for debugging, are "pure", meaning they produce no side effect. A side effect is a behavior with implications outside of the scope of a function, such as modifying a global variable or an input parameter, generating random values or interacting with the outside world. Side effects are important to handle to build applications that are not predetermined at startup, but we will explain later how they are managed withing The Elm Architecture (TEA).

Listing 3.18 gives examples of functions that cannot be directly translated from JavaScript to Elm due to side effects. Pure functions are also sometimes called "referentially transpar-

ent" though the meaning of this terminology is unclear depending on sources. The important property is that calling a pure function with the same arguments will always return the same result. As a consequence many optimizations can be performed such as memoization, or pre-computation of functions with no arguments (which actually are constant expressions). The Elm compiler doesn't precompute constant expressions yet, but memoization in the form of lazy functions can be used for computation of view functions, reducing the amount of work for the diffing algorithm of the virtual DOM.

```
function timeNow() {
  return new Date().getTime();
}

function randomNumber() {
  return Math.random();
}

// Global variable.
lastname = "Doe";

function getFullName(firstname) {
  lastname = " " + lastname;
  return firstname + lastname;
}

console.log(getFullName("John")); // "John Doe"
console.log(getFullName("John")); // "John  Doe"
console.log(getFullName("John")); // "John   Doe"
```

Listing 3.18: Side effects in JavaScript.

3.4.2 Algebraic Data Types (ADT)

Algebraic data types (ADT) initially appeared in 1980 with the Hope programming language, developed by Rod Burstall, Dave MacQueen and Don Sannella [31]. Since then, they have been popularized by functional programming languages such as Haskell or OCaml. Usually, algebraic data types include product types and sum types. Product types are types regrouping multiple data together under the same structure. Tuples like `(Int, Float)`, and records (or objects) are the most common product types. Their names come from the properties on cardinality if we consider types as sets. Indeed, a tuple of three booleans have a cardinality of 8 if you consider all possible combinations, which is the product $2 \times 2 \times 2$. Sum types are referred to as "custom types" in Elm terminology. They are defined with the `type` keyword. Few examples of custom types definitions are provided in Listing 3.19, including the `OneOfThreeBools` sum type, which may contain 6 distinct elements $(2+2+2)$.

```
-- Definition of type Bool as a custom type with two variants
```

```elm
type Bool
    = False
    | True

-- Custom type with one variant
type Unit
    = Unit

-- Custom type with three variants holding other types
type OneOfThreeBools
    = First Bool
    | Second Bool
    | Third Bool

-- Custom type holding data of a generic type (lower case)
type Container a
    = Container a

-- Definition of a generic linked list with a custom recursive type
type List a
    = EmptyList
    | AtLeastOne a (List a)
```

Listing 3.19: Custom types definitions in Elm.

The most important property of custom types in Elm is that they enable modelization of a problem with exactly the correct cardinality for the types, preventing impossible states by design. Concretely, consider that we are modeling accessibility of a site depending on the logged status of users. Typically, in a language without sum types, like JavaScript, the user will be modeled with two fields as in Listing 3.20. Initially the `loggedIn` field will be false and the user name empty. As soon as the user is logged in, the corresponding field will have the value `true`, and the name will be filled with the user name. But what happens when the `loggedIn` field is `false` and the name is filled with something like "John Doe". In theory this state should never be reached if we are careful in our implementation, but in practice, bugs tend to fill every possible crack, requiring more tests to verify that this state is never reached. With custom types in Elm, the user type will be defined as in Listing 3.21. In this definition, a user can either be anonymous or logged in with a name, but never anonymous and with a name. By design, custom types prevents an entire family of bugs.

```javascript
user = {
  loggedIn: false,
  name: ""
};
```

Listing 3.20: User modeled with a product type in JavaScript.

```elm
type User
```

```
2     = Anonymous
3     | LoggedInWithName String
```

Listing 3.21: User modeled with a custom (sum) type in Elm.

3.4.3 Total functions

Functions are qualified as "total" when they are guarantied to return a result for every possible valid input. The Elm language has two properties resulting in functions being total,

- exhaustive pattern matching on custom types,
- and no statement, only expressions.

No statement, only expressions

In most programming languages like JavaScript, programs are composed of successions of statements and expressions. The former do not return values while the latter do. Appart from imports, Elm code contains only top level definitions and expressions. Listing 3.22 provides example of Elm expressions for conditions or loops. All branches of conditions (`if` expressions) must have a value of the same type. Without an `else` branch, the code will not compile. Not having `for` loops statements is among the toughest functional concepts to learn for beginners. In a language with only expressions, loops need to be expressed either with recursive functions, or with higher order functions, like `List.foldl` in the length example.

```
1  localLetDefinitions : Int
2  localLetDefinitions =
3      let
4          x = 14
5          y = 42
6      -- local definitions with the "let" keyword
7      -- are used to resolve the expression
8      -- appearing after the mandatory "in" keyword
9      in
10     x + y
11
12
13 boolToInt : Bool -> Int
14 boolToInt bool =
15     if bool then
16         1
17     -- if expressions must have an "else" branch.
18     -- Both branches must return a value of the same type.
19     else
20         0
21
```

```
length : List a -> Int
length list =
    -- loops are replaced by higher order functions
    -- transforming an intermediate result step by step.
    List.foldl (\x subtotal -> subtotal + 1) 0 list
```

Listing 3.22: Branching control and loops are expressions in Elm.

Pattern matching

Pattern matching is a branching mechanism based on the structure of a type. Any custom type can be matched to one of its different variants with the `case ... of` syntax as shown in Listing 3.23. Any lowercase variable in the pattern will be bound to the corresponding data, like the `name` variable here.

```
username : User -> Maybe String
username user =
    case user of
        LoggedInWithName name ->
            Just name

        Anonymous ->
            Nothing
```

Listing 3.23: Pattern matching in Elm

In javascript, one can fairly easily forget to handle a case, or willingly only process the "happy path" for prototyping speed. In Elm, if I remove the `Anonymous` branch, the compiler will refuse to compile the code with the message showed in Listing 3.24. This is the main reason why the `Nothing` value in Elm, approximately equivalent to the `null` value in JavaScript will never trigger a runtime exception. Everywhere it may appear, i.e. everywhere the `Maybe` type is used, Elm will guaranty at compile time that this case is handled.

```
missing patterns
Line 20, Column 5
This `case` does not have branches for all possibilities:

20|>    case user of
21|>        LoggedinWithName name ->
22|>            Just name

Missing possibilities include:

    Anonymous

I would have to crash if I saw one of those. Add branches for them!

```

```
15 Hint: If you want to write the code for each branch later, use `Debug.todo` as
       a
16 placeholder. Read <https://elm-lang.org/0.19.0/missing-patterns> for more
17 guidance on this workflow.
```

Listing 3.24: Missing pattern compiler error in Elm

As suggested in the hint given in the compiler error, we could also use `Debug.todo "message"` in the `Anonymous` branch, which is very useful for the rapid prototyping phase. Remark that the `Debug.todo` function will make the program crash if reached at runtime. For this reason, all functions from the `Debug` module are forbidden in code compiled in release mode.

No runtime exception

Thanks to expressions and exhaustive pattern matching, Refactoring an Elm code base can be done with confidence. Any place where types do not match with functions will be signalled by the compiler, which becomes a true coding assistant. Noredink, a company based in San Francisco reported in 2018 that after two years of using Elm in production, they got their first runtime exception, compared to 60000 for the JavaScript code [75].

3.4.4 The Elm Architecture (TEA)

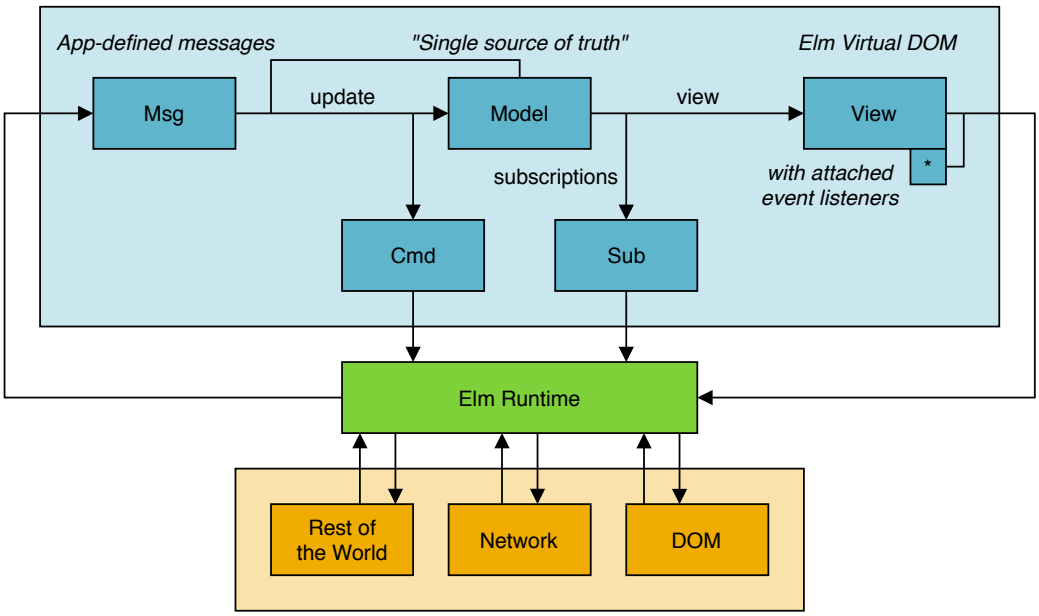


Figure 3.4: The Elm Architecture (TEA).

The Elm Architecture (TEA) enforces a unidirectional data transformation flow, visualized in Figure 3.4. The central entity is the `Model`. It contains all and every information

about our application state. The visual aspect of our application is called the `View` (basically an HTML rendered document) which is generated by the `view` function, from the `Model`. Finally, all events generate messages, of type `Msg`. The `update` function, updates the model by reacting to those messages, closing the loop.

All functions are pure, meaning there is no side effect, outputs of functions are entirely defined by inputs. There cannot be global variables mutations, real world events, network interaction etc. Basically such a program would be running in a predestined way from its start to its end, preventing us from loading images and interacting with them. This is why the application is attached to the Elm runtime, provided by the language, transforming all real world events ("side effects") into our defined set of messages, of type `Msg`.

The main challenge with pure functions is to describe side effects without performing them. Those are described in three locations:

1. View attributes as DOM event listeners for pointer events.
2. Commands (`Cmd`) generated by the update function, like loading of images.
3. Subscriptions (`Sub`) to outside world events like the window resizing.

The Elm runtime takes those side effect descriptions, perform them, and, whenever there is a result / an answer, transforms it into one of our defined messages (`Msg`) and routes it to our update function. After updating the model, the runtime automatically calls the `view` function. This way, the user interface reacts to model modifications similarly than with other one-way data bindings we have previously introduced.

3.4.5 Elm-UI, an alternative layout strategy

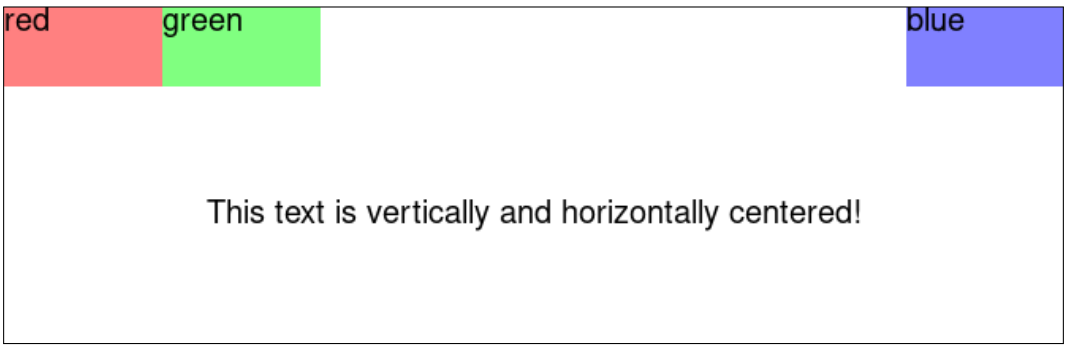


Figure 3.5: User interface specified by Listing 3.25.

We have seen that the Elm architecture enables building Web applications with HTML views. It treats the user interface as data, using a virtual DOM under the hood and managing the DOM side effects with a runtime system. Overall, with Elm guaranties, one is

fairly confident that when the program compiles, it is functionally correct. Layout however, traditionally relies both on HTML and CSS rules, intrinsically hard to debug due to their cascading nature since a CSS rule may apply to all children of a DOM element. In [197], Sinha et al. identified user interface design as the main difficulty in Web development.

At Elm Europe 2017, Matthew Griffith made a presentation entitled "Understanding style" [94]. In this work, he identifies the most problematic aspect of layout being that there is no clear way to identify when it is incorrect. There is not even an unhelpful "undefined is not a function" error in CSS resolution. Basically an error in layout and style is just the unexpected. With this in mind he created a library, now named elm-ui [93], aiming at providing the guaranty that if your code compiles, the layout if fully specified. The key property of the library is that its base building block, the `el` element, only has one child, instead of a list like in the case of a `div` HTML element. Also, building blocks with multiple children must have explicit layout. Listing 3.25 showcases how functional composition of UI elements and an attention on naming enable building of clear and robust user interfaces. The corresponding user interface is provided in Figure 3.5.

```
1  module Main exposing (..)
2
3  import Element as El exposing (Color, Element)
4  import Element.Background as Background
5
6  main = El.layout [] content
7
8  content : Element msg
9  content =
10     El.column [ El.height El.fill, El.width El.fill ]
11         [ header
12         , centeredText
13         ]
14
15 header : Element msg
16 header =
17     El.row [ El.width El.fill ]
18         [ El.el [ El.alignLeft ] (rectangle (El.rgb 1 0.5 0.5) "red")
19         , El.el [ El.alignLeft ] (rectangle (El.rgb 0.5 1 0.5) "green")
20         , El.el [ El.alignRight ] (rectangle (El.rgb 0.5 0.5 1) "blue")
21         ]
22
23 rectangle : Color -> String -> Element msg
24 rectangle color text =
25     El.el
26         [ Background.color color
27         , El.height (El.px 50)
28         , El.width (El.px 100)
29         ]
30         (El.text text)
31
32 centeredText : Element msg
33 centeredText =
34     El.el [ El.centerY, El.centerX ]
35         (El.text "This text is vertically and horizontally centered!")
```

Listing 3.25: Fully specified layout with elm-ui.

3.4.6 Reliable packages

Elm packages are versioned using the semantic versioning specification [175], with the MAJOR.MINOR.PATCH schema. The major number is incremented when changes breaking the API are introduced, such as removing a function or modifying its type signature. The minor number is incremented when new values or functions are introduced. Finally the patch number is incremented when nothing else than internal non-exposed implementation are modified. Since Elm uses total functions, it is able to programmatically compute version number increments with the `elm bump` command. As a consequence, upgrading your dependencies is guarantied by the compiler to not break one code as long as they don't increase the major number.

Another advantage of having pure, total functions is that type signatures are not lying, implying that one can immediately identify functions capable of triggering side effects. At the current state of the Elm language, the only places where side effects can happen are ports to JavaScript (forbidden in packages), commands and HTML. Therefore, a dependency that doesn't expose functions with commands or HTML in their type signatures will not be able to steal bitcoins or launch nuclear warheads.

Conclusion

In this chapter, we provided a guided tour of the Web evolutions since its beginnings in 1991. We identified the tendency to build interfaces in a declarative way to reduce the complexity of user interfaces. And in order to introduce our technology of choice, the Elm programming language, we discussed on the limitations of JavaScript when we value the reliability of our applications. The following chapter combines the knowledge of Web technologies presented here, and the annotation needs required for segmentation datasets. It presents an open-source image annotation Web application, developed with easy deployment to crowdsourcing platforms in mind.

Chapter 4

Interactive Annotation on the Web

Contents

4.1 Introduction 70
4.2 Presentation of the application 71
4.3 Technical choices 73
4.3.1 The model states 73
4.3.2 The messages 73
4.3.3 The view 75
4.3.4 Library and application duality 75
4.4 Crowdsourcing annotations 76
4.5 Acknowledgments 76
4.6 Conclusion 77

4.1 Introduction



Figure 4.1: Screenshot of the interface of our image annotation Web application.

Image annotations are required in a wide range of applications including image classification (which requires textual labels), object detection (bounding boxes), or image segmentation (pixel-wise classification). The rise and successes of deep learning lead to an increasing need for annotations, as training sets should be of a large size for these algorithms to be efficient. Yet, researchers still spend time and resources to create ad hoc tools to prepare those datasets. The application we present in this chapter aims at providing a customizable tool to fulfill most image annotation needs.

Application	Year	Tools
LabelMe	2008	bbox, polygon, iterative semi-automatic segmentation
VIA	2016	bbox, polygon, point, circle, ellipse
Labelbox	2018	bbox, polygon, point, line
Dataturks	2018	bbox, polygon
Ours	2018	bbox, polygon, point, stroke, outline

Table 4.1: Most relevant image annotation Web applications (tools).

Application	Configurable interface	Tasks management	Type	License
LabelMe	no	Mturk integration	server	OSS
VIA	no	no	client	OSS
Labelbox	yes	yes	server	private
Dataturks	no	yes	server	private
Ours	yes	Mturk integration	client	OSS

Table 4.2: Most relevant image annotation Web applications (application type).

Many image annotation applications already exist (Table 4.1). LabelMe [185], one of the most popular, provides an interface for drawing bounding boxes and polygons around objects in an image. It has been used extensively to create datasets for image segmentation. Some more recent softwares share the same goals, with their own specificities. For example, Labelbox [129] and Dataturks [58] provide annotation tasks management, particularly useful when crowdsourcing the annotations; these softwares are proprietary. The VGG Image

Annotator (VIA [64]) is an open-source client application like ours, with the specificity of providing annotation attributes, editable in a spreadsheet format.

We release an open-source application [5], entirely client side, meaning that no data is uploaded to any server. Images are loaded from files and annotated locally, in the browser. The simplest tool, from a user perspective, should be immediately available i.e. should not require any additional installation to be fully functional. Our image annotation software is thus a Web-based application, easily configurable to fit users needs, as well as embeddable in the Mechanical Turk platform to design crowdsourcing campaigns.

We first present the features of our application, then describe its architecture. Finally, we explain how it can be used to start crowdsourcing experiments.

4.2 Presentation of the application

A screenshot of the application can be seen in Figure 4.1. The image to be annotated occupies the central part of the screen; a toolbar is located on top, object classes are available on the left and images to be annotated on the right.

Images. Multiple images can be loaded at the same time using the image icon on the top-right corner of the application. These images are not uploaded on the server, and can either be loaded locally from the client's machine, or from a distant server.

Tools. Our application includes several tools to annotate images. Icons for these tools are depicted in Figure 4.2. From left to right, the first available annotation is the point, that can be useful to designate objects in the image. It can also be used as a seed in region-growing image segmentation methods. The second annotation we included is the bounding box, which provides the localization of objects in the image, and is used in object detection problems. The information we acquire are the left, right, top and bottom coordinates of the bounding box. The third annotation we chose to implement is the stroke, or scribble, which is a popular interaction in image segmentation. It consists in a sequence of points, interpreted as a continuous line. The outline, fourth type of annotation, is a closed shape, typically drawn around objects. It is comparable to a bounding box in essence, but provides a more precise location of objects. Finally, polygons can also be drawn (as in LabelMe, for instance), by successively clicking new points as vertices.

All these tools are available both with a mouse or a touch interaction. As a matter of fact, some tools are better suited to touch devices (for example, outlines) than others (polygons).

Object classes. For most annotation tasks, we also need to differentiate objects in the images. Typically each annotated area is attributed a class, or label. The PASCAL VOC dataset [73], for example, is composed of 20 classes, grouped by categories:

- *Person*: person



Figure 4.2: Annotation tools icons

```
{ "classes":
  [ { "category": "Person"
    , "classes": [ "person" ]
    }
  , { "category": "Animal"
    , "classes": [ "bird", "cat", "cow", "dog", "horse", "sheep" ]
    }
  , { "category": "Vehicle"
    , "classes": [ "aeroplane", "bicycle", "boat", "bus", "car", "motorbike",
      "train" ]
    }
  , { "category": "Indoor"
    , "classes": [ "bottle", "chair", "dining table", "potted plant", "sofa",
      "tv/monitor" ]
    }
  ]
, "annotations": [ "point", "bbox", "stroke", "outline", "polygon" ]
}
```

Listing 4.1: A configuration file to annotate the PASCAL dataset.

- *Animal*: bird, cat, cow, dog, horse, sheep
- *Vehicle*: aeroplane, bicycle, boat, bus, car, motorbike, train
- *Indoor*: bottle, chair, dining table, potted plant, sofa, tv/monitor

In our application, classes are specified in a JSON configuration file. A strict corresponding config for PASCAL VOC classes is presented in Listing 4.1.

To attribute a class to an annotation, a user should first select the class in the left sidebar, then use a tool to create an annotation. Selecting a class in the left sidebar also highlights the annotations corresponding to this class.

Configuration file. The five annotation tools are optionally made available by the configuration file. In Listing 4.1, the last line of the depicted configuration file contains an `annotations` field, listing the tools that should be available. In this case, they all are.

In addition to the five fundamental annotation types, each type can be derived in virtually any number of variations. For example, interactive segmentation algorithms often require *foreground* and *background* scribbles. In our application, this would mean the user would need to draw two types of strokes. This can be achieved using the configuration file, as in Listing 4.2. Such configuration would result in two stroke icons in the toolbar, of different colors, just as in Figure 4.1.

```
{ "classes": []
, "annotations":
    [ "bbox"
    , { "type": "stroke", "variations": [ "fg", "bg" ] }
    ]
}
```

Listing 4.2: A configuration file to include two types of strokes.

```
type State
    = NothingProvided
    | ConfigProvided Config Classes (Zipper Tool)
    | ImagesProvided (Zipper RawImage)
    | AllProvided Config Classes (Zipper Tool) (Zipper AnnotatedImage)
```

Listing 4.3: State type definition.

4.3 Technical choices

The application code is organized in two parts:

- A minimalist Node.js server, located in the `server/` directory. It is statically serving the content of `server/dist/` with compression.
- A complete Elm client application, located in the `client/` directory. It follows the Elm architecture presented in the previous chapter. We present the model, messages, and specific views of this application in this section. The compiled application weighs 150 kB gzipped, which is great for low bandwidth connections.

4.3.1 The model states

The `state` is the main component of the `Model`. It contains the images and configuration loaded as well as the annotations performed. Its type is defined as in Listing 4.3 and can be modeled as a finite state machine, visualized in Figure 4.3.

The application available online starts in state 0 (`NothingProvided`) and enables you to reach state 2 (`AllProvided`) with buttons to load images and configuration. Two messages called `LoadImages` and `ConfigLoaded` produce transitions in the state machine.

4.3.2 The messages

All modifications of the model are understood by looking at the `Msg` type definition (Listing 4.4). The `update` function then performs the modifications described by those messages.

- The `WindowResizes` message is triggered when the application is resized. In the update function, it takes the new size and recomputes some view parameters.

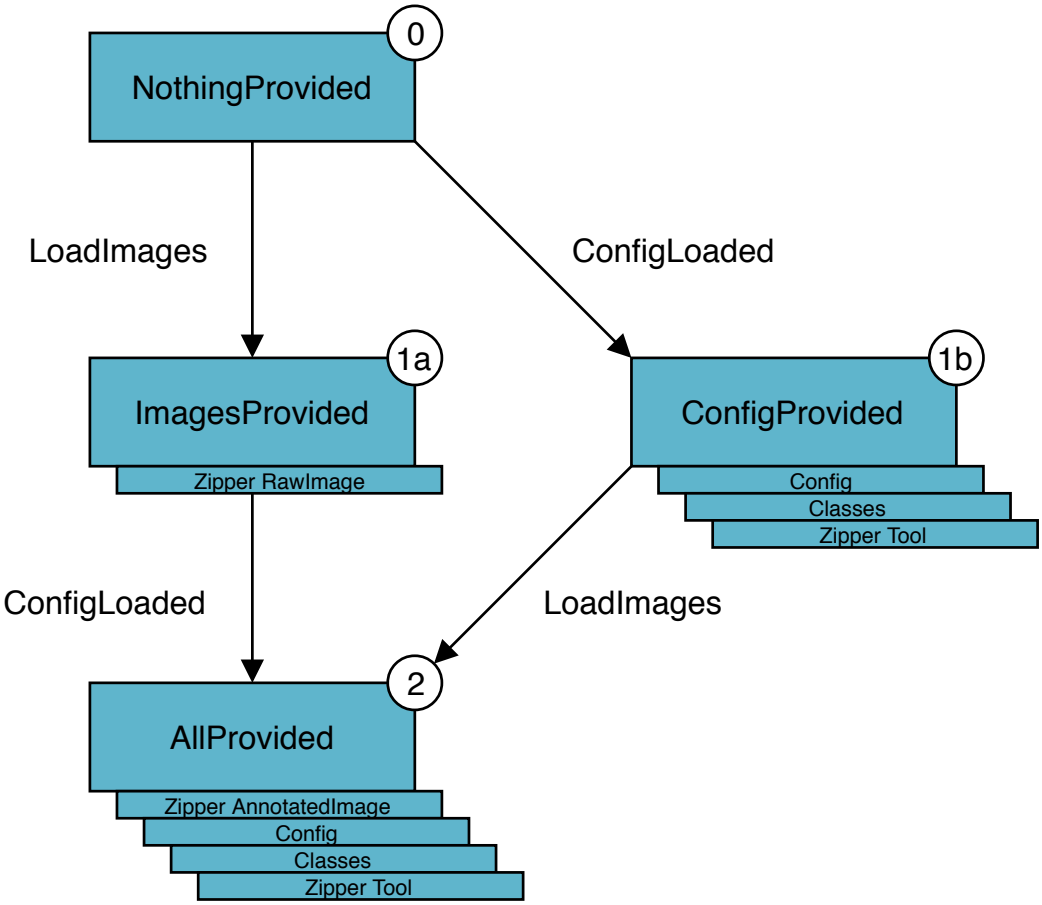


Figure 4.3: The application states.

- A `PointerMsg` message is triggered by pointer events (mouse, touch, etc.). In the update function, this is the message activating all the annotations logic code of our application.
- The messages `SelectImage`, `SelectTool` and `SelectClass` are generated when clicking on images, tools and classes.
- Files are handled by five messages:
  - When loading images from the file explorer, a `LoadImages` message is generated with a list of the images files and their names as identifiers. For each image correctly loaded an `ImageLoaded` message is generated, providing a local url, corresponding to the image in memory.
  - The messages `LoadConfig` and `ConfigLoaded` behave similarly.
  - The `Export` message causes the application to serialize into JSON all the anno-

```
type Msg
    = WindowResizes Device.Size
      -- pointer events
    | PointerMsg Pointer.Msg
      -- select things
    | SelectImage Int
    | SelectTool Int
    | SelectClass Int
      -- files
    | LoadImages (List { name : String, file : Value })
    | ImageLoaded { id : Int, url : String, width : Int, height : Int }
    | LoadConfig Value
    | ConfigLoaded String
    | Export
      -- other actions
    | ZoomMsg ZoomMsg
    | RemoveLatestAnnotation
```

Listing 4.4: Msg type definition.

tations, and asks the user to save the generated file. It is triggered by clicking on the export button of the top action bar.

- Whenever an event should change the zooming level of the drawing area, a `ZoomMsg` message is generated.
- Finally, the `RemoveLatestAnnotation` message is also explicit.

4.3.3 The view

The view of this application is based on four components, each implemented in its own module, with potentially different versions depending on the current state of the application.

- The top action bar (`src/View/ActionBar.elm`).
- The center annotations viewer area (`src/View/AnnotationsArea.elm`).
- The right images sidebar (`src/View/DatasetSideBar.elm`).
- The left classes sidebar (`src/View/ClassesSideBar.elm`).

4.3.4 Library and application duality

In order to offer a turnkey solution to image annotations, we created a configurable application solving most needs. But we also thought of cases where advanced modifications are required. Consequently, the foundation of this application has been extracted in the

independent package elm-image-annotation [7]. It is designed as an API to create, modify and visualize geometric shapes, useful in the context of image annotation.

Modules for manipulation and serialization (in JSON) of annotations are under the `Annotation.Geometry` namespace. It already contains one module for each tool presented earlier. If you want to introduce a new tool, this is where you can create a new module.

This package also contains the following important modules, under the `Annotation` namespace:

- `Annotation.Style`: defines types describing appearance of points, lines and fillings of annotations.
- `Annotation.Svg`: exposes functions rendering SVG elements for each annotation kind.
- `Annotation.Viewer`: manages the central visualization area, supporting zooming and translations, relative to an image frame.

If you are interested in creating another rendering target than SVG, like canvas or WebGL, it would require alternative modules to `Annotation.Svg` and `Annotation.Viewer`. The rest of the code can stay unchanged.

4.4 Crowdsourcing annotations

Image annotation interfaces are often used in the context of large datasets of images to annotate. As such, tasks management for crowdsourcing campaigns is an important feature. Labelbox and Dataturks are all-in-one services providing tasks management directly in their applications. Just like LabelMe, we choose instead to provide a configuration, ready to use with Amazon Mechanical Turk (Mturk).

Mturk comes in two sides. A "requester" is defining a set of tasks while a "worker" is performing them. Workers are payed by requesters through the Mturk service. The concept of a "HIT" (Human Intelligence Task) characterizes the task unit. In our case, one HIT means one image to be annotated. We describe in details how to setup a campaign with our template in the application documentation.

4.5 Acknowledgments

We would like to thank

- @tforgione and @GarciaDelMolino for their wise feedbacks,
- @dncg for their Windows tests,

- the online Elm community for their help along the road: @evancz for the delightful Elm language, @ianmackenzie for the fantastic geometry library, @mdgriffith for the very refreshing layout library, @luke for the amazing tool Ellie, @norpan, @jessta, @loganmac, @antew, for their invaluable help on slack.

4.6 Conclusion

In this chapter we have introduced our Web-based image annotation application. More information is available in the online documentation [6]. Evolutions of this application are still developed in alternative branches to keep the master branch in a stable state. We welcome all forms of feedback and contribution.

Part II

RGB-D Visual Odometry

Chapter 5

Introduction to the RGB-D Visual Odometry Problem

Contents

5.1 Modeling Image Capture in a Camera 82
5.1.1 Historic Remarks 82
5.1.2 Projective Geometry 82
5.1.3 Pinhole Camera Model 83
5.1.4 Intrinsic Parameters 84
5.1.5 Radial Distortion 85
5.2 Modeling Camera Movements 86
5.2.1 Origins of Visual Odometry 86
5.2.2 3D Space & Rigid Body Motion 86
5.2.3 The Lie Group $SO(3)$ and Lie Algebra $\mathfrak{so}(3)$ 88
5.2.4 The Lie Group $SE(3)$ and Lie Algebra $\mathfrak{se}(3)$ 90
5.3 Visual Odometry Approaches 90
5.3.1 Capturing Device 91
5.3.2 Relation to Visual SLAM 92
5.3.3 Reducing Drift 93
5.4 Motion Estimation 94
5.4.1 Feature-Based Motion Estimation 94
5.4.2 Appearance-Based Motion Estimation 96

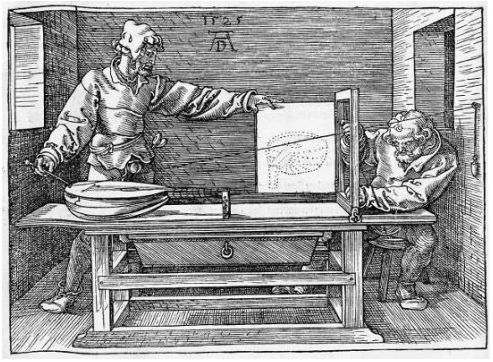
5.1 Modeling Image Capture in a Camera

5.1.1 Historic Remarks

The study of the image formation process has a long history. Traces of geometric formulations of image formation are present in Euclid work (4th century B.C.). These skills also re-emerged in Renaissance art with artists such as Brunelleschi, Donatello and Alberti. A treatise on the projection process, *"Della Pittura"*, was published by Leon Battista Alberti in 1435 and influenced many Renaissance artists such as Leonardo da Vinci and Raphael. In Figure 5.1a the perspective emerges from the vanishing point, an imaginary point at the center of the image, at which all parallel lines in the represented scene cross. Dürer devised a machine to get a perspectively correct image, represented in Figure 5.1b. It is a manual reproduction of what a camera does today.



(a) Raphael, The School of Athens (1509)



(b) Dürer's perspective machine (1525)

Figure 5.1: Usage of perspective projection in Renaissance art.

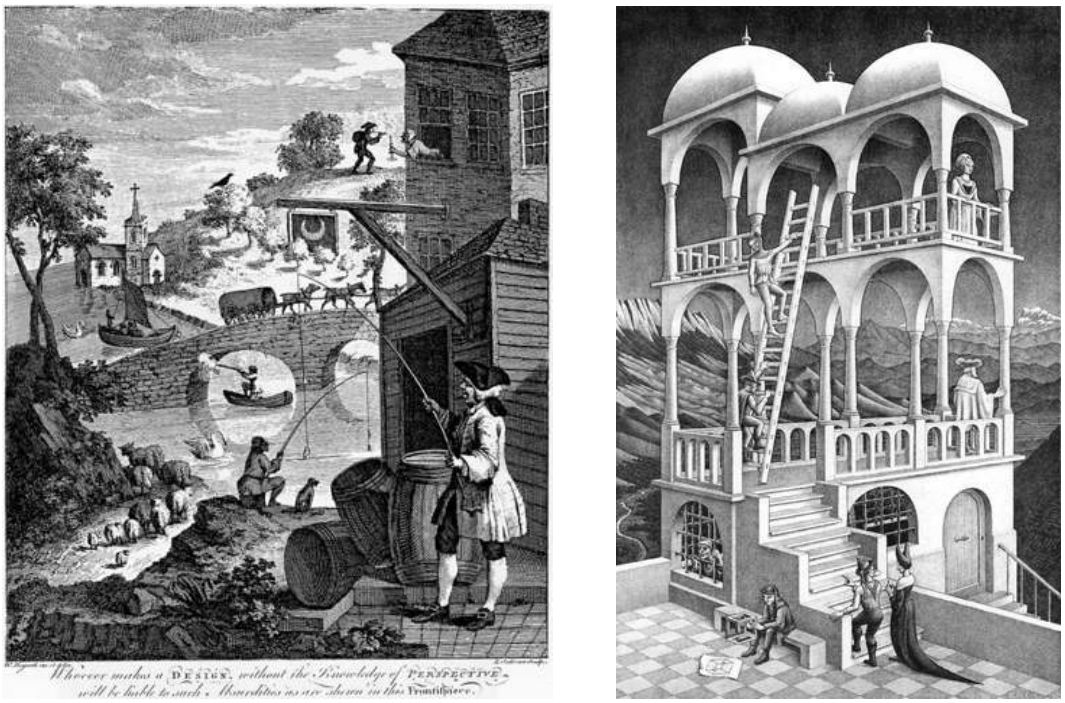
Many artists also played with those perspective rules to create images that seem locally correct but have inconsitent global depth or gravity such as Hogarth (Figure 5.2a) and Escher (Figure 5.2b).

5.1.2 Projective Geometry

In order to formally write transformations by linear operations, we make extensive use of homogeneous coordinates to represent a 3D point (X, Y, Z) as a 4D-vector $(X, Y, Z, 1)$ with the last coordinate fixed to 1. However, this normalization is not always necessary as one can represent 3D points by a general 4D vector

$$\boldsymbol{X} = (XW, YW, ZW, W) \ \in \ \mathbb{R}^4.$$

In general, an n-dimensional projective space $\mathbb{P}^n$ is the set of all one-dimensional subspaces (i.e. lines through the origin) of the vector space $\mathbb{R}^{n+1}$. A point $p \in \mathbb{P}^n$ can then be assigned



(a) Hogarth, Satire (1753) (b) Escher, Belvedere (1958)

Figure 5.2: Conscious circumvention of perspective projection in art.

homogeneous coordinates $\boldsymbol{X} = (x_1, \ldots, x_{n+1})^\top$, among which at least one x is nonzero. For any nonzero $\lambda \in \mathbb{R}$, the coordinates $\boldsymbol{Y} = (\lambda x_1, \ldots, \lambda x_{n+1})^\top$ represent the same point p.

5.1.3 Pinhole Camera Model

Perspective projection emerges from a simplified model of a real camera called the pinhole camera, represented in Figure 5.3. The main issue of such a camera, is that the hole has to be very small to get a sharp image, therefore limiting the amount of light entering the capture device. In order to augment that amount of light, it is possible to use lenses, but just as with a pinhole camera, the image is upside down in the image plan. In order to avoid dealing with minus signs in the equations, we pretend that the image plan is virtually on the same side than the object. The perspective transformation π modeling this projection is given by

$$\pi : \mathbb{R}^3 \to \mathbb{R}^2; \quad \boldsymbol{X} \mapsto x = \pi(\boldsymbol{X}) = \begin{pmatrix} f\frac{X}{Z} \\ f\frac{Y}{Z} \end{pmatrix}$$

where f is the focal length, X, Y, Z are the object coordinates in the 3D world, the z axis being the camera axis. The one challenge we have to overcome, is that this transformation is non linear. In order to do so, we use homogeneous coordinates, which is basically similar

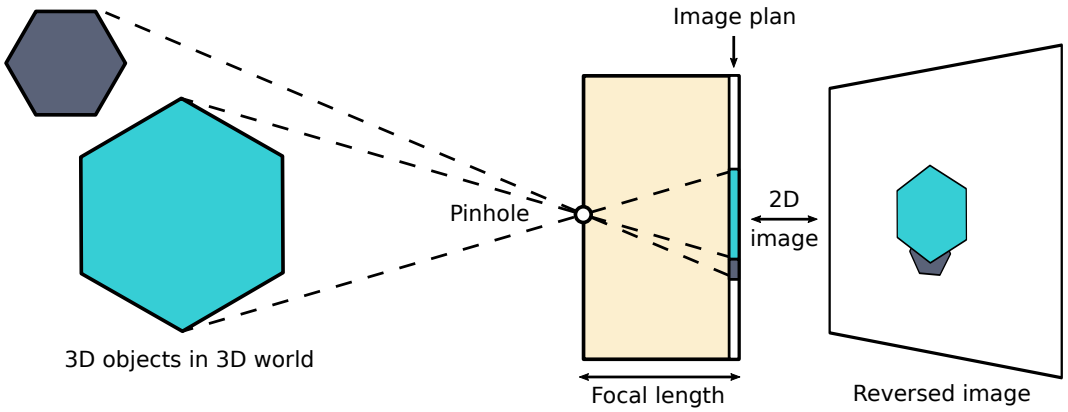


Figure 5.3: Pinhole camera model.

to multiplying everything by Z,

$$Z\boldsymbol{x} = Z\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K_f \Pi_0 \boldsymbol{X}$$

where we have introduced the two matrices

$$K_f = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \Pi_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

The matrix Π_0 is referred to as the standard projection matrix. We often note the distance to the camera along its axis with $\lambda > 0$ so

$$\lambda \boldsymbol{x} = K_f \Pi_0 \boldsymbol{X}.$$

5.1.4 Intrinsic Parameters

If the camera is not centered at the optical center, we have an additional translation o_x, o_y. The point where the optical axis intersects the image plan is called the principal point. If pixels do not have unit scale, we need to introduce additional scaling factors s_x and s_y. And if pixels are not rectangular, we also have a skew factor s_θ. The transformation from coordinates in the frame of the camera to final pixel coordinates has thus the following steps:

$$\text{Camera } (3\text{D}, \boldsymbol{X}) \xrightarrow{K_f \Pi_0} \text{Image } (2\text{D}, \boldsymbol{x}) \xrightarrow{K_s} \text{Pixel } (2\text{D}, \boldsymbol{x'})$$

where the pixel coordinates $\boldsymbol{x'} = (x', y', 1)$ are given by $\lambda\, \boldsymbol{x'} = K_s\ K_f\ \Pi_0\ \boldsymbol{X}$ with

$$K_s = \begin{pmatrix} s_x & s_\theta & o_x \\ 0 & s_y & o_y \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad K_f = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

We call $K = K_s\ K_f$ the intrinsic matrix since it holds parameters intrinsic to the camera system, independent from the outside 3D world.

5.1.5 Radial Distortion

The intrinsic parameters in the matrix K model linear distortions in the transformation to pixel coordinates. In practice however, one can also encounter significant distortions along the radial axis. This is particularly visible in a wide field of view or if one uses cheaper cameras such as webcams. A simple effective model for such distortions is to use

$$x = x_d(1 + a_1 r^2 + a_2 r^4), \quad y = y_d(1 + a_1 r^2 + a_2 r^4)$$

where $\boldsymbol{x} = (x_d, y_d)$ is the distorted point, and $r^2 = \|\boldsymbol{x}\|^2$ is its squared distance to the principal point. Usually, a_1 and a_2 are estimated through a calibration step computed from distortions of straight lines as in Figure 5.4 or simultaneously with a 3D reconstruction [200, 78]. Other more sophisticated models exist [61] but we will not enter in details here since we will consider that images are rectified as if fitting the pinhole model.

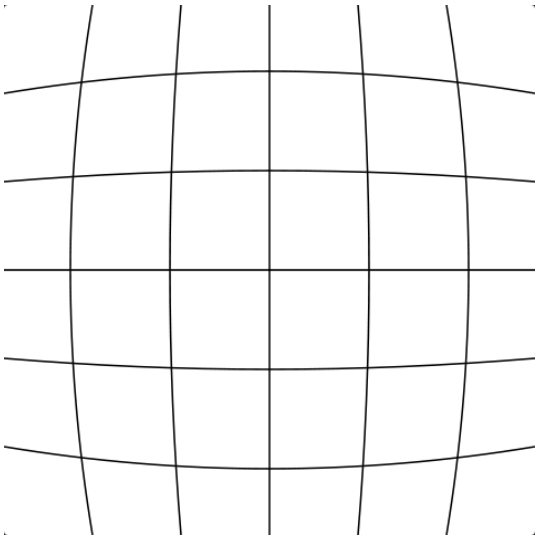


Figure 5.4: Grid projection with radial distortion.

5.2 Modeling Camera Movements

5.2.1 Origins of Visual Odometry

Aiming to reconstruct a three-dimensional structure of the world from a set of two-dimensional views has a long history in computer vision. It is generally considered an ill-posed problem since reconstructions consistent with a given set of observations/images are typically not unique. Therefore, one need to impose additional assumptions. The study of geometric relations between a 3D scene and observed 2D projections is based on two types of mathematic transformations, namely

- Perspective projection, and projective geometry to account for the image formation process we presented in the previous section.
- Euclidean motion or "rigid body motion" representing the motion of the camera from one frame to the next.

The first known work on the problem of multiple view geometry was that of Erwin Kruppa (1913) who showed that two views of five points are sufficient to determine both the relative tansformation ("motion") between the two views and the 3D location ("structure") of the points up to a finite number of solutions. A linear algorithm to recover structure and motion from two views based on the epipolar constraint was proposed by Longuet-Higgins in 1981 [139]. Several summarizing text books and papers were also written on the subject [74, 226]. Extensions to three views [199, 195] and factorization techniques for multiple views and orthogonal projection were also developed [207]. Depending on communities and context, the joint estimation of camera motion and surrounding 3D environment is called structure and motion (also known as structure from motion) or visual SLAM (simultaneous location and mapping). Visual SLAM techniques slightly differ from structure and motion in the sense that they are specialized for timely coherent sequences of images such as videos. Visual odometry, that we will detail later, is the core step of Visual SLAM, consisting of evaluating the camera motion of the next frame in the sequence. Structure and motion however usually refers to situations where no such assumption is done regarding the set of images.

5.2.2 3D Space & Rigid Body Motion

Cross Product & Skew-symmetric Matrices

The cross product of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ in $\mathbb{R}^3$ is a vector orthogonal to both.

$$\times : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3, \quad \boldsymbol{u} \times \boldsymbol{v} = \begin{pmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{pmatrix} \in \mathbb{R}^3.$$

Since $\boldsymbol{u} \times \boldsymbol{v} = -\boldsymbol{v} \times \boldsymbol{u}$, the cross product also introduces an orientation. Fixing $\boldsymbol{u}$ induces a linear mapping $\boldsymbol{v} \mapsto \boldsymbol{u} \times \boldsymbol{v}$ wich can be represented by the skew-symmetric matrix

$$\widehat{u} = u_\times = \begin{pmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{pmatrix} \in \mathbb{R}^{3\times 3}$$

such that $\boldsymbol{u}\times\boldsymbol{v} = \widehat{u}\,\boldsymbol{v}$. In turn, every skew symmetric matrix $M \in \mathbb{R}^{3\times 3}$ verifying $M = -M^\top$ can be identified by a vector $\boldsymbol{u} \in \mathbb{R}^3$. The operator $\wedge$ ("hat") defines an isomorphism between $\mathbb{R}^3$ and the space $\mathfrak{so}(3)$ of the 3×3 skew-symmetric matrices. Since a similar property is true for twists that we introduce later, we will use the notation $u_\times$ instead of $\widehat{u}$, which is a visual reminder that it acts like a cross product. Its inverse is denoted by ("vee") $\vee : \mathfrak{so}(3) \to \mathbb{R}^3$.

Rigid-Body Motion

A rigid-body motion (or rigid-body transformation) is a family of maps preserving the norm and cross product of any two vectors.

$$\begin{gathered} g : \mathbb{R}^3 \to \mathbb{R}^3, \quad \boldsymbol{u} \mapsto g(\boldsymbol{u}), \\ \forall \boldsymbol{u} \in \mathbb{R}^3, \quad \|g(\boldsymbol{u})\| = \|\boldsymbol{u}\|, \\ \forall\ \boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^3, \quad g(\boldsymbol{u}) \times g(\boldsymbol{v}) = g(\boldsymbol{u} \times \boldsymbol{v}) \end{gathered}$$

Since norm and scalar product are related by the polarization identity one can also state that a rigid-body motion is a map which preserves inner product and cross product. As a consequence, rigid-body motions also preserve the triple product, and therefore volumes.

$$\forall\ \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^3, \quad \langle g(\boldsymbol{u}), g(\boldsymbol{v}) \times g(\boldsymbol{w})\rangle = \langle \boldsymbol{u}, \boldsymbol{v} \times \boldsymbol{w}\rangle.$$

Let $\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3} \in \mathbb{R}^3$ be the orthonormal oriented vectors of our initial frame. We note the transformed vectors $\boldsymbol{r_i} = g(\boldsymbol{e_i})$ and R the matrix $R = (\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3})$ The first constraint (preservation of scalar product) implies that R is an orthogonal matrix $R^\top R = RR^\top = I$. The second property (preservation of cross product) implies that $\det(R) = +1$. In other words, R is an element of the group $SO(3) = \left\{ R \in \mathbb{R}^{3\times 3} \,\middle|\, R^\top R = I,\ \det(R) = +1 \right\}$. The motion of the origin can be represented by a translation $\boldsymbol{t} \in R^3$. Thus the rigid-body motion g can be written as

$$g(\boldsymbol{x}) = R\boldsymbol{x} + \boldsymbol{t}, \quad R \in SO(3), \quad \boldsymbol{t} \in \mathbb{R}^3.$$

Image Formation with Camera Movement

Let's consider $\boldsymbol{X_0}$ a point in the World reference frame. Its coordinates in the camera frame are determined by a rigid body motion $\boldsymbol{X} = g(\boldsymbol{X_0}) = R\boldsymbol{X_0} + \boldsymbol{t}$. In homogeneous coordi-

nates, we can write $\boldsymbol{X} = g\boldsymbol{X_0} = \left(\begin{smallmatrix} R & \boldsymbol{t} \\ 0 & 1 \end{smallmatrix}\right) \boldsymbol{X_0}$. In 5.1.4, we identified that pixels coordinates are linked to point coordinates in the camera frame by $\lambda\ \boldsymbol{x'} = K_s\ K_f\ \Pi_0\ \boldsymbol{X}$. In total, the transformation from World coordinates to pixels coordinates is given in homogeneous coordinates by

$$\boxed{\lambda\ \boldsymbol{x'} = K\ \Pi_0\ g\ \boldsymbol{X_0}}$$

where λ is the depth of the point in the camera frame, K is the intrinsics matrix, Π_0 the standard projection matrix and g the rigid body motion characterizing the camera, also sometimes called extrinsics matrix.

5.2.3 The Lie Group $SO(3)$ and Lie Algebra $\mathfrak{so}(3)$

Sophus Lie (1841–1899)



Portrait of Marius Sophus Lie

Marius Sophus Lie was a Norwegian-born mathematician. He created the theory of continuous symmetry, and applied it to the study of geometry and differential equations. He discovered that continuous transformation groups are better understood in their linearized versions (Theory of transformation groups, 1893). These infinitesimal generators form a structure which is today known as a Lie algebra.

The reference C++ implementation to manipulate elements of Lie algebras useful to the computer vision community is a library called `Sophus` [202] in tribute to Sophus Lie. In our work, we also provide a Rust implementation of these Lie algebras.

Lie Algebra $\mathfrak{so}(3)$

One can show that the effect of any infinitesimal rotation $R \in SO(3)$ can be approximated by an element from the space of skew-symmetric matrices $\mathfrak{so}(3) = \left\{ w_\times \,\middle|\, \boldsymbol{w} \in \mathbb{R}^3 \right\}$. The rotation group $SO(3)$ is called a Lie group. The space $\mathfrak{so}(3)$ is called its Lie algebra.

The Exponential Map

Given the infinitesimal formulation of a rotation, obtained from a continuous set of rotations $R(t)$ and by deriving the equation $R(t)R(t)^\top = I$, one can show that $\dot{R}R^\top$ is a skew-symmetric matrix and that we have the differential equation system

$$\begin{cases} \dot{R}(t) = w_\times(t)R(t), \\ R(0) = I. \end{cases}$$

If we assume that $w_\times(t)$ is constant in time ($= w_\times$), this known equation has the solution

$$R(t) = e^{w_\times t} = \sum_{n=0}^{\infty} \frac{(w_\times t)^n}{n!} = I + w_\times t + \frac{(w_\times t)^2}{2!} + \dots$$

which is a rotation around the axis $\boldsymbol{w} \in \mathbb{R}^3$ by an angle of t (if $\|\boldsymbol{w}\| = 1$). One can also absorb the scalar $t \in \mathbb{R}$ into the skew-symmetric matrix $w_\times$. This matrix exponential therefore defines a map from the Lie algebra to the Lie group

$$\exp : \mathfrak{so}(3) \to SO(3), \quad w_\times \mapsto e^{w_\times}.$$

In analogy to the well-known Euler equation $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, there is an expression called Rodrigues' formula for the exponential of skew symmetric matrices,

$$e^{w_\times} = I + \frac{w_\times}{\|\boldsymbol{w}\|}\sin(\|\boldsymbol{w}\|) + \frac{w_\times^2}{\|\boldsymbol{w}\|^2}(1 - \cos(\|\boldsymbol{w}\|)).$$

The Logarithm of $SO(3)$

There is conversely a mapping from the Lie group $SO(3)$ to the Lie algebra $\mathfrak{so}(3)$. For any rotation matrix $R \in SO(3)$, there exists a vector $\boldsymbol{w} \in \mathbb{R}^3$ such that $R = \exp(w_\times)$. Such an element is denoted by $w_\times = \log(R)$. If $R \neq I$, we note r_{ij} its coefficients and $\boldsymbol{w}$ is given by

$$\begin{cases} \|\boldsymbol{w}\| = \cos^{-1}\left(\dfrac{\text{trace}(R) - 1}{2}\right), \\ \dfrac{\boldsymbol{w}}{\|\boldsymbol{w}\|} = \dfrac{1}{2\sin(\|\boldsymbol{w}\|)}\begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix}. \end{cases}$$

The above solution is not unique since for example, increasing the angle by multiples of 2π will give the same rotation.

5.2.4 The Lie Group $SE(3)$ and Lie Algebra $\mathfrak{se}(3)$

The Lie Algebra of Twists $\mathfrak{se}(3)$

Just as with $SO(3)$ one can show that $SE(3)$ has a tangent space, of which the elements are called twists. This tangent space is called the Lie algebra of twists, noted $\mathfrak{se}(3)$.

$$\mathfrak{se}(3) = \left\{ \widehat{\xi} = \begin{pmatrix} w_\times & \boldsymbol{v} \\ 0 & 0 \end{pmatrix} \in \mathbb{R}^{4\times 4} \;\middle|\; w_\times \in \mathfrak{so}(3),\ \boldsymbol{v} \in \mathbb{R}^3 \right\}$$

As with skew-symmetric matrices, we can define operators "hat" $\wedge$ and "vee" $\vee$ to convert between a twist $\widehat{\xi} \in \mathfrak{se}(3)$ and its coordinates $\boldsymbol{\xi} \in \mathbb{R}^6$. The twist coordinates $\boldsymbol{\xi} = \left(\begin{smallmatrix} \boldsymbol{v} \\ \boldsymbol{w} \end{smallmatrix}\right)$ are formed by stacking the "linear velocity" $\boldsymbol{v} \in \mathbb{R}^3$ (related to translation) and the "angular velocity" $\boldsymbol{w} \in \mathbb{R}^3$ (related to rotation).

Logarithm and Exponential Coordinates for $SE(3)$

Twist coordinates are also sometimes called "exponential coordinates". This is due to the fact that, similarly than with $SO(3)$, we can define an exponential map between $\mathfrak{se}(3)$ and $SE(3)$. For a twist $\widehat{\xi} = \left(\begin{smallmatrix} w_\times & \boldsymbol{v} \\ 0 & 0 \end{smallmatrix}\right) \in \mathfrak{se}(3)$ its associated rigid body motion is

$$e^{\widehat{\xi}} = \begin{pmatrix} e^{w_\times} & \frac{(I - e^{w_\times})w_\times \boldsymbol{v} + \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{v}}{\|\boldsymbol{w}\|^2} \\ 0 & 1 \end{pmatrix}.$$

Conversely, for every $g \in SE(3)$ there exists twist coordinates $\boldsymbol{\xi} \in \mathbb{R}^6$ such that $g = \exp(\widehat{\xi})$. Given $g = (R, \boldsymbol{t})$, we can compute $\boldsymbol{w}$ thanks to the association $R = e^{w_\times}$ as explained previously for $SO(3)$. For the linear velocity vector $\boldsymbol{v} \in \mathbb{R}^3$, we merely need to solve the equation

$$\frac{(I - e^{w_\times})w_\times \boldsymbol{v} + \boldsymbol{w}\boldsymbol{w}^\top \boldsymbol{v}}{\|\boldsymbol{w}\|^2} = \boldsymbol{t}.$$

Beware that, just as in $SO(3)$, this representation is not unique. In general, there exists many twists representing the same rigid-body motion.

5.3 Visual Odometry Approaches

Projects with precise localization needs may be of very different nature, such as Mars rover exploration [140], underwater navigation [63], autonomous vehicles [22], or augmented reality [190] to cite only few. Depending on the context and project needs, the capturing device and the localization and mapping requirements vary. In this section, we will briefly review the different visual inputs that a localization and mapping algorithm may have to process, and the tradeoffs between local and global coherence, leading to visual odometry or visual SLAM.

5.3.1 Capturing Device

Most of the earliest visual odometry systems were based on stereo capture such as the binocular setup by Matthies and Shafer [141], or the slider camera presented in Moravec's thesis [147] and used on board of Mars rovers. Those are systems providing or simulating the simultaneous capture of the environment by two or more cameras, and thus mimicking human stereo vision. In a calibrated stereo setup, one can easily triangulate 3D coordinates of points for stereo pairs, i.e. matching points in two or more images. Such triangulation can be obtained by searching a point from the left image onto the right image along its associated epipolar line, which is the right image of the line passing through the optical center of the left camera and the point in the left image. This method is often called disparity search. Due to the geometric constraint visualized in Figure 5.5, all epipolar lines cross at the epipole, which is the image of the optical center of one camera into the other. This constraint is called the epipolar constraint. Points with depth information are then tracked in successive frames, and motion can be estimated based on those 3D point clouds.

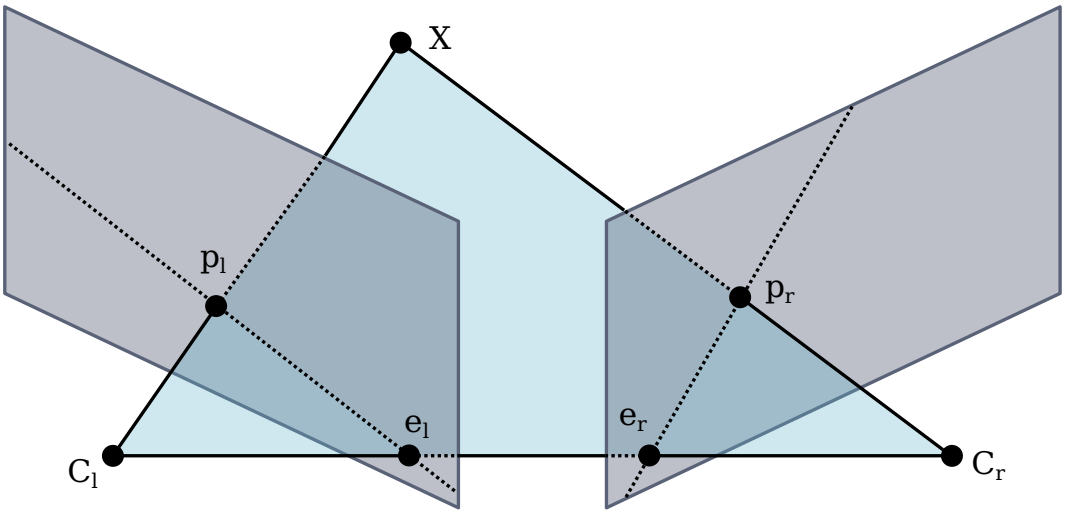


Figure 5.5: Epipolar constraint between two cameras. The center of the left and right cameras are C_l and C_r, X is the observed 3D point, projected onto p_l and p_r respectively in the left and right image. The left and right epipoles are noted e_l and e_r.

Alternatively to the stereo scheme, one can also use a single camera. It is called monocular visual odometry. In that case, the 3D structure is unknown. The relative motion between frames must then be retrieved from the 2D information only. That vector passing through the camera center and a given pixel of the image is usually called a bearing vector. In the monocular case, the motion and 3D structure computed by visual odometry can only be recovered up to an unknown scale when using the pinhole camera model. This is due to the fact that the projection modelization cannot differentiate big distances far away from small distances close to the camera. The first influencial work exploiting monocular camera in a

realtime algorithm is the one of Nister et. al. [157], also coining the term "visual odometry". It provides two important improvements over previous work. First is the use of reprojection error, which is the distance between the detected point in an image and its reprojection from the 3D point with a given camera transformation, while previous work tend to compute motion by aligning 3D point clouds. The second is the use of a five-point minimal solver [156] in a RANSAC scheme for robust motion estimation.

Recently, depth-sensing cameras such as the Microsoft Kinect [233] have appeared in the consumer market under the name RGB-D cameras ("D" stands for depth). Those cameras use an active system based on structured infrared light to compute a depth image. Due to limitations in outside environments, the usage of these cameras is restricted to indoor navigation. One should also avoid very reflective surfaces such as glass and mirrors. In favorable cases, RGB-D cameras offer the advantage of providing realtime depth information without the need of triangulation like in the stereo case.

One should note that in some situations, the camera can be paired with other sensing devices. Inertial measurement units (IMU) are present in smartphone and LIDAR are often provided in autonomous vehicles for example. Any additional information or constraint provided by a particular setup can be exploited to improve performances of the visual odometry system. There are therefore many papers exploring these areas but they are out of the scope of our study.

5.3.2 Relation to Visual SLAM

Visual simultaneous localization and mapping is the name given to the technique consisting in recovering the 3D structure of the environment (the map) and the trajectory of the camera relative to this environment. While visual odometry is only concerned with the local consistency of the map and trajectory, visual SLAM tries to obtain a globally consistent map. This difference is of the utmost importance for autonomous systems navigating for a long period, but less so for short augmented reality experiences for example.

There exists three dominant strategies in visual SLAM. One is based on filtering methods such as in MonoSLAM by Davison et. al. [59]. Those approaches use methods similar to the extended Kalman filter (EKF) to jointly estimate the camera trajectory and the 3D location of a small number of landmarks in a probabilistic scheme. The second strategy consists in keeping a small subset of the frames called keyframes and apply global optimization algorithms such as bundle adjustment on those. One such example is the algorithm of Klein and Murray called parallel tracking and mapping (PTAM) [120]. Finally there are bio-inspired strategies such as RatSLAM by Milfort et. al. [145].

Today, the majority of commonly used visual SLAM methods are keyframe-based. The two most comprehensive solutions are ORB-SLAM [151] and OpenVSLAM [205]. Keyframe-based visual SLAM is usually composed of a visual odometry core, extended with three components providing global consistency. A pose graph is built with the selected keyframes.

Camera locations are the nodes of the graph and transformations between keyframes are stored in the edges. A loop closure mechanism is added to detect when the camera returns to previously visited locations. When a loop is detected, an edge is added to the pose graph. Finally, a global optimization such as bundle adjustment is used to refine both keyframes camera locations and estimated 3D points. The choice between visual odometry and visual SLAM is thus mainly a tradeoff between realtime performances and accuracy and consistency over a long period of time. In our work presented later in this document, we focus on visual odometry since visual SLAM could be tackled later as an extension.

5.3.3 Reducing Drift

In the context of visual odometry, the trajectory is tracked incrementally. Small errors therefore accumulate and quickly deteriorate long term accuracy. Approaches reducing the errors and the drift are thus important.

Outlier Removal with RANSAC

Random sample consensus (RANSAC) is a method to estimate a model from a set of noisy data [77]. It consists in sampling a random subset of the data, estimate a model from it, and classify the rest of the data as fitting (inliers) or not (outliers) that model. If we note s the number of data points in the samples, ϵ the percentage of outliers in all the data, and p the target probability of successfully find the correct model, the number N of samples required is

$$N = \frac{\log(1-p)}{\log(1-(1-\epsilon)^s)}.$$

In a situation where 50% of the data is outliers, and with a target probability of success of 99%, it requires 16 iterations for a sampling size of 2 data points and 145 iterations for a sampling size of 5, growing at an exponential rate. This is one reason why exploiting every available constraint of the system to reduce the degrees of freedom is important. As a consequence, a number of minimal model parameterizations have been studied such as a three-point solver proposed by Fraundorfer et. al. when two of the camera angles are known [81]. Using a robust estimation scheme like the one provided by RANSAC can considerably reduce the drifts accumulated along the sequence.

Windowed Bundle Adjustment

Another method, complementary to outliers removal, is to use windowed bundle adjustment i.e. a local joint optimization of camera poses and point coordinates, over a small number of recent camera poses. It was demonstrated to considerably reduce tracking errors in [122]. The main issue of bundle adjustment, used in structure and motion algorithms as well as in global optimization for visual SLAM, is the computational complexity, growing in cube of

the number of points and camera poses. Therefore, limiting bundle adjustment to a small "window" of frames offers an efficient way of optimizing camera localization and 3D structure with a controlled complexity.

5.4 Motion Estimation

We have explained in Section 5.2.2 that the camera motion between two frames can be represented by a transformation with 6 degrees of freedom (DoF) called rigid body motion

$$g(\boldsymbol{x}) = R\boldsymbol{x} + \boldsymbol{t}, \quad R \in SO(3), \quad \boldsymbol{t} \in \mathbb{R}^3.$$

In homogeneous coordinates, we can write $g = \left(\begin{smallmatrix} R & \boldsymbol{t} \\ 0 & 1 \end{smallmatrix}\right)$. The core challenge of visual odometry is to accurately estimate this transformation g from the observations provided by the camera images. The two main approaches for this estimation are feature-based or appearance-based. Feature-based motion estimation is also sometimes called *indirect*, while appearance-based is called *direct* visual odometry. The distinction is due to the fact that feature-based methods first step is to find and match features in the images. A feature is a local image pattern differentiable from its surrounding and thus recognizable in multiple images. Once features are paired, only the geometric information (location) of correspondences is kept to estimate the camera motion. In contrast, direct (appearance-based) methods formulate the camera motion estimation as a problem directly depending on the image intensity observations.

5.4.1 Feature-Based Motion Estimation

There are three main feature-based approaches, depending on the formulation of the feature correspondences. A detailed presentation of each scheme is provided in the first part of the visual odometry tutorial by Scaramuzza and Fraundorfer [188].

3D to 3D

When features in previous and current frames have a depth information, one can compare the 3D points coordinates. There are two main strategies to estimate the motion from those point clouds. The first is to consider point clouds globally and try to align them as a whole. Variants of the ICP "iterative closest point" algorithm [23], which is a method for registration of 3D shapes, are well suited for this task. This strategy is often used in the case of RGB-D cameras since it avoids the step of matching features in both images.

When features are matched, another strategy consists in formulating the problem as a minimization problem

$$\arg\min_{g_k} \sum_i \|\boldsymbol{X}_k^i - g_k(\boldsymbol{X}_{k-1}^i)\|$$

where g_k is the rigid body motion between frames $k-1$ and k, $\boldsymbol{X}_k^i$ and $\boldsymbol{X}_{k-1}^i$ are the 3D coordinates of point $\boldsymbol{X}^i$ in the camera frames k and $k-1$. This formulation is quite sensible to outliers and requires a robust approach.

2D to 2D

When no depth information is provided by the sensors, it may be preferable to avoid the estimation of 3D coordinates alltogether. In presence of a calibrated camera, its motion can be recovered thanks to the essential matrix

$$E = t_\times R$$

where t is the translation up to an unknown scale factor, and R is the rotation of the transformation. The essential matrix itself is computed thanks to the epipolar constraint, stating that for every matching pair of normalized image coordinates $\boldsymbol{p}$ and $\boldsymbol{p}'$,

$$\boldsymbol{p}'^\top E \boldsymbol{p} = 0.$$

The essential matrix can be recovered with factorization techniques [139]. When robustness to outliers is desired, the most common solution is to use a RANSAC-like scheme, with a five-point algorithm such as the one presented by Nister [156].

3D to 2D

Instead of comparing 3D coordinates of triangulated points, it is also possible to compare the reprojection of a 3D point $\boldsymbol{X}_{k-1}^i$ into the image I_k, that we will note $g_k(\boldsymbol{X}_{k-1}^i)$, with its actual position in the image, noted p_k^i. This error is usually called a reprojection error. As we will explain soon, we can compute another kind of reprojection error, based on appearance, so we will call this a geometric reprojection error in this document. Recovering the camera motion thus consists in minimizing the geometric reprojection error

$$\arg\min_{g_k} \sum_i \|p_k^i - g_k(\boldsymbol{X}_{k-1}^i)\|^2.$$

This problem is called "perspective from n points" (PnP). The minimal case requires at least three points and is called "perspective from three points" (P3P). A P3P solver may return up to four potential solutions that can be disambiguated with other points. In our interactive visual odometry Web application, presented in Chapter 7, we provide a fast P3P implementation based on Persson and Nordberg's solver [169].

5.4.2 Appearance-Based Motion Estimation

Most appearance-based motion estimation methods are also direct, with some exceptions like the algorithm by Goecke et al. [90] which computes the Fourier-Mellin transform of the image. Our implementation, presented in Chapter 7, belongs to the direct image alignment category so we will detail how this works.

Direct Image Alignment

In general, the direct approach formulates the problem as an image registration (alignment) task. Under the photoconsistency assumption, i.e. the appearance of a point does not change between images, aligning them consists in finding the transformation W minimizing the photometric reprojection error

$$\underset{W}{\operatorname{argmin}} \sum_{\boldsymbol{x}} \|I(W(\boldsymbol{x})) - I^*(\boldsymbol{x})\|^2$$

where I^*, I are the reference and new images, and $\boldsymbol{x}$ is the position of a pixel in the reference image. The transformation W is called the warp function and can take many forms. Most of the time it is parametric, such as a 2D affine transformation, visualized in Figure 5.6 and modelled by the matrix $\begin{pmatrix} 1+p_1 & p_3 & p_5 \\ p_2 & 1+p_4 & p_6 \end{pmatrix}$ where p_1 to p_6 are the six parameters of the transformation. The non parametric case is well known under the name optical flow which consists in recovering the vector field describing the movement of all pixels. To model the warp function by a rigid body motion of the camera capturing the image, one could represent the camera motion by a matrix of the form $(R\ \boldsymbol{t})$ with $R \in SO(3)$ and $\boldsymbol{t} \in \mathbb{R}^3$. The main inconvenience of this parameterization is that R is a constrained 3x3 matrix, so using 9 free parameters to optimize is clearly suboptimal. Recent direct visual odometry algorithms are all using the Lie algebra $\mathfrak{se}(3)$ for the parameterization of the rigid body motion [153, 9, 119, 121, 80, 69].



Figure 5.6: Direct image alignment. The 2D affine transformation between the first and second image is represented with the red rectangle.

Minimization of the Photometric Reprojection Error

The minimization of the photometric reprojection error is a non-linear, non-convex problem due to the form of the warping function. It is common to solve it with a non-linear iterative algorithm such as Gauss-Newton. If we add the parameterization to the notation, $W(\boldsymbol{x}) = W(\boldsymbol{x}, \boldsymbol{\xi})$, and consider that $\boldsymbol{\xi}$ is to be found iteratively, we can rewrite the expression to minimize as

$$\sum_{\boldsymbol{x}} \|I(W(\boldsymbol{x}, \boldsymbol{\xi} + \delta\boldsymbol{\xi})) - I^*(\boldsymbol{x})\|^2.$$

In a Gauss-Newton scheme, the minimum is found by performing a first order Taylor expansion of $I(W(\boldsymbol{x}, \boldsymbol{\xi} + \delta\boldsymbol{\xi}))$. Using the chain rule, the gradient of this expression can be written $J = \nabla I \nabla_{\boldsymbol{\xi}} W$ where ∇I is the image gradient and $\nabla_{\boldsymbol{\xi}} W$ is the Jacobian of the warping function. Let H be the Gauss-Newton approximation of the Hessian, $H = \sum_{\boldsymbol{x}} J^\top J$, then the step of the parameters $\delta\boldsymbol{\xi}$ is given by

$$\delta\boldsymbol{\xi} = H^{-1} \sum_{\boldsymbol{x}} J^\top (I^*(\boldsymbol{x}) - I(W(\boldsymbol{x}, \boldsymbol{\xi})))$$

and the new set of parameters is updated as $\boldsymbol{\xi} \leftarrow \boldsymbol{\xi} + \delta\boldsymbol{\xi}$.

In [16] Baker and Matthews provide a detailed review of the direct alignment problem in a unifying framework showcasing the different possible formulations of the expression to minimize. Those formulations are referred to as forward additive, forward compositional, inverse additive and inverse compositional. The inverse additive approach is subtly different but the other three are easily summarized by Table 5.1. The compositional approaches model the increment as a composition so it is required that the set of warps forms a group. The inverse compositional approach exchanges the roles of the reference image with the new image to align. Image gradients are thus precomputed once on the reference image I^* instead of at each iteration, which provides a computational and accuracy advantage. For this reason, the majority of direct visual odometry algorithms, including our own implementation use that inverse compositional scheme.

Formulation	Expression of residual	Warp update
Forward additive	$I(W(\boldsymbol{x}, \boldsymbol{\xi} + \delta\boldsymbol{\xi})) - I^*(\boldsymbol{x})$	$W(\boldsymbol{x}, \boldsymbol{\xi}) \leftarrow W(\boldsymbol{x}, \boldsymbol{\xi} + \delta\boldsymbol{\xi})$
Forward compositional	$I(W(W(\boldsymbol{x}, \delta\boldsymbol{\xi}), \boldsymbol{\xi})) - I^*(\boldsymbol{x})$	$W(\boldsymbol{x}, \boldsymbol{\xi}) \leftarrow W(\boldsymbol{x}, \boldsymbol{\xi}) \circ W(\boldsymbol{x}, \delta\boldsymbol{\xi})$
Inverse compositional	$I^*(W(\boldsymbol{x}, \delta\boldsymbol{\xi})) - I(W(\boldsymbol{x}, \boldsymbol{\xi}))$	$W(\boldsymbol{x}, \boldsymbol{\xi}) \leftarrow W(\boldsymbol{x}, \boldsymbol{\xi}) \circ W(\boldsymbol{x}, \delta\boldsymbol{\xi})^{-1}$

Table 5.1: Formulation of the expression to minimize and the warp update step depending on the optimization scheme, as detailed in [16].

About the Depth Requirement

In the previous formulation, we eluded writing the actual expression of the warp function but it can be decomposed as

$$\text{Pixel 1 } (2\text{D}, \boldsymbol{x}_1) \longrightarrow \text{Camera 1 } (3\text{D}, \boldsymbol{X}_1) \longrightarrow \text{Camera 2 } (3\text{D}, \boldsymbol{X}_2) \longrightarrow \text{Pixel 2 } (2\text{D}, \boldsymbol{x}_2).$$

The first step of this transformation is to retrieve the coordinates of the 3D point $\boldsymbol{X}_1$ in the frame of the first camera from its pixel coordinates in the first image. However, this is only possible if the depth of the point is known and the camera is calibrated. We make the assumption that the camera is or can be calibrated. There are thus two approaches for direct visual odometry. Either use a sensor configuration easing the process of getting depth information such as RGB-D [119] and stereo cameras, or alternatively estimate camera motion and point depths [69]. This second scheme needs special attention for the initialization. The work we present in Chapter 7 uses RGB-D cameras and is thus in the first category.

We can add that though end-to-end deep learning approaches are just beginning [219], some authors use deep neural networks to estimate depth information in a monocular camera setup [229], thus providing a similar framework than in the RGB-D camera case.

Extensions and Richer Modelizations

The previous developments are based on few assumptions, including photoconsistency. However in reality, those assumptions are incorrect. The appearance of a point in an image for example depends on the surface materials and orientation, and on the lighting conditions. An entirely different field of computer vision called photometric stereo aims at retrieving 3D geometry with advanced modelization of the observed 3D surface and lighting conditions. In our work led by Y. Quéau [176], we show how a dermatoscope can be repurposed to compute precise surface reconstruction of the skin. In visual odometry, some adjustments are possible to better take into account changes in lighting conditions. Global illumination changes, resulting from camera automatic exposure, can be modelled by an affine change of the pixel intensities in the image [69]. Intensity variations due to reflective bright surfaces, occlusions or objects motion can be ruled out or neglected by robust weights [121].

Another incorrect assumption is that an image has been taken at a fixed camera position. In practice, most camera sensors are based on a rolling shutter, i.e. the pixels of the sensor are discharged row after row. The time period of the accumulated light, and thus camera location is different for every row of the image. One can take advantage of rolling shutter for artistic effects such as in Figure 5.7. When tracking high speed camera motions however the bending of straight vertical lines visible in Figure 5.8 will result in errors in the photometric reprojection. Some papers thus model the camera movement as a continuous-time location problem [118] or with piecewise linear velocity [191]. Such extensions are out of scope for our work.



Figure 5.7: Effect of an iPhone rolling shutter when taking a picture of a propeller. Image provided by Soren Ragsdale under creative commons.



Figure 5.8: Effect of rolling shutter bending fences due to the high speed motion of the camera. Image provided by BrayLockBoy under creative commons.

Chapter 6

Performant Web Applications

Contents

6.1 A Brief History of Native Code in the Client 102
6.1.1 Java Applets 102
6.1.2 Flash 104
6.1.3 Google Native Client (NaCl) 104
6.1.4 Emscripten and asm.js 105
6.2 WebAssembly 106
6.2.1 Relation to Previous Technologies 107
6.2.2 Compilation to WebAssembly 108
6.2.3 WebAssembly Minimum Viable Product (MVP) 108
6.2.4 WebAssembly Bright Future 108
6.2.5 Why this Matters for Research 109
6.3 C++ Portability Pitfalls 110
6.3.1 Web Limitations 110
6.3.2 Low Level Native or Architecture Specific Code 110
6.3.3 No Dynamic Linking to OS Libraries 110
6.4 Rust and WebAssembly 111
6.4.1 The Rust Programming Language 111
6.4.2 WebAssembly in Rust 112
6.5 Conclusion 112

In the previous chapter we layed the foundation necessary to modelize the visual odometry problem. Similarly than in the first part of this thesis, we want to explore an interactive scenario through the Web. Visual odometry however, is slightly different from annotation in terms of computational needs. Before reviewing our library and interactive application, we thus make a small detour toward an analysis of performance capabilities in client Web applications. Our tour starts with a brief history of "native" code in browsers, followed by a detailed presentation of WebAssembly, a recent technology about to change how we share and distribute high performance code. Finally we provide an explanation for the choice of the Rust programming language for the development of our visual odometry library.

6.1 A Brief History of Native Code in the Client

Being able to run high performance code in the browser is useful in many use cases, including for example scientific computing. Yet, it remains a challenging task. Resources often refer to this as "native" code but the terminology is rather vague. Depending on the context, "native" may have one of the following meanings,

1. code statically compiled directly to the target architecture and running from the browser,
2. code compiled from a typical "native" language such as C or C++,
3. or anything that is not generating JavaScript.

The distinction between those and how they relate to "native" code will be made clearer after the following brief history of high performance code in browsers.

6.1.1 Java Applets

In 1995, just four years after the birth of the Web, the Java programming language was created. It appeared along with a companion technology called Java Applet, designed to run Java applications in the browser. The Java Virtual Machine (JVM) was hosted by browsers, enabling much better performances than JavaScript at that time. As an example, Brendon C. Glazer worked on interactive ray tracing of VRML scenes with Java applets in 1999 [88]. Figure 6.1 depicts how the applet would appear in a Web page at that time. Since 1998 Java applets have also had access to 3D hardware acceleration [144] whereas JavaScript waited until 2011 for WebGL in HTML5 canvas.

On the down side, Java applets would break accessibility of the Web. Screen readers would not be able to parse the content of the dedicated applet area in the page. The security model of Java applets also had some weaknesses. Applets would have to get approved by a browser user and then gain rights equivalent to a native desktop application. Unfortunately, just like terms of service, most people click "agree" and move on [160]. In addition, the Java Runtime Environment (JRE) has had hundreds of security vulnerabilities [115] in its

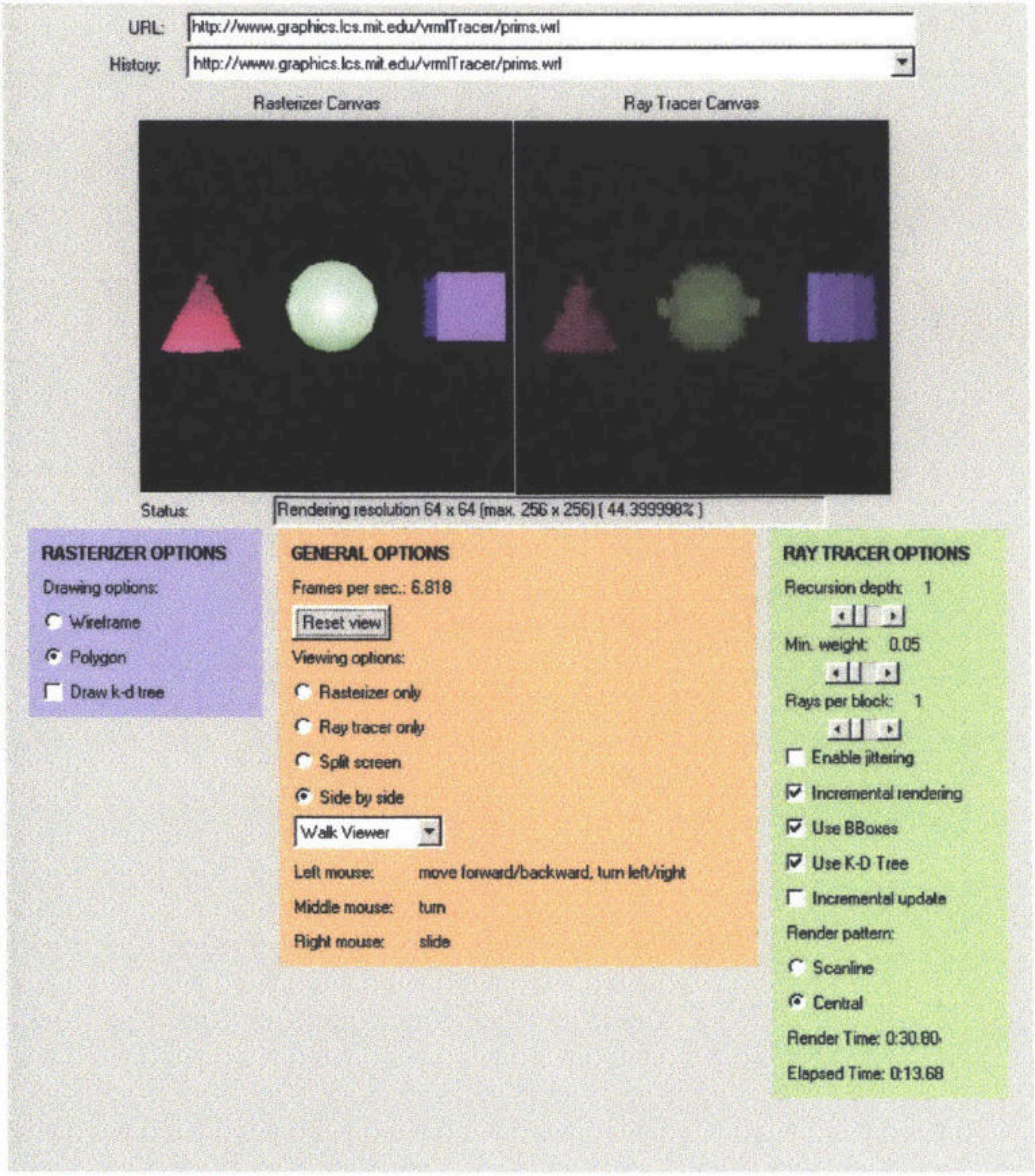


Figure 4.1.1 - Applet
This is a scaled down image of the applet as it would appear in a web page.

Figure 6.1: Interactive ray tracing applet from Brendon C. Glazer master thesis (1999).

lifetime. This represents a serious security threat for browser vendors. Java applets were eventually entirely removed from Java SE 11, in 2018.

6.1.2 Flash

In his "History of Flash" [83], Jonathan Gay retraces the early days of Flash. In 1993, he, Charlie Jackson and Michelle Welsh founded FutureWave Software but their initial vector drawing application did not draw much attention (pun intended). After discussions at Siggraph 1995, the company decided to focus on a web animation product named "FutureSplash Animator". It gained reputation with Microsoft and Disney using it and as a consequence, was acquired by Macromedia and rebranded "Flash" in 1996.

Contrary to Java applets, Flash use cases started very narrow. It provided a simple and efficient solution to build and share animations on the web, offering a visual advantage over pure HTML documents. In year 2000, Flash 5 was released, with the ActionScript programming language, bringing even more potential to Flash-based websites. Two years later, in 2002, Flash 6 brought support for real-time messaging protocols, enabling video and audio streaming capabilities. This represents roughly 8 years until 2010 when such capabilities are also supported through HTML5 in most browsers. In the meantime, highly influencial projects shaped the Web thanks to Flash. For example, Chad Hurley, Steve Chen and Jawed Karim launched YouTube in 2005, based on Flash.

In spite of the many advantages of Flash and ActionScript over classic Web pages, it also had similar flaws than Java applets. In October 2000, usability consultant Jakob Nielsen published a short article entitled "Flash: 99% Bad" [155] stating that "Flash technology tends to discourage usability for three reasons: it makes bad design more likely, it breaks with the Web's fundamental interaction style, and it consumes resources that would be better spent enhancing a site's core value." Apple also played a huge role in Flash decline. In 2007 the iPhone launched without Flash support, leading to Steve Jobs, then Apple CEO, writing in 2010 an oppen letter called "Thoughts on Flash" [114] in which he explains why Flash was doomed to disappear. Among those reasons, Flash is proprietary, going against open Web standards and it has numerous security flaws [79].

Consequently, with the advent of HTML5 surrounding technologies regarding multimedia capabilities, and the improved performances of JavaScript, Flash became obsolete. So in July 2017, Adobe announced end support of Flash in 2020.

6.1.3 Google Native Client (NaCl)

Java applets and Flash were never trully considered "native" since they involved third party virtual machines. One would not compile code directly to machine instructions but to an intermediate bytecode representation for Java applets, or just using a JIT compiler for ActionScript (Flash) code. They provided however great performances improvements when compared to JavaScript before 2008 and the arrival of JavaScript JIT compilers with the V8 engine. But around 2009 and 2010, two new projects named Native Client (NaCl) and Emscripten respectively emerged from research at Google and Mozilla. We will hold on

Emscripten for now and explain first what NaCl was about.

At this period of time, native code was already running in browsers, usually via an old plugin interface called the "Netscape Plugin API" (NPAPI). This is how the JVM, the Flash player or PDF readers for example would be integrated in browsers. So from the observation that the Web had trully become a rich multimedia platform, and that native code was already running in browsers via plugins, a team of Google engineers explored a way to generalize and improve security of any native C or C++ code execution in browsers [230]. This would in theory unlock high performances for any Web application. But it would need a sandboxed and secure new set of APIs to make sure that those applications would not execute any malicious code via the browser. That is how the Native Client (NaCl) project, and its associated API called Pepper Plugin API (PPAPI) started in 2009.

The two main downsides of this approach are the security and portability concerns. Even though NaCl code would run into a sandboxed environment, enforcing both security and accessibility to classical desktop resources requires a continuous effort from browser vendors, now needing to secure two different sandboxed environment, NaCl and JavaScript, instead of one. Portability was also a concern since any NaCl code would need to be compiled to all target architectures where the browser need to run. In practice, only x86 Intel architectures were fully supported, going against the nature of the Web, supposed to run on all platforms. This limitation was the reason for the birth of the Portable Native Client project.

Portable Native Client (PNaCl, pronounced like the word "pinnacle"), was a work by A. Donovan et al. [62] aiming at distributing NaCl programs in an intermediate pre-compiled neutral instruction-set format, preventing the need to compile directly to all target architectures. Concretely, the format chosen was the Low-Level Virtual Machine (LLVM) bitcode. Unfortunately, this intermediate representation bitcode is a fast moving target, and retro-compatibility is not a main objective of the LLVM project. It means that PNaCl code could become obsolete at a fast pace, also a deal breaker for Web standards. With the reluctance of other vendors to adopt a non-specified Google initiative, and the appeal of another rising approach coming from Emscripten, PNaCl was finally deprecated in 2017.

6.1.4 Emscripten and asm.js

Out of curiosity, Mozilla engineer Alon Zakai started to work around 2010 on a project to explore the limitations of compiling C++ code to JavaScript. It was the start of the Emscripten project [232]. Emscripten base idea consists in converting LLVM bitcode to JavaScript. LLVM originated from the work of Chris Lattner and Vikram Adve [130] on a compiler framework and an intermediate code representation, optimized for compiler transformations. It reduces the language-architecture complexity from $\mathcal{O}(mn)$ to $\mathcal{O}(m+n)$ since languages do not need to compile to every target like x86 or Arm, and instead can just target LLVM which in turn knows how to compile to each instruction set architecture (ISA) as depicted in Figure 6.2.

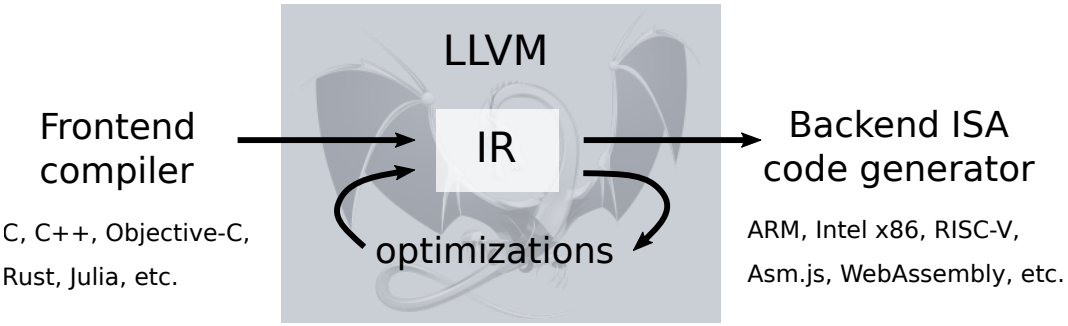


Figure 6.2: LLVM Compiler Infrastructure

After few iterations with the help of Luke Wagner, working on JavaScript compilation at Mozilla, Emscripten was able to output JavaScript code running roughly two to three times slower than the equivalent C code. This specific subset of JavaScript was formalized under the name asm.js in 2013 [103]. The asm.js code for a sum function is presented in Listing 6.1. As you can see, it makes use of JavaScript operations doing nothing, like the bitwise-or with 0, except cohercing values to certain types. You would not want to write asm.js code by hand, but for a compiler, it enables certain kinds of optimizations not possible with traditional hand-written JavaScript code. Yet it is a strict subset of JavaScript and can still run in any browser, whether or not they have a specialized handling of asm.js.

```
1 function add(x, y) {
2     var x = x | 0;  // x is a 32-bit value
3     var y = y | 0;  // so is y
4     return (x+y) | 0; // 32-bit addition, no type or overflow checks
5 }
```

Listing 6.1: Sum of two 32-bit integers in asm.js.

The performances without the need of any plugin convinced companies with huge code bases to join the effort. The Unreal game engine was ported [71], Autodesk AutoCAD [101], Adobe Lightroom [134], OpenCV [206] and many others. With the success of Emscripten and asm.js as a proof of concept, all major browser vendors came together to formalize a new specification known today as WebAssembly. Figure 6.3 summarizes main events leading to the creation of WebAssembly.

6.2 WebAssembly

WebAssembly, abreviated Wasm, is an instruction set with a binary format, and a stack-based virtual machine able to execute those programs [97].

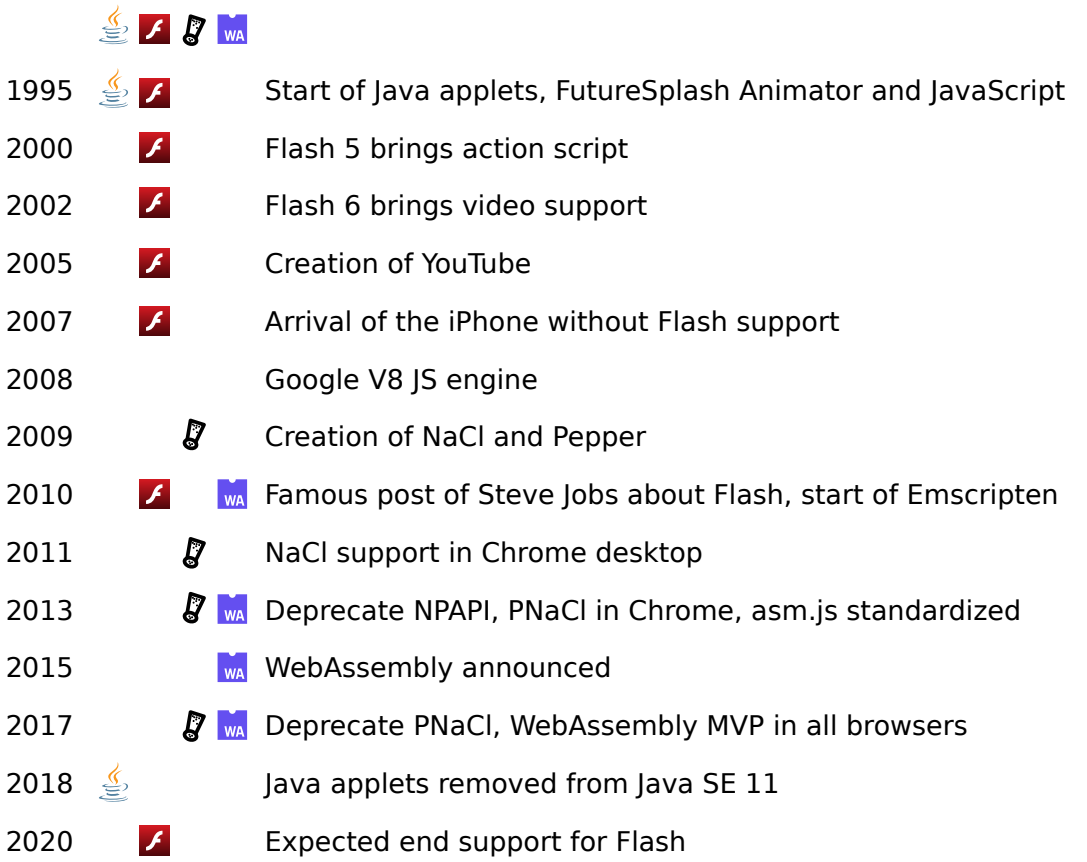


Figure 6.3: Events releated to the birth of WebAssembly.

6.2.1 Relation to Previous Technologies

Just like Flash and Java applets, WebAssembly code is executed by a virtual machine. Similarly to NaCl, the Wasm runtime is sandboxed, preventing it from accessing data outside of its context and compromising the security of the underlying machine. And as asm.js, Wasm does not require any plugin. It has been developed as a Web standard and implemented by all major Web browsers, desktop and mobile, since 2017. Unlike asm.js however, Wasm is distributed in a binary format, meaning it is more efficient to send over the network and to parse, drastically reducing the loading time of Web pages. WebAssembly has been built drawing lessons from the past and is likely to endure for the following reasons.

- It is a Web standard and a collective effort from all browser vendors.
- It is the low level missing piece of the Web, complementing JavaScript where more control over performances is needed.
- It offers strong security guaranties, and a sound type system [224].

- It theoretically enables any programming language to run on the Web.

6.2.2 Compilation to WebAssembly

Currently WebAssembly does not provide a garbage collector. Therefore, there are two main approaches to compile to WebAssembly. If a programming language enables memory management at compile time, such as C, C++ or Rust, the simplest way is to target LLVM intermediate representation. Emscripten enables compilation to WebAssembly from C and C++. Rust can either use Emscripten as a backend or directly the wasm32 target of LLVM. Otherwise, when a programming language requires a runtime for features such as garbage collection, the entire runtime needs to be ported to WebAssembly in addition to the actual program. This is for example the case of Blazor, a .NET (Microsoft) library enabling the creation of Single Page Applications written in C#.

6.2.3 WebAssembly Minimum Viable Product (MVP)

The first version of WebAssembly, initially released on March 2017, is called a minimum viable product (MVP). This consensus aimed at producing a simple yet functional specification and implementation for all major browser vendors. Concretely, the Wasm MVP offers roughly the same capabilities than asm.js. It has only four types, integers and floating point numbers in 32 and 64 bits, and manages memory as a unique contiguous block of bytes. New features such as threads, SIMD, exception handling, reference types, garbage collection or tail call optimization, are currently being worked on and delayed to the after-MVP phase.

6.2.4 WebAssembly Bright Future

A common pun in WebAssembly communities is to describe it as “neither Web, nor assembly” [170]. Although slightly caricatural, the WebAssembly specification does not require any Web component. As a matter of fact, just like JavaScript broke out of the browser with Node, many non-browser WebAssembly runtimes have already been created, such as Wasmer [221] and Wasmtime [223].

Since Wasm is basically a performant sandboxed calculator, interoperability capabilities with files or the network must be provided in the form of imports and exports of a Wasm module. In the browser, those are passed to the module at the initialization call from JavaScript. But outside of the browser, as in Wasmer for example, imports and exports have to be provided by the runtime environment. If we want to be able to use the same Wasm module in those two different contexts, we need to have a common interface specification, which can have different implementations in each runtime environment.

In order to adress those points, the WebAssembly working group is standardizing a system interface under the name WebAssembly System Interface (WASI [47]) and a types interface known as WebAssembly Interface Types [48]. As shown in the demonstration video

available on YouTube [49], it is now possible to create a Wasm library in Rust for example, and then call it from the Web, Node, Rust, but even Python or any language capable of embedding a WebAssembly runtime.

6.2.5 Why this Matters for Research

A growing number of actors are worrying about research reproducibility [43, 29] in computational sciences. Nick Barnes published an article on Nature News [18] encouraging researchers to publish even bad code instead of none.

IPOL Journal [135], Image Processing On Line, is an initiative supported by the French space agency CNES and the European Research Council. Its goal is to publish both precise algorithm descriptions and certified implementations, and making them available on an online platform. Due to performance constraints, they rejected most garbage collected languages interpreted or based on virtual machines such as Python and Java, and only accept Fortran, C and C++ implementations with strict portability requirements. In order to mitigate security risks, authors are uniquely identified. Exposure to malicious data, such as images uploaded by online users, is reduced by system restrictions and careful examination of the source code during review. WebAssembly would have been a perfect fit for this project if it had existed in 2010 since it provides a solution for secure, performant and portable code.

The Association for Computing Machinery (ACM) recently revised its artifact review and badging policies [2], based on the International Vocabulary for Metrology. A work is considered repeatable if the results are obtainable on multiple trials, within the same team and experimental setup. It is considered replicable if another team is capable of getting the same results while reusing the provided artifact and same experimental setup. Finally it is considered reproducible if the results can be reproduced with different teams and experimental setup, without using the provided artifacts. In accordance to these guidelines, ACM Multimedia, the ACM conference targetting multimedia research, has setup a reproducibility review process, based on companion papers [150]. Companion papers are distinct from the main paper. They must be provided with associated artifacts and describe precisely how to replicate the findings of the associated original paper contribution.

Containarization of the development environment with tools such as Docker is a new practice reducing dependency conflicts. It has been picked up by the machine learning community, especially when needing to use different versions of languages and tools for deep learning tasks in separate projects. Carl Boettiger even wrote a paper entitled “An introduction to Docker for reproducible research” [26] describing the different aspects of computer environments preventing work reproducibility. Yet, Solomon Hykes, the original author of Docker shared on twitter that if WebAssembly and its system interface existed in 2008, he wouldn’t have needed to create Docker [109]. “That’s how important it is” as he said.

6.3 C++ Portability Pitfalls

Not every C or C++ project can be ported to WebAssembly. There are three main factors of non portability: Web limitations, no low-level architecture specific code and no system dynamic linking.

6.3.1 Web Limitations

High performance code may rely on parallelization to speed up processing. Due to security concerns, parallelization methods such as multithreading or SIMD are currently a work in progress but not usable by default in browsers.

6.3.2 Low Level Native or Architecture Specific Code

Other high performance code may rely on architecture specific constraints. For example, x86 assembly is not portable to other platforms. Code requiring a big-endian physical ordering of memory, i.e. starting multi-bytes data types with the most significant byte, will not work since WebAssembly makes the assumption of a little-endian machine, such as in the case of x86, ARM or RISC-V.

6.3.3 No Dynamic Linking to OS Libraries

Emscripten is at its core an LLVM backend (cf Figure 6.2) targetting the WebAssembly instruction set. A WebAssembly module must run in isolation inside the browser and thus needs access to every dependency inside the WebAssembly virtual machine. Therefore, Emscripten requires the LLVM bitcode of every dependency used. In traditional C and C++ environments however, it is common practice to depend on precompiled shared dynamic libraries (.so and .dll files) provided by the OS packages. In contrast, Emscripten needs the source code of every direct and transitive dependency to produce the LLVM bitcode and then the Wasm code. To ease the process, Emscripten already includes most of the C and C++ standard libraries, in addition to few common libraries for graphical applications such as Simple DirectMedia Layer (SDL), a cross-platform multimedia library.

DVO [201] for example relies on OpenCV, Eigen, Boost and Sophus. Unfortunately, OpenCV has only been partially ported to WebAssembly [206]. The porting team assumed that, in the context of the Web, media would be generated from Web APIs. Reading an image for example, is performed through the creation of an HTML canvas, loading and decoding the image data, which is then transferred to an OpenCV matrix. Native OpenCV however would use the `imdecode` function, itself depending on a dozen of other libraries such as libjpg, libtiff and libpng, not all of which have been ported to WebAssembly. When trying to add the `imdecode` function to the set of ported OpenCV code, one currently hits "missing symbol" errors due to similar dependency and linking issues. In the case of DVO

and other RGB-D visual odometry algorithms, there is a need to decode 16 bit PNGs for the depth images. Until they are supported in Web APIs it will not be possible with OpenCV.

This limitation does not mean that DVO can never be ported to WebAssembly, but that it would require a significant amount of work to either complete the OpenCV port, or find another image decoding library already ported to WebAssembly and adapt DVO code to accomodate it. The same could happen for parts of Eigen, Boost and Sophus used in DVO.

6.4 Rust and WebAssembly

WebAssembly and the Rust programming language are both technologies that emerged from research projects at Mozilla. Though independent, their core communities are thus sharing similar values and working in synergy within the Rust and WebAssembly Working Group.

6.4.1 The Rust Programming Language

Rust is a programming language focused on safety, speed and concurrency, that begun in 2006 as a side project by Mozilla employee Graydon Hoare. It has been evolving drastically until 2015, when version 1.0 was announced with stability goals in mind. There are many reasons why Rust is among the best contestants for writing efficient and correct computer vision algorithms. We will focus on those related to our task, interactive computer vision on the web.

First, Rust uses an LLVM backend so it is capable of targetting WebAssembly by default. In addition, Rust has an automatic memory management system called ownership. It enables memory safety guarantees at compile time without the need of a garbage collector or manual memory management. This is important in the context of WebAssembly since it means that Rust code can be compiled to Wasm without embedding a big runtime for memory management. Ownership in Rust is based on the concept of linear types, highly influenced by the work of Girard [85], Wadler [216], Baker [14], and Clark et al. [50] among others. The main idea behind Rust ownership is that every value has a unique variable called its owner. When this variable goes out of scope, the associated value is released from memory. Other variables can access this value during its lifetime through a borrowing mechanism, similar to references. At any time, there must be at most one mutable borrow, and there cannot be mutable and immutable borrows at the same time. Those constraints are very similar to the read and write constraints usually enforced on shared values in concurrent programming.

Finally Rust is not affected by the dynamic linking issues we mentioned earlier when targetting WebAssembly in C++. Rust 1.0 shipped with a project manager called “cargo” and an online package registry “crates.io”. Cargo manages compilation, modules and external dependencies with the use of a very simple project description file named `Cargo.toml` placed at the root of a Rust project. As a consequence, compiling a pure Rust project will almost

always work with a single command `cargo build`. It downloads dependencies, locally build and statically link them.

In contrast to the Elm package registry we discussed in Section 3.4.6, Rust packages follow but do not enforce semantic versioning. Similarly to Elm, Rust also provides algebraic data types with pattern matching, and immutability by default, thus giving much more confidence than C++ that when a program compiles it is correct.

6.4.2 WebAssembly in Rust

The WebAssembly tooling ecosystem in Rust is composed of four main projects.

- `wasm-bindgen` [220] facilitates interoperability between Rust and JavaScript with the help of simple code annotations.
- `wasm-pack` [222] complements `wasm-bindgen` by generating all the JavaScript glue code currently necessary to load and call a WebAssembly module from JavaScript as if it was an ES6 module. `wasm-pack build` basically serves as a replacement to `cargo build` when compiling to WebAssembly.
- `js-sys` [117] provides raw bindings to JavaScript global APIs for projects using wasm-bindgen.
- `web-sys` [225] provides raw bindings to Web APIs for projects using wasm-bindgen.

Out of those four projects, `wasm-pack` is considered a tool, installed alongside the rest of Rust tooling. The three others simply are libraries, added to a project dependencies with one line each in the description file `Cargo.toml`.

6.5 Conclusion

The Web is in constant evolution. It is also the most ubiquitous platform, accessible from multitude of devices such as laptops, tablets, phones and embedded devices. Being able to run performant code in Web browsers has thus being a competitive advantage, and many technologies have tried to corner that market. As we explained however, all those past technologies such as Java applets, Flash or Google NaCl have had huge drawbacks, preventing them from standard adoption. Recently with the arrival of WebAssembly, we finally arrived at a point where high performance code can run in Web browsers with very few inconvenients. For this reason, the four major browser vendors, Google, Mozilla, Microsoft and Apple all rallied together to define this new standard and implement a minimum viable product in few months. We also explained why the current best two languages to target WebAssembly are C++ and Rust. While C++ has the original WebAssembly code generator available, called Emscripten, the language struggles with building dependencies and legacy

non portable code. Combined with the intrinsic difficulties of that language to safely build on other contributions, we decided to avoid C++ and start fresh with the Rust programming language. We can also note that in 2019, for the fourth year in a row, Rust has been elected the most loved programming language by respondents of the Stack Overflow's annual developer survey [186]. So we hope that this choice will also bring more joy to potential future contributors. In the next chapter, we will thus introduce our visual odometry library in Rust and present its port to an interactive Web application.

Chapter 7

Interactive Visual Odometry on the Web

Contents

7.1 Visual Odometry in Rust (VORS) 116
7.1.1 Overview of VORS 116
7.1.2 Intuition on Direct Image Alignment 116
7.1.3 Sparse Points Selection 118
7.1.4 Multi-Resolution Direct Image Alignment 119
7.1.5 Limits of the Implementation 122
7.2 RGB-D Visual Odometry Evaluation 122
7.2.1 Dataset Creation / Acquisition 123
7.2.2 Evaluation Metrics 126
7.2.3 Setup and Algorithms Evaluation 128
7.3 Interactive VORS on the Web 133
7.3.1 Port of VORS to WebAssembly 133
7.3.2 Interactive VORS Web Interface 135
7.3.3 Human in the Loop Closure 137
7.4 Conclusion 140

7.1 Visual Odometry in Rust (VORS)

7.1.1 Overview of VORS

VORS is a sparse, keyframe-based, RGB-D, direct visual odometry algorithm. As presented in Section 5.4.2, VORS belongs to the family of direct visual odometry, based on the image alignment optimization problem. Contrary to monocular visual odometry, which requires an estimation of points depth, we focus on the RGB-D case, where the necessary depth information for reprojection is provided by a sensor. In order to reduce drift when the camera moves slowly, image alignment is performed from a fixed keyframe instead of the previous frame. Decision to change the keyframe is done heuristically. Finally, our algorithm uses a sparse subset of points in the image, which has been shown [69] to be sufficient to track the camera motion. Disregarding sparsity and robustness, discussed later, our algorithm is very similar to DVO [119], and its predecessor [201]. An overview of the full pipeline of VORS is presented in Figure 7.1.

7.1.2 Intuition on Direct Image Alignment

As presented in Section 5.4.2, under the photoconsistency assumption, direct image alignment consists in finding the warp function W minimizing

$$\sum_{\boldsymbol{x}} \|I(W(\boldsymbol{x})) - I^*(\boldsymbol{x})\|^2$$

where I^*, I are the reference and new images, and $\boldsymbol{x}$ is the position of a pixel in the reference image. As explained in Baker and Matthews [16], this can be minimized with an iterative optimization. If we note $\boldsymbol{\xi}$ the parameters of the warp function, and $\delta\boldsymbol{\xi}$ the iterative increment, the expression of the residual in an inverse compositional formulation is

$$I^*(W(\boldsymbol{x}, \delta\boldsymbol{\xi})) - I(W(\boldsymbol{x}, \boldsymbol{\xi})).$$

In a Gauss-Newton scheme, the iterative increment $\delta\boldsymbol{\xi}$ is computed as

$$\delta\boldsymbol{\xi} = H^{-1} \sum_{\boldsymbol{x}} J^\top (I^*(\boldsymbol{x}) - I(W(\boldsymbol{x}, \boldsymbol{\xi})))$$

where $J = \nabla I^* \cdot J_{\boldsymbol{\xi}} W$, ∇I^* is the reference image gradient and $J_{\boldsymbol{\xi}} W$ is the Jacobian of the warp function. The Hessian is computed as the Gauss-Newton approximation $H = \sum_{\boldsymbol{x}} J^\top J$. For readability of the expression inside the sum on all pixels $\boldsymbol{x}$, we did not indicate in the notation that J, composed of ∇I^* and $J_{\boldsymbol{\xi}} W$, is computed at pixel $\boldsymbol{x}$. The image gradient ∇I^* at $\boldsymbol{x}$ is a 1x2 vector, while the Jacobian of the warp $J_{\boldsymbol{\xi}} W$ at $\boldsymbol{x}$ is a 2x6 vector (2 coordinates, 6 parameters). We leave aside the expression of the warp Jacobian, detailed in Appendix A.1. Two important components of $\delta\boldsymbol{\xi}$ are the reference image gradient ∇I^* and the residual

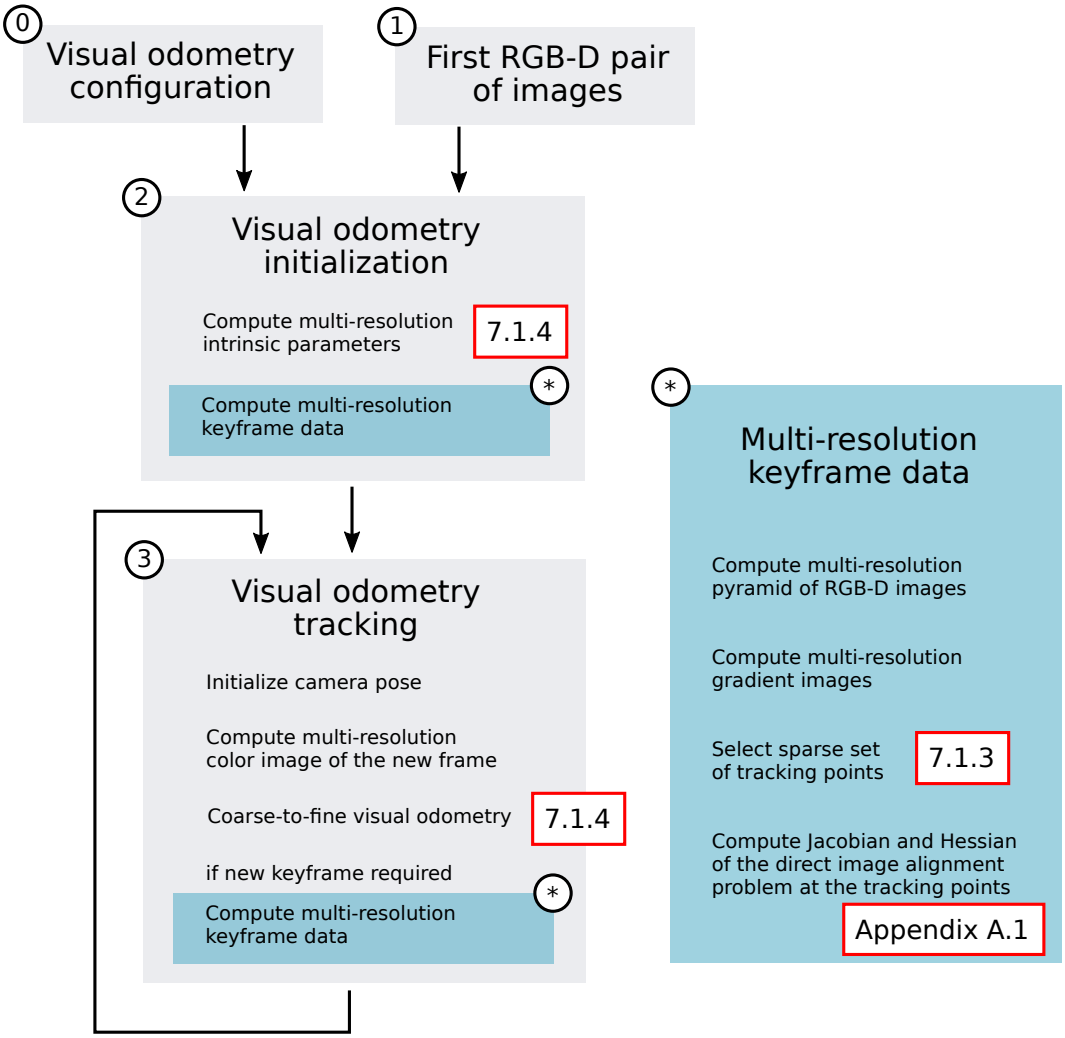


Figure 7.1: Overview of VORS pipeline.

image $\delta I = I^*(\boldsymbol{x}) - I(W(\boldsymbol{x}, \boldsymbol{\xi}))$ which is the difference between the reference image and the reprojected pixels on the second image. We represented both in Figure 7.2 for a couple of images in the ICL-NUIM sequence. As we can see there, not all pixels are participating equally in the expression. The most informative pixels are all near high gradient locations which actually makes sense. Indeed if we, as humans, compare two homogeneous areas, it will be impossible to tell how the camera moved. If instead we are presented heterogeneous regions of the image with textures it is possible to visualize the movement. The same occurs for the expression of the warp increment $\delta\boldsymbol{\xi}$. Since pixels with high gradient magnitudes contain the most information, it should be possible to sample only those for direct image alignment, which leads us to the presentation of sparse points selection mechanisms.

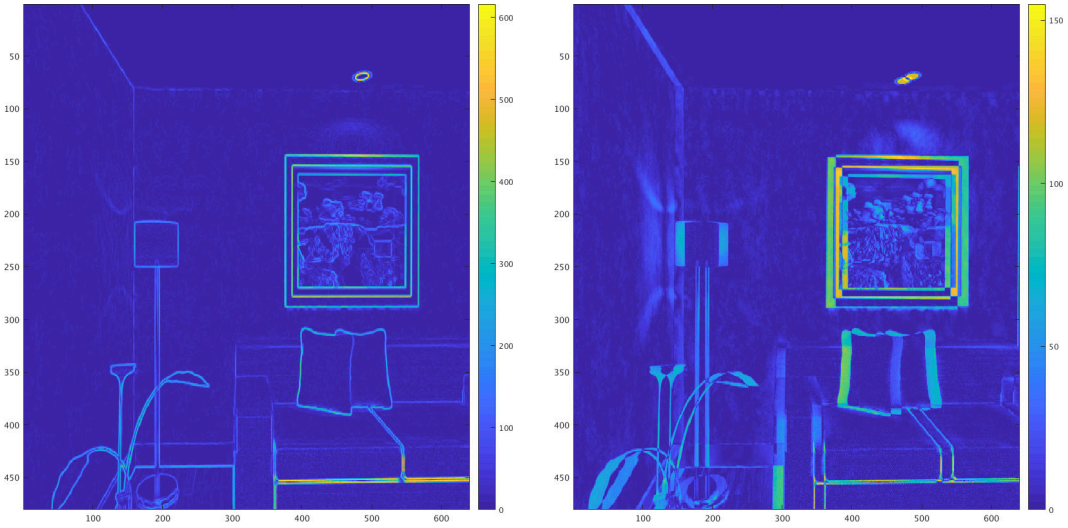


Figure 7.2: Gradient norm of the reference image (left), and absolute residual image (right).

7.1.3 Sparse Points Selection

Engel et al. demonstrated in DSO [69] that using a sparse subset of points for the direct image alignment problem still results in precise camera locations at the condition that points satisfy two properties,

- they should be well distributed in the image,
- and they should be located at positions with higher gradient magnitudes.

In DSO, selected points are called candidate points. Though presented quickly in the paper, their selection mechanism is much complex, based on at least ten parameters. The core idea is to regularly divide an image in tiles. One subdivision produces big tiles, called regions, and another generates small tiles, called blocks. One pixel is selected per block if its gradient is higher than a local threshold depending on the region containing the pixel. In practice, blocks are actually multi-scale, with three default levels. If none of the four "level n" blocks composing a "level $(n+1)$" block elected a candidate point, the "level $(n+1)$" block checks whether a pixel satisfies a lower threshold. The factor between block thresholds at different levels is one of the parameters. Another mechanism aims at obtaining a target amount of candidate points. That amount can be approximated from blocks base size but it might differ. Depending on the difference with the target count, the algorithm will choose between the three following options, (1) keep candidates, (2) randomly select a sample of the candidates, or (3) change the block base size and restart from scratch.

Since we care about the complexity of VORS, we tried a different approach, much simpler, yet resulting in a distribution satisfying the two expected properties. Our sparse selection mechanism is based on a multi-resolution pyramid of images. We embraced the idea that each level of the image pyramid should exhibit that property of well distributed points

with higher density near higher gradients. We thus propose a simple coarse-to-fine selection mechanism, depicted in Figure 7.3. At the lowest resolution, all points are candidates. For each candidate pixel at one pyramid level, we elect one or two candidates in the four subpixels of the next level. A threshold based on gradients is used to determine which pixels should be selected. This sparse selection scheme ensures that there is at least one candidate per region corresponding to one pixel at the lowest resolution. It also increases the density of candidates in higher gradients areas. The number of levels of the image pyramid is a common parameter for candidates selection and for the multi-resolution direct image alignment presented next. As a result, we obtain a sparse selection of points with only one parameter, the gradient threshold to pick one or two subpixels as candidates.

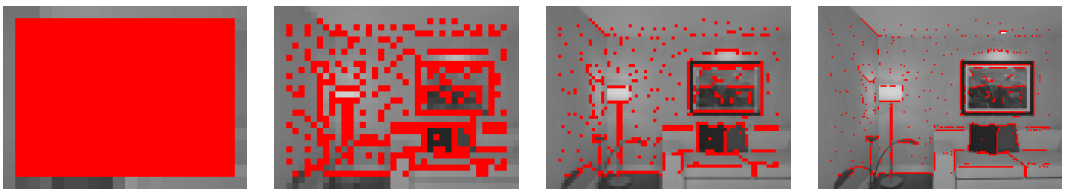


Figure 7.3: Coarse-to-fine candidate selection of VORS. Candidates are represented in red. All points are kept at the lowest resolution. Each higher resolution elects one or two candidates per parent candidate.

Figure 7.4 depicts a zoomed-in view of the same image region for both DSO and VORS candidates selection mechanisms. As visible on the left image, DSO candidate points are regularly spaced, except in homogeneous areas. VORS candidate points however tend to form contiguous lines, which probably generates redundant information. Limiting the maximum number of subpixel candidates to one at some levels (in the higher resolutions), could both help control the maximum amount of candidates and limit the information redundancy generated by neighbour candidates.

7.1.4 Multi-Resolution Direct Image Alignment

Convergence of the Optimization Problem

In theory, the iterative algorithm computing motion increments is only correct under the assumption that the initialization is already near the solution. Convergence to the correct minimum is thus a hard problem and under some circumstances, the optimization may drift to another local minimum. One way to help convergence is to use the Levenberg-Marquardt approximation of the Hessian. It consists in multiplying the diagonal elements of the Gauss-Newton approximation of the Hessian by $(1 + \lambda_{lm})$. The Levenberg-Marquardt coefficient λ_{lm} is dynamically updated toward 0 when converging or toward $+\infty$ when diverging.

Another method to improve convergence consists in solving the optimization with a coarse-to-fine multi-resolution approach. Indeed, the image gradient used for the Jacobian

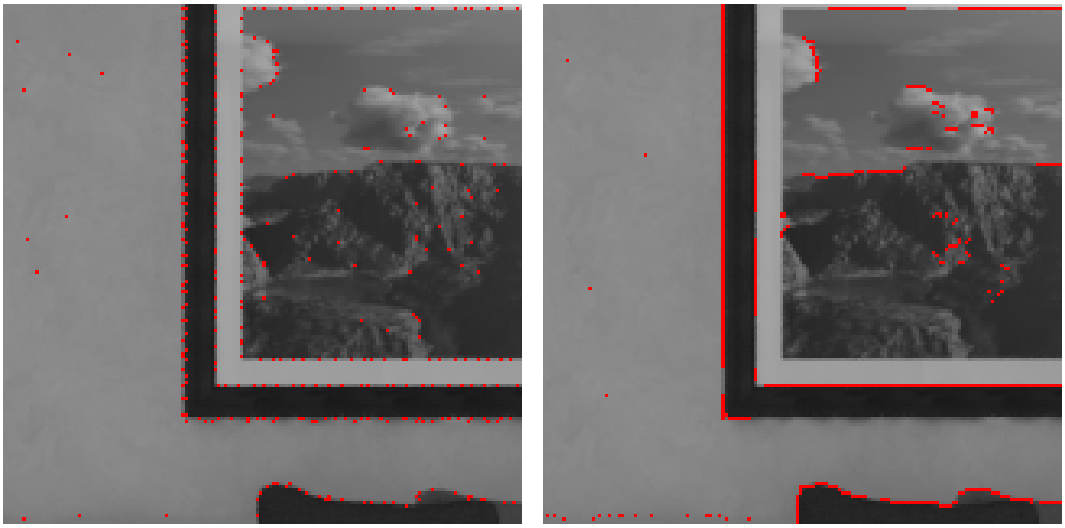


Figure 7.4: DSO (left) and VORS (right) candidate points (in red).

contains information of larger areas of the original image when computed at lower resolutions. As a consequence, the vicinity of the global optimum is artificially increased. For the direct image alignment, we thus use a pyramid of images, where each level has half the resolution of the previous one. As explained previously the number of levels also impacts candidate points selection. Indeed, at the lowest resolution, all pixels serve as candidates for the optimization. The number of levels is thus chosen as a compromise between the minimum amount of candidate points and the desired convergence rate. Starting from 640x480 images in the ICL-NUIM and TUM RGB-D datasets, we found that 6 levels is a good compromise. The lowest resolution has 20x15 images, which implies a minimum of 300 candidate points.

Multi-Resolution Intrinsics

Since we use a coarse-to-fine optimization, we must be able to initialize one level from the result of the previous one. In Section 5.1.4, we presented the image formation as the succession of two transformations, first the projection from the camera frame to the image frame, and then the projection onto the pixels frame. When halving the image resolution, the second transformation K_s changes.

$$K_s = \begin{pmatrix} s_x & s_\theta & o_x \\ 0 & s_y & o_y \\ 0 & 0 & 1 \end{pmatrix}.$$

We note K'_s the new pixels projection with a resolution multiplied by α. In our case α is of the form 2^{-n}. On the one hand, scaling parameters are all multiplied by α i.e. $s'_x = \alpha s_x$, $s'_y = \alpha s_y$ and $s'_\theta = \alpha s_\theta$. The principal point parameters on the other hand depend on

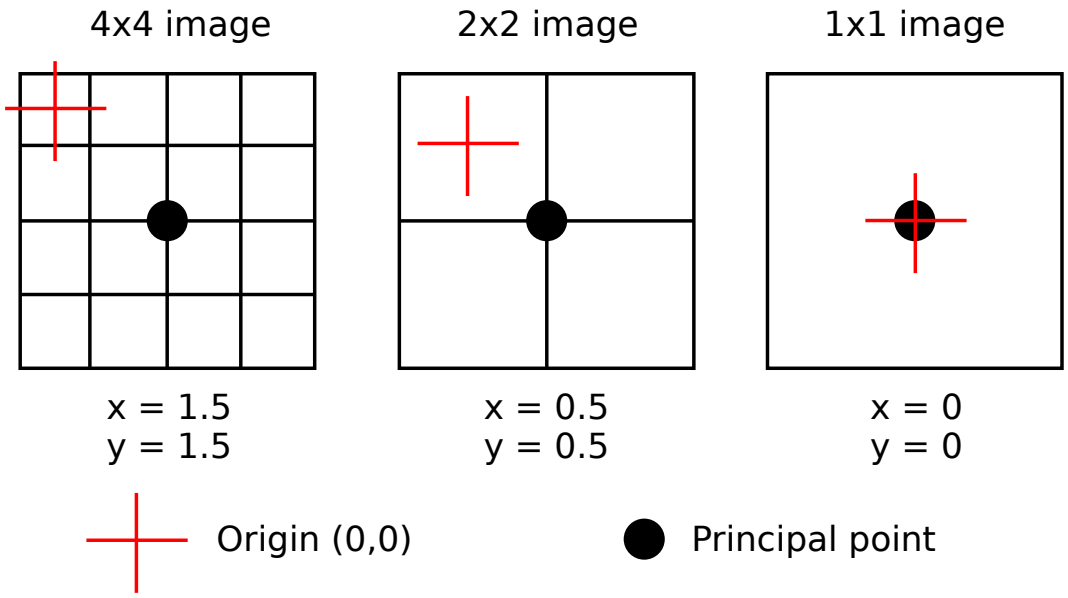


Figure 7.5: Effect of a centered coordinate system on the coordinates of the principal point at multiple resolutions.

the origin of the coordinate system. In most modelizations, the value of pixel (x, y) is interpreted as the integration of the light hitting the sensor on the surface located between $(x - 0.5, y - 0.5)$ and $(x + 0.5, y + 0.5)$. The coordinates of the top left corner of the image is thus $(-0.5, -0.5)$ instead of $(0, 0)$. Figure 7.5 illustrates how the principal point is transformed in such coordinate system. As a result, the location of the principal point is updated as $o'_x = \alpha(o_x + 0.5) - 0.5$ and $o'_y = \alpha(o_y + 0.5) - 0.5$. In the end, the updated intrinsic pixel transformation is

$$K'_s = K_\alpha K_s \quad \text{with} \quad K_\alpha = \begin{pmatrix} \alpha & 0 & 0.5(\alpha - 1) \\ 0 & \alpha & 0.5(\alpha - 1) \\ 0 & 0 & 1 \end{pmatrix}.$$

The presented approach is the one used by the TUM RGB-D coordinate system for the principal point given in the intrinsics matrix. It is thus also the one we use when initializing the multi-resolution intrinsic parameters in step 2 of Figure 7.1. The camera motion estimated at one resolution can thus be reused as-is for the initialization of the next resolution. It is the projection that changes, due to a different intrinsic matrix.

Keyframe Update

We mentioned in VORS overview that the decision to change the keyframe is done heuristically. Similarly to ORB-SLAM [151] and DSO [69], we use the optical flow of tracked points,

which is the distance in pixels between their locations in the reference and in the tracked images, to make that decision. For convergence reasons, we established that the reference and the tracked frames should not be too different from each other. We therefore use a mean threshold distance of one pixel at the lowest image resolution.

7.1.5 Limits of the Implementation

To our knowledge, VORS is the first Rust-only complete direct visual odometry stack. We thus value simplicity for this important milestone. Incidentally, our algorithm lacks a few significant features, left as later improvements.

One notable missing component is robustness to outliers. It is common that the photoconsistency assumption does not hold with real-world images. Among the many possible reasons, two of them are the presence of dynamic moving objects, and the appearance of bright spots due to light reflections. One possible solution is to use a robust M-estimator instead of a least square estimation. The energy to minimize then takes the form

$$\sum_{\boldsymbol{x}} \rho(I(W(\boldsymbol{x})) - I^*(\boldsymbol{x}))$$

where ρ has a few interesting properties. In part 2 of their series on direct image alignment [15], Baker et al. explain in details how a robust M-estimator can be used within an inverse compositional iteratively reweighted least squares (IRLS) algorithm. Possible estimators include Huber, Geman-McClure, Tukey or even t-distribution estimators. Klose et al. [121] use Huber and Tukey M-estimators, while for example, Kerl et al [119] and Gutierrez et al. [95] prefer the t-distribution.

Another commonly appearing problem with cameras is automatic exposure variations. With changes in lighting conditions, exposure parameters of cameras are often automatically adjusted, resulting in global photoconsistency errors. Different affine exposure parameterizations have successfully been integrated in the expression to minimize, such as in [121] and [69]. We did not add such additional parameters in our modelization and thus expect VORS to have difficulties tracking the camera motion in scenes with brightness changes.

7.2 RGB-D Visual Odometry Evaluation

As previously explained, our implementation belongs to the family of direct visual odometry from RGB-D images. In this section, we will detail how it has been evaluated against comparable algorithms by introducing the available datasets, presenting the different evaluation metrics, and finally lay out the testing setup we provide with the evaluation results.

7.2.1 Dataset Creation / Acquisition

Comparing different algorithms is a complex task for many reasons, one of them being the ability to run those algorithms on the same set of data. The existence of well built reference datasets is thus a crucial point, and understanding their characteristics is valuable to correctly compare and interpret evaluation results.

Overview of Available Datasets

We saw in Chapter 1 that the principal difficulty for building annotation datasets is the required human annotation time. For visual odometry, algorithms are strongly related to capture devices, be it stereo, mono, RGB-D cameras, or cameras paired with other sensors such as inertial measurement units (IMU), GPS or lidar. As a consequence, different datasets focus on different acquisition systems. For every evaluated acquisition system, there must exist another measurement system, more precise and reliable than the one being evaluated. The main difference is thus that the challenge is technological for visual odometry, while it is mostly time consumption for image annotation.

The availability of datasets for visual odometry first came from the mobile robotics community, mostly interested in SLAM from laser sensors (lidar). The data is thus collected from sensors attached to a mobile robot or a car. In the New College [198] and NCLT [35] datasets, the robotic platform is based on a Segway, the KITTI dataset [84] recorded data from a sensors equipped car, while the EuRoC MAV [30] dataset is based on a micro aerial vehicle (MAV). These platforms are depicted in Figure 7.6. The ground truth was recorded with three different approaches for these datasets. Visual odometry ground truth was an after thought for the New College dataset, only available a year later on their website and not discussed in the paper. It appears to have been obtained from dead-reckoning, i.e. from wheel and IMU odometry, provided by the Segway platform, and is thus not very reliable. In the NCLT dataset, the mobile robot trajectory ground truth is computed from a high precision realtime kinematic GPS (RTK GPS) and a graph SLAM based on lidar measurements. The accuracy of the trajectory ranges from a centimeter to a decimeter approximately for a total travelled distance of roughly 147 km. The ground truth trajectory of the KITTI dataset was also obtained from high precision GPS/IMU sensors and is thus also accurate at the decimeter scale. The setup for the EuRoC MAV dataset is a bit different since the mobile vehicle is flying in indoor environment and its trajectory is obtained from motion capture devices. The total travelled distance is thus way shorter, less than a kilometer, but the trajectories are accurate at approximately a millimeter.

A second wave of datasets, represented in bold in Table 7.1, has especially been targetting RGB-D cameras, becoming popular after the launch of the Microsoft Kinect. While the TUM RGB-D [203] and Bonn RGB-D [164] datasets are recorded with real RGB-D sensors, the ICL-NUIM [99] dataset was generated (ray-traced rendered) from synthetic 3D models of indoor scenes. We are going to explain in more details the specificities of the TUM RGB-D

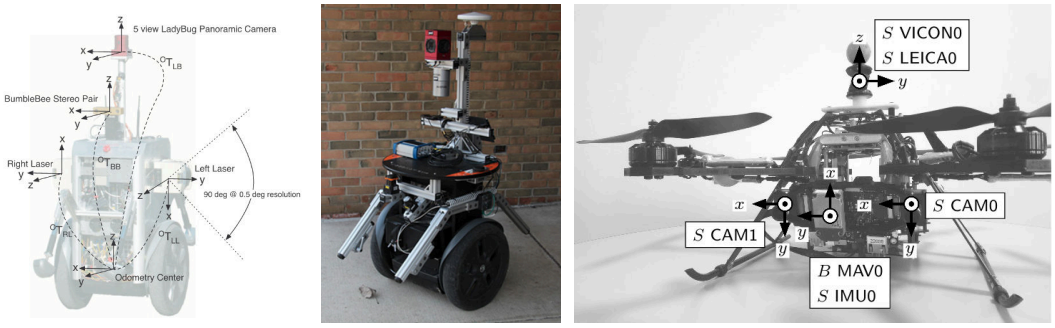


Figure 7.6: Mobile robots used for SLAM datasets. From left to right, the Segway platform used in the New College dataset [198], the Segway platform used in the NCLT dataset [35], the MAV used in the EuRoC MAV dataset [30].

and ICL-NUIM datasets since these are the two we used to evaluate our direct RGB-D visual odometry algorithm.

Finally, with the popularity of inertial sensors coupled with cameras in modern smartphones enabling new augmented reality functionalities, a regain in interest has been visible for 6 degrees of freedom visual inertial odometry (VIO). While the EuRoC MAV [30] and TUM VI [192] datasets are using high quality sensors, the ADVIO dataset [53] is actually using regular smartphones sensors, showing the importance of datasets with less precise data to improve algorithms robustness. We provide a brief summary of these datasets properties in Table 7.1. Note that this is not an exhaustive list of available datasets but an overview of the main ones for visual odometry.

Dataset	Year	Available data	Ground truth
New College [198]	2009	GPS, IMU, wheel odometry, lidar, omnidirectional, stereo	Dead-reckoned
TUM RGB-D [203]	2012	IMU, **RGB-D**	Motion capture
KITTI [84]	2013	GPS, IMU, lidar, stereo	High precision GPS/IMU
ICL-NUIM [99]	2014	**RGB-D**, 3D surface	Synthetic
NCLT [35]	2016	GPS, IMU, wheel odometry, lidar, omnidirectional	RTK GPS and lidar SLAM
EuRoC MAV [30]	2016	Stereo camera	Motion capture
TUM Mono [70]	2016	Mono camera	Closed loop
TUM VI [192]	2018	IMU, Mono camera	Motion capture and closed loop
ADVIO [53]	2018	Smartphone IMU and video	IMU with position fixes
Bonn RGB-D [164]	2019	IMU, **RGB-D**, lidar	Motion capture

Table 7.1: Visual odometry datasets

Motion Capture for RGB-D Visual Odometry

The TUM RGB-D dataset [203] was the first complete and rigorously detailed dataset for RGB-D visual odometry. It is composed of 80 sequences, 47 for training with ground truth,

and 33 for validation only evaluated online, without ground truth. The training sequences are arranged in six groups,

- testing and debugging (4 sequences), simple translation or rotation movements,
- handheld movements (11 sequences),
- robot movements (4 sequences),
- structure and texture (8 sequences), with difficult structure or texture patterns,
- dynamic objects (9 sequences),
- and object reconstruction (11 sequences).

As we can see, the dataset provides both easy sequences and sequences with more difficult situations such as dynamic movements or poor textures in the field of view of the camera. It is thus a good benchmark of the robustness of visual odometry algorithms. These sequences have been acquired by three different Kinect devices, named fr1 (for "Freiburg 1"), fr2 and fr3. All their intrinsic parameters are available in the dataset. The required robustness of the tested algorithms is also increased by the fact that the color camera of the Kinect sensor has a rolling shutter. We do not expect our algorithm to perform very well under those circumstances.

The ground truth camera poses are obtained thanks to an external motion capture system based on MotionAnalysis [149] hardware and software. This setup is composed of eight 300 Hz Raptor-E high definition cameras, equipped with infrared lights to illuminate passive markers attached to the Kinect. After a detailed intrinsic and extrinsic parameters calibration of the system, the authors estimate the relative position errors to be lower than 1 mm and 0.5 degrees.

Synthetic Dataset Creation

Most of the available visual odometry datasets lack a dense 3D surface ground truth to be able to evaluate both the camera trajectory and the structure reconstruction. To this end, Handa et al. created the ICL-NUIM dataset [99]. Contrary to most other datasets, the video sequences provided here are completely generated by computer graphics, using the open-source ray tracing algorithm POV-Ray [174]. The dataset is split into two rooms, a living room and an office, and two scenarios, one noiseless and one with simulated noise. The full pipeline is also provided as open source, if one wishes to customize parts of it. The geometry and some renders of the living room are displayed in Figure 7.7.

In order to test some visual odometry algorithms, we are only going to use the eight noise-free sequences since realistic noisy sequences are already evaluated with the TUM RGB-D dataset. The trajectory ground truth generated by this ICL-NUIM dataset is thus error-free. The color and depth images are also perfectly aligned, timely synchronized and have no light exposure variations. They still contain reflective surfaces and few other ligthing effects not taken into account in our direct visual odometry modelization, but the overall circumstances should be very favorable for our simple implementation.

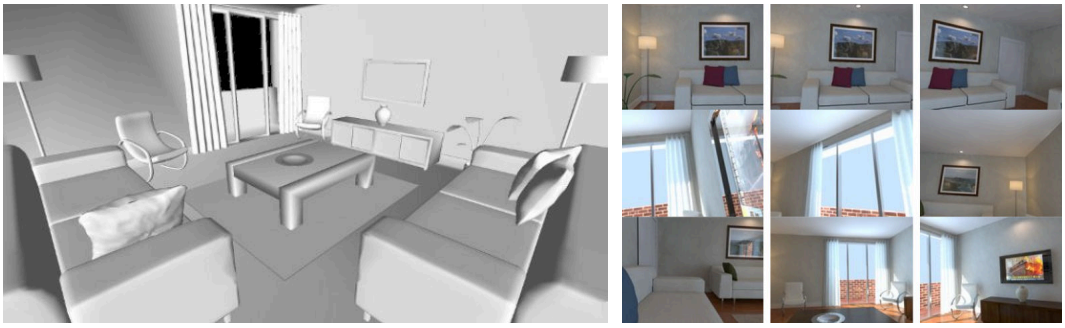


Figure 7.7: Geometry (left) and few rendered images (right) of the living room used for the ICL-NUIM dataset [99].

7.2.2 Evaluation Metrics

There are basically two types of evaluation metrics, those based on a ground truth, and those that are not.

Metrics with Ground Truth, ATE and RPE

The most straightforward method to evaluate a visual odometry trajectory is to compare it with a reference trajectory, called ground truth. This reference trajectory needs to be acquired with more precision than the one being evaluated for the measure to make sense. In our case, the TUM RGB-D dataset uses a motion capture system with sub-millimeter accuracy, and the ICL-NUIM provides exact trajectories since it is a synthetic dataset.

The first evaluation metric commonly used is the absolute trajectory error (ATE). We can consider the trajectory as a discrete set of camera poses

$$G_\Omega = \{\, g_\tau \,|\, \tau \in \Omega,\ g_\tau \in SE(3) \,\}$$

where Ω is the set of discrete time events when camera poses are available. In theory, the ATE can be computed as

$$\text{ATE} = \sum_{\Omega} d(g_\tau, \widehat{g}_\tau)$$

where $d(g_\tau, \widehat{g}_\tau)$ can be thought of as a distance between the estimated transformation g_τ and the ground truth transformation $\widehat{g}_\tau$. Usually, we can consider two types of errors, the translation error

$$\|\text{trans}(\widehat{g}_\tau^{-1} \cdot g_\tau)\|$$

and the rotation error

$$\measuredangle\,(\widehat{g}_\tau^{-1} \cdot g_\tau)$$

where $\measuredangle$ is the amplitude (≥ 0) of the rotation angle. Since the rotation errors will also im-

pact translation errors later, it is common to only use the translation error when computing the ATE. The two most widely used ATE metrics are the RMSE and the median scores

$$\text{ATE}_{\text{rmse}} = \left(\overline{\|\text{trans}(\widehat{g}_\tau^{-1} \cdot g_\tau)\|^2}\right)^{1/2} \quad \text{and} \quad \text{ATE}_{\text{median}} = \text{median}\left(\|\text{trans}(\widehat{g}_\tau^{-1} \cdot g_\tau)\|\right).$$

The median is a better indicator of the algorithm precision, while the RMSE better reflects the presence (or absence) of outliers, i.e. the global robustness of the algorithm.

In practice, we should note that the ground truth and estimated trajectory are not expressed in the same reference frame. It is thus necessary to first align the two trajectories. This is usually done with a principal component analysis (PCA) of the trajectories. It is also important to note that the ground truth and estimated trajectories are not sampled at the same timings and frequency. Therefore, it is also necessary to correctly associate poses of each trajectory, which is reasonably easy when timestamps have been synchronized in the dataset. One of the main issues of the ATE is that drifts have higher impact at the beginning of the sequence than at the end. To better evaluate long sequences we thus use another metric, the relative pose error (RPE).

Contrary to the ATE, which evaluates absolute errors on associated pairs of poses, the RPE compares the relative motion in a sliding window along the sequence. The size of the window is usually a fixed travelled distance or time interval. We will detail for example the computation of the RPE at 1 second. For each associated pair of estimated and ground truth poses $(g_\tau, \widehat{g}_\tau)$, we consider another pair $(g_{\tau'}, \widehat{g}_{\tau'})$ delayed by 1 second ($\tau' \approx \tau + 1s$). The camera motion between τ and τ' in the estimated trajectory is $\Delta_{\tau\tau'} g = g_\tau^{-1} \cdot g_{\tau'}$ and the camera motion in the ground truth trajectory is $\Delta_{\tau\tau'} \widehat{g} = \widehat{g}_\tau^{-1} \cdot \widehat{g}_{\tau'}$. The relative pose error is thus computed as

$$\text{RPE} = \sum_{\Omega'} d(\Delta_{\tau\tau'} g, \Delta_{\tau\tau'} \widehat{g})$$

where $d(\cdot, \cdot)$ is similar than for the ATE, and Ω' is the set of regularly spaced pairs, whether it is a duration, like 1 second, or a distance like 1 meter.

Metrics without Ground Truth

In some conditions, such as long handheld outdoor trajectories, retrieving a ground truth might be problematic. Yet it is still possible to partially evaluate the precision of an algorithm, or rather its robustness to drifts and losses. Indeed, in absence of a full sequence ground truth, it is still possible to evaluate the tracking of loops. It is important to note that global loop closure detections must be deactivated in the algorithms for these measures to make sense.

One method consists in computing loop closure and pose graph optimization and then compare the trajectory before and after. If the drifts of the tracking are not too extreme and do not prevent the pose graph optimization, this error can be representative of the visual

odometry performance.

Another approach consists in specifically designing the dataset sequences to start and end at the same location, such that both ends of the trajectory can be precisely aligned. This is the approach taken in [70]. The error computed is then the accumulated drift between the sequence when aligned to the start and when aligned to the end of the partial ground truth trajectory.

7.2.3 Setup and Algorithms Evaluation

In this section, we describe how we evaluated our VORS implementation and compared it to other open source C++ algorithms. We focused on RGB-D visual odometry, and therefore used both the TUM RGB-D [203] and ICL-NUIM [99] datasets.

The TUM RGB-D Dataset Format

The TUM RGB-D dataset is well specified and as a result, others including ICL-NUIM follow the same format. We therefore also base our evaluation setup on the TUM RGB-D format. The general structure of such dataset is as follows.

```
dataset/
  rgb.txt            # List of rgb images with their timestamps
  depth.txt          # List of depth images with their timestamps
  groundtruth.txt    # List of ground truth camera poses
  associations.txt   # List of associated rgb and depth images with timestamps
  accelerometer.txt  # List of accelerometer measurements with timestamps
  rgb/               # Directory containing all color images
    timestamp_rgb_0.png
    timestamp_rgb_1.png
    ...
  depth/             # Directory containing all depth images
    timestamp_depth_0.png
    timestamp_depth_1.png
    ...
```

Listing 7.1: General structure of the TUM RGB-D dataset format.

The files `rgb.txt` and `depth.txt` list all images with their associated timestamps. The RGB-D visual odometry algorithms also need the correspondences between color and depth images, so if not present, the first step is usually to run a provided `associate.py` script to generate an `associations.txt` file from the `rgb.txt` and `depth.txt` files. Each line of that file contains a pair of RGB and depth images with their respective timestamps. Color images are stored as 640x480 8-bit RGB images in the PNG format. Depth images are stored as 640x480 16-bit monochrome images in the PNG format. Depth images are scaled by a factor of 5000, i.e. a pixel value of 5000 corresponds to a distance of 1 meter. Theoretically

depth images thus have a precision of 0.2 mm for a range of 0 to roughly 13 meters. The ground truth trajectory is provided in the `groundtruth.txt` file as follows.

```
# timestamp tx ty tz qx qy qz qw
1305031449.7996 1.2334 -0.0113 1.6941 0.7907 0.4393 -0.1770 -0.3879
1305031449.8096 1.2334 -0.0111 1.6939 0.7911 0.4393 -0.1768 -0.3872
...
```

Listing 7.2: Ground truth trajectory format.

The timestamp is the POSIX time of the given pose, i.e. the number of seconds elapsed since 1970, January 1st. The coefficients `tx`, `ty`, `tz`, are the coordinates of the optical center of the color camera with respect to a given world frame. The coefficients `qx`, `qy`, `qz`, `qw` are the parameters of the quaternion describing the orientation of the color camera. The last coefficient `qw` is the real part of the quaternion.

Open Source Algorithms and Provided Setup

In addition to VORS, we evaluated five other open source algorithms for RGB-D visual odometry, namely DVO [119], Fovis [108], and three variants of the OpenCV RGB-D odometry module based on [152, 201]. For each one of those six algorithms, we implemented a small program performing the following operations (pseudo code).

```
# Usage: track camera_id path/to/associations.txt
# where camera_id is one of [icl|fr1|fr2|fr3]

intrinsic_matrix = create_camera(camera_id)
tracker = initialize_tracker(intrinsic_matrix)
for (rgb_image, depth_image) in extract_images(association_file):
    tracker.track(rgb_image, depth_image)
    print(tracker.camera_pose)
```

Listing 7.3: Outline of the common RGB-D odometry program implemented for every tested algorithm.

All those algorithms except VORS are C++ programs with different complexity of build, due to dependency issues. DVO for example would not compile anymore with recent versions of the Sophus [202] library for Lie algebra due to missing re-orthogonalization of the rotation matrix after optimization increments. Therefore, we are providing clear installation instructions as well as a Docker container ready for compilation of all 6 programs. This setup is open sourced under GPL license at `https://github.com/mpizenberg/rgbd-tracking-evaluation`.

Evaluation Results

We evaluated all algorithms both on the ICL-NUIM dataset and on the TUM RGB-D dataset, constituting a total of 55 sequences. As visible in Figure 7.8, our implementation

(VORS) performs similarly than DVO, Fovis and OCV-RGBD. It is coherent since these four algorithms use both the visual information of the color image and the depth map to estimate the camera motion. OCV-ICP however, which only uses the depth map, is unable to track the camera movements in the majority of sequences. The OCV-RGBD-ICP variant, which is a mixed approach minimizing both energies of OCV-RGBD and OCV-ICP, inherits from the same tracking difficulties as OCV-ICP.

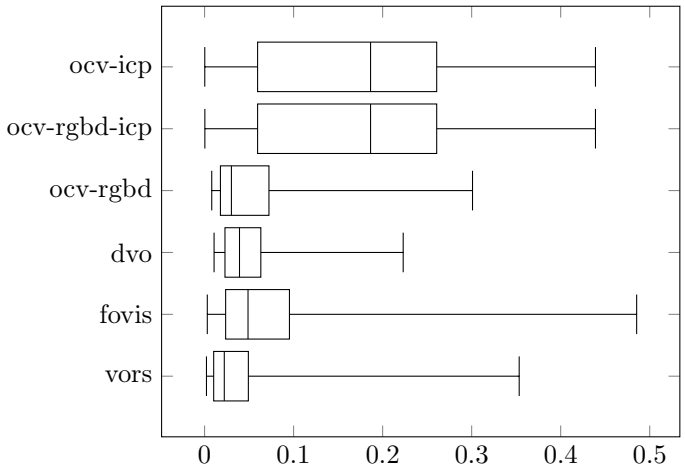


Figure 7.8: Distribution of the median relative pose error (RPE) at 1 second on all 55 sequences, composed of 47 TUM sequences and 8 ICL-NUIM sequences.

Out of 55 sequences taken into account in the Figure 7.8, the wide majority (47) comes from the TUM RGB-D dataset. VORS, which does not implement robust approaches, should perform better for the synthetic sequences of the ICL-NUIM dataset than for the real sequences of the TUM dataset. Figure 7.9 thus presents the same plot than Figure 7.8 but restricted to the 8 ICL-NUIM sequences. As we can see, VORS is performing remarkably in these conditions. Other noteworthy details are exacerbated by this plot. Both geometric odometries (OCV-ICP and OCV-RGBD-ICP) perform extremely well thanks to the high precision dense depth maps provided by the dataset. One should note that this precision comes at the price of time. The first ICL sequence for example takes three times longer with OCV-ICP than with Fovis, VORS and DVO, all comparable in speed. It is also notable that Fovis is having more issues tracking those sequences. It can be explained by the nature of this algorithm, which is indirect, based on FAST features, contrary to VORS, DVO and OCV-RGBD which are direct visual odometry algorithms. The synthetic nature of the images, and lack of unique textures compared to real images is degrading the detection rate of Fovis FAST descriptors.

Instead of using the median RPE, which pictures the overall precision of the algorithms, Figure 7.10 represents the RMSE RPE, which better reflects the robustness to outliers. As visible in that figure, VORS has a long tail of highly erroneous motion tracking. High RMSE

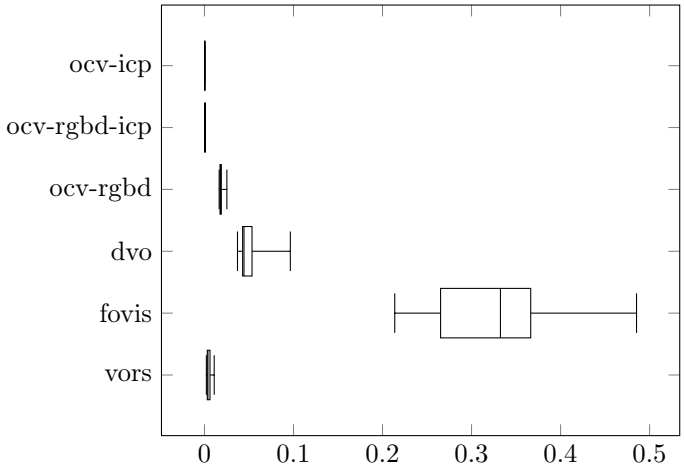


Figure 7.9: Distribution of the median relative pose error (RPE) at 1 second on the 8 ICL-NUIM sequences.

errors are better understood when looking at the detailed RPE of one sequence. Figure 7.11 contains plots of the RPE along the third sequence of the ICL-NUIM dataset. As visible there, VORS exhibits lower errors in general, but the absence of drift detection or robustness may generate much bigger drifts such as those around frame 600 of the sequence. In the end, even though our visual odometry implementation in Rust (VORS) has robustness issues due to previously discussed challenges, we provide a precise, state of the art RGB-D visual odometry, easy to compile, to use and as we will develop next, easy to port to WebAssembly. VORS thus enables the exploration of efficient interactive visual odometry on the Web, which we finally present next.

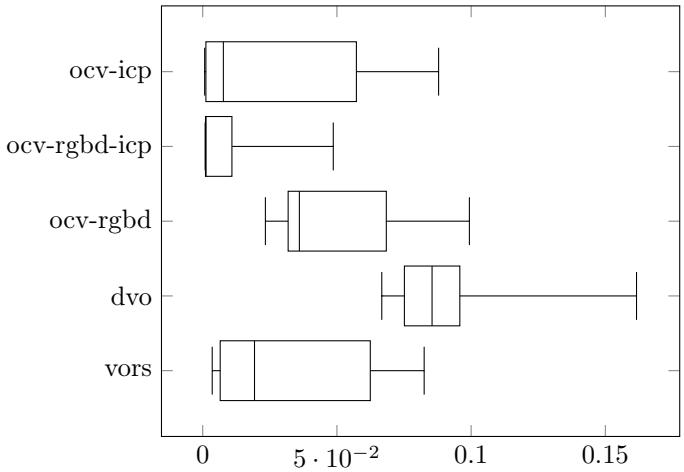


Figure 7.10: Distribution of the RMSE relative pose error (RPE) at 1 second on the 8 ICL-NUIM sequences.

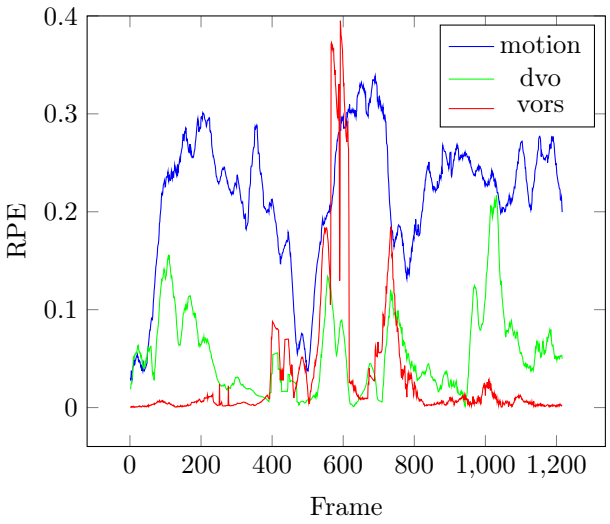


Figure 7.11: Relative pose error (RPE) at 1 second for the third living room sequence of the ICL-NUIM dataset.

7.3 Interactive VORS on the Web

We previously quantitatively validated VORS viability for the visual odometry task. In this final section we detail how it can be made usable directly on the Web. We also attempt at showcasing the potential of such exposure with an example use case of manual human intervention to detect and rectify drifts.

7.3.1 Port of VORS to WebAssembly

VORS in WebAssembly

Globally, the port of VORS to WebAssembly was straightforward, as expected for a Rust-only project. VORS code base is organized as a library providing a rich API, accompanied by a small binary program performing the actual visual odometry on a dataset provided as a command line argument. Porting VORS is thus a matter of two tasks, (1) being able to compile the library to the WebAssembly target, (2) rewrite a small WebAssembly program replacing the command line program in the context of a Web browser. The first task is immediately validated by running `cargo build --target wasm32-unknown-unknown`, confirming the ability to compile VORS to WebAssembly. The second task requires more work, due to the different natures of native command line applications and Web browsers.

Loading Data as a Tar Archive

Native programs have the ability to interact with the underlying operating system. Web applications however are sandboxed, for security reasons reminded in Chapter 6 on performant Web applications. The seemingly simple task of loading images for the dataset directory thus becomes impossible. The simplest alternative, which is the one we chose, consists in loading a `tar` archive of the entire dataset through the file upload mechanism, making the archive content available in the browser memory. This has two constraints, one on memory and the other on the application program.

Current Web APIs prevent us from loading the archive directly on the WebAssembly memory. As a consequence, the archive is loaded inside the application JavaScript memory, and we then duplicate it in the WebAssembly module memory. For an unknown period of time, until the browser decides to release the JavaScript buffer, the application will consume a memory of double the archive size. The full first sequence of the ICL-NUIM dataset for example weighs 800 MB, resulting in 1.6 GB of allocated memory by the application in the browser.

The other change has to occur in the program code, now reading files from an archive instead of from the file system. For this task, we used the `tar-rs` library [54], which brought the first hurdle by not compiling to WebAssembly due to the presence of OS-specific code for file time management. Fortunately, simply adding compiler guards in that library to

deactivate the incriminating functions from the public API when compiling to WebAssembly was sufficient.

PNG Image Reading

The next technical challenge was a performance one. We managed to get VORS compiling and running as a WebAssembly module, but the performances dropped drastically, from 40 frames per second (fps) in the native case to less than 10 fps in WebAssembly. After some performance analysis, we identified the culprit as the PNG image decoding of frames. Figure 7.12 spotlights that the majority of the time is spent decoding both images instead of running the visual odometry algorithm.

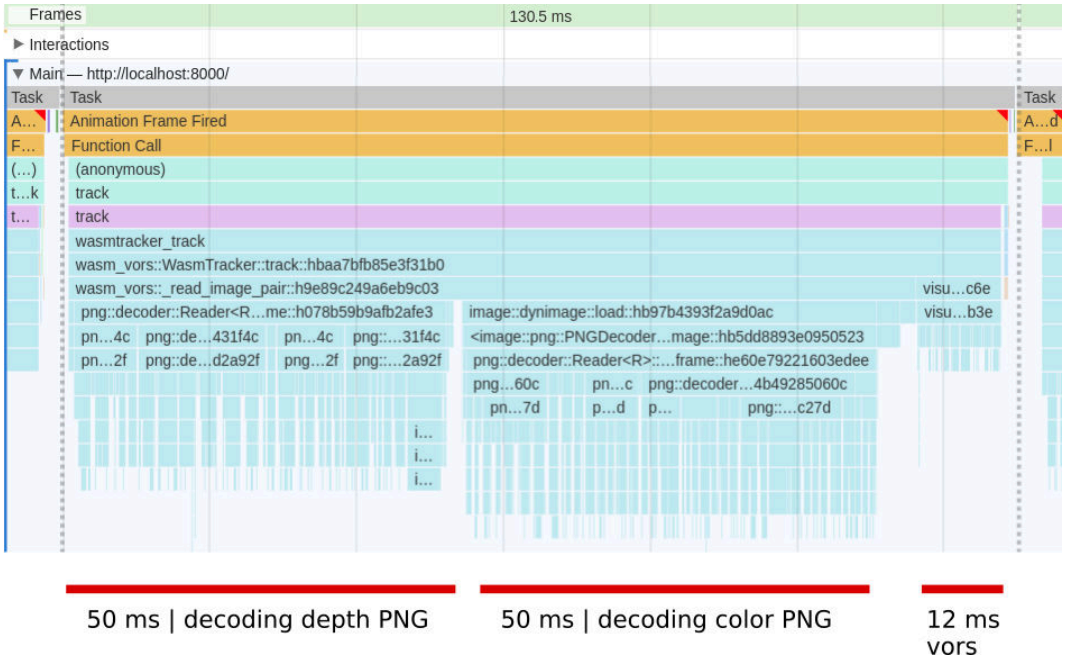


Figure 7.12: Flame graph of VORS WebAssembly performances with the PNG Rust crate. The majority of the time is spent decoding RGB-D images.

Even though it may appear anecdotal, the usability of realtime interactive applications is hindered by low refresh rates so it felt important to improve on the default PNG decoder available in Rust. Since OpenCV decodes images much faster, and Rust should have similar performances than C++, we decided to implement a new Rust PNG decoder. Our implementation keeps in mind that it should be "Wasm-friendly", meaning we limit the amount of memory allocations, which are more costly in WebAssembly.

The PNG specification is available online. Under the hood, the data is compressed with the deflate algorithm, whose specification is also available online. PNG is a simple structured format. A file contains successive data blocks called chunks of different types.

The most interesting types are IHDR (header), IDAT (data) and IEND (end of file). The body of the image is composed by successive IDAT blocks containing transformed lines of the image (called "filtered") to reduce entropy, then compressed with the deflate algorithm. We have implemented the parsing with a fast parser combinator library, and the unfiltering is rather simple. We did not reimplement the inflate algorithm since there are already high performance Rust-only libraries for this task. After a few round of optimizations, we managed to get very good decoding performances, especially in WebAssembly compared to the library we used initially. Table 7.2 summarizes performances on few images we used locally and clearly shows the improvement from the default Rust PNG library. OpenCV performances are included for comparison. As a result, we managed to get VORS compiling and running at similar speed in the browser. After this first milestone, the next step is handling visualization of visual odometry data such as trajectories and 3D point clouds. All this must be tightly integrated in a standard Web application.

Image	us (native)	default (native)	OpenCV	us (wasm)	default (wasm)
depth.png	4.0 ms	9.1 ms	4.0 ms	5.5 ms	30.7 ms
rgb.png	6.6 ms	16.0 ms	6.5 ms	13.7 ms	52.1 ms

Table 7.2: Decoding performances of our PNG decoder.

7.3.2 Interactive VORS Web Interface

Overview of the Interface

Interactive VORS Web interface is visible in Figure 7.13 and is composed of three parts,

- the control bar at the bottom, with a timeline and some buttons,
- the point cloud 3D visualization in the center,
- and the image thumbnail of the current keyframe on the left.

Contrary to structure from motion where images are not guaranted to be in any specific order, visual odometry has the advantage of dealing with video data. The timeline thus enables movement along the temporal axis. It is a one-dimensional control that is well known thanks to its pervasive usage for videos. The frames accessible from the timeline are restricted to the keyframes of the visual odometry. As explained in Section 7.1.4, they are the frames used as reference for the direct image alignment. A thumbnail of the current keyframe is located at the top left corner of the Web interface. For each keyframe, we use the depth information of the sparse candidate points to retrieve their 3D coordinates. The number of candidate points is variable but in the order of a thousand to ten thousands of points per keyframe. Those 3D points are stored in an array buffer, then used for a point cloud visualization, located at the center of the interface. The sparse candidate points corresponding to the current keyframe are visualized in red, to ease the mental association

between the keyframe thumbnail and the corresponding location in the complete point cloud. The reader may notice in Figure 7.13 that the point cloud and the keyframe thumbnail are mirrored. This is due to the ray tracer employed in the ICL-NUIM dataset, which uses a left-handed coordinate system. Finally, the camera trajectory is visible in purple in the point cloud visualization.

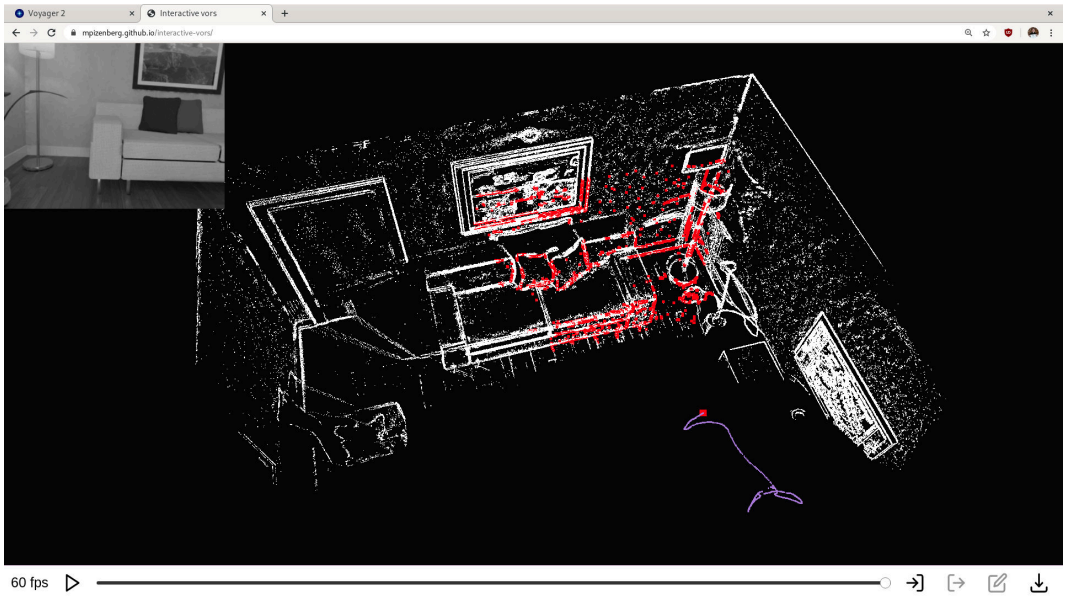


Figure 7.13: Interactive VORS Web Interface.

Visualization Challenges

Is Rust mature enough to write the visualization code in Rust native, and compile it to WebAssembly? Even though the field is exciting and buzzing, the short answer is no. The Rust ecosystem is still lacking in the domain of graphical user interfaces (GUI). As of 2019, only a handful of libraries enable 3D graphics, often as bindings to other C++ libraries.

Currently, the four main APIs for graphics are OpenGL, Vulkan, DirectX and Metal. OpenGL is an open standard developed by the Khronos group since 1992. It is easy and high level, but becoming obsolete when efficient use of current graphics card hardware is needed. DirectX is Microsoft alternative to OpenGL. Up until DirectX 11, it was also a high level API, but starting from DirectX 12, became lower level and more efficient. Vulkan and Metal are also part of this renewal of graphics APIs targetting more efficient usage of the hardware architecture. Vulkan is open and developed by Khronos while Metal is Apple property.

Just as WebGL was introduced as a Web version of OpenGL, a new standard is rising from the Vulkan API, called WebGPU. Unfortunately, as of August 2019, WebGPU is only

supported on Chrome canary for OSX. The only viable option for our visualization is thus the OpenGL/WebGL duo. After trying few Rust libraries, the conclusion is that currently, the only way of reaching decent performances with WebGL for large point clouds is to directly use a WebGL framework and not an automatic conversion from a Rust native OpenGL code. As a consequence, our point cloud visualization is based on ThreeJS [33], a JavaScript 3D library. The points buffers used for the visualizations are efficiently referenced directly from the WebAssembly memory. This mutable memory manipulation has been challenging and error-prone but we eventually managed to get the visualizations working correctly.

7.3.3 Human in the Loop Closure

As discussed in Section 7.2.3, VORS is a precise visual odometry algorithm, prone to punctual large drifts. The resulting point cloud visualizations contain parts that seem to be duplicated, as in Figure 7.14. This kind of observations is common in the presence of loops in the sequence. In a SLAM context, presented in Section 5.3.2, this type of drifts may be corrected by loop closure detection and pose graph optimization. In this section however, we define an interactive correction procedure that puts humans in the loop, instead of automatic loop closure.

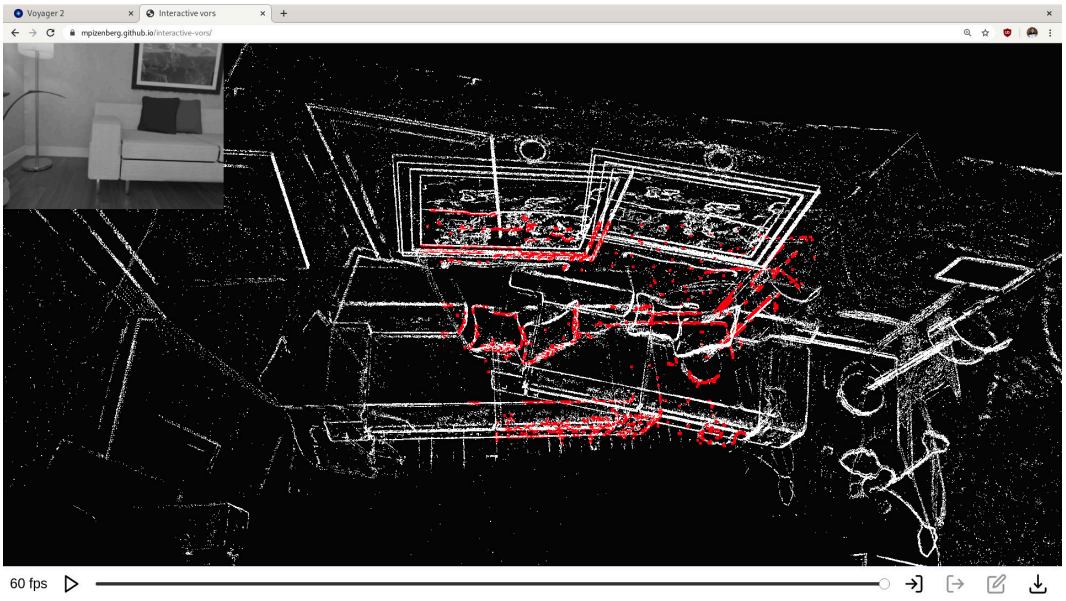


Figure 7.14: Duplication of the point cloud due to drifts.

Automatic Registration

In traditional indirect SLAM algorithms, points of interests are retrieved for all keyframes and their descriptors are classified with techniques such as “bag of words”. Typical indexing

and search algorithms are then applied to identify similar keyframes. In our case, the identification is performed by a human interaction, clicking on the buttons in the toolbar for the reference keyframe and the one that needs re-ajusting. Once two keyframes are selected, the next step consists in computing the camera motion between those.

Usually, one would match all keypoints in the pair of images and use a robust version of the 8-point algorithm if there is no depth information, or a robust PnP algorithm if depth information is known. We thus tried to perform keypoints detection and matching for the pair of keyframes. There exists many keypoint descriptors for this task. Some well known are ORB, SIFT, FAST and A-KAZE, with a Rust implementation of A-KAZE features already available. Unfortunately, due to the low textured images of the synthetic dataset, the number of matches for selected pairs of keyframes are in the order of 20, with multiple wrong matches, leading to unreliable camera motion. It is the same issue that the one degrading Fovis performances in the ICL-NUIM dataset.

The second approach consists in computing direct image alignment from the reference frame. Unfortunately, the keyframes matched by the user are not always similar enough for the direct image alignment to converge to the correct motion. This leads us to another manual human intervention.

Manual P3P Intervention

As presented in Section 5.4.1 on feature-based motion estimation, P3P is a minimal PnP ("perspective-n-points") solver. It enables motion estimation from a set of three points with depth information in the reference frame and three associated points in the other frame. We therefore use the keyframe thumbnails for an annotator to click on three corresponding points, as visualized in Figure 7.15.



Figure 7.15: Clicking interaction on keyframes thumbnails to perform P3P.

The P3P algorithm may return 0 to 4 potential solutions. Usually, those are disambiguated by looking at additional associated points. In our case, we decided to let the

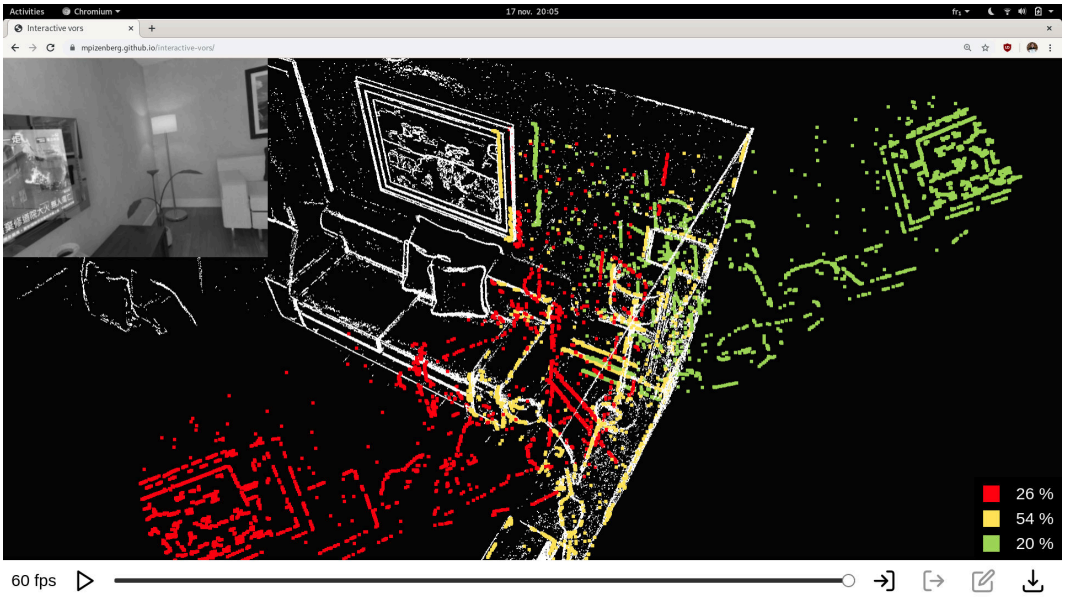


Figure 7.16: Visualization of potential poses for the second keyframe. The misaligned attempt at direct image alignment is presented in red. Two potential P3P poses are colored in yellow and green. An estimation of the probability of each pose to be correct is provided in the lower right corner of the visualization canvas.

human user pick the correct option. Each potential solution is displayed with a different color. The photometric reprojection error of the lowest image resolutions of the pyramid is computed to estimate the probability of each camera position, displayed in the lower right corner of the interface. The user has to click on the correct option to validate it. Figure 7.16 depicts an example situation where two potential P3P poses are proposed in addition to a misaligned initial proposition in red. Once a new camera pose is chosen for the second keyframe, the visual odometry may be restarted for all subsequent frames. Figures 7.17 and 7.18, and Table 7.3 provide qualitative and quantitative results showing the improvements provided by the manual human intervention. Our Rust implementation of P3P is based on Persson and Nordberg's solver [169] and published as an open source library under the Rust Computer Vision (rust-cv) Github organization [171].

measure	before manual intervention	after manual intervention
ATE rmse	0.125126 m	0.100411 m (-20%)
ATE mean	0.113021 m	0.086285 m (-24%)
ATE median	0.102886 m	0.076897 m (-25%)
ATE std	0.053692 m	0.051354 m (-4%)
ATE min	0.049635 m	0.019642 m (-60%)
ATE max	0.267245 m	0.195351 m (-27%)

Table 7.3: ATE before and after manual intervention on the first ICL-NUIM sequence.

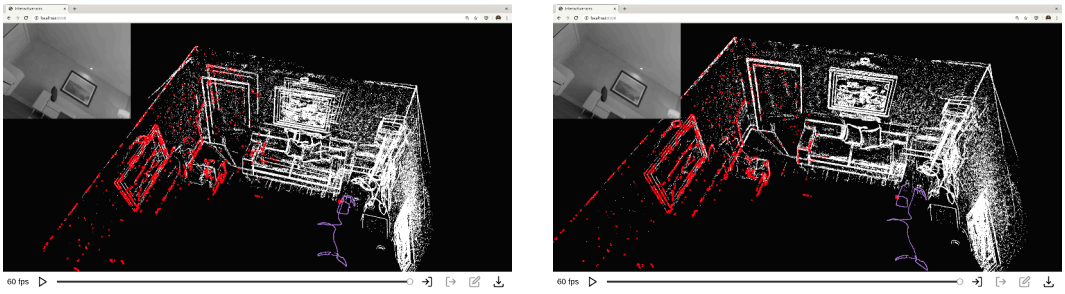


Figure 7.17: Point cloud before (left) and after (right) manual intervention.

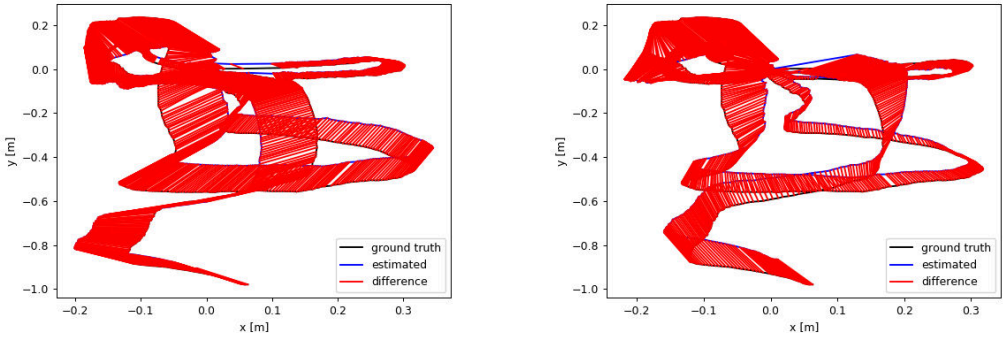


Figure 7.18: Absolute trajectory error (ATE) before (left) and after (right) manual intervention.

7.4 Conclusion

In this chapter, we presented VORS, the first complete direct visual odometry algorithm in Rust. We have paid particular attention to simplicity, and reduction of parameters in all the steps of the algorithm, such as for the sparse candidate points selection. Then, by comparing its precision to other available algorithms, we demonstrated that VORS achieves state of the art odometry precision on RGB-D datasets, especially for synthetic data, presenting less outliers to our modelization. Finally, we presented "Interactive VORS", our interactive Web application, based on the compilation of VORS to WebAssembly. It provides easy access to visual odometry, directly in a browser, with interaction in multiple modalities. A one-dimensional navigation along the temporal axis is provided by a timeline while three-dimensional space navigation is possible with the point cloud visualization. We implemented correction of drifts in conditions where loops are present in the sequence. In general, our goal was to showcase the potential of interactive Web applications to better understand the huge amount of data generated by visual odometry, and eventually offer methods to correct automatic results with guided human intervention.

Conclusion

At the beginning of this journey, we asked ourselves how user interactions and the Web can improve computer vision research. Throughout this work, we showcased different approaches to combine those capabilities within interactive computer vision Web applications.

In the case of image annotation, we provided an adaptation of the GrabCut algorithm combined with medial axis transforms to retrieve precise segmentations from outlining. This interaction has been integrated into a reliable Web application to crowdsource segmentation annotations of the PASCAL-VOC dataset. The clear benefit illustrated here is the scaling and the reach that the Web offers for the creation of learning datasets.

Visual odometry is a computationally intensive task. Due to previous limitations of the Web platform, running those applications directly in a browser was not sufficiently efficient to be seriously considered. The advent of new technologies such as WebAssembly, or WebGL changed this vision completely. Therefore, we started a new visual odometry library in a recent programming language named Rust with excellent support of WebAssembly. We showed how this library can be integrated in a Web application, and profit from its interactive nature to improve the tracking results. This example paves the way for easy access to computer vision algorithms, both for research reproducibility and availability to a non technical public such as creators.

We described how image segmentation and visual odometry can take advantage of user interactions when made accessible through the Web. To this end, we provided concrete explanations on how to build reliable Web applications, and how to port computationally intensive tasks to run the browser. Generally, this approach has the potential to improve machine learning and other computer vision fields thanks to exposure to a wider, more varied set of data. Yet I don't see this practice being picked up immediately. In fact, mainstream solutions for computer vision have a strong inertia due to the massive amount of already available libraries in C++ and Python. It will thus take time before performant languages better suited for the Web like Rust obtain fair usage share and push more research to public exposure through the Web. Our specific contribution in image segmentation also have limitations. The segmentations obtained with single outlines for example do not always provide perfect masks. One could then reasonably question the viability of that interaction to create learning datasets.

Future work could address that point about the loss of precision when using outlines or other imperfect interactive segmentation methods. The aim would be to find a balance point between annotation speed and prediction quality. In particular, we would like to study how loss in precision for the training data translates into loss of quality for machine learning algorithms, and how regularization techniques can minimize the effect of noisy training data. Regarding the visual odometry library, there are many improvements that we could work on, both on the research and the platform sides. One future goal is to be able to run reliably and smoothly on mobile devices. Smartphones typically run on ARM processors, a target supported natively by the Rust toolchain, and also through WebAssembly runtimes so there should not be huge platform hurdles. They are often equipped with more sensors that dedicated cameras such as inertial measurement units (IMU). Integrated smartphones cameras however, usually feature rolling shutter sensors, which change the image formation modelization. Among the many possible research extensions, the priorities are the adaptation to the RGB case (no depth), photometric variation modelization to account for automatic exposure, modelization of rolling shutter and fusion with IMU data. Another very exciting avenue to explore could be collaborative SLAM through the Web. Since smartphones are now connected to high speed 4G and 5G networks, we could imagine situations where groups of individuals all take part in one global map optimization process.

These are exciting times to be alive! Admittedly, we will not radically change every aspect of our lives with better segmentation maps or increased visual odometry precision. But as we mentioned in the introduction, those are key aspects of technological evolutions that could impact our society such as autonomous vehicles. From 1830 to 1890 the distance of railroad in operation in the United States grew from 23 to 166706 miles [60]. With that growth, the number of brakemen deaths (cf Figure 7.19) also radically increased until the introduction of air brakes which replaced their jobs by safer systems. It is a matter of time, continued research and social acquaintance, but autonomous vehicles will eventually provide a similar transformation from manual to automatic systems; and I hope this work will be a positive contribution to a better future.



Figure 7.19: Representation of brakemen, who used to work in all weather conditions. "A Picnic", engraving by Peckwell, published on the cover of The Railroad Conductor, vol. 7, no. 15 (Aug. 1, 1890). Image in public domain.

Appendix A

Gradients of Reprojection Function

A.1 Notation

Let's define a few variables and notations for the rest of the computations. We note

$$K = \begin{pmatrix} f_u & s & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{pmatrix}$$

the matrix of camera intrinsic parameters. Extrinsic camera parameters are expressed as twist coordinates $\boldsymbol{\xi} \in \mathbb{R}^6$, formed by stacking the linear velocity $\boldsymbol{\nu} \in \mathbb{R}^3$ (related to translation), and the angular velocity $\boldsymbol{\omega} \in \mathbb{R}^3$ (related to rotation). Let θ be the norm of $\boldsymbol{\omega}$. We note

$$\boldsymbol{\xi} = \begin{pmatrix} \boldsymbol{\nu} \\ \boldsymbol{\omega} \end{pmatrix}, \quad \boldsymbol{\nu} = \begin{pmatrix} \nu_1 \\ \nu_2 \\ \nu_3 \end{pmatrix}, \quad \boldsymbol{\omega} = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} \quad \text{and} \quad \theta = ||\boldsymbol{\omega}||.$$

We will note $\wedge$ the "hat" operator converting from the twist coordinates $\boldsymbol{\xi} \in \mathbb{R}^6$ to the twist $\widehat{\xi} \in \mathfrak{se}(3)$ in the Lie algebra associated with rigid body motions $SE(3)$. Let $\omega_\times \in \mathfrak{so}(3)$ be the element of the Lie algebra associated with the rotation group $SO(3)$. We then have

$$\widehat{\xi} = \begin{pmatrix} \omega_\times & \boldsymbol{\nu} \\ 0 & 0 \end{pmatrix} \quad \text{and} \quad \omega_\times = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix}.$$

The exponential map from the Lie algebra of twists $\mathfrak{se}(3)$ to the Lie group of rigid body motions $SE(3)$ is of the form

$$\exp(\widehat{\xi}) = \begin{pmatrix} \exp(\omega_\times) & V\boldsymbol{\nu} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} R & \boldsymbol{T} \\ 0 & 1 \end{pmatrix}.$$

According to Rodrigues' formula we can detail the expressions of $\exp(\omega_\times)$ and V as in equation A.1 and equation A.2.

$$\exp(\omega_\times) = I + \frac{\sin\theta}{\theta}\omega_\times + \frac{1-\cos\theta}{\theta^2}\omega_\times^2 \tag{A.1}$$

$$V = I + \frac{1-\cos\theta}{\theta^2}\omega_\times + \frac{\theta-\sin\theta}{\theta^3}\omega_\times^2 \tag{A.2}$$

A.2 Reprojection Using Twist Coordinates

The transformation flow of the reprojection of a pixel consists of the following steps: Pixel 1 (2D, $\binom{u_1}{v_1}$) $\mapsto$ Camera 1 (3D, $\boldsymbol{X}_1$) $\mapsto$ Camera 2 (3D, $\boldsymbol{X}_2$) $\mapsto$ Pixel 2 (2D, $\binom{u_2}{v_2}$). Let g be the rigid body motion from camera 1 to camera 2, defined by

$$g\colon \mathbb{R}^3 \to \mathbb{R}^3, \quad \boldsymbol{X} \mapsto R\boldsymbol{X} + \boldsymbol{t}.$$

If we note λ the depth associated with the reprojected pixel point and $\boldsymbol{x_2}$ its homogeneous coordinates, then all the following expressions are equivalent.

$$\boldsymbol{x_2} = \lambda \begin{pmatrix} u_2 \\ v_2 \\ 1 \end{pmatrix}, \tag{A.3a}$$

$$\boldsymbol{x_2} = K\boldsymbol{X_2}, \tag{A.3b}$$

$$\boldsymbol{x_2} = K(R\boldsymbol{X_1} + \boldsymbol{t}), \tag{A.3c}$$

$$\boldsymbol{x_2} = K(\exp(\omega_\times)\boldsymbol{X_1} + V\boldsymbol{\nu}). \tag{A.3d}$$

A.3 Jacobian Expression

We note $J_{\boldsymbol{\xi}}(\boldsymbol{x_2})$ the Jacobian of $\boldsymbol{x_2}$ relative to the twist coordinates $\boldsymbol{\xi}$. Since $\boldsymbol{x_2} \in \mathbb{R}^2$ ($\mathbb{R}^3$ in homogeneous coordinates) and $\boldsymbol{\xi} \in \mathbb{R}^6$, this Jacobian is a 2x6 matrix (3x6 in homogeneous coordinates). We note $\nabla_{\boldsymbol{\xi}}(\boldsymbol{x_2})$ the 3x6 homogeneous version. From (A.3d) we can write

$$\nabla_{\boldsymbol{\xi}}(\boldsymbol{x_2}) = K(\nabla_{\boldsymbol{\xi}}(\exp(\omega_\times)\boldsymbol{X_1}) + \nabla_{\boldsymbol{\xi}}(V\boldsymbol{\nu})). \tag{A.4}$$

Let's split $J_{\boldsymbol{\xi}}(\boldsymbol{x_2})$ into its components relative to $\boldsymbol{\nu}$ and $\boldsymbol{\omega}$, respectively $J_{\boldsymbol{\nu}}(\boldsymbol{x_2})$ and $J_{\boldsymbol{\omega}}(\boldsymbol{x_2})$, which are both 2x3 Jacobian matrices (3x3 in homogeneous coordinates). We note $\nabla_{\boldsymbol{\nu}}(\boldsymbol{x_2})$ and $\nabla_{\boldsymbol{\omega}}(\boldsymbol{x_2})$ their respective homogeneous versions. Since $\exp(\omega_\times)$ and V only depends on $\boldsymbol{\omega}$, equation A.4 leads to

$$\nabla_{\boldsymbol{\nu}}(\boldsymbol{x_2}) = KV. \tag{A.5}$$

For $\nabla_{\boldsymbol{\omega}}(\boldsymbol{x_2})$, we will have to develop its expression so for now we will settle for

$$\nabla_{\boldsymbol{\omega}}(\boldsymbol{x_2}) = K(\nabla_{\boldsymbol{\omega}}(\exp(\omega_\times)\boldsymbol{X_1}) + \nabla_{\boldsymbol{\omega}}(V\boldsymbol{\nu})). \tag{A.6}$$

A.4 Partial Derivatives Relative to Linear Velocity Terms

Let α, β and λ be the three components of $\boldsymbol{x_2}$. From (A.3a) we have $u_2 = \alpha/\lambda$ and $v_2 = \beta/\lambda$. Thus the partial derivatives relative to linear velocity terms are

$$\frac{\partial u_2}{\partial \boldsymbol{\nu}} = \frac{1}{\lambda^2}\left(\frac{\partial \alpha}{\partial \boldsymbol{\nu}}\lambda - \alpha\frac{\partial \lambda}{\partial \boldsymbol{\nu}}\right) \tag{A.7}$$

and

$$\frac{\partial v_2}{\partial \boldsymbol{\nu}} = \frac{1}{\lambda^2}\left(\frac{\partial \beta}{\partial \boldsymbol{\nu}}\lambda - \beta\frac{\partial \lambda}{\partial \boldsymbol{\nu}}\right). \tag{A.8}$$

Since we use an inverse compositional approach for the alignment problem, we are only interested in the gradient at $\boldsymbol{\xi} = 0$, i.e. $\boldsymbol{\nu} = 0$ and $\boldsymbol{\omega} = 0$. Consequently all terms of V are null in (A.2) except the identity, and equation A.5 leads to

$$\nabla_{\boldsymbol{\nu}}(\boldsymbol{x_2})(0) = K. \tag{A.9}$$

As a consequence of equation A.9, we have

$$\frac{\partial \alpha}{\partial \boldsymbol{\nu}}(0) = \begin{pmatrix} f_u & s & c_u \end{pmatrix}, \quad \frac{\partial \beta}{\partial \boldsymbol{\nu}}(0) = \begin{pmatrix} 0 & f_v & c_v \end{pmatrix}, \quad \text{and} \quad \frac{\partial \lambda}{\partial \boldsymbol{\nu}}(0) = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}. \tag{A.10}$$

We can also note that at $\boldsymbol{\xi} = 0$, the point $\boldsymbol{x_1}$ is projected onto itself. Let λ_1 be the depth of the original point $\left(\begin{smallmatrix} u_1 \\ v_1 \end{smallmatrix}\right)$. Then we have $\boldsymbol{x_2}(0) = \boldsymbol{x_1} = \lambda_1 \left(\begin{smallmatrix} u_1 & v_1 & 1 \end{smallmatrix}\right)^\top$. Therefore, we can write

$$\alpha(0) = \lambda_1 u_1, \quad \beta(0) = \lambda_1 v_1, \quad \text{and} \quad \lambda(0) = \lambda_1. \tag{A.11}$$

From equations A.7, A.10, and A.11, it follows that

$$\frac{\partial u_2}{\partial \boldsymbol{\nu}}(0) = \frac{1}{\lambda_1^2}\left(\lambda_1 \begin{pmatrix} f_u & s & c_u \end{pmatrix} - \lambda_1 u_1 \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}\right), \tag{A.12a}$$

$$\frac{\partial u_2}{\partial \boldsymbol{\nu}}(0) = \frac{1}{\lambda_1}\begin{pmatrix} f_u & s & c_u - u_1 \end{pmatrix} \tag{A.12b}$$

and

$$\frac{\partial v_2}{\partial \boldsymbol{\nu}}(0) = \frac{1}{\lambda_1^2}\left(\lambda_1 \begin{pmatrix} 0 & f_v & c_v \end{pmatrix} - \lambda_1 v_1 \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}\right), \tag{A.13a}$$

$$\frac{\partial v_2}{\partial \boldsymbol{\nu}}(0) = \frac{1}{\lambda_1}\begin{pmatrix} 0 & f_v & c_v - v_1 \end{pmatrix}. \tag{A.13b}$$

Therefore, the 2x3 Jacobian relative to linear velocity terms is

$$\boxed{J_{\boldsymbol{\nu}}(\boldsymbol{x_2})(0) = \frac{1}{\lambda_1}\begin{pmatrix} f_u & s & c_u - u_1 \\ 0 & f_v & c_v - v_1 \end{pmatrix}} \tag{A.14}$$

A.5 Partial Derivatives Relative to Angular Velocity Terms

The purpose here is to obtain a computable expressions of $\nabla_{\boldsymbol{\omega}}(\boldsymbol{x_2})$. We will develop both terms appearing in equation A.6, i.e. $\nabla_{\boldsymbol{\omega}}(\exp(\omega_\times)\boldsymbol{X_1})$ and $\nabla_{\boldsymbol{\omega}}(V\boldsymbol{\nu})$. Since $\boldsymbol{\nu}$ does not depend on $\boldsymbol{\omega}$,

$$\nabla_{\boldsymbol{\omega}}(V\boldsymbol{\nu}) = \sum_i \nu_i \nabla_{\boldsymbol{\omega}} \boldsymbol{V_i}, \tag{A.15}$$

where $\boldsymbol{V_i}$ is the column i of V. We remind that we are interested in the gradient at $\boldsymbol{\xi} = 0$. A first order Taylor expansion of V at 0 shows that all terms are polynomial in ω_i so since $\boldsymbol{\nu}$ is also null, this all gradient is null.

$$\nabla_{\boldsymbol{\omega}}(V\boldsymbol{\nu})(0) = 0. \tag{A.16}$$

We are thus left with the first part $\nabla_{\boldsymbol{\omega}}(\boldsymbol{x_2})(0) = K\nabla_{\boldsymbol{\omega}}(\exp(\omega_\times)\boldsymbol{X_1})(0)$. If we develop $\exp(\omega_\times)$ as in equation A.1, we have

$$\nabla_{\boldsymbol{\omega}}(\exp(\omega_\times)\boldsymbol{X_1}) = 0 + \nabla_{\boldsymbol{\omega}}\left(\frac{\sin\theta}{\theta}\omega_\times \boldsymbol{X_1}\right) + \nabla_{\boldsymbol{\omega}}\left(\frac{1-\cos\theta}{\theta^2}\omega_\times^2 \boldsymbol{X_1}\right)$$

Let's analyze the last term of this expression. We can remark that $(1-\cos\theta)/\theta^2$ tends to $1/2$ when θ tends to 0. Additionally, all terms of $\omega_\times^2$ are polynomials of degree 2 with no term of degree 1. So all partial derivatives will be polynomials of degree 1, with no degree 0. As a result, the evaluation at $\boldsymbol{\omega} = 0$ will lead to

$$\nabla_{\boldsymbol{\omega}}\left(\frac{1-\cos\theta}{\theta^2}\omega_\times^2 \boldsymbol{X_1}\right)(0) = 0. \tag{A.17}$$

So we are left with

$$\nabla_{\boldsymbol{\omega}}(\exp(\omega_\times)\boldsymbol{X_1})(0) = \nabla_{\boldsymbol{\omega}}\left(\frac{\sin\theta}{\theta}\omega_\times \boldsymbol{X_1}\right)(0). \tag{A.18}$$

We can show that the derivative of the "hat" function has the following interesting form

$$\forall \boldsymbol{\omega}, \boldsymbol{y} \in \mathbb{R}^3, \quad \nabla_{\boldsymbol{\omega}}(\omega_\times \boldsymbol{y}) = -y_\times. \tag{A.19}$$

We can derive from equations A.18 and A.19 that

$$\nabla_{\boldsymbol{\omega}}(\exp(\omega_\times)\boldsymbol{X_1})(0) = -X_{1\times}. \tag{A.20}$$

From equations A.6 and A.20, and knowing that $\boldsymbol{X_1} = K^{-1}\boldsymbol{x_1}$ the gradient expression relative to $\boldsymbol{\omega}$ at 0 is

$$\nabla_{\boldsymbol{\omega}}(\boldsymbol{x_2})(0) = -K(K^{-1}\boldsymbol{x_1})_\times. \tag{A.21}$$

The 2x3 Jacobian can be obtained by computing derivatives of a quotient as we did in equations A.7 and A.8 for ν. Using symbolic computations, we obtain

$$\frac{\partial u_2}{\partial \boldsymbol{\omega}}(0) = \begin{pmatrix} \frac{-ab}{f_v} - s & \frac{ac}{f_u f_v} + f_u & \frac{-f_u^2 b + sc}{f_u f_v} \end{pmatrix}, \tag{A.22}$$

$$\frac{\partial v_2}{\partial \boldsymbol{\omega}}(0) = \begin{pmatrix} \frac{-b^2}{f_v} - f_v & \frac{bc}{f_u f_v} & \frac{c}{f_u} \end{pmatrix}, \tag{A.23}$$

where

$$a = u_1 - c_u, \quad b = v_1 - c_v \quad \text{and} \quad c = af_v - sb.$$

The 2x3 Jacobian relative to angular velocity terms is

$$\boxed{J_{\boldsymbol{\omega}}(\boldsymbol{x_2})(0) = \begin{pmatrix} \frac{-ab}{f_v} - s & \frac{ac}{f_u f_v} + f_u & \frac{-f_u^2 b + sc}{f_u f_v} \\ \frac{-b^2}{f_v} - f_v & \frac{bc}{f_u f_v} & \frac{c}{f_u} \end{pmatrix}} \tag{A.24}$$

with

$$a = u_1 - c_u, \quad b = v_1 - c_v \quad \text{and} \quad c = af_v - sb.$$

A.6 Partial Derivatives with Normalized Coordinates

We can hint from the expression of $\nabla_{\boldsymbol{\nu}}(\boldsymbol{x_2})$ in equation A.9 and the expression of $\nabla_{\boldsymbol{\omega}}(\boldsymbol{x_2})$ in equation A.21 that computing the Jacobian in the frame of the normalized coordinates $\lambda(\tilde{u}\ \tilde{v}\ 1)^\top = \lambda K^{-1}(u\ v\ 1)^\top$ will result in a simpler expression since we avoid the multiplications by K and K^{-1}. Let's thus make the change of variables

$$\begin{pmatrix} u_2 \\ v_2 \end{pmatrix} = K' \begin{pmatrix} \tilde{u_2} \\ \tilde{v_2} \end{pmatrix} + \begin{pmatrix} c_u \\ c_v \end{pmatrix} \quad \text{with} \quad K' = \begin{pmatrix} f_u & s \\ 0 & f_v \end{pmatrix}.$$

The chain rule means that $J_{\boldsymbol{\xi}}(\boldsymbol{x_2}) = K' \cdot J_{\boldsymbol{\xi}}(\tilde{\boldsymbol{x_2}})$ and we will show that $J_{\boldsymbol{\xi}}(\tilde{\boldsymbol{x_2}})$ has a way simpler expression. The same reasoning leading to equation A.9, now leads to

$$\nabla_{\boldsymbol{\nu}}(\tilde{\boldsymbol{x_2}})(0) = I_3 \tag{A.25}$$

and the one leading to equation A.21 now leads to

$$\nabla_{\boldsymbol{\omega}}(\tilde{\boldsymbol{x_2}})(0) = -\tilde{\boldsymbol{x_1}}_{\times}. \tag{A.26}$$

Writing the derivatives of the quotient as in equations A.7 and A.8 we thus obtain

$$\boxed{J_{\boldsymbol{\nu}}(\tilde{\boldsymbol{x_2}})(0) = \frac{1}{\tilde{\lambda_1}} \begin{pmatrix} 1 & 0 & -\tilde{u_1} \\ 0 & 1 & -\tilde{u_1} \end{pmatrix}} \tag{A.27}$$

and

$$\boxed{J_{\boldsymbol{\omega}}(\tilde{\boldsymbol{x_2}})(0) = \begin{pmatrix} -\tilde{u_1}\tilde{v_1} & 1 + \tilde{u_1}^2 & -\tilde{v_1} \\ -1 - \tilde{v_1}^2 & \tilde{u_1}\tilde{v_1} & \tilde{u_1} \end{pmatrix}} \tag{A.28}$$

Since the final objective is to compute the Jacobian of the photometric reprojection error, which is

$$J = \nabla I \cdot J_{\boldsymbol{\xi}}(\boldsymbol{x_2}) = \nabla I \cdot K' \cdot J_{\boldsymbol{\xi}}(\tilde{\boldsymbol{x_2}}),$$

note that the matrix K' is often directly multiplied with the image gradient ∇I and thus does not appear in the function computing the Jacobian of the warping function, actually computing $J_{\boldsymbol{\xi}}(\tilde{\boldsymbol{x_2}})$.

Bibliography

[1] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *IEEE transactions on pattern analysis and machine intelligence*, 34(11):2274–2282, 2012.

[2] ACM. Artifact review and badging. `https://www.acm.org/publications/policies/artifact-review-badging`, April 2018. Accessed: 2019-10-28.

[3] Harsh Agrawal, Karan Desai, Yufei Wang, Xinlei Chen, Rishabh Jain, Mark Johnson, Dhruv Batra, Devi Parikh, Stefan Lee, and Peter Anderson. nocaps: novel object captioning at scale. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 8948–8957, 2019.

[4] Eirikur Agustsson, Jasper RR Uijlings, and Vittorio Ferrari. Interactive full image segmentation. *arXiv preprint arXiv:1812.01888*, 2018.

[5] Application source code. `https://github.com/mpizenberg/annotation-app`, 2018. Accessed: 2018-05-20.

[6] Application documentation. `https://reva-n7.gitbook.io/annotation-app/`, 2018. Accessed: 2018-05-20.

[7] Image annotation package. `https://github.com/mpizenberg/elm-image-annotation`, 2018. Accessed: 2018-05-20.

[8] Jeremy Ashkenas. Coffeescript. `http://coffeescript.org/`, 2009. Accessed: 2019-06-25.

[9] Cedric Audras, A Comport, Maxime Meilland, and Patrick Rives. Real-time dense appearance-based slam for rgb-d sensors. In *Australasian Conf. on Robotics and Automation*, volume 2, pages 2–2, 2011.

[10] Sebastian McKenzie (Babel author). 2015 in review. `https://medium.com/@sebmck/2015-in-review-51ac7035e272`, January 2016. Accessed: 2019-06-26.

[11] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495, 2017.

[12] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[13] Lars Bak and Kasper Lund. Dart: a language for structured web programming. `http://googlecode.blogspot.com/2011/10/dart-language-for-structured-web.html`, October 2011. Accessed: 2019-06-25.

[14] Henry G Baker. Lively linear lisp:"look ma, no garbage!". *ACM Sigplan notices*, 27(8):89–98, 1992.

[15] Simon Baker, Ralph Gross, Takahiro Ishikawa, and Iain Matthews. Lucas-kanade 20 years on: A unifying framework: Part 2. In *International Journal of Computer Vision*. Citeseer, 2003.

[16] Simon Baker and Iain Matthews. Lucas-kanade 20 years on: A unifying framework. *International journal of computer vision*, 56(3):221–255, 2004.

[17] Vincent Balat. Ocsigen: typing web interaction with objective caml. In *Proceedings of the 2006 workshop on ML*, pages 84–94. ACM, 2006.

[18] Nick Barnes. Publish your computer code: it is good enough. *Nature News*, 467(7317):753–753, 2010.

[19] Adela Barriuso and Antonio Torralba. Notes on image annotation. *arXiv preprint arXiv:1210.3448*, 2012.

[20] D. Batra, A. Kowdle, D. Parikh, J. Luo, and T. Chen. iCoseg: Interactive co-segmentation with intelligent scribble guidance. In *2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3169–3176, June 2010.

[21] Rodrigo Benenson, Stefan Popov, and Vittorio Ferrari. Large-scale interactive object segmentation with human annotators. In *CVPR*, 2019.

[22] Massimo Bertozzi, Alberto Broggi, Elena Cardarelli, Rean Isabella Fedriga, Luca Mazzei, and Pier Paolo Porta. Viac expedition toward autonomous mobility [from the field]. *IEEE Robotics & Automation Magazine*, 18(3):120–124, 2011.

[23] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606. International Society for Optics and Photonics, 1992.

[24] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[25] Harry Blum and Roger N Nagel. Shape description using weighted symmetric axis features. *Pattern recognition*, 10(3):167–180, 1978.

[26] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[27] Y. Y. Boykov and M. P. Jolly. Interactive graph cuts for optimal boundary amp; region segmentation of objects in N-D images. In *Eighth IEEE International Conference on Computer Vision, 2001. ICCV 2001. Proceedings*, volume 1, pages 105–112 vol.1, 2001.

[28] Worldwide browser market share. `http://gs.statcounter.com/browser-market-share/all/worldwide/2019`, 2019. Accessed: 2019-06-15.

[29] Jonathan B Buckheit and David L Donoho. Wavelab and reproducible research. In *Wavelets and statistics*, pages 55–81. Springer, 1995.

[30] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35(10):1157–1163, 2016.

[31] Rod M Burstall, David B MacQueen, and Donald T Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143. ACM, 1980.

[32] Draft iso specification for the c programming language, 2018 edition. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2346.pdf`, March 2019. Accessed: 2019-06-20.

[33] Ricardo Cabello. Threejs, a javascript 3d library. `https://threejs.org/`. Accessed: 2020-01-26.

[34] Nicolas Cannasse. Haxe interview. `http://ncannasse.fr/blog/haxe_interview`, June 2009. Accessed: 2019-06-25.

[35] Nicholas Carlevaris-Bianco, Arash K Ushani, and Ryan M Eustice. University of michigan north campus long-term vision and lidar dataset. *The International Journal of Robotics Research*, 35(9):1023–1035, 2016.

[36] Axel Carlier, Vincent Charvillat, Amaia Salvador, Xavier Giro-i Nieto, and Oge Marques. Click'n'cut: Crowdsourced interactive segmentation with object candidates. In *Proceedings of the 2014 International ACM Workshop on Crowdsourcing for Multimedia*, pages 53–56. ACM, 2014.

[37] Axel Carlier, Vincent Charvillat, Amaia Salvador, Xavier Giro-i Nieto, and Oge Marques. Click'N'Cut: Crowdsourced Interactive Segmentation with Object Candidates. In *Proceedings of the 2014 International ACM Workshop on Crowdsourcing for Multimedia*, CrowdMM '14, pages 53–56, New York, NY, USA, 2014. ACM.

[38] Axel Carlier, Amaia Salvador, Ferran Cabezas, Xavier Giro-i Nieto, Vincent Charvillat, and Oge Marques. Assessment of crowdsourcing and gamification loss in user-assisted object segmentation. *Multimedia tools and applications*, 75(23):15901–15928, 2016.

[39] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):48, 2017.

[40] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[41] Xinlei Chen, Hao Fang, Tsung-Yi Lin, Ramakrishna Vedantam, Saurabh Gupta, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*, 2015.

[42] Adam Chlipala. Ur/web: A simple model for programming the web. *ACM SIGPLAN Notices*, 50(1):153–165, 2015.

[43] Jon F Claerbout and Martin Karrenbach. Electronic documents give reproducible research a new meaning. In *SEG Technical Program Expanded Abstracts 1992*, pages 601–604. Society of Exploration Geophysicists, 1992.

[44] Lin Clark. A crash course in just-in-time (jit) compilers. `https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/`, February 2017. Accessed: 2019-06-17.

[45] Lin Clark. Inside a super fast css engine: Quantum css (aka stylo). `https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/`, August 2017. Accessed: 2019-07-15.

[46] Lin Clark. The whole web at maximum fps: How webrender gets rid of jank. `https://hacks.mozilla.org/2017/10/the-whole-web-at-maximum-fps-how-webrender-gets-rid-of-jank/`, October 2017. Accessed: 2019-07-15.

[47] Lin Clark. Standardizing wasi: A system interface to run webassembly outside the web. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, March 2019. Accessed: 2019-10-27.

[48] Lin Clark. Webassembly interface types: Interoperate with all the things! https://hacks.mozilla.org/2019/08/webassembly-interface-types/, August 2019. Accessed: 2019-10-27.

[49] Lin Clark. Webassembly interface types: Interoperate with all the things! https://youtu.be/Qn_4F3foB3Q, August 2019. Accessed: 2019-10-27.

[50] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[51] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 24(5):603–619, 2002.

[52] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *International Symposium on Formal Methods for Components and Objects*, pages 266–296. Springer, 2006.

[53] Santiago Cortés, Arno Solin, Esa Rahtu, and Juho Kannala. Advio: An authentic dataset for visual-inertial odometry. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 419–434, 2018.

[54] Alex Crichton. Tar file reading/writing in rust, tar-rs. https://github.com/alexcrichton/tar-rs. Accessed: 2020-01-26.

[55] Evan Czaplicki. A farewell to frp. https://elm-lang.org/blog/farewell-to-frp, May 2016. Accessed: 2019-07-13.

[56] Evan Czaplicki and Stephen N Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation-PLDI'13*. ACM Press, 2013.

[57] Ryan Dahl. Original node.js presentation. https://youtu.be/ztspvPYybIY, November 2009. Accessed: 2019-06-17.

[58] Dataturks. https://dataturks.com/, 2018. Accessed: 2018-05-20.

[59] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 29(6):1052–1067, 2007.

[60] Chauncey Mitchell Depew. *One Hundred Years of American Commerce*. DO Haynes, 1895.

[61] Frédéric Devernay and Olivier D Faugeras. Automatic calibration and removal of distortion from scenes of structured environments. In *Investigative and Trial Image Processing*, volume 2567, pages 62–72. International Society for Optics and Photonics, 1995.

[62] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. Pnacl: Portable native client executables. *Google White Paper*, 2010.

[63] Matthew Dunbabin, Jonathan Roberts, Kane Usher, Graeme Winstanley, and Peter Corke. A hybrid auv design for shallow water reef navigation. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 2105–2110. IEEE, 2005.

[64] A. Dutta, A. Gupta, and A. Zissermann. VGG image annotator (VIA). `http://www.robots.ox.ac.uk/~vgg/software/via/`, 2016. Accessed: 2018-05-20.

[65] Ecmascript standard specification, 5.1 edition. `http://www.ecma-international.org/ecma-262/5.1/#sec-11.6.1`, June 2011. Accessed: 2019-06-17.

[66] Ecmascript standard specification document, 2018 edition. `https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, June 2018. Accessed: 2019-06-20.

[67] Death of elaine herzberg. `https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg`, March 2018. Accessed: 2019-11-27.

[68] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[69] Jakob Engel, Vladlen Koltun, and Daniel Cremers. Direct sparse odometry. *IEEE transactions on pattern analysis and machine intelligence*, 40(3):611–625, 2017.

[70] Jakob Engel, Vladyslav Usenko, and Daniel Cremers. A photometrically calibrated benchmark for monocular visual odometry. *arXiv preprint arXiv:1607.02555*, 2016.

[71] Unreal Engine. Epic games releases "epic citadel" on the web. `https://www.unrealengine.com/en-US/blog/epic-games-releases-epic-citadel-on-the-web`, May 2013. Accessed: 2019-11-05.

[72] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[73] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[74] Olivier Faugeras and OLIVIER AUTOR FAUGERAS. *Three-dimensional computer vision: a geometric viewpoint*. MIT press, 1993.

[75] Richard Feldman. First elm runtime error at noredink. `https://twitter.com/rtfeldman/status/961051166783213570?lang=en`, February 2018. Accessed: 2019-07-08.

[76] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient graph-based image segmentation. *International journal of computer vision*, 59(2):167–181, 2004.

[77] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[78] Andrew W Fitzgibbon. Simultaneous linear estimation of multiple view geometry and lens distortion. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[79] Vulnerability statistics for flash. `https://www.cvedetails.com/product/6761/Adobe-Flash-Player.html?vendor_id=53`. Accessed: 2019-10-23.

[80] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. In *2014 IEEE international conference on robotics and automation (ICRA)*, pages 15–22. IEEE, 2014.

[81] Friedrich Fraundorfer, Petri Tanskanen, and Marc Pollefeys. A minimal case solution to the calibrated relative pose problem for the case of two known orientation angles. In *European Conference on Computer Vision*, pages 269–282. Springer, 2010.

[82] Jesse James Garrett. Ajax: A new approach to web applications. `https://web.archive.org/web/20190507051447/http://adaptivepath.org/ideas/ajax-new-approach-web-applications/`, February 2005. Accessed: 2019-07-04.

[83] Jonathan Gay. The history of flash. `https://web.archive.org/web/20080120200126/http://www.adobe.com:80/macromedia/events/john_gay/index.html`. Accessed: 2019-10-23.

[84] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[85] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[86] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[87] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[88] Brendon C. Glazer. Interactive ray tracing of vrml scenes in java. Master's thesis, Massachusetts Institute of Technology, 1999.

[89] Go specification, 2018 edition. `https://golang.org/ref/spec`, November 2018. Accessed: 2019-06-20.

[90] Roland Goecke, Akshay Asthana, Niklas Pettersson, and Lars Petersson. Visual vehicle egomotion estimation using the fourier-mellin transform. In *2007 IEEE Intelligent Vehicles Symposium*, pages 450–455. IEEE, 2007.

[91] Google announce chrome. `https://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html`, September 2008. Accessed: 2019-06-17.

[92] Luke Gottlieb, Jaeyoung Choi, Pascal Kelm, Thomas Sikora, and Gerald Friedland. Pushing the limits of mechanical turk: qualifying the crowd for video geo-location. In *Proceedings of the ACM multimedia 2012 workshop on Crowdsourcing for multimedia*, pages 23–28. ACM, 2012.

[93] Matthew Griffith. Elm-ui library. `https://github.com/mdgriffith/elm-ui`, 2017. Accessed: 2019-07-08.

[94] Matthew Griffith. Understanding style. `https://youtu.be/NYb2GDWMIm0`, June 2017. Accessed: 2019-07-08.

[95] Daniel Gutiérrez-Gómez, Walterio Mayol-Cuevas, and Josechu J Guerrero. Inverse depth for accurate photometric and geometric error minimisation in rgb-d dense visual odometry. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 83–89. IEEE, 2015.

[96] Google web toolkit. `http://www.gwtproject.org/`, May 2006. Accessed: 2019-06-25.

[97] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM.

[98] Peter Hallam and Alex Russel. The future of js and you. `https://youtu.be/ntDZa7ekFEA`, May 2011. Accessed: 2019-07-17.

[99] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. In *2014 IEEE international conference on Robotics and automation (ICRA)*, pages 1524–1531. IEEE, 2014.

[100] Rich Harris. Svelte, cybernetically enhanced web apps. `https://svelte.dev/`, 2016. Accessed: 2019-07-14.

[101] Dania El Hassan. Autocad's journey to the web. `https://youtu.be/BfkL3WgOPdI`, October 2019. Accessed: 2019-10-25.

[102] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[103] David Herman, Luke Wagner, and Alon Zakai. asm.js: Working draft. `http://asmjs.org/spec/latest`, 2013. Accessed: 2019-10-24.

[104] Miško Hevery and Adam Abrons. Declarative web-applications without server: demonstration of how a fully functional web-application can be built in an hour with only html, css & javascript library. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 801–802. ACM, 2009.

[105] Miško Hevery and Brad Green. Google i/o 2013 - design decisions in angularjs. `https://youtu.be/HCR7i5F5L8c?t=197`, 2013. Accessed: 2019-07-04.

[106] Chien-Ju Ho, Tao-Hsuan Chang, Jong-Chuan Lee, Jane Yung-jen Hsu, and Kuan-Ta Chen. Kisskissban: a competitive human computation game for image annotation. *ACM SIGKDD Explorations Newsletter*, 12(1):21–24, 2010.

[107] Jeff Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[108] Albert S Huang, Abraham Bachrach, Peter Henry, Michael Krainin, Daniel Maturana, Dieter Fox, and Nicholas Roy. Visual odometry and mapping for autonomous flight using an rgb-d camera. In *Robotics Research*, pages 235–252. Springer, 2017.

[109] Solomon Hykes. Solomon hykes tweet about the importance of webassembly. https://twitter.com/solomonstre/status/1111004913222324225?lang=en, March 2019. Accessed: 2019-10-28.

[110] SAE International. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, revision of june 2018. https://www.sae.org/standards/content/j3016_201806/, June 2018. Accessed: 2019-11-26.

[111] Anil K Jain and Balakrishnan Chandrasekaran. 39 dimensionality and sample size considerations in pattern recognition practice. *Handbook of statistics*, 2:835–855, 1982.

[112] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N Smith. Property models: from incidental algorithms to reusable components. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 89–98. ACM, 2008.

[113] Java specification, java se 12 edition. https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf, February 2019. Accessed: 2019-06-20.

[114] Steve Jobs. Thoughts on flash. https://www.apple.com/hotnews/thoughts-on-flash/, April 2000. Accessed: 2019-10-23.

[115] Vulnerability statistics for oracle jre. https://www.cvedetails.com/product/19117/Oracle-JRE.html?vendor_id=93. Accessed: 2019-10-22.

[116] Jsfuck, an esoteric subset of javascript with only six characters. https://en.wikipedia.org/wiki/JSFuck, 2010. Accessed: 2019-06-21.

[117] js-sys, raw bindings to js global apis for projects using wasm-bindgen. https://github.com/rustwasm/wasm-bindgen/tree/master/crates/js-sys. Accessed: 2019-10-30.

[118] Christian Kerl, Jorg Stuckler, and Daniel Cremers. Dense continuous-time tracking and mapping with rolling shutter rgb-d cameras. In *Proceedings of the IEEE international conference on computer vision*, pages 2264–2272, 2015.

[119] Christian Kerl, Jürgen Sturm, and Daniel Cremers. Robust odometry estimation for rgb-d cameras. In *2013 IEEE International Conference on Robotics and Automation*, pages 3748–3754. IEEE, 2013.

[120] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 1–10. IEEE Computer Society, 2007.

[121] Sebastian Klose, Philipp Heise, and Alois Knoll. Efficient compositional approaches for real-time robust direct visual odometry from rgb-d data. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1100–1106. IEEE, 2013.

[122] Kurt Konolige, Motilal Agrawal, and Joan Sola. Large-scale visual odometry for rough terrain. In *International symposium on robotics research*, volume 19, pages 23–43, 2007.

[123] Christoph Korinke. Intuitive Input Methods for Interactive Segmentation on Mobile Touch-Based Devices. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 645–648, New York, NY, USA, 2015. ACM.

[124] Christoph Korinke, Nils Steffen Worzyk, and Susanne Boll. Exploring Touch Interaction Methods for Image Segmentation on Mobile Devices. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia*, MUM '15, pages 84–93, New York, NY, USA, 2015. ACM.

[125] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*, 2017.

[126] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[127] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[128] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, and Vittorio Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv:1811.00982*, 2018.

[129] Labelbox. `https://www.labelbox.com/`, 2018. Accessed: 2018-05-20.

[130] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[131] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[132] Ryan Levick. We need a safer systems programming language. `https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/`, July 2019. Accessed: 2019-11-26.

[133] Alex Levinshtein, Adrian Stere, Kiriakos N Kutulakos, David J Fleet, Sven J Dickinson, and Kaleem Siddiqi. Turbopixels: Fast superpixels using geometric flows. *IEEE transactions on pattern analysis and machine intelligence*, 31(12):2290–2297, 2009.

[134] Web version of adobe lightroom. `https://lightroom.adobe.com`. Accessed: 2019-11-04.

[135] Nicolas Limare and Jean-Michel Morel. The ipol initiative: Publishing and testing algorithms on line for reproducible research in image processing. *Procedia computer science*, 4:716–725, 2011.

[136] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[137] Peng Liu, Hui Zhang, and Kie B Eom. Active deep learning for classification of hyperspectral images. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(2):712–724, 2017.

[138] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[139] H Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133, 1981.

[140] Mark Maimone, Yang Cheng, and Larry Matthies. Two years of visual odometry on the mars exploration rovers. *Journal of Field Robotics*, 24(3):169–186, 2007.

[141] Larry Matthies and STEVENA Shafer. Error modeling in stereo navigation. *IEEE Journal on Robotics and Automation*, 3(3):239–248, 1987.

[142] Michael McAleer. Audi's self-driving a8: drivers can watch youtube or check emails at 60km/h. `https://www.irishtimes.com/life-and-style/motors/audi-s-self-driving-a8-drivers-can-watch-youtube-or-check-emails-at-60km-h-1.3150496`, July 2017. Accessed: 2019-11-26.

[143] Kevin McGuinness and Noel E O'connor. A comparative evaluation of interactive segmentation algorithms. *Pattern Recognition*, 43(2):434–444, 2010.

[144] Sun microsystems. Java 3d api specification, version 1.1.2. `https://docs.oracle.com/cd/E17802_01/j2se/javase/technologies/desktop/java3d/forDevelopers/j3dguide/j3dTOC.doc.html`, June 1999. Accessed: 2019-10-22.

[145] Michael J Milford, Gordon F Wyeth, and David Prasser. Ratslam: a hippocampal model for simultaneous localization and mapping. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 1, pages 403–408. IEEE, 2004.

[146] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[147] Hans P Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical report, Stanford Univ CA Dept of Computer Science, 1980.

[148] Eric N Mortensen and William A Barrett. Intelligent scissors for image composition. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 191–198. ACM, 1995.

[149] Motionanalysis. `https://www.motionanalysis.com`. Accessed: 2020-01-17.

[150] ACM Multimedia. Overall acm multimedia reproducibility objectives. `https://project.inria.fr/acmmmreproducibility/`, 2019. Accessed: 2019-10-28.

[151] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

[152] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136. IEEE, 2011.

[153] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. Dtam: Dense tracking and mapping in real-time. In *2011 international conference on computer vision*, pages 2320–2327. IEEE, 2011.

[154] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[155] Jakob Nielsen. Flash: 99% bad. `https://www.nngroup.com/articles/flash-99-percent-bad/`, October 2000. Accessed: 2019-10-23.

[156] David Nister. An efficient solution to the five-point relative pose problem. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages II–195. IEEE, 2003.

[157] David Nistér, Oleg Naroditsky, and James Bergen. Visual odometry. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 1, pages I–I. Ieee, 2004.

[158] Speaker description, ryan dahl: Node.js, evented i/o for v8 javascript. `https://www.jsconf.eu/2009/speaker/speakers_selected.html`, September 2009. Accessed: 2019-06-17.

[159] Details about the event-stream incident. `https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident`, November 2018. Accessed: 2019-06-19.

[160] Jonathan A Obar and Anne Oeldorf-Hirsch. The biggest lie on the internet: Ignoring the privacy policies and terms of service policies of social networking services. *Information, Communication & Society*, pages 1–20, 2018.

[161] Max Ogden. Callback hell, a guid to writing asynchronous javascript programs. `https://github.com/maxogden/callback-hell`, February 2016. Accessed: 2019-06-20.

[162] David Oleson, Alexander Sorokin, Greg Laughlin, Vaughn Hester, John Le, and Lukas Biewald. Programmatic gold: Targeted and scalable quality assurance in crowdsourcing. In *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[163] Opal ruby to javascript compiler. `https://opalrb.com/`, 2011. Accessed: 2019-06-25.

[164] E. Palazzolo, J. Behley, P. Lottes, P. Giguère, and C. Stachniss. ReFusion: 3D Reconstruction in Dynamic Environments for RGB-D Cameras Exploiting Residuals. In *iros*, 2019.

[165] Dim P Papadopoulos, Jasper RR Uijlings, Frank Keller, and Vittorio Ferrari. Extreme clicking for efficient object annotation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4930–4939, 2017.

[166] George Papandreou, Liang-Chieh Chen, Kevin P Murphy, and Alan L Yuille. Weakly- and semi-supervised learning of a deep convolutional network for semantic image segmentation. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1742–1750. IEEE Computer Society, 2015.

[167] Sean Parent. A possible future of software development. `https://youtu.be/4moyKUHApq4`, July 2007. Accessed: 2019-07-12.

[168] Sean Parent. A possible future of software development. `https://stlab.cc/legacy/figures/Boostcon_possible_future.pdf`, May 2007. Accessed: 2019-07-12.

[169] Mikael Persson and Klas Nordberg. Lambda twist: an accurate fast robust perspective three point (p3p) solver. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 318–332, 2018.

[170] Jay Phelps. Webassembly, neither web nor assembly, but revolutionary. `https://www.infoq.com/presentations/webassembly-intro/`, March 2019. Accessed: 2019-11-05.

[171] Matthieu Pizenberg. Camera pose estimation given 3d points and corresponding pixel coordinates. `https://github.com/rust-cv/p3p`. Accessed: 2020-01-26.

[172] Matthieu Pizenberg, Axel Carlier, Emmanuel Faure, and Vincent Charvillat. Outlining objects for interactive segmentation on touch devices. In *Proceedings of the 25th ACM International Conference on Multimedia*, MM '17, pages 1734–1742, New York, NY, USA, 2017. ACM.

[173] Matthieu Pizenberg, Axel Carlier, Emmanuel Faure, and Vincent Charvillat. Web-based configurable image annotations. In *Proceedings of the 26th ACM International Conference on Multimedia*, MM '18, pages 1368–1371, New York, NY, USA, 2018. ACM.

[174] Pov-ray, persistence of vision (tm) raytracer. `https://www.povray.org`. Accessed: 2020-01-20.

[175] Tom Preston-Werner. Semantic versioning specification, version 2.0.0. `https://semver.org/spec/v2.0.0.html`, June 2013. Accessed: 2019-07-10.

[176] Yvain Quéau, Mathieu Pizenberg, Jean-Denis Durou, and Daniel Cremers. Microgeometry capture and rgb albedo estimation by photometric stereo without demosaicing. In *Thirteenth International Conference on Quality Control by Artificial Vision 2017*, volume 10338, page 103380O. International Society for Optics and Photonics, 2017.

[177] Sarunas J Raudys and Anil K. Jain. Small sample size effects in statistical pattern recognition: Recommendations for practitioners. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 13(3):252–264, 1991.

[178] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[179] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[180] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[181] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[182] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[183] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "GrabCut": Interactive Foreground Extraction Using Iterated Graph Cuts. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 309–314, New York, NY, USA, 2004. ACM.

[184] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[185] Bryan C Russell, Antonio Torralba, Kevin P Murphy, and William T Freeman. Labelme: a database and web-based tool for image annotation. *International journal of computer vision*, 77(1-3):157–173, 2008.

[186] Rust elected most loved programming language in stackoverflow 2019 developer survey. `https://insights.stackoverflow.com/survey/2019`, 2019. Accessed: 2019-10-30.

[187] Philippe Salembier and Luis Garrido. Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. *IEEE transactions on Image Processing*, 9(4):561–576, 2000.

[188] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry [tutorial]. *IEEE robotics & automation magazine*, 18(4):80–92, 2011.

[189] Isaac Z. Schlueter. How npm works. `https://youtu.be/ShRDgdvlZQ8`, May 2011. Accessed: 2019-06-19.

[190] Thomas Schöps, Jakob Engel, and Daniel Cremers. Semi-dense visual odometry for ar on a smartphone. In *2014 IEEE international symposium on mixed and augmented reality (ISMAR)*, pages 145–150. IEEE, 2014.

[191] David Schubert, Nikolaus Demmel, Lukas von Stumberg, Vladyslav Usenko, and Daniel Cremers. Rolling-shutter modelling for direct visual-inertial odometry. *arXiv preprint arXiv:1911.01015*, 2019.

[192] David Schubert, Thore Goll, Nikolaus Demmel, Vladyslav Usenko, Jörg Stückler, and Daniel Cremers. The tum vi benchmark for evaluating visual-inertial odometry. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1680–1687. IEEE, 2018.

[193] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2. 0. In *OOPSLA Companion*, pages 975–985, 2006.

[194] Piyush Sharma, Nan Ding, Sebastian Goodman, and Radu Soricut. Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2556–2565, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[195] Amnon Shashua. Trilinearity in visual recognition by alignment. In *European Conference on Computer Vision*, pages 479–484. Springer, 1994.

[196] Jamie Shotton, Andrew Fitzgibbon, Mat Cook, Toby Sharp, Mark Finocchio, Richard Moore, Alex Kipman, and Andrew Blake. Real-time human pose recognition in parts from single depth images. In *CVPR 2011*, pages 1297–1304. Ieee, 2011.

[197] Nishant Sinha, Rezwana Karim, and Monika Gupta. Simplifying web programming. In *Proceedings of the 8th India Software Engineering Conference*, pages 80–89. ACM, 2015.

[198] Mike Smith, Ian Baldwin, Winston Churchill, Rohan Paul, and Paul Newman. The new college vision and laser data set. *The International Journal of Robotics Research*, 28(5):595–599, 2009.

[199] Minas E Spetsakis and Yiannis Aloimonos. Closed form solution to the structure from motion problem from line correspondences. In *AAAI*, pages 738–743, 1987.

[200] Gideon P Stein. Lens distortion calibration using point correspondences. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 602–608. IEEE, 1997.

[201] Frank Steinbrücker, Jürgen Sturm, and Daniel Cremers. Real-time visual odometry from dense rgb-d images. In *2011 IEEE international conference on computer vision workshops (ICCV Workshops)*, pages 719–722. IEEE, 2011.

[202] Hauke Strasdat and Steven Lovegrove. Sophus, c++ implementation of lie groups using eigen. `https://github.com/strasdat/Sophus`, 2012. Accessed: 2020-01-08.

[203] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. A benchmark for the evaluation of rgb-d slam systems. In *2012 IEEE/RSJ*

International Conference on Intelligent Robots and Systems, pages 573–580. IEEE, 2012.

[204] Hao Su, Jia Deng, and Li Fei-Fei. Crowdsourcing annotations for visual object detection. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[205] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. Openvslam: A versatile visual slam framework. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 2292–2295. ACM, 2019.

[206] Sajjad Taheri, Alexander Vedienbaum, Alexandru Nicolau, Ningxin Hu, and Mohammad R Haghighat. Opencv. js: Computer vision processing for the open web platform. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 478–483. ACM, 2018.

[207] Carlo Tomasi and Takeo Kanade. Shape and motion from image streams under orthography: a factorization method. *International journal of computer vision*, 9(2):137–154, 1992.

[208] Alan Turing. Computing machinery and intelligence. *Mind*, 59(236):433, 1950.

[209] Jasper RR Uijlings, Koen EA Van De Sande, Theo Gevers, and Arnold WM Smeulders. Selective search for object recognition. *International journal of computer vision*, 104(2):154–171, 2013.

[210] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[211] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. Captcha: Using hard ai problems for security. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–311. Springer, 2003.

[212] Luis Von Ahn and Laura Dabbish. Esp: Labeling images with a computer game. In *AAAI spring symposium: Knowledge collection from volunteer contributors*, volume 2, 2005.

[213] Luis Von Ahn and Laura Dabbish. Designing games with a purpose. *Communications of the ACM*, 51(8):58–67, 2008.

[214] Luis Von Ahn, Ruoran Liu, and Manuel Blum. Peekaboom: a game for locating objects in images. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 55–64. ACM, 2006.

[215] Luis Von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[216] Philip Wadler. Is there a use for linear logic? *PEPM*, 91:255–273, 1991.

[217] Jordan Walke. Reason home page. `https://reasonml.github.io/`, May 2016. Accessed: 2019-06-25.

[218] Jue Wang, Maneesh Agrawala, and Michael F. Cohen. Soft scissors: An interactive tool for realtime high quality matting. *ACM Trans. Graph.*, 26(3), July 2007.

[219] Sen Wang, Ronald Clark, Hongkai Wen, and Niki Trigoni. Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2043–2050. IEEE, 2017.

[220] wasm-bindgen, facilitating high-level interactions between wasm modules and javascript. `https://github.com/rustwasm/wasm-bindgen`. Accessed: 2019-10-30.

[221] Wasmer, the universal webassembly runtime. `https://wasmer.io/`. Accessed: 2019-10-25.

[222] Your favorite rust -> wasm workflow tool! `https://github.com/rustwasm/wasm-pack`. Accessed: 2019-10-30.

[223] Wasmtime, a small and efficient runtime for webassembly & wasi. `https://wasmtime.dev/`. Accessed: 2019-10-25.

[224] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 53–65. ACM, 2018.

[225] web-sys, raw bindings to web apis for projects using wasm-bindgen. `https://github.com/rustwasm/wasm-bindgen/tree/master/crates/web-sys`. Accessed: 2019-10-30.

[226] Juyang Weng, Narendra Ahuja, and Thomas S Huang. Optimal motion and structure estimation. *IEEE Transactions on pattern analysis and machine intelligence*, 15(9):864–884, 1993.

[227] Jianxiong Xiao, James Hays, Krista A Ehinger, Aude Oliva, and Antonio Torralba. Sun database: Large-scale scene recognition from abbey to zoo. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3485–3492. IEEE, 2010.

[228] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.

[229] Nan Yang, Rui Wang, Jorg Stuckler, and Daniel Cremers. Deep virtual stereo odometry: Leveraging deep depth prediction for monocular direct sparse odometry. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 817–833, 2018.

[230] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[231] Peter Young, Alice Lai, Micah Hodosh, and Julia Hockenmaier. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. *TACL*, 2:67–78, 2014.

[232] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

[233] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE multimedia*, 19(2):4–10, 2012.

Popularized Abstract

Computer vision is the science of understanding the environment from images, which is very useful in domains such as augmented reality or autonomous vehicles. In this thesis, we explore how specific Web applications can help improving some 2D and 3D subfields of computer vision.

In the first part, we focus on the problem of annotation for image segmentation. This task consists in identifying the pixels corresponding to an object of interest in an image to help a computer to be able to do it automatically later. We present our method to efficiently annotate datasets of images through an online crowdsourcing service.

In the second part of this thesis, we focus on the problem of visual odometry. It consists in tracking the trajectory of a camera and reconstructing a virtual 3D map of its environment just from the images provided by the camera itself. We present our open-source library, and a Web application able to improve the algorithm results thanks to user input.

Résumé vulgarisé

La vision par ordinateur est la science de la compréhension de l'environnement à partir d'images, très utile dans des domaines tels que la réalité augmentée ou les véhicules autonomes.

Dans un premier temps, nous traitons le problème de l'annotation pour la segmentation d'image. Cette tâche consiste à identifier les pixels correspondants à un objet d'intérêt dans une image pour entraîner un ordinateur à le faire automatiquement plus tard. Nous présentons notre approche pour annoter efficacement des grands ensembles d'images par le biais d'un service de crowdsourcing en ligne.

Dans un second temps, nous étudions le problème d'odométrie visuelle, qui consiste à retracer la trajectoire d'une caméra et à reconstruire une carte virtuelle en 3D de son environnement à partir des images capturées par la caméra. Nous présentons notre bibliothèque logicielle libre, ainsi qu'une application Web capable d'améliorer les résultats de l'algorithme grâce à des interactions utilisateur.

Abstract

Computer vision is the computational science aiming at reproducing and improving the ability of human vision to understand its environment. In this thesis, we focus on two fields of computer vision, namely image segmentation and visual odometry and we show the positive impact that interactive Web applications provide on each.

The first part of this thesis focuses on image annotation and segmentation. We introduce the image annotation problem and challenges it brings for large, crowdsourced datasets. Many interactions have been explored in the literature to help segmentation algorithms. The most common consist in designating contours, bounding boxes around objects, or interior and exterior scribbles. When crowdsourcing, annotation tasks are delegated to a non-expert public, sometimes on cheaper devices such as tablets. In this context, we conducted a user study showing the advantages of the outlining interaction over scribbles and bounding boxes. Another challenge of crowdsourcing is the distribution medium. While evaluating an interaction in a small user study does not require complex setup, distributing an annotation campaign to thousands of potential users might differ. Thus we describe how the Elm programming language helped us build a reliable image annotation Web application. A highlights tour of its functionalities and architecture is provided, as well as a guide on how to deploy it to crowdsourcing services such as Amazon Mechanical Turk. The application is completely open-source and available online.

In the second part of this thesis we present our open-source direct visual odometry library. In that endeavor, we provide an evaluation of other open-source RGB-D camera tracking algorithms and show that our approach performs as well as the currently available alternatives. The visual odometry problem relies on geometry tools and optimization techniques traditionally requiring much processing power to perform at realtime framerates. Since we aspire to run those algorithms directly in the browser, we review past and present technologies enabling high performance computations on the Web. In particular, we detail how to target a new standard called WebAssembly from the C++ and Rust programming languages. Our library has been started from scratch in the Rust programming language, which then allowed us to easily port it to WebAssembly. Thanks to this property, we are able to showcase a visual odometry Web application with multiple types of interactions available. A timeline enables one-dimensional navigation along the video sequence. Pairs of image points can be picked on two 2D thumbnails of the image sequence to realign cameras and correct drifts. Colors are also used to identify parts of the 3D point cloud, selectable to reinitialize camera positions. Combining those interactions enables improvements on the tracking and 3D point reconstruction results.

Résumé

La vision par ordinateur est un domaine de l'informatique visant à reproduire et à améliorer la capacité de la vision humaine à comprendre son environnement. Dans cette thèse, nous nous concentrons sur deux domaines de la vision par ordinateur, à savoir la segmentation d'image et l'odométrie visuelle. Nous montrons l'impact positif qu'apporte l'usage d'applications Web interactives pour chacun d'eux.

La première partie de cette thèse porte sur l'annotation et la segmentation d'images. Nous définissons dans un premier temps le problème de l'annotation d'images et les défis que cela représente pour des grands ensembles de données. De nombreuses interactions ont été utilisées dans la littérature pour aider les algorithmes de segmentation. Les plus courantes consistent à désigner explicitement des contours, dessiner des boîtes englobantes, ou marquer des traits à l'intérieur et à l'extérieur des objets d'intérêt. Dans un contexte de crowdsourcing, les tâches d'annotation sont déléguées à un public non-expert. Pour cette raison, nous avons mené une étude utilisateur montrant les avantages d'une interaction que nous appelons entourage par rapport aux autres types d'interactions. Nous décrivons comment le langage de programmation Elm nous a aidé à construire une application Web d'annotation d'images qui soit fiable. Un tour d'horizon des fonctionnalités et de son architecture est proposé, ainsi qu'un guide pour le déploiement dans des services de microtâches comme Amazon Mechanical Turk. Cette application est entièrement libre et mise à disposition en ligne.

Dans la seconde partie de cette thèse, nous présentons notre bibliothèque libre d'odométrie visuelle directe. Nous fournissons une évaluation comparative montrant que notre approche est aussi performante que les alternatives actuellement disponibles. La formulation du problème d'odométrie visuelle repose sur des outils géométriques et des techniques d'optimisation nécessitant une grosse puissance de calcul pour fonctionner à 25 images par seconde. Puisque nous aspirons à exécuter ces algorithmes sur le Web, nous passons en revue les technologies passées et courantes fournissant des bonnes performances directement au sein du navigateur Web. En particulier, nous détaillons comment cibler une nouvelle plateforme appelée WebAssembly à partir des langages de programmation C++ et Rust. Notre bibliothèque a été implémentée entièrement dans le langage de programmation Rust, ce qui en a facilité le portage vers WebAssembly. Cette propriété nous a permis de mettre en place une application Web d'odométrie visuelle proposant différents types d'interactions. Une barre de temps permet une navigation unidimensionnelle le long de la séquence vidéo. Des paires de points peuvent être sélectionnées sur deux images de la séquence pour réaligner les caméras et corriger l'éventuelle dérive. Des couleurs sont également utilisées pour identifier des parties sélectionnables du nuage de points 3D pour réinitialiser les positions de la caméra. La combinaison de ces interactions permet d'apporter des améliorations sur les résultats du suivi et de la reconstruction du nuage de points 3D.