SmartCanvas:
a Drawing Assistance Application for Android Devices

Basile Maret Simon Scherrer

Distributed Systems Lab, Spring Semester 2017

Supervisors:
Pankaj Khanchandani
Prof. Dr. Roger Wattenhofer

Distributed Computing Group
ETH Zurich

July 31, 2017

Abstract

In recent years, modern hand-held devices have begun to offer a wide range of multi-media applications that have long been deemed computationally too expensive for such machines. While increasing computation speeds have been utilized for widely-used *entertainment* applications of various forms, leveraging this power for user *enhancement* remains a challenge and has yet to produce applications that are similarly popular.

In this project, we devised, implemented, and evaluated an application that aims to enhance a user's drawing capabilities. SmartCanvas, as we name the application, pursues the goal of assisting a user in drawing an arbitrary image onto an arbitrary medium, e.g., a sheet of paper or a whiteboard. Assistance shall be provided by overlaying an image on live camera recording. Dynamic position adjustment of the overlay should then have the effect for the user that the image appears as being projected to a fixed position on the medium. Given this illusion, the user can then trace features in the projected image to craft a drawing.

Numerous challenges had to be overcome in order to make the application usable in a simple, efficient, and pleasant manner, most importantly performance and stability of the camera frame processing. By building on GPU acceleration and heavily optimized image processing algorithms, however, we achieved to develop an application that makes the drawing process both faster and more precise for a user with little experience in drawing.

Contents

Chapter 1

Introductory remarks

In this chapter, we present some introductory considerations that were made in advance to the development of the application. In Section 1.1, we present the motivation and the general idea behind SmartCanvas. In Section 1.2, the design goals for the application are outlined. The requirements which have to be observed in order to achieve these goals are listed in Section 1.3.

1.1 Introduction

In recent years, increasing speeds and decreasing costs in hardware of handheld mobile devices have given rise to a wide variety of new applications. Also tasks that have long been deemed computationally too expensive for mobile devices can now be carried out by standard applications. This is for example demonstrated by applications like video live-streaming from a device, off-line image editing and especially analysis and modification of live camera recording. Regarding the latter functionality, applications like the popular application *Snapchat* [7] allow a user to detect features and apply filters on camera frames in real-time.

It is an interesting and open challenge to use these relatively new capabilities of mobile devices not only for *user entertainment*, but also for *user enhancement*, i.e., providing assistance to a user who performs tasks in the physical world. An obvious example of such user enhancement is to support a user in crafting a product, where the support is offered by displaying visual instructions. Thereby, precision, proportionality, and plan conformity of the final product should be improved. Furthermore, the user should also learn from the hints provided by the application and become better at crafting the product even without the assistance application.

A possible crafting activity in which an application could assist is given by *drawing*, which is also the goal of the application presented here. The following two sections state more precisely how this assistance should be provided.

1.2 Goals

The application developed in this work aims at supporting the user in crafting a *drawing*, i.e., an image drawn by hand onto a physical object like a piece of paper or a whiteboard (in the following called the *medium*).

By projecting an arbitrary image onto the medium, the application should allow the user to copy the image. The projection should be performed by overlaying a semi-transparent version of the image on frames coming from a camera in real-time. In projection, it is crucial that size, position, and perspective of the overlay image are continuously adjusted when the user moves the camera. This dynamic adjustment should create the illusion that the image sits on a fixed position on the medium (cf. Figure 1). This is necessary such that the user can resume drawing an image from any perspective.

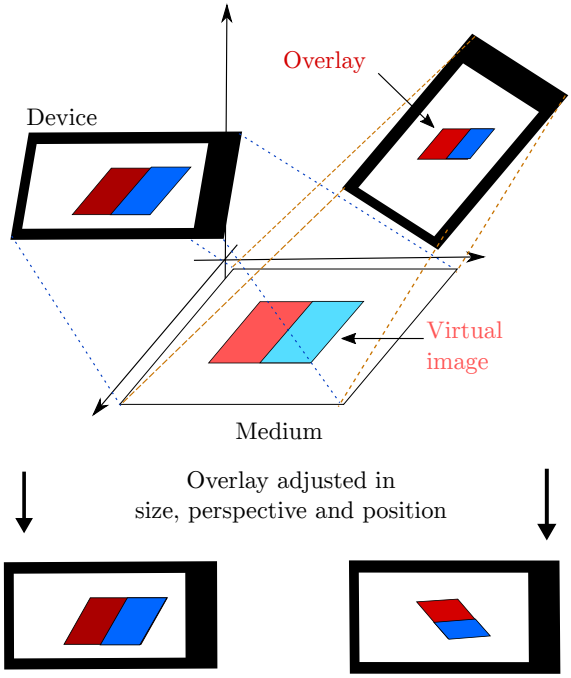

Figure 1: Illustration of overlay adjustment in size, position and projection such that the projection always stays in the same place for the user

The user should be able to mark on the medium where the image should be projected to, e.g., by marking the corner positions. Given a stable projection to this position in physical space, the user can then copy the image or certain of its features by tracing edges of the overlay with a pen. Since the visibility of the user's hand and the drawing pen is essential for eye-hand coordination, the overlay image is semi-transparent.

Furthermore, the user should be able to choose an arbitrary image from her device and preprocess it in a way that simplifies drawing, e.g., by highlighting edges.

1.3 Requirements

In this section, we outline the requirements with which the application has to conform in order to provide the functionality described in Section 1.2. The two main requirements that shaped the design decisions in this work were *usability* (Section 1.3.1) and *compatibility with widely-used platforms* (Section 1.3.2).

1.3.1 Usability

In order to helpfully support the user in drawing, *usability* is essential. Therefore, the usability of the application is also the central objective that was pursued in the development of SmartCanvas. In this context, an application is usable if it can be used in a simple, efficient and pleasant manner. The following subrequirements have to be respected in order to enable trouble-free utilization of the application

- **Real-time frame processing:** In image projection, it is of paramount importance that the modified camera recording can be displayed smoothly. For this to hold, the number of frames per second must be high enough such that the human eye can not distinguish individual frames. The computation per frame must thus be performed as efficiently as possible, which has implications for both the amount (minimal) and the mode (by means of specialized hardware architectures like a GPU) of frame processing.
- **Stability of projection:** In order to help the user improve her precision in drawing, the application itself has to be precise in projection. The limiting corners for the image have to be localized reliably such that the overlay image is always projected to the same position on the medium. If precision and continuity of edge detection are guaranteed, the user can change perspective without provoking distortions or flickering in the image.
- **Simplicity:** Since the goal of the application is to simplify drawing, learning to operate the application itself should be as simple as possible. Both picking the overlay image and retracing the image edges should be feasible intuitively. A potential source of operating complexity could stem from the problem that the drawing hand may cover the limiting corners of the projected image, thus making the projection less stable. Allowing the user to cover edges improves simplicity.

1.3.2 Platform compatibility

A secondary requirement that influenced the design of SmartCanvas deals with the question of the platform on which the application should be available. In-

spired by the numerous similar applications available for standard smartphones, we also developed the application at hand for the very widely-used Android platform. Special devices like devices with *Tango* enhancements [6] might offer helpful functionality for building the application. However, such devices are not widespread consumer products and thus not desirable according to our requirement.

Chapter 2

Used technology

In this chapter, we mention the technology components upon which the developed application builds. Technology components here mean libraries, implementation techniques and frameworks that we used for SmartCanvas without developing them ourselves. To summarize, the frame processing logic of SmartCanvas builds heavily on the *OpenCV* library (Section 2.1), into which GPU acceleration of the *OpenCL* framework (Section 2.2) was integrated. The frame processing was embedded into the application through the *Java Native Interface* (JNI, Section 2.3).

2.1 OpenCV

OpenCV (Open Source Computer Vision Library, [9]) is an open-source project which contains over 2500 algorithms for computer vision and image processing. It has interfaces to numerous programming languages, amongst others also C++ and Java, which were used in the development of the application at hand.

OpenCV represents images as matrices with color values as matrix entries (potentially multi-dimensional). We apply feature detection algorithms (see Section 3.3), image processing operations (see Sections 3.2 and 3.5) and matrix transformations (see Section 3.5), which are offered by the OpenCV library and take such image matrices as inputs.

2.2 OpenCL

By default, OpenCV performs matrix operations by means of the CPU. However, regarding speed, especially image processing operations can profit massively from highly fine-grained parallelization as it is done in a Graphical Processing Unit (GPU). Matrix addition can serve as an illustration: theoretically, all element-wise additions can be executed at the same time due to the lack of result interdependence.

Programming a GPU to absorb such inherently parallelizable workloads can be done by means of *OpenCL* (Open Computing Language, [3]). OpenCL perceives the GPU as a collection of multiple *processing elements* (PEs), each of which have local memory and can execute a *kernel* independently from other PEs. By writing appropriate kernels and performing data assignment to PEs, substantial speed-up can be obtained via OpenCL.

OpenCV provides a *Transparent API* (T-API) to OpenCL, meaning that GPU acceleration through OpenCL is automatically used if enabled and available. Enabling is necessary for the following reason: Although OpenCV implements numerous OpenCL kernels that achieve GPU acceleration of its algorithms, these kernels are not part of the OpenCV library by default. Instead, the OpenCV library with OpenCL kernels has to be compiled manually with an additional compiling flag set.

Further requirements have to be observed such that the transparent API to OpenCL can be used:

- **OpenCL library:** The device-specific OpenCL library (`libOpenCL.so`, usually located in `/system/vendor/lib/` on the device) has to be linked to the application.
- **Matrix data type:** A special matrix data type has to be used in code (`cv::UMat` instead of `cv::Mat`).
- **GLES and EGL:** If OpenCL should have access to the graphics subsystem of Android, both the Android rendering library OpenGL for Embedded Systems (GLES, [4]) and the EGL API to the Android windowing system [2] have to be linked to the application. Furthermore, an *OpenCL context* has to be initialized (cf. `app/src/main/cpp/CLprocessor.cpp` in project code). In the application, this so-called *CL-GL sharing* is used to obtain the current camera frame in an efficient manner.

2.3 Java Native Interface (JNI)

Since the OpenCV library is written in C and C++, there have to exist C bindings for a command when using OpenCV in Java, which is the main language in application development for Android. As OpenCL functionality is not included in OpenCV by default, there are no interfaces that would enable using OpenCL from Java code. All parts of the code for which GPU acceleration was desirable thus have to be written in C++ itself.

Source code in C or C++ can be integrated into an Android Java application via the *Java Native Interface* (JNI, [8]). For every Java method specified as *native*, a C++ method with a matching name has to be implemented. A library can then be built from the C++ source code by means of the CMake build tool (integrated in Android Studio). This library then has to be loaded in the Java code at runtime such that the native C++ methods are found. In the project code, the relevant files on both sides of the interface are `app/src/main/cpp/jni.c` and `app/src/main/java/ch/ethz/dslfs17/smartcanvasocl/NativePart.java`.

Chapter 3

Application design

In this chapter, we describe the design of our application in detail. Each of the following sections will focus on a set of design decisions that were taken to solve aspects of the problem at hand. Section 3.1 will give an overview of the application workflow and its structure. Section 3.2 will describe which kind of preprocessing is necessary for the user-chosen image. Section 3.3 aims at explaining the algorithms for detecting the edges, which are then later structured by a process that we call 'edge localization' (Section 3.4). Finally, Section 3.5 concludes the chapter by presenting the programming logic needed to overlay the image on the frame.

3.1 Overview

Figure 2 displays the workflow of SmartCanvas. The application starts with a plain Android main activity, which conveys the user to an activity where the image to draw can be selected. In that activity, the images that have been previously selected by the user are displayed and can thus be reselected directly. If the user wants to select a new image, she can do so from various sources like the device camera, the file system, a cloud, and more.

After selecting the image, the user can crop and rotate it. This is helpful if the user only wants to draw a certain part of the original image.

In a next step, the user can then apply different filters on the image. The image filters, which are borrowed from the Android GPUImage library [1], include cartoonization and edge-highlighting filters. These filters all aim at simplifying the image and thus the drawing process. The filters are described more extensively in Section 3.2.1.

Since the image may be rich in detail and contain fine-grained structures, it makes sense to focus only on parts of the image at a time. This can be achieved by *segmenting* the image, which can be done in the subsequent step of the workflow. The user can arbitrarily split the image into segments with image-spanning lines in horizontal and vertical orientation. Section 3.2.2 explains this

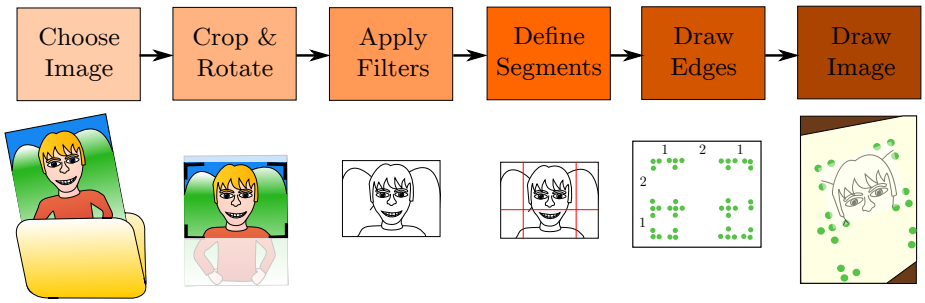

Figure 2: Illustration of application workflow

segmentation process in closer detail.

The image with applied modifications is then saved to a folder for the recently used images. This persistence enables the re-use of previously selected images, as is mentioned above. This functionality is useful when a user wants to continue drawing an image that she started drawing in a previous execution of the application. In order to avoid distortions in the drawn image, the image has to be exactly the same in both executions. Saving the modified image ensures this equality.

After defining image segments, the preparation of the image is concluded. However, the medium also has to be prepared: The edge points of all image segments have to be marked. The instruction step of the work-flow helps in this preparation: It presents the constellation of the edge points to be drawn, especially the distances between them.

In the final and most important step of the work-flow, the actual projection of the image onto the medium is performed. This projection is based on the repeated execution of edge detection (Section 3.3), edge localization (Section 3.4) and image overlay (Section 3.5).

3.2 Image preprocessing

In this section, we explain how the non-trivial parts of image preprocessing are implemented, i.e., image filters (Section 3.2.1) and image segmentation (Section 3.2.2), as opposed to image cropping and rotation, which have a straightforward implementation.

3.2.1 Filters

When drawing, it is often difficult to know which edge or which line to draw, since many images rather consist of shades and gradients than lines. This observation is particularly true for photographs. In order to help the user, SmartCanvas offers a set of filters that simplify the image by emphasizing edges. The available filters are *Sketch*, *Toon*, and *Smooth Toon*. All of them use a Sobel

operator to highlight edges in an image. The Sobel operator performs two spatial gradient measurements on the image to detect the changes in intensity. A measurement G_x measures the changes on the horizontal axis, the other measurement G_y measures them on the vertical axis. The gradient measurements G_x and G_y are each performed by convolving a 3×3 kernels with the original image:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A,$$

where A is the original image and $*$ is the convolution operation. The two gradient measurements can then be combined into a gradient magnitude $G = \sqrt{G_x^2 + G_y^2}$ which measures the change in intensity in both directions.

The *Sketch* filter replaces the colors of the source image with the value of the gradient magnitude at each pixel, ranging from black for a magnitude of 0 to white for the highest magnitude of the image, in a grayscale. The colors of the resulting image are then inverted for the edges to be black and the background to be white.

The *Toon* filter works similarly to the *Sketch* filter but colors the pixels either in black or white instead of a grayscale. The filter uses a threshold to make the decision for the gray pixels. Once the edges are black and the background is white, the filter fills the white spaces between the edges with a quantized set of colors from the original image.

The *Smooth Toon* filter is the same as the *Toon* filter but a Gaussian blur is applied before the filter.

All of these filters use the implementations from the GPUImage library [1].

3.2.2 Segmentation

In order to simplify and improve the sketching of the image, conducting a segmentation can help in two major aspects. First, if the image is split into segments and these segments can be used individually, the user only has to focus on a single image part at a time. Second, if only a single image segment is displayed, the details in that segment can be represented in a larger fashion, making it easier to trace them. This is especially relevant in the face of our usage requirement that the edges have to be visible while drawing: Fitting a large image into a single set of edges may result in projected image features that are too fine to trace.

In image segmentation, the challenge arises that there is no single best segmentation for all images. Which segmentation is suitable for an image depends heavily on the desired quantity of detail in every segment and the presence of contiguous image features, which should not be spread over multiple segments.

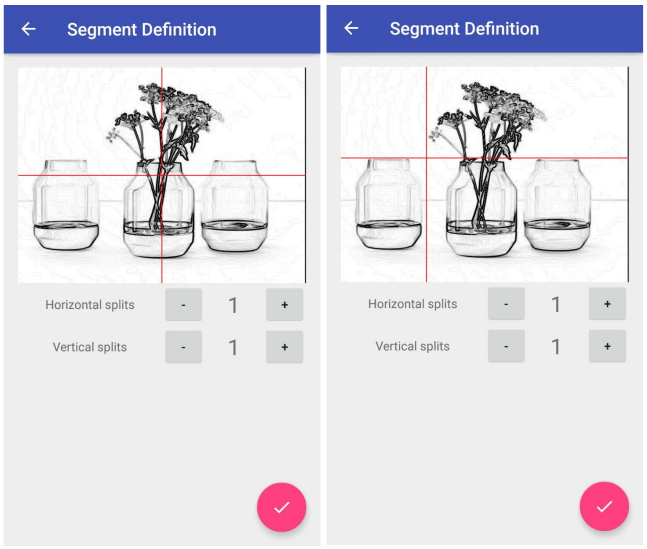

Figure 3: Screenshot of the application's segment definition functionality (left: default, right: arbitrary segmentation achievable by line dragging)

The application thus offers the possibility to perform an arbitrary segmentation of the image by means of image-spanning horizontal and vertical lines. These splitting lines can be defined by the user in `SegmentDefinitionActivity`. As perceivable in Figure 3 on the left, this activity first gives the user a choice by how many lines each dimension should be split. The application then places these lines on the image in equal distance from each other. To further customize the segmentation, the user can then drag these lines to the desired position (as on the right in Figure 3). The offsets of these lines are then saved and propagated to subsequent steps in the application workflow.

One such subsequent step is the `InstructionActivity`, which instructs the user how to draw the edge points onto the medium. The user can prepare the medium by copying the required formation of edge points. In order to guarantee correct ratios between segment sizes, the side lengths of segments are displayed as part of the formation. These side lengths, given in centimeters, are calculated on the basis of the image's larger side length, chosen by the user (cf. Figure 4 on the left).

As soon as the application starts its drawing mode, the image is split according to the line offsets. The produced sub-images are then kept in memory. In drawing mode, the user can pick the segment that she wants to draw next by tapping on the appropriate segment on the *segment panel* (highlighted in Figure 4 on the right). This segment panel can be made visible through a setting's option and disappears five seconds after it was last touched.

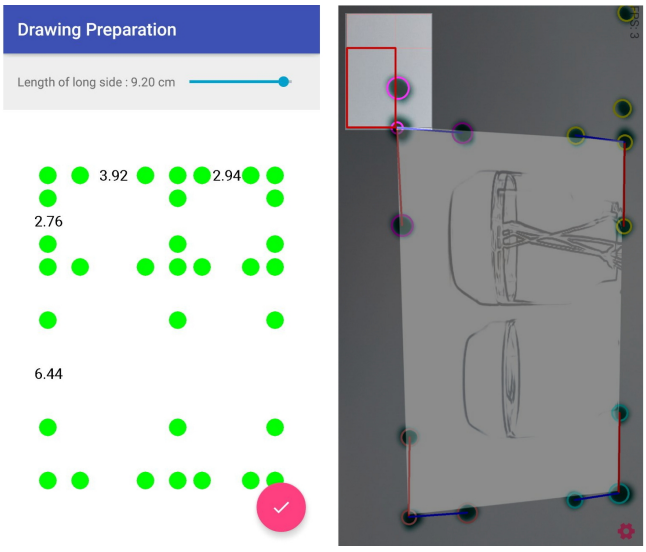

Figure 4: Screenshot of the `InstructionActivity` (left, with the edge point formation defined by segmentation) and drawing mode (right, with highlighted segment panel)

3.3 Edge detection

The following section will elucidate the process by which the edge points in the camera frame image are detected. This process is based on two major components, namely HSV color filtering (explained in Section 3.3.1) and Hough circle detection (explained in Section 3.3.2). The goal of the edge detection process is to produce a complete and sound list of all edge points in the frame image, given by their coordinates. These points are then structured and sorted by the process of edge localization (see Section 3.4 below).

3.3.1 HSV color filtering

False positives are a substantial challenge in edge detection: A simple detection algorithm could mistake numerous features in the frame image for an edge point. Examples include features of the partially drawn image itself or features on the user's hand. Furthermore, even if false positives could be reduced by some algorithm, the respective algorithm would have to judge every edge point candidate, which makes per-frame processing computationally more expensive and decreases the smoothness of the camera preview.

In order to avoid slow or even incorrect edge detection, one partial solution is to give a suitably preprocessed input to the edge detection process itself. HSV color filtering is such a suitable method, which builds on the assumption that the user marks the edge points in a distinct color, i.e., a color different from

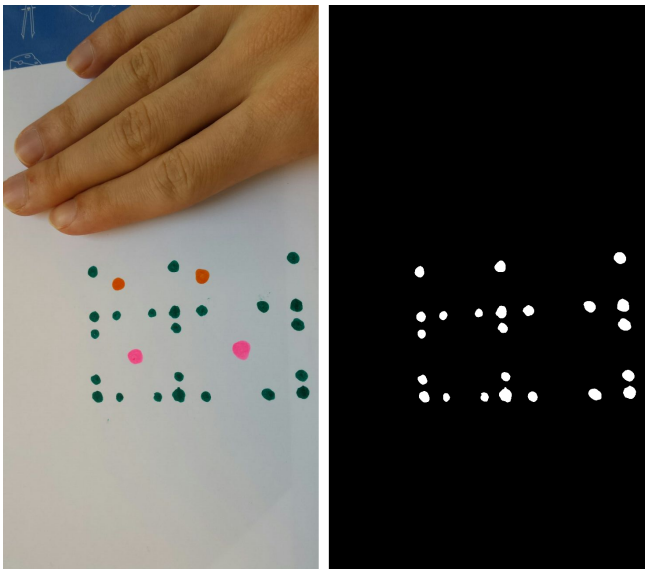

Figure 5: Effect of the HSV color filtering: A frame image with many distracting elements (left) can be translated into a simplified image with far more easily detectable edge points (right)

colors in the image itself, on the user's hand, and on other distracting elements.

The general idea of HSV color filtering is to detect all pixels in the frame that are reasonably similar to a certain color, i.e., within a certain distance in the HSV color space. These pixels can then be set to white, whereas all other pixels are set to black. Figure 5 serves as an example. Then feeding this simplified image to Hough circle detection (see next section) makes the edge detection process both faster and more precise.

The target color can be defined by the user through the application settings. As every color in the HSV color space, it is determined by the following three parameters: first, a *hue* parameter, which identifies a certain combination of three primary colors (red, green, blue); second, a *saturation* value, which identifies a combination of the hue with white; and third, a *value* parameter, which identifies a combination of the hue-saturation color with black.

In the implementation, the OpenCV method `inRange()` is used to detect all pixels with a color between (H_{min}, S, V) and $(H_{max}, 255, 255)$, where H_{min}, S, V and H_{max} are selected by the user. Intuitively, this range corresponds to the space between a color mixed from primary colors and a less intense and darker version of this color.

3.3.2 Hough circle detection

Similar to edge points having a specific *color*, it is advantageous in edge detection if edge points have a specific *shape*. If the edge points are restricted to a certain

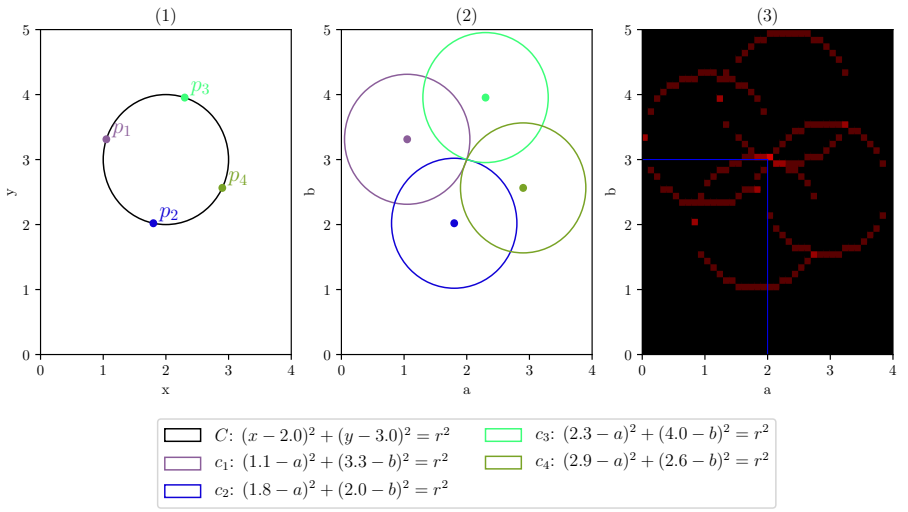

Figure 6: Visualization of Hough circle transform

shape, the edge detection can avoid using generic feature detection algorithms, e.g., SIFT or SURF. Both these algorithms have major drawbacks concerning our performance requirement. The SIFT algorithm is not even designed to run in real-time, as in live per-frame processing. SURF, in fact, is meant to deliver good results in real-time. However, it is not available in the GPU-optimized version of OpenCV due to patent protection, which blocks a valuable option for optimization.

Instead of using generic feature detection algorithms, we can resort to shape-oriented detection algorithms. Such algorithms exist for almost every simple shape, are usually provided by OpenCV in a heavily optimized form, and thus have a runtime that fits the real-time requirement well. We decided to use *circles* as a simple shape that is suitable for edge points. The application can detect these circular edge points by the highly efficient *Hough circle transform*, which we will briefly illustrate by means of Figure 6.

We assume that a circle C, as given in subplot (1) of Figure 6, is to be detected. Furthermore, we assume that the radius of C is known to be r. With C formalized as

$$C(x,y): \quad (x-a_C)^2+(y-b_C)^2=r^2, \tag{1}$$

,

the Hough transform has to find the center (a_C, b_C) of C.

The Hough transform starts by picking random points p_i, $i \in \mathbb{N}$, from the circle edge, which can be found on the basis of color differences. The coordinate pairs (x_i, y_i) of all p_i satisfy Equation 1. Replacing the unknowns a_C and b_C then by variables, every point p_1 defines a circle in the parameter space of

Equation 1, given by

$$c_i(a,b): \quad (x_i - a)^2 + (y_i - b)^2 = r^2. \tag{2}$$

These circles c_i are depicted in subplot (2) of Figure 6. As can be noticed from this subplot, the intersection of all circles c_i represents the desired circle center (a_C, b_C), since this is the only point that is in distance r of more than two arbitrarily chosen points on the circle.

This intersection is then found by splitting the parameter space into *accumulator cells*, as shown in subplot (3) of Figure 6. Every circle with a line passing through a specific accumulator cell contributes to that accumulator cell. In (3), the red-intensity of a cell represents the value of the local accumulator cell. The accumulator cell with the maximum value is then the cell which contains (a_C, b_C). In the numerical example of Figure 6, this is the cell at position (2,3), which is also the center point of circle C in subplot (1). We thus detected circle C correctly.

If r is not known, the Hough transform can be easily extended to include r as an additional dimension in the parameter space. Instead of circles, the points p_i then define *cones* with an intersection to be found. The OpenCV function `cv::HoughCircles()`, upon which SmartCanvas builds, expects a minimum and maximum radius, thus constraining the radius dimension of the parameter space.

In a frame image that is color-filtered according to Section 3.3.1 and scaled down to a smaller size, the circular edge points can be detected within 20-40 milliseconds. This time complexity does not destroy the smoothness of the camera frame sequence and thus conforms to the performance requirement. As output of edge detection, the application produces a list of circle center coordinates. These coordinates are then interpreted by the process of edge localization, which is explained in the following section.

3.4 Edge localization

In this section, we describe how the detected edge points are *interpreted* by the application. Interpretation is here understood as finding out how an edge point should influence the image projection. The function of an edge point is determined by its location in the frame image. For example, if an edge point is classified as the upper left edge, its function is to indicate where the upper left corner of the overlay image should be projected to. Since determining the relative locations of all edge points is key to interpretation, this process is henceforth referred to as *edge localization*. The goal of edge localization is to discover the upper left edge, the upper right edge, the lower left edge, and the lower right edge, even in the presence of detection errors and invisibility of edge points.

In the following subsections, we refer to the edge locations, e.g., upper left, as *ordinal directions*, a term borrowed from geography where it denotes the compass points northwest, northeast, southwest, and southeast.

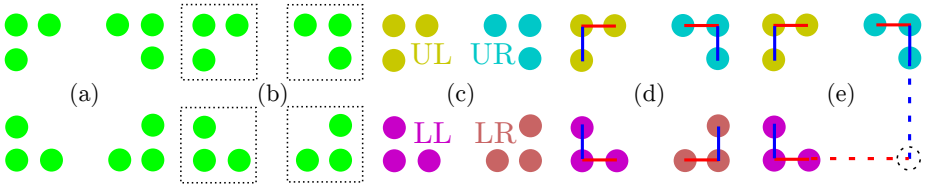

Figure 7: Stages of the edge localization process

Figure 7 shows the stages the designed edge localization process. In stage (a), the initial stage of the process, only the coordinates of the edge points are known. These edge points, which were found in the edge *detection* process (cf. Section 3.3), are then divided into groups by the subprocess of *edge group clustering* (Section 3.4.1). In a next step, the produced edge groups in stage (b) are locationally sorted both among each other (*inter-group sorting*, stage (c), Section 3.4.2) and within edge groups (*intra-group sorting*, stage (d), Section 3.4.3). The goal of *inter-group sorting* is to distinguish edge groups into the ordinal directions, e.g., upper left. In contrast, the goal of *intra-group sorting* is to distinguish different types of edge points within an edge group. If stage (d) can be reached for three edge groups but fails for the last one, the fourth edge can be predicted as visualized in stage (e) and described in Section 3.4.4. Finally, the new edge positions are computed as a weighted average of previous and current positions, where the respective weights depend on the camera motion (cf. Section 3.4.5).

3.4.1 Edge group clustering

In order to divide edge points into groups, a clustering algorithm in the form of the *k-Means algorithm* [5] is sufficient. The *k-Means algorithm* clusters elements together on the basis of Euclidean distance between them. Edge points with a similar location are thus perceived as belonging to one edge group, which is a reasonable assumption.

The general idea behind the algorithm is straightforward. The algorithm is initialized with a number k of clusters (here, $k = 4$) and a set of data elements (here, edge points). The cluster centers are initialized randomly. Then, the algorithm fits the clusters to the data by alternating between the *assignment step* and the *adjustment step*. In the assignment step, all data elements are assigned to the cluster with the nearest center. In the adjustment step, every cluster center is recomputed as the mean of all data elements that are assigned to the respective cluster. This alternation stops if assignments do not change anymore or if a fixed maximum number of iterations is reached. Part (a) of Figure 8 illustrates an example run of the k-Means algorithm.

OpenCV implements the *k-Means algorithm* in the method `cv::kMeans()`. This method returns both the cluster centers as coordinates and the cluster labels for all input points.

Since the number of clusters k has to be defined in advance, a problem as

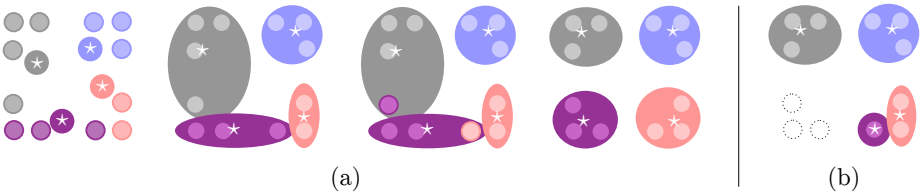

Figure 8:
(a) Example illustration of k-Means algorithm convergence
(b) Algorithm-specific problem arising if an edge group is invisible

depicted in part (b) of Figure 8 arises if an edge group is not visible: The algorithm then splits the actually existing three edge groups into four clusters. The application solves this problem by merging a collapsed cluster (a cluster with at most 1 assigned edge point) into the nearest other cluster *if* the other cluster is within a certain short distance.

3.4.2 Inter-group sorting

Once the edge group clustering process as described in the previous section has produced the edge groups, these edge groups need to be localized. The application has to find a bijection between detected edge groups and ordinal directions. We refer to this process as *inter-group sorting*.

Inter-group sorting is performed by running a simple *scanline algorithm* over the detected cluster centers. For every ordinal direction, the cluster center with a minimal approximate distance to the respective corner is found, as depicted in part (a) of Figure 9. The approximate distance between the cluster center and the frame corner is computed as the sum of distances in both dimensions, i.e., the distance from the lower left corner at $(0, 0)$ is $x + y$, whereas the distance from the upper left corner at $(0, h_{frame})$ is $x + (h_{frame} - y)$, and so forth.

Similar to edge group clustering (3.4.1), a problem arises if a whole edge group is not visible. The problem is illustrated in part (b) of Figure 9. In the figure, one cluster center is assigned to two ordinal directions due to one edge group missing. If these assignments were adopted, the upper left and the lower left edge of the overlay image would be wrongly projected to the same location.

The application prevents such double assignments as follows. Each pair of ordinal directions is checked on whether they reference the same cluster center. If a double assignment exists, the assignment is only kept at the ordinal direction for which the cluster center is more *typical*, i.e., closer to the respective frame corner. In the example above, the cluster center would be assigned to the upper left ordinal direction.

3.4.3 Intra-group sorting

When entering the *intra-group sorting* step of the edge localization process, the application has already found groups of edge points and assigned these groups

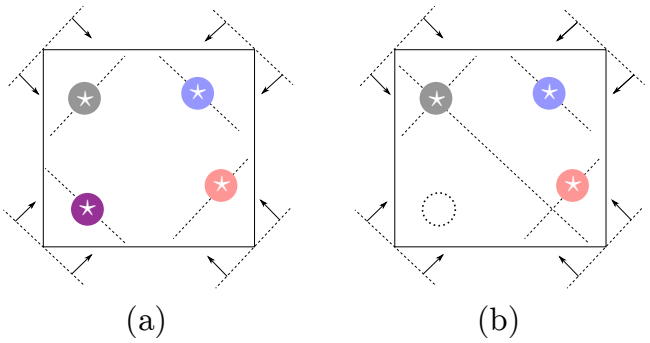

Figure 9:
(a) Illustration of scanline algorithm to find the minimal edge in all ordinal directions
(b) Algorithm-specific problem arising if an edge group is invisible

to the ordinal directions, e.g., one edge group may be 'upper left'.

In order to help with prediction of invisible edges, each such edge group does not only consist of one edge point (main edge point), but also has multiple *orientation* edge points in both the horizontal and vertical dimension. As described below in Section 3.4.4, the combination of the main edge point and a certain orientation point defines a line that can be used for prediction. The goal of intra-group sorting then is to discover the internal structure of an edge group. This goal is not trivial: an edge group consists of up to 5 edge points (cf. the internal edge in the edge point pattern of Figure 4 on page 12), of which not all may be even relevant for the current image segment. A multiple scanline algorithm thus has to be applied to classify all edge points according to their function.

We will now illustrate this within-group sorting process at the example in Figure 10 on the next page. It is assumed, without loss of generality, that the displayed edge group was assigned to the upper left ordinal direction. The displayed edge group consist of a main edge point e_m, a horizontal orientation point e_h, a vertical orientation point e_v, and two irrelevant orientation points e_i. Because of the classification as upper-left, we know that the edge point with the maximal x-coordinate is the horizontal orientation point and that the edge point with the minimal y-coordinate is the vertical orientation point. In step (b), these two points can thus be found by straightforward coordinate sorting. As to find the main edge point, the already localized orientation points have then to be ignored and a diagonal-scanline approach has to be applied, similar as in inter-group sorting. However, contrarily to inter-group sorting, the scanline now has to find the point which is closest to the corner of the *opposite* ordinal direction, i.e., here lower right. Step (c) thus delivers the main edge point. All remaining edge points can be ignored; they are guaranteed to be irrelevant. In step (d), the edge group thus can be reconstituted with all relevant edge points and their function.

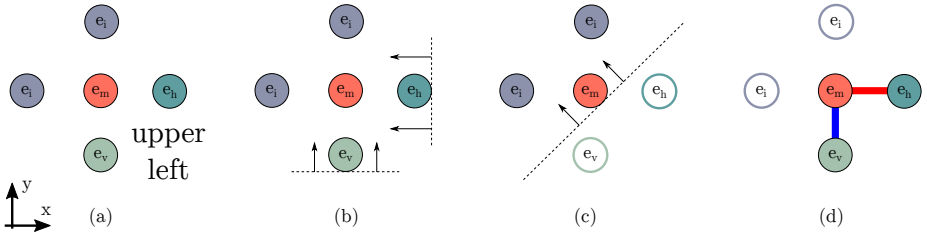

Figure 10: Illustration of intra-group sorting process

Note that it depends on the ordinal direction which horizontal, vertical and diagonal scanlines to use. These variations are analogous to the example presented in the previous paragraph.

3.4.4 Edge prediction

In edge detection, it may happen that an edge group is not detected, for several reasons: the edge group may be covered by the user's hand, it may be out of the frame, or it may simply not have been detected due to instability of the Hough circle detection (Section 3.3.2).

In these cases, it is necessary to *predict* the missing edge, i.e., to make an educated guess on where the missing edge is located, relying on other edge information from current and previous frames. Such information has to be provided in sufficient form. For example, the application is only capable of predicting one edge at a time. Three edge groups thus have to be completely detected and localized in order to predict a fourth one. However, if this condition holds, the requested edge group structure ensures that prediction is possible and accurate most of the time.

There are two base cases in the edge prediction algorithm of the application. We will now illustrate them by means of the example in Figure 11 on the following page. In the example, the missing main edge ur_m of the upper right corner has to be predicted.

In case (a) of Figure 11, not a single edge point of the upper right edge group has been detected. The missing edge point ur_m thus has to be predicted on the basis of neighbor edge groups. From the known fact that the missing edge is at the upper right corner, we get the upper left edge group as its *horizontal neighbor* and the lower right edge group as its *vertical neighbor*. Both these edge groups then define vectors. The intersection of these vectors is the prediction. In the horizontal neighbor group, the main edge and the horizontal orientation point define a horizontal orientation vector $\vec{o_h}$. In the vertical neighbor group, the main edge and the vertical orientation point define a vertical orientation vector $\vec{o_v}$. The goal is then to find $t, s \in \mathbb{R}$ such that:

$$\vec{o_h} = \overrightarrow{ul_m ul_h} \qquad \vec{o_v} = \overrightarrow{lr_m lr_v} \qquad t \cdot \vec{o_h} = s \cdot \vec{o_v} \tag{3}$$

However, as the user does not necessarily arrange the edge points in a strictly

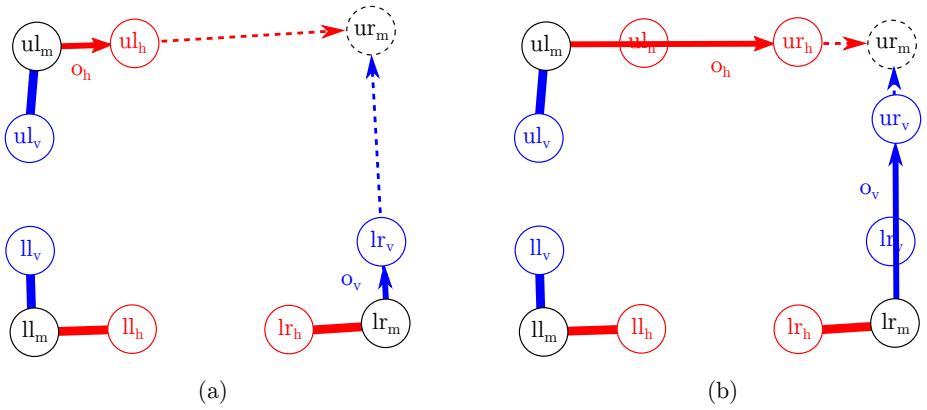

Figure 11: Illustration of the two base cases for edge prediction

rectangular fashion, the prediction in case (a) may not be optimally accurate. Due to the heavy stretching of the vectors, even a small angular deviation in the neighboring edge groups translates into a large deviation at the intersection. Both if the orientation points are inaccurately drawn or detected, these errors are multiplied.

Whenever possible, case (b) in Figure 11 should thus be applied in order to achieve better accuracy and stability of the prediction. Case (b) tries to avoid the source of this imprecision, namely the stretching of the vectors by large values for t and s. This large scaling can be avoided if the missing edge group is *partially* detected, i.e., orientation points have been detected, but not the main edge. If any orientation points of the missing edge group have been localized, the orientation vectors for the intersection are defined differently to case (a). The horizontal orientation vector $\vec{o_h}$ is defined by the main edge of the horizontal neighbor group and the horizontal orientation point of the *missing* edge group. The vertical orientation vector is spanned by the main edge of the vertical neighbor group and the vertical orientation point of the *missing* edge group. Now $t, s \in \mathbb{R}$ have to be found in the following setting:

$$\vec{o_h} = \overrightarrow{ul_m ur_h} \qquad \vec{o_v} = \overrightarrow{lr_m ur_v} \qquad t \cdot \vec{o_h} = s \cdot \vec{o_v} \tag{4}$$

Base cases (a) and (b) can also be combined when only one orientation point in the missing edge group has been localized. The orientation vectors are then defined differently, each according the base case which is feasible.

However, inaccurate edge point detection can also in case (b) lead to unacceptably large prediction errors: If edge point detection is unstable, the predicted edge may move around erratically, distort the image repeatedly, and therefore severely impede the drawing process.

In order to avoid this volatility problem, the predicted edge coordinates are never fully adopted. Instead, a moving average is computed for the predicted edge: The new edge position is determined by a weighted average of the old edge

position and the prediction. This measure has the following two advantages. First, if the respective edge has once been detected, the known edge position still influences the new edge position and likely makes it more accurate. Second, since detection errors are random, averaging tends to cancel them out.

3.4.5 Motion-aware moving averages

The same reasoning that makes the case for applying moving averages in edge prediction also justifies the application of moving averages for all edge positions: The instability of edge detection can lead to volatile edge positions, which would translate into a flickering projection image. Therefore, the application computes a moving average for every edge point position, which is a weighted average of the current edge position and the position of the corresponding edge point in the last frame. Note that this moving average can only be done at the end of the edge localization process; otherwise, finding the corresponding edge point from the last frame would not be possible.

If the device is stationary above the drawing, this moving average can compensate for unstable edge detection and trembling of the user's hand, which is desirable. However, when the device moves, applying a moving average can slow down the adjustment of the edge positions, which is undesirable. Hence, the application has to be aware of device motion in order to decide how the moving average should be applied. We thus call the mechanism by which edge positions are smoothed over time *motion-aware moving averages*.

Technically, this motion-awareness is implemented by listening to the *accelerator*, a device component that captures the device movement in space and is usable through a simple interface in Android. From the movement intensity, the weighted average coefficients are determined: If the movement is small, then large weight is placed on the old edge point position; if it is large, then the current edge position is integrated into the average with a larger weight. Through this mechanism, edge positions can be stabilized during stationary periods and quickly adjusted during device dislocation periods.

3.5 Image overlay

This section aims at explaining the process of overlaying the image on the camera frame, given the positions of four edge points. Since this overlay constitutes an image processing operation, it is the most expensive step in the per-frame operation pipeline and the limiting factor of the frame-processing speed in the application. Thus, special attention has to be turned to the efficiency of the image overlay.

In Figure 12, the image overlay process is illustrated. The process starts by obtaining edge positions from the edge localization process (Section 3.4 on page 15) and an image to overlay, which is the current image segment. From the edge positions, the application finds the limiting rectangle in the frame, which is defined by minimum and maximum edge point values in both dimensions.

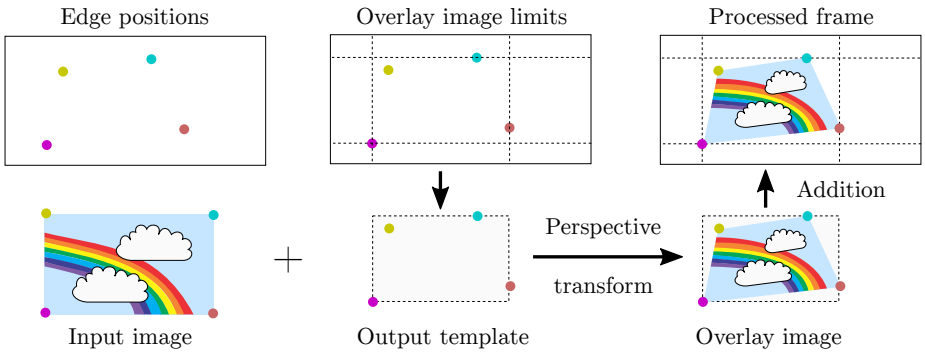

Figure 12: Illustration of the image overlay process

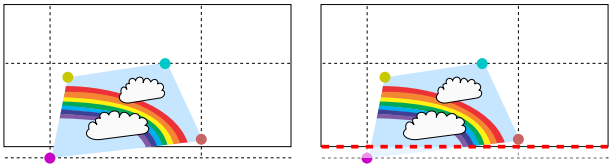

Figure 13: Illustration: Cropping the overlay image when predicted edge is outside of frame

The edge position values are then recomputed to match the coordinate system given by the frame sub-image.

The adjusted edge points then form the *output template*, meaning a set of points to which the corresponding image corners have to be mapped. This mapping is performed by finding a *perspective transformation* between the origin and the destination points (OpenCV method `cv::getPerspectiveTransform()`) and applying it (OpenCV method `cv::warpPerspective()`). The transformed image can be overlaid on the frame at the position of the previously found limiting rectangle (OpenCV method `cv::add()`).

Extracting the limiting rectangle from the frame is crucial, because working on a smaller sub-image speeds up the image overlay considerably. The image overlay works by plain matrix addition. If the overlay image had the same size as the frame image, two larger matrices would have to be added, which would be computationally more complex. Furthermore, doing so would serve no purpose, since the overlay image matrix would contribute a summand of 0 at many indices. Additional optimization is achieved by preprocessing the image in a way that makes it look semi-transparent when overlaid. This preprocessing saves the complexity of multiplying the overlay image by an opacity factor in every frame (as in OpenCV method `cv::addWeighted()`).

One more intricacy of the overlaying process consists in the handling of edge points that are outside of the frame image. Such edge points are always the result of an edge prediction (see Section 3.4.4 on page 19). Given such an

out-of-frame edge, simple addition to a submatrix of the frame image matrix is infeasible: The out-of-frame edge point would result in an invalid row or column range for the submatrix. However, the solution to this problem is straightforward: If the limiting rectangle overshoots the frame image, both the limiting rectangle and the overlay image are cropped such that they fit into the frame image (cf. Figure 13).

Finally, it is possible that in a single frame, edges can not be localized. However, simply abstaining from image overlay in these cases would lead to unpleasant flickering of the projection. In order to smooth the projection display, the last transformed image is thus overlaid even if no edges have been localized. The cached transformed image is projected to the last known position until (1) new valid edges are found, (2) no valid edges have been found for a certain number of frames, or (3) the user clears the cache by tapping on the device screen.

Chapter 4

Evaluation

In this chapter, we will evaluate the application in its final state according to the requirements outlined in Section 1.3, namely usability and its sub-aspects (cf. Section 4.1) and compability with widely-used platforms (cf. Section 4.2). In Section 4.3, we will present examples of drawings which have been produced with the aid of SmartCanvas.

4.1 Usability

It is central that the application can be used in simple, efficient, and pleasant manner. Multiple sub-requirements had to be observed in order to provide good *usability*. In the following, we will briefly state if and how each of these sub-requirements has been fulfilled.

- **Real-time frame processing:** Thanks to highly optimized algorithm both in OpenCV and our application code as well as GPU hardware acceleration by means of OpenCL , the application achieves to display 11-14 frames per second. This frame rate is sufficient to make the processed camera recording look smooth.
- **Stability of projection:** There are numerous challenges in guaranteeing guarantee a stable projection of the image onto the medium. These challenges include unstable edge detection, distracting influences in the frame, trembling of the user's hand, and edge points moving out of the frame. By applying a bundle of measures, however, these challenges could be overcome in a satisfactory manner: Distracting influences can be neutralized using HSV color filtering (Section 3.3.1), unstable edge detection and hand trembling can be compensated for with motion-aware moving averages (Section 3.4.5), and the problem of edges moving out of the frame can be addressed by predicting edges (Section 3.4.4). Given that corner points have been arranged in the required formation, we consider the image projection stable enough to trace lines in the image.

- **Simplicity:** Operating the application is simple in the sense that picking and preprocessing the image are intuitively feasible (Section 3.2). From our own experience, however, we believe that a certain amount of practice is required to craft satisfactory drawings with the application. The user has to learn how to perform an appropriate segmentation of the image, how to draw corner points in an exact fashion, and how to avoid covering multiple corner points while drawing. In order to reduce operating complexity, future work on the application should address these complications.

4.2 Platform compatibility

With the requirement of compatibility with widely-used platforms, we formulated a desire to avoid basing our application on special hardware architectures and special devices (cf. Section 1.3.2). Instead, we wanted SmartCanvas to be usable on every device that is capable of running the Android OS and possesses a camera.

To run the application at a pleasant performance, the device has to include a GPU which can be programmed using the OpenCL framework. Since all mobile processors to our knowledge offer this possibility, we consider this requirement fulfilled.

4.3 Example drawings

In this section, we present Figure 14, Figure 15, and Figure 16 as proof of the drawing capability enhancement through SmartCanvas. The example drawings show that both cartoon and photo portraits can be drawn with the assistance of the application. Table 1 presents details about the individual drawings.

Figure	**Motif**	**Format on medium**	**Drawing time**
Figure 14	Donald Duck	20cm × 14cm	30 min
Figure 15	Autoportrait	9cm × 6cm	20 min
Figure 16	Lino Guzzella	15cm × 11.7cm	40 min

Table 1: Details about example drawings

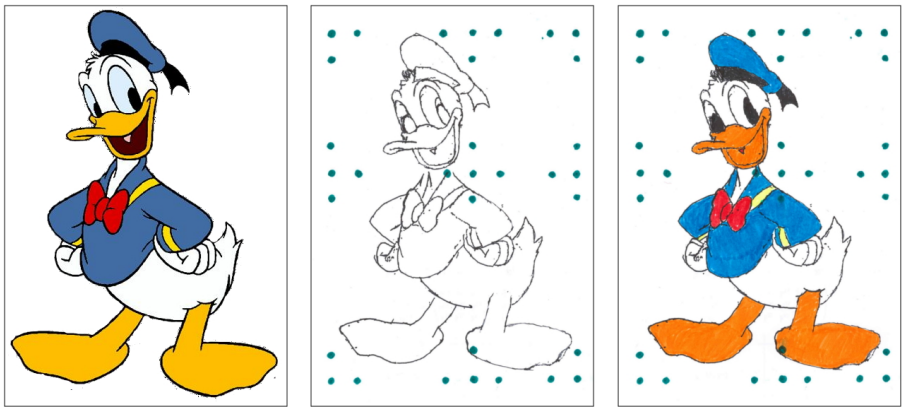

Figure 14: Cartoon picture drawn with the aid of SmartCanvas

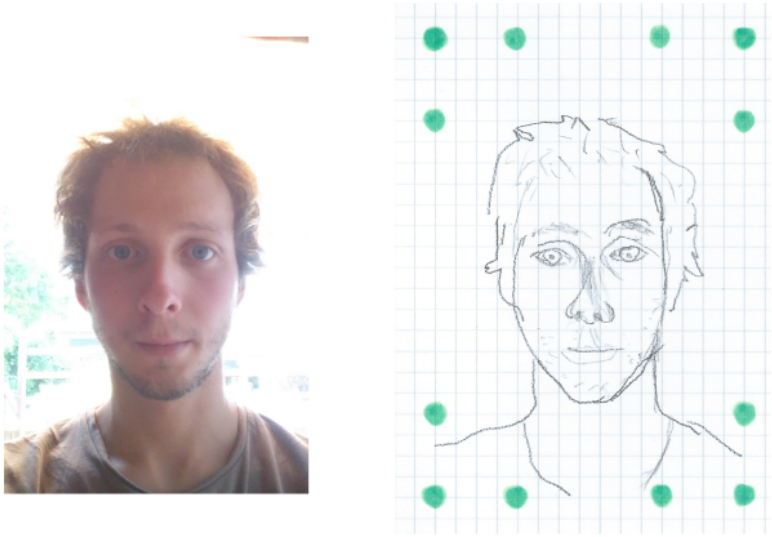

Figure 15: Autoportrait drawn with the aid of SmartCanvas

Figure 16: Photo portrait drawn with the aid of SmartCanvas

References

[1] CyberAgent. GPUImage for Android. URL: https://github.com/CyberAgent/android-gpuimage.

[2] Khronos Group. EGL (Native Platform Interface). URL: https://www.khronos.org/egl/.

[3] Khronos Group. OpenCL. URL: https://www.khronos.org/opencl/.

[4] Khronos Group. OpenGL for Embedded Systems (GLES). URL: https://www.khronos.org/opengles/.

[5] John A Hartigan and JA Hartigan. *Clustering algorithms*, volume 209. Wiley New York, 1975.

[6] Google Inc. Project Tango. URL: https://developers.google.com/tango/.

[7] Snap Inc. Snapchat. URL: https://www.snapchat.com/.

[8] Oracle Inc. *Java Native Interface - Specification*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html.

[9] OpenCV team. OpenCV. URL: http://opencv.org/.