SketchADoodle: Engineering Touch-surface Multi-stroke Gestures by Bézier Curves

DONATIEN GROLAUX, ICHEC Brussels Management School, Université catholique de Louvain, Belgium
JEAN VANDERDONCKT, Université catholique de Louvain, Belgium
THANH-DIANE NGUYEN, ICHEC Brussels Management School, Belgium
IYAD KHADDAM, Université catholique de Louvain, Belgium

Touch-surface multi-stroke gestures, as well as freehand drawings, are typically acquired by devices and sensors as a suite of timestamped points on a plane. This Cartesian coordinate system, although useful for computation like complexity analysis, gesture classification and recognition, becomes complex and inefficient when gestures need to be visualized and directly manipulated for editing. To address these challenges, a new mathematical representation of these gestures via a Bézier curve is defined to initiate a model-based approach for gesture direct manipulation (*e.g.*, cut, copy, paste, translate, scale, rotate, deform, crop, compose, decompose). SketchADoodle, an Android-based mobile application for drawing, gesturing, demonstrates how the pseudo-code of the Bézier-based operations are engineered for real-time direct manipulation. We release the programming code for further development of gesture-based user interfaces based on Bézier curves.

CCS Concepts: • **Human-centered computing** → **Gestural input**; **Graphical user interfaces**; *Interactive systems and tools*; • **Software and its engineering** → **Graphical user interface languages**; Runtime environments; • **Computing methodologies** → *Model development and analysis*;

Additional Key Words and Phrases: Bézier curve, bitmap-based drawing, control point, direct manipulation of gestures, gesture recognition, multi-stroke gesture, uni-stroke gesture, stroke gesture, vector drawing.

ACM Reference Format:
Donatien Grolaux, Jean Vanderdonckt, Thanh-Diane Nguyen, and Iyad Khaddam. 2020. SketchADoodle: Engineering Touch-surface Multi-stroke Gestures by Bézier Curves. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 2 (June 2020), 30 pages. https://doi.org/10.1145/3397875

1 INTRODUCTION

Touch-surface multi-stroke gestures [29, 56] are involved in a wide variety of gesture-based user interfaces [38, 43], ranging from 2D menu items [2] and motion gestures [3] to 3D object manipulation [4, 8], Computer-Aided Design (CAD) [19], and geometric modelling [14]. In these fields, a gesture is often captured as a sequence of timestamped spatial points, perhaps associated with some actions:

Authors' addresses: Donatien Grolaux, ICHEC Brussels Management School, Université catholique de Louvain, Rue au Bois, 365a, Brussels/Louvain-la-Neuve, 1150/1348, Belgium, donatien.grolaux@{ichec,uclouvain}.be; Jean Vanderdonckt, Université catholique de Louvain, LouRIM Institute, Place des Doyens, 1, ICTeam Institute, Avenue Georges Lemaître, 4, Louvain-la-Neuve, 1348, Belgium, jean.vanderdonckt@uclouvain.be; Thanh-Diane Nguyen, ICHEC Brussels Management School, Rue au Bois, 365a, Brussels, 1150, Belgium, thanh-diane.nguyen@ichec.be; Iyad Khaddam, Université catholique de Louvain, LouRIM Institute, Place des Doyens, 1, Louvain-la-Neuve, 1348, Belgium, iyad.khaddam@uclouvain.be.

2573-0142/2020/6-ART2 $15.00
https://doi.org/10.1145/3397875

$$G = \{p_i{=}(x_i, y_i, z_i, t_i)\}, \forall i \in \overline{n}{=}\{1, ..., n\} \quad (1)$$

where $(x_i, y_i, z_i) \in \mathbb{R}^3$ are the 3D coordinates of each gesture point, and t_i is the timestamp. When a gesture is issued on a touch-surface, z_i is not recorded. This simple representation serves as the basis for raw gesture capturing and for sending them to stroke-gesture recognizers, such as $P+ [51] (the most accurate recognizer for the moment), $Q [52] (the fastest recognizer for the moment), G-Gene [11], and !FTL [50] to name a few. This bitmap representation remains suitable for any internal machine processing and computation, such as gesture analysis, classification, and recognition [3, 23]. However, when it comes to visualizing and rendering such stroke gestures for direct manipulation, especially for editing, the bitmap representation is no longer convenient because the sampling is discrete, not continuous, thus preventing the end user to logically edit them. For example, cutting a gesture into two parts requires point re-sampling to determine the cutting point, which was possibly not captured during gesture acquisition; composing two gestures to form a new one is almost impossible apart from reissuing it completely.

Historically speaking, the Rotate-Scale-Translate (RST) paradigm [28] became a *de facto* standard to manipulate gestures in a 2D space: from two reference points captured or more, meaningful parameters of a gesture are computed to ensure *rotation* (relying on angular data), *scaling* (based on length ratio), and *translation* (linked to the barycenter of the translation). These three fundamental operations can be effectively performed by any stroke gesture. Invariance with respect to translation, scaling, and rotation is crucial to maintain when a gesture captured on one device should be effectively and efficiently compared against reference gestures belonging to the training set.

All recent stroke-gesture recognizers, such as $Q [52], ensure RST invariance, control them or both, such as !FTL [50]. RST processes only geometrical data [12] in an efficient way. For example, moving an object using two fingers on a touch-surface, represents and produces four Degrees-of-Freedom (DoF): two translations, one rotation, and one scaling. RST can be even extended to more DoF, such as for 6 DoF [34] or by applying it on an axis to manipulate the object [4].

While RST still represents today an attractive method to perform computations on stroke-gestures by matrix transformations on points, it is not so convenient for other end-user operations, such as editing, cutting, copying, pasting, deforming, and composing gestures: calculating matrix transformations in real-time is cost-expensive [14] and could induce latency increasing the response time. Some operations like composition and decomposition are hard to achieve with these transformations, they are error-prone due to rounding error, and they create artificial points [20].

Various model-based approaches have been introduced to manipulate 2D objects in a logical way, such as Bézier curves [7], Bézier surfaces [35], B-Splines [14], or Non-Uniform Rational B-spline (NURBS) surfaces [19]. Thanks to these model-based approaches, 2D contents are logically manipulated: deforming a shape relies on manipulating varying logical parameters of the model, such as moving the control points of a Bézier curve. Such a curve relies on a limited number of control points to edit the gesture either globally or locally, which enables efficient computer visualization and control flexibility. However, manipulating an underlying model, as induced by these approaches, could become resource consuming and complex to understand, design, and develop. The user interface designer and/or developer should possess a thorough understanding of the mathematical representation and operations to optimally process 2D contents [53].

To address the aforementioned challenges, this paper applies Bézier curves to represent, store, and manipulate touch-surface multi-stroke gestures to benefit from the advantages associated to a model-based approach and provides a series of high-level algorithms to visualize and directly manipulate gestures at a higher level of abstraction than the RST method. In this way, visualizing,

editing, and composing such gestures are becoming possible in a logical way while not suffering from the underlying complexity.

To this end, the contributions of this paper are as follows:

C_1. We revisit the concept of Bézier curves in the light of multi-stroke gesture to transform the RST representation into an analytical form by Bézier curves.

C_2. We define a set of algorithms for common gesture operations, such as draw, move, rotate, resize, compose, slice, colorize at a higher level of abstraction than with RST [28]. For each algorithm, the method is described, the pseudo-code is provided and explained to facilitate reusability.

C_3. We present SketchADoodle, an Android-based mobile application developed in Java that demonstrates the engineering of the above contributions into a drawing application that is efficient enough to run smoothly on an average smartphone. This application can be downloaded from the Android Play repository.

C_4. The core library for the drawing abstraction containing the algorithms described in this document was translated to JavaScript and released on GitHub.

The remainder of this paper is structured as follows: Section 2 discusses the related work by providing the background required for the contributions and compares the bitmap-based approach (RST) with our vector-based approach (Bézier curve); Section 3 elaborates on constraints and design considerations raised by a model-based approach for direct manipulation of multi-stroke gestures by Bézier curves; Section 4 describes the implementation of SketchADoodle, an Android-based mobile application for drawing, gesturing, which demonstrates how the pseudo-code of Bézier-based operations could be engineered in a user interface; Section 5 reports on the results obtained by a task-based evaluation of SketchADoodle by new end users; Section 6 concludes the paper by summarizing the contributions and discusses some future avenues to this work.

2 RELATED WORK AND BACKGROUND

Our work mainly touches two scientific areas: geometric modelling [14] based on Bézier curves [34, 35] and stroke-gesture manipulation, which are respectively discussed in the next two subsections with a focus on how Bézier-related work may be exploited in gesture manipulation, such as vector drawing. The final subsection compares how operations could be performed either with a bitmap-based approach with RST [28] against the vector-based approach with Bézier.

2.1 Bézier Curves and B-Splines

Drawing a curve in a vector-based approach typically relies on Bézier curves. A Bézier curve of a degree n [7] is defined by n+1 *points* $P_i = (x_i, y_i)$, the first and the last points being considered as the starting point and the ending point of the curve. The remaining $n-1$ points are called *control points* as they act as magnets to control the curve: the curve naturally tends to be attracted by them and moving the control points directly modifies the curve. Fig. 1a-b illustrates the process of drawing a Bézier curve of degree two, which have three control points. The first and second control points define a segment of line, and the second and third control points define a second segment. For any $t \in [0, 1]$, we have get two points on these segments by using t as a proportion over them. These two points define a segment of line, and recursively we obtain a point on that segment by considering t as a proportion over it. This point is on the quadratic Bézier curve defined by the control points for the parameter t. This process is applied recursively for Bézier curves of orders higher than two. Formally, a Bézier curve for the control points $P = (P_i) = (P_0, ..., P_n)$ is defined by:

$$P(t) = \sum_{i=0}^{n} B_i^n(t) \bullet P_i \; \forall t \in [0,1] \tag{2}$$

where the *Bernstein polynomials* are defined by [27]:

$$B_i^m(u) = \binom{m}{i} u^i (1-u)^{m-i} \tag{3}$$

The higher the degree of the Bézier curve, the more complex the curve shape can be. A very high degree Bézier curve could be advantageously replaced by a sequence of curves (*e.g.*, Fig. 1c) of a more limited degree, generally 3, where the ending point of a curve coincides with the starting point of the next curve, which determines a *B-spline* [14]. We adopt the same approach, but with B-splines composed of quadratic Bézier curves only to simplify the algorithms, to make their computation faster, and to allow users to issue curves in any order, direction, or position.

Favreau *et al.* [20] performed a significant step in pursuing the goal of transforming bitmap-based inputs into vector-based curves. Previous algorithms for this problem tend to produce complex drawings composed of many short curves and many control points, especially with thick or sketchy lines. They proposed the first vectorization algorithm that balances fidelity to the bitmap input with the simplicity of the output, as measured by the number of curves and their degree. By casting this trade-off as a global optimization, less curves are produced with more accuracy. It also disambiguates curve topology at junctions by favoring the simplest interpretations overall. [16].

Eigenwillig *et al.* [18] extended the problem of snap rounding from straight-line segments [25] to Bézier curves of arbitrary degree, and thus the first method for geometric rounding of curvilinear arrangements. From a set of intersecting Bézier curves, they computed a geometric rounding of their true arrangement, with an acceptable precision. The underlying deformation of the arrangement set all Bézier control points at integer points and comes with the same geometric invariance properties as in straight line snap rounding: during rounding, objects do not move further than the radius of a pixel, and features of the arrangement could collapse but do not invert.

Yap *et al.* [55] introduced the first subdivision algorithm for efficiently compute the intersection of two Bézier curves F, G, possibly with tangential intersections. Based on geometric separation bounds, and using a criterion for detecting non-crossing intersection of curves, their algorithm is adaptive in that it exploits exact float computations. In contrast, most other algorithms assume F,G to be relatively prime. Cheng *et al.* addressed the problem of efficient clipping of Bézier curves [13], which is a fundamental contribution to cut a Bézier curve at any point in a logical manner.

In conclusion, there are several contributions that effectively and efficiently solve the direct manipulation of such curves, but limited to one operation at time or a small set of operations (like [16], while we need integration of these operations for direct manipulation. For this purpose, we identified several implementations of such operations on Bézier curves that will be transposed, revisited, and modified for our purpose of gesture direct manipulation.

2.2 Gesture Manipulation

The bitmap format (Eq. 1) is the predominant representation in many environments for various types of gesture manipulation, particularly design [3], gesture recognition [2, 8, 11, 26, 51, 52], and development [36, 37], for composition [46], but not really for direct manipulation. Two exceptions exist for gesture recognition where the initial bitmap format is transformed into vectors: PennyPincher [47] transforms pairs of points into vectors to compute a scalar product between them; !FTL [50] considers subsequent sets of three points to form a triangle and compute the Local Shape Distance (LSD) between the reference and the candidate. Others also privilege a model-based approach for handling gestures [21], such as Proton++ [26], GestIT [46], and GestUI [38].

None of these environments exploit Bézier curves for uni-stroke or multi-stroke gestures. Several successful works exist however that rely on Bézier curves for recognizing other types of gestures (*e.g.*, hand poses) or for calculating other types of analysis (*e.g.*, classification). Shin *et al.* [45] present a hand gesture recognition system for exploring large registered 3D data sets. A geometric method using Bézier curves is used for the trajectory analysis and gesture classification and recognition from trajectories based on variations in hand speed. The recognition rate reported from the experiment is on average beyond 97%.

Operations	Bitmap-based	Vector-based
Drawing simple shapes (lines, curves, rectangles, circles, text, etc.)	Yes	Yes
Editing individual objects (move, reshape, rotate, resize, change coloring, etc.)	No: objects drawn are fused on the underlying bitmap, and they do not exist by themselves.	Yes: all objects drawn exists by themselves and can be modified.
Editing a region of the drawing (select an area and cut/copy/paste, rotate, resize, blur effect, white balance manipulation, noise reduction, special coloring effects, etc.)	Yes: bitmaps are matrixes of color dots well suited for these manipulations.	No: the colors of individual objects can be manipulated; however, it is specific to each object and not a region of the drawing.
Coloring closed areas	Yes: bucket tool fills a region defined by pixels of the same (or close) color.	No: once again, the fill color if individual objects can be manipulated; however, it is specific to each object and not a region of the drawing.

Table 1. Comparison of bitmap-based *vs.* vector-based representations.

Rashid *et al.* [40] also recognize hand gesture by modelling them through Bézier curves. Firstly, a skin-based hand segmentation is performed using a normal Gaussian distribution. Then, the hand centroid points are identified and fitted with Bézier curves. These subsequent points are finally concatenated to build the full Bézier descriptors, that are further classified by Hidden Markov Models (HMM) using Left-Right Banded (LRB) topology. The control points of the resulting Bézier curves therefore correspond to the various centroid points (i.e., control points). Au *et al.* [4] handle multi-touch gestures, but for transforming 3D objects in a constrained way. Shen *et al.* [44] improve the problem of memory storage occupied with the current method of multi-touch gesture recognition by relying on a quadratic Bézier curve.

Quadratic Bézier curves are also investigated for face and hand gesture recognition [17]. Firstly, face and hands images are extracted by skin color and their centroids are tracked by a special Greedy Exchange algorithm. Then, the centroids are similarly fitted into a quadratic curve and six invariants from this quadratic curve are computed. In this case, a gesture feature vector composed of $6n$ invariants is constructed, where n is the number of the strokes.

Sapek [41] is able to recognize a gesture from points sequences encoded according to Eq. 1 without any marker indicating the starting and ending points of a stroke. Their points are fitted into a Bézier curve with constant memory usage by computing a least-squares method.

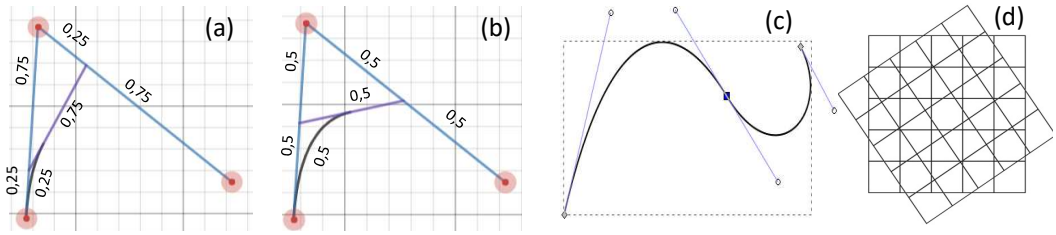

Fig. 1. Creation of a Bézier curve with three control points: (a)-(b); (c) B-Spline of degree three; (d) Rotation of a bitmap-based representation.

2.3 Comparison of Bitmap-based *vs.* Vector-based approaches

Two main types of drawing applications exist on computers (Table 1):

(1) *Bitmap-based* drawing applications applying the RST paradigm [28], like Gimp [49] and Photoshop [1]. At their core, they manipulate a rectangular matrix of dots represented by points, where each dot holds its own color.
(2) *Vector-based* drawing applications like Inkscape [48] and CorelDraw [15], which manipulate mathematical objects like vectors and splines. At their core, they turn drawings into computational problems, with properties such as precision, compactness and editability [20].

Both approaches actually work very well, albeit differently. They have their individual strengths and weaknesses (Table 1). For example, simulating a paint brush is straightforward in bitmap-based applications, but hard to edit apart from fine-grained point editing. Contrarily, editing the objects shape in vector-based applications is easier, while editing colors at the pixel level is not possible.

Indeed, the bitmap-based approach represents an object by a fixed matrix of pixels. To rotate part or whole of the object, the corresponding part in the matrix should be rotated as well (Fig. 1d). The rotated dots must be projected back onto the fixed matrix of pixels. For most angles, no one-to-one matching between the initial and the final matrices exist, which introduces visual artefacts, like an aliasing effect. The problem for scaling is similar: for most scale sizes, there is no one-to-one matching.

Vector-based approaches do not have this limitation. The transformations are applied on the mathematical object describing the shapes, with a precision as high as needed. What the user sees is the projection of the resulting shape on the matrix of pixels of the screen. However, the bucket tool purpose is to fill an area of pixels of the same color, which is not well suited for a vector-based approach. This article contributes an algorithm that overcomes this difficulty. This drawing abstraction is implemented for mobile devices, which gives an access to multi-touch. Moreover, the algorithms should be designed for limited memory and CPU capabilities such as found on mobile devices.

Both approaches are fundamentally based on how a drawing, a gesture is internally represented inside the application and the tools provided by the application reflect their intrinsic capabilities.

In this work, we create an alternate drawing abstraction trying to get closer to how young children would draw using their school tools. Typically, they have pencils to draw lines, they have scissors and glue that serve as their version of cut and paste. They also like to paint completely enclosed areas in different colors. And because we are creating a digital version of this drawing approach, it would be nice to propose manipulations of the drawing that would not be possible in the real world. We decided to focus our drawing abstraction on the set of tools explained in Table 2.

Operations	Real World	Bitmap-based	Vector-based	SketchADoodle
Freehand drawing	Pen	Yes	Yes	Yes: 
Color enclosed areas	Pen	Yes: bucket tool	No	Yes: The area pointed by the user is filled 
Cut	Scissors	Yes: free selection+cut and paste	No	Yes: The underlying picture is cut along the line 
Move and rotate	The cut figures are moved and rotated by hand	Partial: free selection + move and rotate but yields aliasing	Yes	Yes: A multitouch interaction allows moving and rotating objects 
Resize	Not possible	Partial: free selection+resize but yields aliasing	Yes	Yes: A multitouch interaction allows resizing objects 
Group	Glue stick	No	Yes	Yes: All objects in contact with the line are grouped together. They will move, rotate, and scale together 

Table 2. Transposition of operations from the real world to bitmap-based *vs.* vector-based approaches.

3 DIRECT MANIPULATION OF MULTI-STROKE GESTURES USING B-SPLINES

For our purpose, gestures or freeform drawings are captured using B-Splines that are a succession of Bézier curves. Internally, B-Splines are represented by the coordinates of their control points. All operations performed by an end user on the multi-stroke gesture, say compose two gestures, is therefore transposed into a corresponding operation performed on this representation, say slice one B-Spline at some control point to form two new and independent B-Splines.

For this purpose, Table 2 gives an overview of such operations by comparing its real-world representation, its typical implementation found in bitmap-based and vector-based approaches, and our capability to perform the same operation. These operations will be detailed one by one in the next subsections by motivating their need, introducing the algorithm to support it, and the pseudocode of these algorithms. This pseudocode will then service in implementing SketchADoodle, an Android-based mobile application that will be discussed in Section 4.

3.1 Freehand Drawing

When an end user issues a gesture on a touch-surface, only points are captured by the device with some lag [9] depending on the capturing device, which may affect the system response time, and the end-user's perception of it. To achieve the illusion that a line is drawn following the movement of a gesture, we could join all points by successive segment of lines. This yields a problem if the sampling of points is too low as the smooth curve appears as a bumpy poly-line. This problem is even worse for SketchADoodle as it is possible to scale up any drawn curve: no sampling can stay good enough under sufficient scaling up. We solve this issue by fitting the captured points to a B-Spline as they keep their smoothness when scaled up or down. The problem of fitting points to B-Spline is well studied [13, 24, 57]. However, these algorithms yield to several problems for our purpose. Firstly, if we apply the fitting while the user is still drawing, the same part of a gesture can be interpolated by different B-Splines over time, yielding to strange visual effects that are potentially confusing for the end user. Secondly, if we wait for the user to complete its gesture before doing the fitting, what do we display during the drawing itself? Finally, these algorithms can be cost intensive yielding to slow response times on low-end devices.

To solve these issues, a streaming algorithm [5] processes points as they arrive. As the user is inputting points P_i, the algorithm fits them on the fly to a Bézier curve until the curve is no longer considered accurate enough. The last fitting Bézier curve is then added to the B-spline, and the corresponding points are removed from P. More formally, let B the target quadratic B-spline defined by the control points $B_0, B_1, ..., B_{length(B)-1}$ so that each Bézier curve of index i in B is defined by $(B_{i\times2}, B_{i\times2+1}, B_{i\times2+2})$. Two successive Bézier curves i and i+1 share the common point $B_{i\times2+2} = B_{(i+1)\times2}$ so that they appear as a continuous spline. The corresponding pseudocode is FreeHand () (Fig.2), which consists in an event-based loop simplified as a while loop (lines 2-19). We define || as the Append operator (line 4). The algorithm splits the shape drawn into two parts: a B-spline B and a sequence of segments of lines P. At line 5, P is fitted to the Bézier curve BT. If the fitting is not good enough according to the D distance function defined further (line 6), the previous fitting to B is added and the corresponding points from P (lines 6-16) are removed. γ is an arbitrarily chosen value: a high value accommodates more points in a fitting which makes the B-spline smoother, but less accurate. Inversely, a low value makes the B-spline more accurate, but less smooth. Lines 18 and 19 provide the user with a continuous visual feedback by drawing B and P. When the user stops, lines 21-28 ensure that all remaining points in P are fitted to a curve added to B, thus completing the operation.

3.2 Fitting

The pseudocode of Fitting (P) (Fig. 3) creates a quadratic Bézier curve fitting the points P. The parametric quadratic equation for the Bézier curve is:

$$C(t) = C_0 \times t^2 + C_1 \times t \times (1-t) + C_2 \times (1-t)^2 \tag{4}$$

The extreme points of the curve (C_0 and C_2) are set to the extreme points of P (P_0 and $P_{length(P)-1}$) [12]. For C_1, we force the curve to pass by the middle point of P: we approximate t as the relative

FreeHand ()

1 $P \leftarrow \emptyset,\ B \leftarrow \emptyset$
2 **while** *the user is drawing* **do**
3 **let** X the new input point
4 $P \leftarrow P \,||\, X$
5 **let** $BT \leftarrow fitting(P)$
6 **if** $D(B,P) > \gamma$ **then**
7 **let** $P\prime \leftarrow (P_i | i \in \{0, length(P) - 2\})$
8 // $P\prime$ is a copy of all points of P except the last one.
9 **let** $BT\prime \leftarrow fitting(P\prime)$
10 **if** $B = \emptyset$ **then**
11 $B \leftarrow BT\prime$
13 **else**
$B \leftarrow B \,||\, BT\prime_1 \,||\, BT\prime_2$
15 **end**
16 $P \leftarrow P \setminus (P\prime_i | i \in \{0, length(P\prime) - 2\})$
// Remove all points from P belonging to $P\prime$ except the last one to keep the connection to B.
18 **end**
draw B-spline B
19 $\forall i \in \{1, length(P) - 1\},\ draw\ segment\ (P_{i-1}, P_i)$
end
21 **if** $length(P) > 1$ **then**
22 **let** $BT \leftarrow fitting(P)$
23 **if** $B = \emptyset$ **then**
24 $B \leftarrow BT$
else
26 $B \leftarrow B \,||\, BT_1 \,||\, BT_2$
end
end
29 draw B-spline B

Fig. 2. Free Line Drawing Algorithm.

position of this point on the length of all the segments of P (lines 3 to 5) and then solving for C_1 (line 7).

Fitting (P)

1 **let** $C_0 \leftarrow P_0$
2 **let** $C_2 \leftarrow P_{length(P)-1}$
3 **let** $DT \leftarrow \sum_{i=0}^{i<length(P)-1} d(P_i, P_{i+1})$ // d is the Euclidian distance between two points
4 **let** $DM \leftarrow \sum_{i=0}^{i<floor((length(P)-1)/2)} d(P_i, P_{i+1})$
5 **let** $t = \frac{DM}{DT}$
6 **let** $Pt \leftarrow P_{floor(length(P)-1)/2)}$
7 **let** $C_1 \leftarrow Pt/(2 \times t \times (1-t)) - C_0 \times t/(2 \times (1-t)) - C_2 \times (1-t)/(2 \times t)$
8 **return** (C_0, C_1, C_2)

Fig. 3. Fitting Algorithm.

D (B,P) (Fig. 4) computes a distance between the Bézier curve B and the set of points P. Since there are many ways to define this distance, we selected one that is efficient and good enough for our purpose. D returns the maximum Euclidean distance between each point pt in P and their equivalent point on B. By computing the ratio between the sum of euclidean distances for all segments in P up to the point pt and the total Euclidean distance for all successive segments of P, we obtain an estimated t for the equivalent point on B to pt.

D (B,P)

1 **return** $max(\forall j \in \{1, length(P) - 2\} : d(P_j, B_j))$ *where* $B_j = B\left(\frac{\sum_{i=0}^{j-1} d(P_i,P_{i+1})}{\sum_{i=0}^{length(P)-2} d(P_i,P_{i+1})}\right)$

Fig. 4. Distance Algorithm.

3.3 Translating, Scaling, and Rotating Gestures

A nice property of the Bézier curves is that their affine transformations follows exactly the affine transformations of their control points. Therefore, implementing the basic affine operations is resumed to applying them on their control points. A mouse pointer can drag any control point from one position to another, which can be described as a displacement vector between the two positions. When a multi-point interaction is possible, from the displacement of only two points, we can compute translating, scaling, and rotation in a straightforward way. Fig. 5 depicts this phenomenon where a user dragged a gesture from (A, B) to (C, D). E is the translation of B by the vector $\vec{AC}$. The shape should be moved along the vector $\vec{AC}$, rotated by σ, and scaled by $d(C, D)/d(A, B)$.

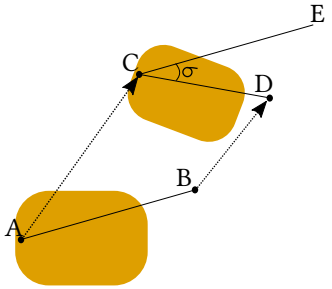

Fig. 5. Affine transformations for translating two control points of a Bézier curve.

To achieve a direct manipulation behavior, the algorithm transitions automatically between an interaction based on one or two control points, *i.e.*, using one or two fingers on a multitouch device. The implementation is based on the state design pattern [22]. In a system where the same action results in different execution paths depending on the state of the system, the state design pattern uses dynamic binding to objects implementing each state instead of bare *if* statements on a state value. The implementation benefits from a clearer code where each state's specific behavior is isolated into its own object.

For the purpose of our user interface, we have three states depending on the number of control points currently active: zero, one, or two. The possible actions are: down (add one control point), move (move any of the control points), and up (remove a control point).

MoveResizeRotate () (Fig. 6, 7, and 8) is an event-based loop organized by the state value. The interaction begins when a first point is given to the interaction (lines 7-11 of Fig. 6). The function

getObject (line 10) determines the topmost object at position (x, y) which is the object pointed to by the user.

MoveResizeRotate () initialization and NONE state

```
1  let state ← NONE
2  let x0, y0, x1, y1 ← 0
3  let ox0, oy0, ox1, oy1 ← 0
4  let shape ← ∅
5  when state is NONE :
6      event down(x, y, index) :
7          if index = 0 and getObject(x, y) <> ∅ then
8              x0 ← x
9              y0 ← y
10             object ← getObject (x, y)
11             state ← MOVE
           end
13     event move(x, y, index) : ∅
14     event up(x, y, index) : ∅
```

Fig. 6. The Interactive algorithm for move, resize, and rotate to ensure translation, scaling, and rotation.

MoveResizeRotate () MOVE state

```
1  when state is MOVE :
2      event down(x, y, index):
3          if index = 1 then
4              x1 ← x
5              y1 ← y
6              let ox0 ← x0, oy0 ← y0, ox1 ← x1, oy1 ← y1 // save as reference
7              let refObject ← copy(object)
8              state ← SCALEANDROTATE
           end
10     event move (x, y, index):
11         if index = 0 then
12             move (object, x − x0, y − y0)
13             x0 ← x
               y0 ← y
           end
16     event up(x, y, index):
17         if index = 0 then
               object ← NONE
           end
```

Fig. 7. Algorithm for interactive translate, scale, and rotate.

When a single point is moved, we compute a delta between its last and new coordinates and apply this delta to the selected object (line 10-15 of Fig. 7). When a second point is added to the interaction (lines 2-9 of Fig. 7), the two control points are saved as the reference points for the move, scale, and rotate operations, along with a reference copy of the selected object. These points

MoveResizeRotate () ScaleAndRotate state

```
1  when state is SCALEANDROTATE :
2      event down(x, y, index) : ∅
3      event move (x, y, index):
4          if index = 0 then
5              x_0 ← x
               y_0 ← y
           end
8          if index = 1 then
9              x_1 ← x
               y_1 ← y
           end
12         object ← copy(refobject)
13         move(object, −ox_0, −oy_0)
14         scale(object, d((x_0, y_0), (x_1, y_1))/d((ox_0, oy_0), (ox_1, oy_1)) )
15         let oa ← angle((ox_1, oy_1), (ox_0, oy_0), (ox_1, oy_0))
16         let a ← angle((x_1, y_1), (x_0, y_0), (x_1, y_0))
17         rotate(object, oa − a)
18         move(object, ox_0, oy_0)
19     event up(x, y, index):
20         if index = 1 then
21             state ← MOVE
           end
```

Fig. 8. Interactive translate, scale, and rotate Algorithm.

correspond to points A and B of Fig. 5. The reference points and object avoids floating point issues that would arise from continuously transforming the original object. When one of the two points is moved (lines 3-18 of Fig. 8), the selected object is transformed. First, we restore the reference object (line 12), then we center the object according to the first reference point (line 13), scale it (line 14), rotate it (lines 15-17), and finally move it to its correct position (line 18).

3.4 Glue Tool

The glue tool consists in two actions: firstly, finding elements designated by the user and, secondly, grouping them together so that they behave as one. The second point is just an issue of using a proper data structure. The elements manipulated by the editor should not just be B-splines, but collections of B-splines. Elements glued together are put in the same collection. When moved, scaled or rotated, the affine transformations are applied to the whole collection. The first point is dependent on how we let the user select elements. To simulate a glue tape, we let the user draw a segment of line between two points. All elements that intersect this segment of line should be glued together. A Bézier spline intersects a segment of line if any of its curves intersects this segment. There exist many approaches to solve this problem [16, 54, 55]. The problem can be even solved directly analytically because we are just intersecting a segment of line to a quadratic Bézier curve. However, this yields to two problems when run on a low-end device:

(1) Floating point arithmetic has approximation errors; in some case these errors can be big enough to fail the computation.

(2) Floating point arithmetic can be time consuming; this is heavily dependent of the type of operations that is used and the performance of the hardware running it. In particular, the operations required for an analytical solution where often too slow on real mobile devices.

There are several tricks that we can use to accelerate the computation of this algorithm:

- Bézier curves are so that they are fully contained in the bounding box defined by their control points. Consequently, a quick check can eliminate the possibility of an intersection if the bounding box does not intersect the segment [54].
- De Casteljau's algorithm [39] decomposes a Bézier curve into a B-spline composed of two Bézier curves [6]. This algorithm can be applied iteratively on the individual curves of the B-spline as many times as we want. Consequently, we can create a B-spline composed of Bézier curves as small as we want. A small enough Bézier curve is very close to the segment defined by its start and end points. In fact, when rendered on an actual screen, a small enough Bézier curve will be drawn on the exact same pixels as the segment of line defined by its start and end points. In other words, we can turn a B-spline into a succession of segment of lines that look exactly the same to the end user due to the way they are rendered on the screen. Computing if two segments of line intersect each other is faster than solving quadratic equations, and yields less errors due to floating point approximations. Of course, the transformation of the B-spline into lots of tiny segments takes time, but the result can be cached in memory and reused many times. In practice, the cached version is created when needed, by any tool that has a use for it, and invalidated when the object is scaled or rotated.

The pseudocode ApproximateCurveBySegments (curve,tolerance) (Fig. 9) transforms a Bézier curve into a collection of segments according to a provided *tolerance*. In practice, tolerance is determined by the pixel density of the screen. *segmentsCount* is the number of segments used to approximate the curve, and is incremented at each iteration (line 4). Lines 5-8 creates the segments, by adding intermediate points directly from the parametric equation of the Bézier curve (Fig. 10). Line 9 computes an *error* value, by the sum of the Euclidean distances between the middle of each segment and their equivalent point on the curve. The algorithm stops when *error* is above the *tolerance* (line 10). For simplicity, this algorithm returns the first *failed* approximation; in practice this approximation is good enough for our application.

ApproximateCurveBySegments (curve,tolerance)

1 **let** $segmentsCount \leftarrow 1$
2 **let** $segments \leftarrow \emptyset$
3 **repeat**
4 $\quad segmentsCount \leftarrow segmentsCount + 1$
5 $\quad segments \leftarrow (curve_0)$
6 $\quad$ **for** $i \leftarrow 1$ **to** $segmentsCount$ **do**
7 $\quad\quad segments \leftarrow segments \,||\, pointOnCurveAt(curve, i/segmentsCount)$
$\quad$ **end**
9 $\quad$ **let** $error \leftarrow \sum_{i=1}^{segmentsCount} d\left(\frac{segments_{i-1}+segments_i}{2}, pointOnCurveAt\left(curve, \frac{i\times 2-1}{segmentsCount\times 2}\right)\right)$
until $error < tolerance$
11 **return** $segments$

Fig. 9. Approximate a Bézier curve by segments

pointOnCurveAt (curve,t)

return

$$((1-t)^2 \times curve_{x_0} + 2 \times (1-t) \times t \times curve_{x_1} + t^2 \times curve_{x_2}, \\ (1-t)^2 \times curve_{y_0} + 2 \times (1-t) \times t \times curve_{y_1} + t^2 \times curve_{y_2}) \quad (5)$$

Fig. 10. Get a point on a Bézier curve

3.5 The Bucket Tool for Coloring

In a bitmap drawing application, the bucket tool changes the color of all pixels that are in contact of the same color (or within a threshold). As we are using a vector-based approach, this algorithm must be adapted. To be as close as possible to the usual bucket tool behavior, we decided to use a mixed approach between a bitmap-based algorithm and a vector-based one.

Firstly, we use the classic flood fill algorithm on a bitmap version of the drawing. This algorithm is slightly modified to capture the bordering pixels of the flooded area. Secondly, we walk these pixels matching them to the actual splines they correspond to. When passing from one spline to another, we figure out the intersection point between them and use de Casteljau's algorithm the keep the relevant part from each spline. This walking creates a new closed spline which contains the flooded area. Finally, to give consistency to the result, we glue together the flooded area and all the curves that are in its contact.

Let us cover these steps in more details: the first step is the flood fill algorithm. Internally, we have a repository containing all the B-splines of the current drawing. For simplicity, we assume this repository is an indexed array. Instead of running the flood fill algorithm on a rendering of the real drawing, we run it on a buffer where all elements are drawn using a color that is their index in the repository, starting at 1. In other words, the color of a pixel in the buffer is the index of the element in the repository for which this pixel was drawn for. No element is drawn with the color 0 which is the color that has to be filled.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	0	0	0	0	0	0	3	0	0
1	1	2	2	2	2	2	2	2	2	2
2	0	1	0	0	0	0	0	3	0	0
3	4	1	0	0	**0**	0	0	3	0	0
4	0	1	4	0	0	0	0	0	3	0
5	0	1	0	4	4	0	0	0	3	0
6	1	0	0	0	0	4	4	4	4	3
7	1	0	0	0	0	0	0	0	0	3

In the above example, there are four splines and the user has selected the pixel $(4, 3)$ as the starting point for the flood fill. This point is displayed in bold for readability. The version of our algorithm is modified in a single way: each time a pixel different than 0 is met, we append the coordinates and the color to an edge collection.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	0	0	0	0	0	0	3	0	0
1	1	2	2	2	2	2	2	2	2	2
2	0	1	0	0	0	0	0	3	0	0
3	4	1	0	0	**X**	X	X	3	0	0
4	0	1	4	0	0	0	0	0	3	0
5	0	1	0	4	4	0	0	0	3	0
6	1	0	0	0	0	4	4	4	4	3
7	1	0	0	0	0	0	0	0	0	3

We assume that the flood fill algorithm goes right first ; by symmetry it does not matter. Eventually, the algorithm meets the color 3 at position $(7, 3)$ and appends $((7, 3), 3)$ to the edge collection.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	0	0	0	0	0	0	3	0	0
1	1	2	2	2	2	2	2	2	2	2
2	0	1	0	0	0	0	0	3	0	0
3	4	1	X	X	**X**	X	X	3	0	0
4	0	1	4	0	0	0	0	0	3	0
5	0	1	0	4	4	0	0	0	3	0
6	1	0	0	0	0	4	4	4	4	3
7	1	0	0	0	0	0	0	0	0	3

Once an edge is met, the flood fill algorithm fills the same line in the other direction. Eventually, it meets another edge pixel and appends $((1, 3), 1)$ to the edge collection.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	0	0	0	0	0	0	3	0	0
1	1	2	2	2	2	2	2	2	2	2
2	0	1	X	X	X	X	X	3	0	0
3	4	1	X	X	**X**	X	X	3	0	0
4	0	1	4	X	X	X	X	X	3	0
5	0	1	0	4	4	X	X	X	3	0
6	1	0	0	0	0	4	4	4	4	3
7	1	0	0	0	0	0	0	0	0	3

Eventually, the flood fill algorithm completes, and the edge collection contains all the border pixels and their respective colors. To be able to walk around the edges, we first need to split them into tracks (Fig. 11) that are composed of pixels that are neighbours of each other (lines 5-9 of Fig. 12 and Figure 13). Also, each track stays on the same colour for as long as possible: the tracks stay on the same color at first (lines 2-6 of Fig. 11) before being merged together (lines 7-13). In the example, we have these tracks:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	0	0	0	0	0	0	3	0	0
1	1	2	**2**	**2**	**2**	**2**	**2**	2	2	2
2	0	**1**	X	X	X	X	X	**3**	0	0
3	4	**1**	X	X	X	X	X	**3**	0	0
4	0	1	**4**	X	X	X	X	X	**3**	0
5	0	1	0	**4**	**4**	X	X	X	**3**	0
6	1	0	0	0	0	**4**	**4**	**4**	4	3
7	1	0	0	0	0	0	0	0	0	3

Starting from (7, 3), the first track is of color 3 and goes downwards. This track is followed by the track for the color 4, then 1, then 2, and then finally a track of a single pixel of color 3. Finally, these tracks are merged together when the ending of a track is contiguous to the starting of another.

EdgesToTracks (edges)

```
1  let tracks ← ∅
2  while edges ≠ ∅ do
3  |  let track ← consume (edges)
4  |  edges ← edges \ {p ∈ track}
   |  tracks ← tracks || track
   end
7  foreach a ∈ tracks do
8  |  foreach b ∈ tracks | b ≠ a do
9  |  |  if abs(a_{length(a)−1_x}, b_{0_x}) < 2 and abs(a_{length(a)−1_y}, b_{0_y}) < 2 then
10 |  |  |  tracks ← tracks \ a \ b || concat(a, b)
   |  |  end
   |  end
   end
14 return tracks
```

Fig. 11. Transform the edges of a flood fill into tracks.

Consume (edges))

```
1  let track ← (edges_0) // track starts with first element of edges
2  edges ← edges \ (edges_0)
3  let point ← ∅
4  repeat
5  |  point ← findNeighbourPoint(edges, track_{length(track)−1})
6  |  if point ≠ ∅ then
7  |  |  track ← track || {point}
8  |  |  edges ← edges \ {point}
   |  end
   until point = ∅
11 return track
```

Fig. 12. Follow the edges to create a track of pixels of the same colour.

FindNeighbourPoint (edges,pa)

```
1  foreach pb∈ edges do
2  |  if abs (pa_x, pb_x) < 2  abs (pa_y, pb_y) < 2  pa_color = pb_color then
3  |  |  return b
   |  end
   end
6  return ∅
```

Fig. 13. Find a neighbour point among the edges.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	1	0	0	0	0	0	0	3	0	0
1	1	2	2	2	2	2	2	2	2	2
2	0	1	X	X	X	X	X	3	0	0
3	0	1	X	5	5	5	X	3	0	0
4	0	1	X	6	0	5	X	X	3	0
5	4	1	X	6	6	6	X	X	3	0
6	1	4	X	X	X	X	X	X	3	3
7	1	0	4	4	4	4	4	4	0	3

The next step consists in walking along these tracks, finding the parts of the B-splines corresponding to the pixels of the tracks, and stitching them together. Several edge cases need to be considered, such as when an inside area is excluded from the flood fill, a small island is delineated above by two splines (5 and 6). The flood fill algorithm tracks them as a second track separated from the first one. The graphical toolkit supports filling B-splines with exclusion zones. The algorithm generates a data structure that contains the B-splines corresponding to all the tracks (Fig. 14).

```
CreateFilledArea (tracks, repository)
1  // repository is an array of the existing B-splines
2  let area ← ∅
3  foreach track ∈ tracks do
4  |  area ← area || createClosedBSpline (track, repository)
   end
6  return area
```

Fig. 14. Aggregate the closed B-splines together.

For a given track, we have a sequence of pixel coordinates and the index of the B-spline they correspond to in the memory repository where the drawing is stored. However, the pixel coordinates are not accurate positions of the B-spline points. Indeed, they are the coordinates of the projection of these points on the matrix of pixels of the device screen. Although finding the real point on a B-spline analytically is feasible, we prefer approximating the segments for efficiency (Fig. 9). In SketchADoodle, this approximation is saved in a cache. Fig. 15 returns the point on a B-spline closest to a reference point. The B-spline is first decomposed into a collection of segments (line 1), and the function returns the closest point to the reference point among all the segments (lines 2-3). Fig. 16 returns the point on a segment closest to a reference point. First, it projects the reference point on the line defined by the segment (lines 1-5), and restricts the response to the extreme points of the segment if the projection is outside (lines 6-12) of it.

```
ClosestPoint (bspline,x,y)
1  let segments ← approximateCurveBySegments(bspline, tolerance)
2  let points ← {∀segment ∈ segments : closestPointOnSegment(segment, x, y)}
3  return point : d(point, (x, y)) = min_{p∈points} d(p, (x, y))
```

Fig. 15. Get the point on the B-spline closest to (x,y).

ClosestPointOnSegment (segment,x,y)

```
1  let x1, y1 the bottom left coordinate of the segment
2  let x2, y2 the top right coordinate of the segment
3  let t ← ((x−x1)×(x2−x1)+(y−y1)×(y2−y1)) / ((x1−x2)²+(y1−y2)²)
4  let cx ← x1 + t × (x2 − x1)
5  let cy ← y1 + t × (y2 − y1)
6  if cx < x1 or cy < y1 then
7  |  return (x1, y1)
   end
   if cx > x2 or cy > y2 then
10 |  return (x2, y2)
   end
12 return (cx, cy)
```

Fig. 16. Get the point on the segment closest to (x, y).

By following a track, the algorithm can detect a jump from a B-spline to another. On the example above, the track will jump from color 3 at $(8, 6)$ to color 4 at $(7, 7)$. There is probably an intersection between these B-splines and this point is where we should end B-spline 3 and start B-spline 4. Fig. 17 returns the intersection between two B-splines that are closest to a reference point. First, line 1 captures all possible intersections between the B-splines. If there is at least one (line 2), then we get closest one to the reference point (line 3) and check if that point is close enough to the reference point to be valid (line 4). Indeed, if the point is too far away from the reference point, then the intersection found is not where the jump between the B-splines takes place. Fig. 18 returns all possible intersections by computing all intersections between all the possible combinations of the segments of line.

Intersection (bspline1, bspline2, x, y)

```
1  let intersections ← intersectAll (approximateCurveBySegments (bspline1, tolerance),
   approximateCurveBySegments(bspline2, tolerance)
3  if intersections ≠ ∅ then
4  |  let point : d(point, (x, y)) = min_{p∈intersections} d(p, (x, y))
5  |  if d((x, y), point) < tolerance then
6  |  |  return (point_x, point_y, true)
   |  end
   end
9  return (0, 0, false)
```

Fig. 17. Get the intersection between two B-splines closest to (x, y).

IntersectAll (segments1, segments2)

```
1  return {point : ∃ s1 ∈ segments1 and ∃ s2 ∈ segments2 and point is the intersection between s1 and s2}
```

Fig. 18. Get all intersection points between the two list of segments.

A track can jump from one B-spline to another even in the absence of an intersection between them if they are close enough that there is no gap at the pixel level. This situation is detected

when there is no intersection between the B-splines close to the jump point. In that situation, the algorithm stitches the gap by adding a small segment of line. A quadratic Bézier curve is reduced to a segment of line if the middle control point is colinear and inside the two others (Fig. 19).

```
SEGMENT (x1,y1,x2,y2)
1  return ((x1, y1), (((x1+x2)/2), ((y1+y2)/2)), (x2, y2))
```

Fig. 19. Creates a Bézier curve matching a segment of line.

With these functions in mind, the general idea of turning a track into a closed B-Spline is (Fig. 20):

(1) Start from the first point of the track (lines 1-5).
(2) Stay on the current track until there is a jump to another B-spline (line 7).
(3) Detect the intersection (line 8) and add the relevant part of the first B-spline to the closed B-spline (line 10). In the absence of intersection, stitch the two B-splines (lines 12-15).
(4) Jump on the new B-spline and loop to 2 until the track is completely traversed (lines 6-21).
(5) Add the rest of the current track (lines 22-23).

Finally, Fig. 21 returns a B-spline that is cut from another B-spline at two reference points. Lines 1 and 2 obtain the Bézier curve for the two reference points, and their respective t on these curves. $curveForPoint(bspline, x, y)$ is similar to $closestPoint(bspline, x, y)$ but instead of returning the point on the B-spline, $curveForPoint$ returns the Bézier curve where (x, y) is closest to and the t value for that point on this curve. Line 3 detects the direction of the reference points on the B-spline: if the two Bézier curves are different and the first one is before the second one in the B-spline, then we create the *out* B-spline accordingly (lines 4-5). If the two Bézier curves are the same (line 9), then we cut this curve accordingly (line 10). Finally, in all other cases, we just apply this algorithm on the reverse of the B-spline.

This algorithm uses $deCasteljauRight(curve, t)$ and $deCasteljauLeft(curve, t)$. They apply de Casteljau's algorithm [6] at t on the curve and return respectively the curve split after or before t. The algorithm of $createClosedBSpline(track, repository)$ is incomplete: a B-spline can intersect itself, which is not taken into consideration here. In the real implementation, $getClosestPoint$ also returns the index of the Bézier curve inside the B-spline. Jumping from index i to index i+1 or i−1 is normal as the track follows the B-spline in the same or opposite direction. If the jump is higher, then we met an intersection of the B-spline with itself which is treated similarly to an intersection between two different B-splines.

And finally, to achieve an intuitive behavior, the closed B-spline is automatically glued to all other splines that were tracked so that they now act as a group.

createClosedBSpline (track, repository)

```
1  // track_0 is the first point of the track
2  // each point_color is the index of the B-spline in the repository
3  let current ← repository_{track_{0 color}}
4  let (startx, starty) ← getClosestPoint(current, track_{0 x}, track_{0 y})
5  let out ← ∅
6  ∀ point ∈ track \ track_0
7      if current ≠ repository_{point_color} then
8          let (x, y, found) ← intersection(current, repository_{point_color}, point_x, point_y)
9          if found then
10             out ← out || cutBSpline(current, startx, starty, x, y)
11         else
12             let (cx, cy) ← closestPoint(current, point_x, point_y)
13             let (nx, ny) ← closestPoint(repository_{point_color} t, point_x, point_y)
14             out ← out || cutBSpline(current, startx, starty, cx, cy)
15             out ← out || segment(current, cx, cy, nx, ny)
16         end
17         current ← repository_{point_color}
18         startx ← nx
19         starty ← ny
20     end
21 end
22 let (cx, cy) ← closestPoint(current, current_{length(current)−1_x}, current_{length(current)−1_y})
23 out ← out || cutBSpline(current, startx, starty, cx, cy)
```

Fig. 20. Create a closed B-spline according to a track.

cutBSpline (bspline,x1,y1,x2,y2)

```
1  let (curve1, t1) ← curveForPoint(bspline, x1, y1)
2  let (curve2, t2) ← curveForPoint(bspline, x2, y2)
3  if curve1 < curve2 then
4      let out ← out || deCasteljauRight(bspline_{curve1}, t1)
5      ∀_{i=curve1+1}^{i<curve2−1} : out ← out || bspline_i
6      out ← out || deCasteljauLeft(bspline_{curve2}, t2)
7      return out
8  end
9  if (curve = curve2 and t1 ≤ t2) then
10     return deCasteljauLeft(deCasteljauRight(bspline_{curve1}, t1), t2)
11 else
12     return cutBSpline(reverse(bspline), x1, y1, x2, y2)
13 end
```

Fig. 21. Cut a B-spline at points $(x1, y1)$ and $(x2, y2)$.

3.6 The Scissor Tool

To cut a drawing, the end-user draws a segment of line. All objects in the drawing repository that intersect the segment are cut by it. Cutting open B-splines is easy: by simplifying *getIntersection (bspline1,bspline2,x,y)*, we obtain $getIntersection\,(bspline, segment, x, y)$. At each intersection point, we use $deCasteljauRight(curve, t)$ and $deCasteljauLeft(curve, t)$ to split the B-spline in two or more parts recursively. For closed B-splines, we also need to close back the parts that are cut by stitching at the cut points using $segment(x1, y1, x2, y2)$.

4 IMPLEMENTATION OF SKETCHADOODLE

Now that we have exposed the various algorithms for direct manipulation of multi-stroke gestures by Bézier curves and their corresponding pseudocode, this section demonstrates how to implement them in SketchADoodle, an Android-based mobile application for drawing and capturing gestures. The name SketchADoodle has been chosen for combining two concepts: a *sketch* consists of a rapidly executed freehand drawing that is not intended as a finished work, often consisting of a multitude of overlapping lines, whereas a *doodle* is often considered as a small mindless sketch.

SketchADoodle is accessible as follows:

- It can be freely downloaded from the Google Play store at https://play.google.com/store/apps/details?id=com.grayswindle.sketchadoodle.
- A YouTube video demonstrating its usage is accessible at https://youtu.be/91slSprZsrI.
- A presentation of its functionalities is summarized in https://www.slideshare.net/jeanvdd/sketchadoodle.
- The algorithms described in this paper were ported to JavaScript and can be also accessed and downloaded from GitHub at https://github.com/GrolauxDonatien/SketchADoodleJS.

The UML V2.5 Class Diagram [10], as defined by OMG[1], reproduced in Fig. 22, provides an overview of SketchADoodle's main classes and relationships. As required by the Android SDK, *MainActivity* is the class that manages the main screen of the application. *DrawingArea* is a custom *View* that occupies most of the screen. This class displays the current drawing, and receives the touch events generated by the user. As the action of each touch heavily depends on the active tool, *DrawingArea* forwards the events to a class representing the active tool at the time.

ToolListener defines the four events the tools are interested in. The first three (*up*, *down*, and *move*) correspond to touch events. The last one (*onDraw*) allows the tool to draw on top of the current drawing. For instance, the glue tool draws a rectangular shaped gluing tape over the current drawing while the user is dragging his finger. All these classes have a reference towards the *UIModel* class. *UIModel* has a role similar to a view-model of the MVVM software architecture by encapsulating the data required by the view. This class implements the observer pattern for the model of the UI state: which tool is selected, what is the current line color, etc. *UIModel* also provides a reference towards the *Repository* class that contains the data specific to the current drawing, and the methods for its manipulation. It plays a role similar to a use case controller by providing the high level commands used by the UI layer.

Internally, each drawn object is of one of three types: a B-Spline, a closed B-Spline with an inside color and optionally one or more exclusion zones (also defined by closed B-Splines), or a composition of them. The mathematical parts of the algorithms are delegated to the *BezierToolkit* class while the manipulation of the repository data structure is kept in *Repository*. This separation allows unit testing the algorithms of *BezierToolkit* independently of the whole application. In practice, this proved to be very beneficial: even if an algorithm is mathematically proven correct,

[1] See http://www.omg.org/spec/UML/2.5/

its realisation using floating-point calculus can yield errors in edge cases. Debugging tools helps a lot in identifying precisely the specifics of these errors. Even if Android Studio is a very good IDE with an advanced debugging tool, having to run the whole application on a mobile phone (be it a real or an emulated one) makes the debugging session unnecessarily slow and inconvenient. As the application is written for Java, it was easy to isolate the edge cases and create unit tests specifically for them, and run them natively on a proper computer. This paper focuses mainly on the algorithms provided by *BezierToolkit*, however we also describe the way *HandTool* transits between different states depending on the number of fingers in action by the user.

The algorithms introduced in Section 3 are further refined in the SketchADoodle application to improve their overall efficiency and usability as follows (Fig. 23):

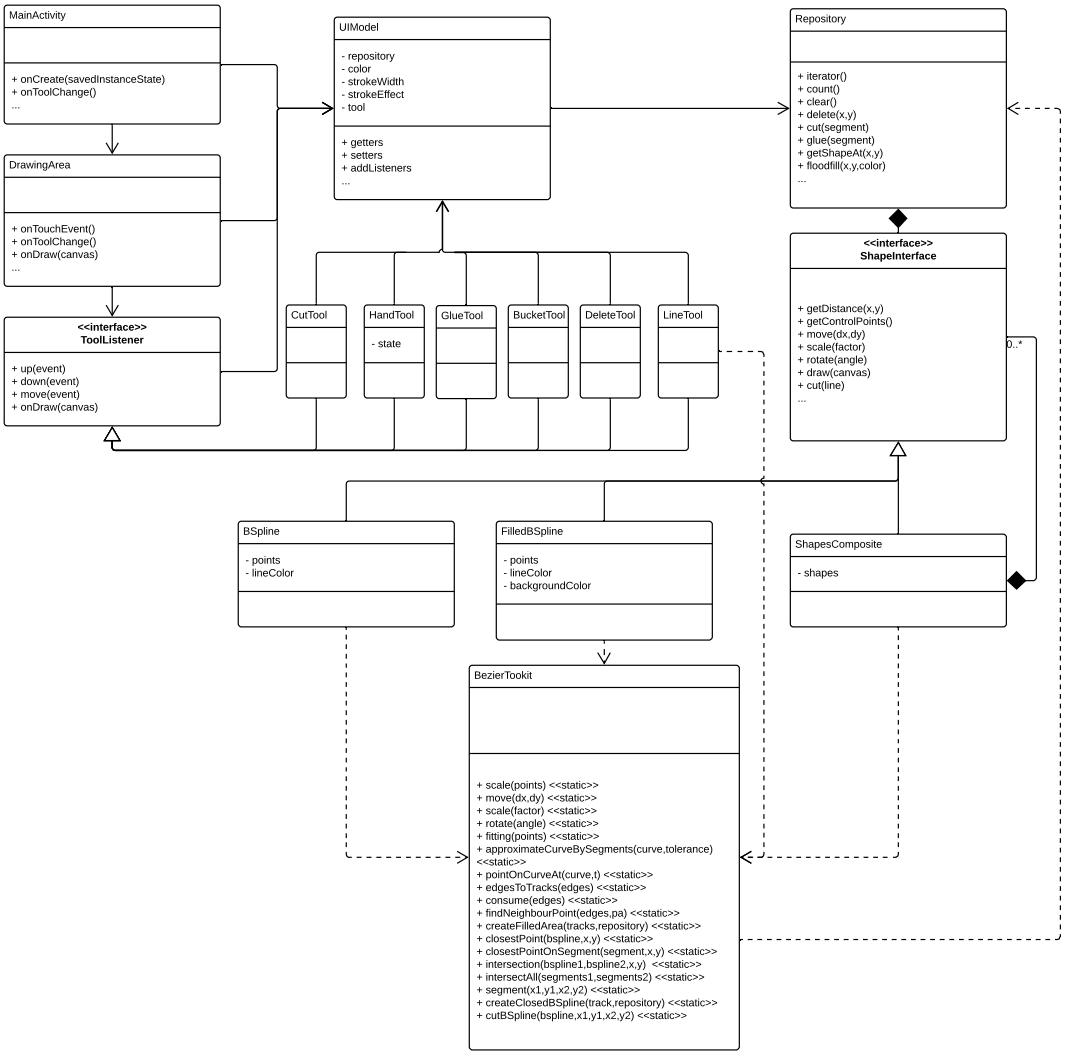

Fig. 22. UML class diagram of SketchADoodle.

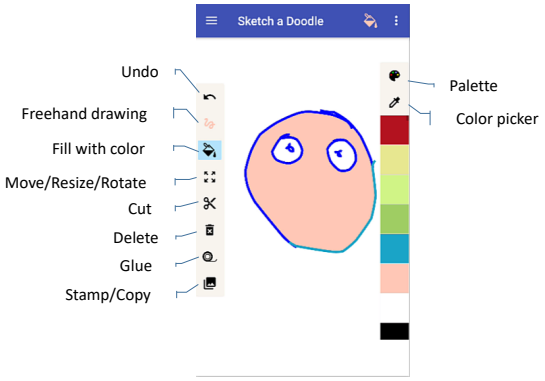

Fig. 23. SketchADoodle and its tools corresponding to operations.

- Undo operations are implemented thanks to the Memento pattern[2], a software design pattern restoring an object to its previous state via rollback [22].
- The free hand drawing tool supports an arbitrary number of touches which draw their own independent B-spline.
- The move, resize and rotation tool supports all variations of touches. For example, one can start dragging a shape with the left finger, add the middle finger to initiate a rotation, lift the left finger and move the shape using the middle finger, add back the left finger and so on.
- Splines are drawn with a configurable width, color, and style. In particular, the algorithms described in this paper do not take the width into account.
- The scissor tool has a rectangular shape instead of a line with no depth. With a line, a B-spline cut in two results in two B-splines that are so close together they still look like a single one. This was rather confusing for users. Using a rectangle instead of a line removes a whole part of the B-spline and the result is more obvious. The implementation involves cutting according to the four sides of the rectangle, and remove the parts left under it.
- Because the drawing abstraction is unusual w.r.t. other applications, the first time a tool is selected a small animation is played that exemplifies its use. We used the interpreter design pattern to implement a tiny language that supports these animations.

The implementation is also optimized for speed and memory usage:

- We have already addressed the approximation of B-splines by collections of segments of line. For all B-splines, the application creates and caches the approximations on demand, and invalidates them when moved, scaled, or rotated.
- Frequently, the algorithms look for intersections between two B-splines. Bézier curves are so that they are fully contained in the bounding box of their control points. Consequently, two Bézier curves cannot intersect if their bounding boxes are not intersecting. The same conclusion holds for two B-splines. Computing a bounding box is very fast and checking that two boxes intersect each other is also very fast. The implementation uses this fact to avoid unnecessary computations when possible.
- The implementation uses the right data structures for the right use. For example, findNeighbourPoint looks like an exhaustive search over a list. In fact, the information is directly available on the buffer onto which we ran the flood fill algorithm. For a given point, we just need to check the points directly above, below, at the left or at the right.

[2]See http://w3sdesign.com/?gr=b06&ugr=proble

- We used the Android Profiler[3] software to detect bottlenecks and correct them when necessary. For example, we detected that the reference flood fill algorithm implemented at first was very memory intensive. On slower devices, this algorithm would take up to 5 seconds to fill an area roughly the size of the whole screen (worst case scenario). Consequently, we implemented a well-known variation of the algorithm where a single object manages a contiguous line of pixels instead of having an individual object for each pixel. The speedup is around 10 times faster in practice.

The current implementation exhibits a satisfactory behavior in most cases, even on slower devices. For simple drawings, most operations complete almost instantly (<300ms), and at worst stay reasonably fast (<1000ms). As the drawings become more complex (hundreds of B-splines), the response time starts declining depending on the CPU of the device and the memory available.

5 EVALUATION OF SKETCHADOODLE

As pointed out in Section 4, SketchADoodle's response time remains below 300 ms most of the time on a normal smartphone and below 1000 ms on a low-end device. In order to test this fluidity and the overall usability of SketchADoodle as an application implementing multi-stroke gestures by Bézier curves, we conducted a user study based on a task to determine to what extent they assess the results. This section therefore describes the method followed for this user study, reports on the results, and discusses its output.

5.1 Method

5.1.1 Recruitment. We conducted a user trial of N=26 participants (11 females, 15 males), aged between 19 and 66 years (M=32.54, SD=11.53 years), who were recruited from a database of volunteers, employees of our organisation, students coming from computer-related disciplines (*e.g.*, digital strategy or ECG'2020 conference) and having different ages and backgrounds. Some participants were invited via e-mail to participate in the experiment: an e-mail was sent to them along with an introduction text about the problem (scenario), instructions about the task to be performed with the SketchADoodle application, and a form to fill-in the IBM PSSUQ (Post-Study System Usability Questionnaire) questionnaire [31], an evolution of the IBM CSUQ questionnaire [30]. Other respondents were not submitted to any selection procedure, apart from checking that they are a regular smartphone user and no hand disability (*e.g.*, a dexterity problem) is self-reported.

5.1.2 Consent. Each participant performed the experiment in a controlled environment : a quiet office environment. Prior to the task, each participant was welcomed, had the process explained to them, signed a GDPR-compliant consent form, and filled in a questionnaire on their demographic background.

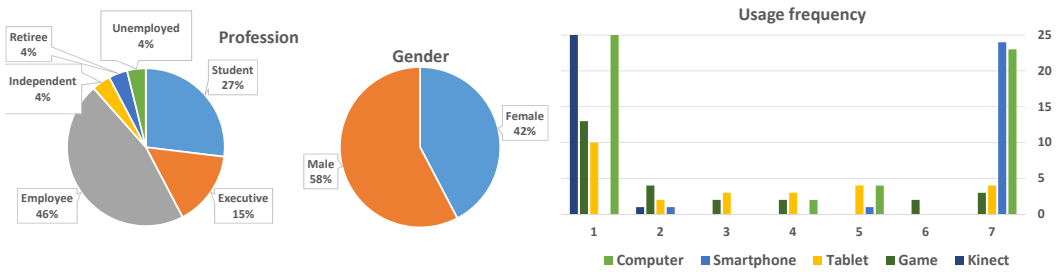

Fig. 24. Distribution of the population sampling in the experiment: (a) by profession, (b) by gender, (c) by frequency of device usage.

[3]See https://developer.android.com/studio/profile/android-profiler

5.1.3 Demographic measurements. Through a demographic questionnaire, the participants were asked to provide some personal information for statistics such as age at the time of the experiment, gender, profession, domain of activity, their background. Each participant was also asked to self-report her familiarity with five devices (1=minimum, 7=maximum, on a Likert scale [33]): computer, smartphone, tablet, game console, Kinect-like device. All participants revealed a frequency of device usage matching the desired profile: they are all sufficiently frequent computer users, particularly the smartphone (Fig. 24).

5.1.4 Training. After the questionnaire was completed, the participant installed SketchADoodle from the Android Play Store on her smartphone to preserve familiarity with the device. Each participant spends up to 5 minutes to get acquainted with SketchADoodle, but could finish early if she wanted.

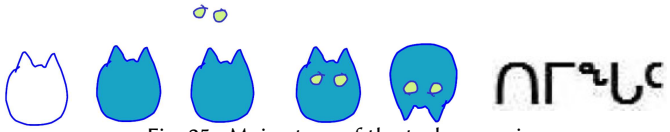

Fig. 25. Main steps of the task scenario.

5.1.5 Task. Participants were instructed to perform a task comprised of the following steps:

(1) *Perform the drawing scenario* (Fig. 25): draw a cat's face using the Freehand drawing; paint the interior with the bucket fill tool; draw two separate green cat eyes outside the face; drag them on their very right position on the face; glue them on the face to create a complete face; rotate the resulting face downside using the Resize tool; issue an Inuktitut[4] word as specified; save the results into a file sent to the experimenter using the Share tool. Drawing a cat was assumed to be a familiar animal that does not require any pre-requisites.
(2) *Fill a IBM PSSUQ form*: this questionnaire [31] enables participants to express their level of satisfaction with the usability of the setup and the testing process. This 16-question questionnaire has been empirically validated with a large number of participants on a significant set of stimuli [32], it is widely applicable for any interactive system [31], it benefits from a proved $\alpha = 0.89$ reliability coefficient between its results and the perceived system usability [30], and a rigorous procedure for administrating this questionnaire is recommended and followed [42].

5.2 Results

Each session is observed by the experimenter equipped with a chronometer to capture the task completion time. The task completion rate is measured in percentage (%) with respect to the goal of completing the entire task. Each IBM PSSUQ closed question is measured using a 7-point Likert scale [33] (1=strongly disagree, 2=largely disagree, 3=disagree, 4=neutral, 5=agree, 6=largely agree, 7=strongly agree) and the following measures are computed: system usefulness (SysUse: Items 1-6), quality of the information (InfoQual: Items 7-12), quality of the interaction (InterQual: Items 13-15), and system quality (Overall: Items 1-16). The answers provided by participants are entered into a MS Excel Spreadsheet.

[4]One of the principal Inuit languages of Canada spoken in all areas north of the tree line, including parts of the provinces of Newfoundland and Labrador, Quebec. Source: https://en.wikipedia.org/wiki/Inuktitut. This word was selected as it was expected to be unknown by participants and to issue gesture recognition based on another software, which is outside the scope of this paper.

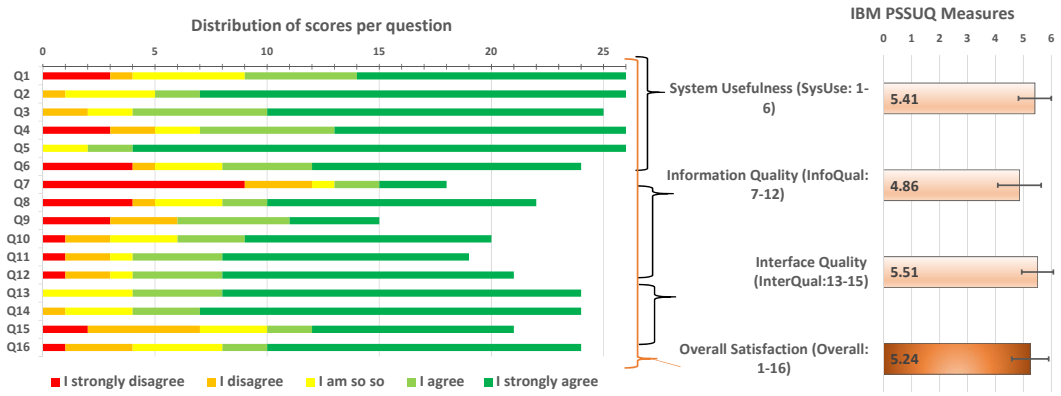

Fig. 26. Distribution of IBM PSSUQ answers and aggregated metrics.

5.3 Discussion

Overall, all participants succeeded to complete the scenario (for all of them, the task completion rate is 100%) in a reasonable amount of time between 3 minutes and 15 minutes. (M=6.42 minute, SD=3.76 minute). Fig. 26 shows the distribution of participants' answers per IBM PSSUQ question, ranging from Q1 to Q16. The average values of these questions are then aggregated into the four PSSUQ metrics, which are computed in the right part: SysUse (M=5.41, SD=0.59), InfoQual (M=4.89, SD=0.78), InterQual (M=5.51, SD=0.57), and Overall (M=5.24, SD=.66), all computed with a 95% interval of confidence (α=.05). A general observation of Fig. 26 suggests that the subjective satisfaction of participants involved in the experiment follows a rather positive trend. In particular, *"Q1. Overall, I am satisfied with how easy it is to use this system."* suggests that the general attitude of participants regarding the whole process supported by SketchADoodle could be interpreted as satisfying (M=5.81, Md=6, SD=0.72) with respect to the goal of the question Q2 *"It is simple to use this system."* which *no "very bad"* score has been given by nobody, which is also the case of Q3 *"I can effectively complete my work using this system."* and Q5 *("I am able to efficiently complete my work using this system."*. For the error management, some users found it difficult to correct their errors because they could not find the correct information by *("Q7. The system gives error messages that clearly tell me how to fix problems."* and *("Q8. The information (on-screen messages and guidance or other documentation) provided with this system is clear."* However, the limitation of the error management, in addition to "the easy way to use the system", participants also shows that the interface of this system is pleasant with their all positive answer in Q13 and a vast majority of positive trend in the *"Q14.I like using the interface of this system."*

This is also confirmed by the relatively acceptable scores aggregated in the 4 categories, *i.e.*, System usefulness, quality of the information,quality of the interaction and system quality, from the PSSUQ questionnaire [30]. Indeed, the results of the presented analysis show that the average score for all categories is between 4.8 and 6. Overall, participants did appreciate the easy use and quality interaction they have had. All values are beyond the threshold of 4 (which represents a good value), but below 5 (which represents an excellent value in average), a range that is empirically assessed as positive [42].

6 CONCLUSION AND FUTURE WORK

In this paper, we first revisited the concept of Bézier curves to characterize touch-surface multi-stroke gestures and to store them into a new Bézier-based data structure that is different from the traditional bitmap-based approach. This characterization was not just for expressing gestures as

vectors instead of points, but mainly to engineer their real-time manipulation and editing by end-users, which was not doable with the bitmap representation. For example, simple editing operations like cut, crop, deform, compose, decompose are hardly achieved in the bitmap representation: it is nearly impossible to cut a multi-stroke gesture into pieces without interpolating points in an intelligent way and it is even more complicated to select several strokes and to compose a new gesture by concatenating them. All these operations, particularly those requiring direct manipulation, are made possible thank to the Bézier structure, which then required to invent a new set of algorithms working on the Bézier curves for the purpose of gesture manipulation and engineering. We therefore defined a set of algorithms for common gesture operations, such as draw, move, rotate, resize, compose, slice, colorize at a higher level of abstraction than with the bitmap representation. For each algorithm, the method is described, the pseudo-code is provided and explained to facilitate reusability. To demonstrate such an engineering of gesture operations, we developed SketchADoodle, a free-to-download Android-based mobile application developed in Java that is efficient enough to run smoothly on an average smartphone. To foster reusability of this engineering approach, the core library for the drawing abstraction containing the algorithms (some being new, some others being adaptation of existing algorithms [6]) described in this document was translated to JavaScript and released on GitHub. We make no claim that our approach is better or worse than other Bézier-based approaches which are popular in commercially available vectorial drawing applications. These software are developed for another purpose. We however show that our approach can accommodate the real-time constraints imposed by direct manipulation of gestures, which was not permitted in the same way by bitmap-based approaches.

In order to investigate these capabilities, we are for the moment creating several vocabularies of gestures, some of them being compositions of some others, rescaling, or rotations, or more complex combinations. For instance, several alphabets are representative candidates for this purpose, such as the Moore alphabet and the Inuktikut language.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments on earlier versions of this manuscript.

REFERENCES

[1] Adobe. 2019. *PhotoShop*. https://www.adobe.com/products/photoshop.html

[2] Caroline Appert and Shumin Zhai. 2009. Using Strokes as Command Shortcuts: Cognitive Benefits and Toolkit Support. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems (CHI '09)*. Association for Computing Machinery, New York, NY, USA, 2289–2298. https://doi.org/10.1145/1518701.1519052

[3] Daniel Ashbrook and Thad Starner. 2010. MAGIC: A Motion Gesture Design Tool. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2159–2168. https://doi.org/10.1145/1753326.1753653

[4] Oscar Kin-Chung Au, Chiew-Lan Tai, and Hongbo Fu. 2012. Multitouch Gestures for Constrained Transformation of 3D Objects. *Computer Graphics Forum* 31, 2pt3 (May 2012), 651–660. https://doi.org/10.1111/j.1467-8659.2012.03044.x

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/543613.543615

[6] P.J. Barry and R.N. Goldman. 1988. de Casteljau-Type Subdivision is Peculiar to Bézier Curves. *Computer-Aided Design* 20 (1988). Issue 3. https://doi.org/10.1016/0010-4485(88)90018-8

[7] Pierre Bézier. 1978. General distortion of an ensemble of biparametric surfaces. *Computer-Aided Design* 10, 2 (1978), 116–120. https://doi.org/10.1016/0010-4485(78)90088-X

[8] Birgit Bomsdorf, Rainer Blum, and Daniel Künkel. 2015. Towards ProGesture, a Tool Supporting Early Prototyping of 3D-Gesture Interaction. *International Journal of People-Oriented Programming* 4, 2 (July 2015), 54–70. https://doi.org/10.4018/IJPOP.2015070103

[9] Ugo Braga Sangiorgi, Vivian Genaro Motti, François Beuvens, and Jean Vanderdonckt. 2012. Assessing Lag Perception in Electronic Sketching. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design (NordiCHI '12)*. Association for Computing Machinery, New York, NY, USA, 153–161. https://doi.org/10.1145/2399016.2399040

[10] Ashley Bush and Sandeep Purao. 2011. Mapping UML Techniques to Design Activities. In *Information Modeling in the New Millennium*, Matti Rossi and Keng Siau (Eds.). IDEA Publishing Group, Chapter 12, 199–217.

[11] Alessandro Carcangiu and Lucio Davide Spano. 2018. G-Gene: A Gene Alignment Method for Online Partial Stroke Gestures Recognition. *Proceedings of ACM Human-Computer Interaction* 2, EICS, Article 13 (June 2018), 17 pages. https://doi.org/10.1145/3229095

[12] Hung-Hsin Chang and Hong Yan. 1998. Vectorization of Hand-Drawn Image using Piecewise Cubic Bézier Curves Fitting. *Pattern Recognition* 31, 11 (1998), 1747 – 1755. https://doi.org/10.1016/S0031-3203(98)00045-4

[13] Fuhua Cheng and Chi-Cheng Lin. 1985. Clipping of Bézier Curves. In *Proceedings of the 1985 ACM Annual Conference on The Range of Computing: Mid-80's Perspective (ACM '85)*. ACM, New York, NY, USA, 74–84. https://doi.org/10.1145/320435.320462

[14] E. Cohen, R. F. Riesenfeld, and G. Elber. 2001. *Geometric Modeling with Splines: An Introduction*. A. K. Peters, Ltd., Natick, MA, United States.

[15] Corel Corporation. 2019. *CorelDRAW Graphics Suite*. https://www.coreldraw.com/

[16] Victor A. Debelov and Aleksandr M. Matsokin. 2000. Implementation of Set Operations and Intersection of Bézier Curves. *Computers & Graphics* 24 (2000). Issue 1. https://doi.org/10.1016/s0097-8493(99)00137-5

[17] Qiulei Dong, Yihong Wu, and Zhanyi Hu. 2006. Gesture Recognition using Quadratic Curves. In *Computer Vision (ACCV '06)*, P. J. Narayanan, Shree K. Nayar, and Heung-Yeung Shum (Eds.). Springer, Berlin, Heidelberg, 817–825.

[18] Arno Eigenwillig, Lutz Kettner, and Nicola Wolpert. 2007. Snap Rounding of Bézier Curves. In *Proceedings of the 23rd ACM Annual Symposium on Computational Geometry (SCG '07)*. ACM, New York, NY, USA, 158–167. https://doi.org/10.1145/1247069.1247101

[19] Gerald Farin. 1993. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide* (3 ed.). Academic Press Professional, Inc., San Diego, CA, USA.

[20] Jean-Dominique Favreau, Florent Lafarge, and Adrien Bousseau. 2016. Fidelity vs. Simplicity: A Global Approach to Line Drawing Vectorization. *ACM Transactions on Graphics* 35, 4, Article 120 (July 2016), 10 pages. https://doi.org/10.1145/2897824.2925946

[21] Sebastian Feuerstack, Mauro Dos Santos Anjo, and Ednaldo B. Pizzolato. 2011. Model-based Design and Generation of a Gesture-based User Interface Navigation Control. In *Proceedings of the 10th Brazilian Symposium on Human Factors in Computing Systems and the 5th Latin American Conference on Human-Computer Interaction (IHC+CLIHC '11)*. Brazilian Computer Society, Porto Alegre, Brazil, 227–231. http://dl.acm.org/citation.cfm?id=2254436.2254475

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional. http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

[23] Bogdan-Florin Gheran, Jean Vanderdonckt, and Radu-Daniel Vatavu. 2018. Gestures for Smart Rings: Empirical Results, Insights, and Design Implications. In *Proceedings of the ACM International Conference on Designing Interactive Systems (DIS '18)*. ACM, New York, NY, USA, 623–635. https://doi.org/10.1145/3196709.3196741

[24] Yves Mineur; Tony Lichah; Jean Marie Castelain; Henri Giaume. 1998. A shape controled fitting method for Bézier curves. *Computer Aided Geometric Design* 15 (1998). Issue 9. https://doi.org/10.1016/s0167-8396(98)00025-9

[25] L. J. Guibas and J. Stolfi. 1982. A Language for Bitmap Manipulation. *ACM Trans. Graph.* 1, 3 (July 1982), 191–214. https://doi.org/10.1145/357306.357308

[26] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton++: A Customizable Declarative Multitouch Framework. In *Proceedings of the 25th ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 477–486. https://doi.org/10.1145/2380116.2380176

[27] P.P. Korovkin. 2001. *Encyclopedia of Mathematics*. Springer Science+Business Media/ Kluwer Academic Publishers, Berlin, Germany, Chapter Bernstein polynomials. https://www.encyclopediaofmath.org/index.php/Bernstein_polynomials

[28] Gordon Kurtenbach, George Fitzmaurice, Thomas Baudel, and Bill Buxton. 1997. The Design of a GUI Paradigm Based on Tablets, Two-hands, and Transparency. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems (CHI '97)*. ACM, New York, NY, USA, 35–42. https://doi.org/10.1145/258549.258574

[29] Luis A. Leiva, Daniel Martín-Albo, and Réjean Plamondon. 2015. Gestures à Go Go: Authoring Synthetic Human-Like Stroke Gestures Using the Kinematic Theory of Rapid Movements. *ACM Trans. Intell. Syst. Technol.* 7, 2, Article Article 15 (Nov. 2015), 29 pages. https://doi.org/10.1145/2799648

[30] James R. Lewis. 1995. IBM computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use. *International Journal of Human-Computer Interaction* 7, 1 (1995), 57–78. https://doi.org/10.1080/10447319509526110

arXiv:https://doi.org/10.1080/10447319509526110

[31] James R. Lewis. 2002. Psychometric Evaluation of the PSSUQ Using Data from Five Years of Usability Studies. *International Journal of Human–Computer Interaction* 14, 3-4 (2002), 463–488. https://doi.org/10.1080/10447318.2002.9669130 arXiv:https://doi.org/10.1080/10447318.2002.9669130

[32] James R. Lewis. 2006. Sample Sizes for Usability Tests: Mostly Math, Not Magic. *interactions* 13, 6 (Nov. 2006), 29–33. https://doi.org/10.1145/1167948.1167973

[33] Rensis Likert. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 22, 140 (1932), 55–. http://psycnet.apa.org/record/1933-01885-001

[34] Jingbo Liu, Oscar Kin-Chung Au, Hongbo Fu, and Chiew-Lan Tai. 2012. Two-Finger Gestures for 6DOF Manipulation of 3D Objects. *Computer Graphics Forum* 31, 7pt1 (sep 2012), 2047–2055. https://doi.org/10.1111/j.1467-8659.2012.03197.x

[35] Yingbin Liu and Stephen Mann. 2008. Parametric Triangular Bézier Surface Interpolation with Approximate Continuity. In *Proceedings of the ACM Symposium on Solid and Physical Modeling* (June 2008) *(SPM '08)*. Association for Computing Machinery, New York, NY, USA, 381–387. https://doi.org/10.1145/1364901.1364956

[36] Hao Lü, James A. Fogarty, and Yang Li. 2014. Gesture Script: Recognizing Gestures and Their Structure Using Rendering Scripts and Interactively Trained Parts. In *Proceedings of the 32nd ACM International Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 1685–1694. https://doi.org/10.1145/2556288.2557263

[37] Hao Lü and Yang Li. 2013. Gesture Studio: Authoring Multi-touch Interactions Through Demonstration and Declaration. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 257–266. https://doi.org/10.1145/2470654.2470690

[38] Otto Parra, Sergio Espa na, Jose Ignacio Panach, and Oscar Pastor. 2019. An empirical comparative evaluation of gestUI to include gesture-based interaction in user interfaces. *Science of Computer Programming* 172 (2019), 232 – 263. https://doi.org/10.1016/j.scico.2018.12.001

[39] Hartmut Prautzsch. 1989. A Round Trip to B-Splines via de Casteljau. *ACM Trans. Graph.* 8, 3 (July 1989), 243–254. https://doi.org/10.1145/77055.77061

[40] O. Rashid and A. Al-Hamadi. 2015. Utilizing the Bézier Descriptors for Hand Gesture Recognition. In *Proceedings of the IEEE International Conference on Image Processing (ICIP '15)*. 3525–3529. https://doi.org/10.1109/ICIP.2015.7351460

[41] Micha Sapek and Szczepan Paszkiel. 2017. Detection of Gestures Without Begin and End Markers by Fitting into Bézier Curves with Least Squares Method. *Pattern Recognition Letters* 100, C (dec 2017), 83–88. https://doi.org/10.1016/j.patrec.2017.10.006

[42] Jeff Sauro and James R. Lewis. 2016. *Quantifying the User Experience. Practical Statistics for User Research* (2 ed.). Morgan Kaufmann, Burlington, MA, USA. https://www.sciencedirect.com/book/9780123849687/quantifying-the-user-experience

[43] Ovidiu Andrei Schipor, Radu-Daniel Vatavu, and Jean Vanderdonckt. 2019. Euphoria: A Scalable, event-driven architecture for designing interactions across heterogeneous devices in smart environments. *Information & Software Technology* 109 (2019), 43–59. https://doi.org/10.1016/j.infsof.2019.01.006

[44] L. Shen, S. Huo, M. Chen, F. Li, F. Yao, and H. Shen. 2017. Multi-touch Gesture Recognition Algorithm of Vehicle Electronic Devices-based on Bézier Curve Optimization Strategy. In *Proceedings of the 32nd Youth Academic Annual Conference of Chinese Association of Automation (YAC '17)*. 720–723. https://doi.org/10.1109/YAC.2017.7967503

[45] Min C. Shin, V. Tsap, Leonid, and Dmitry B. Goldgof. 2004. Gesture Recognition using Bézier Curves for Visualization Navigation from Registered 3-D Data. *Pattern Recognition* 37, 5 (2004), 1011–1024. https://doi.org/10.1016/j.patcog.2003.11.007

[46] Lucio Davide Spano, Antonio Cisternino, Fabio Paternò, and Gianni Fenu. 2013. GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition. In *Proceedings of the 5th ACM International Symposium on Engineering Interactive Computing Systems (EICS '13)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2494603.2480307

[47] Eugene M. Taranta, Andres N. Vargas, and Joseph J. LaViola. 2016. Streamlined and accurate gesture recognition with Penny Pincher. *Computers and Graphics* 55 (2016), 130 – 142. https://doi.org/10.1016/j.cag.2015.10.011

[48] Inkscape Team. 2019. *Inkscape - Draw Freely.* https://inkscape.org/

[49] The GIMP Team. 2019. *GIMP - GNU Image Manipulation.* https://www.gimp.org/

[50] Jean Vanderdonckt, Paolo Roselli, and Jorge Luis Pérez-Medina. 2018. !FTL, an Articulation-Invariant Stroke Gesture Recognizer with Controllable Position, Scale, and Rotation Invariances. In *Proceedings of the 20th ACM International Conference on Multimodal Interaction (ICMI '18)*. ACM, New York, NY, USA, 125–134. https://doi.org/10.1145/3242969.3243032

[51] Radu-Daniel Vatavu. 2017. Improving Gesture Recognition Accuracy on Touch Screens for Users with Low Vision. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (May 2017) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 4667–4679. https://doi.org/10.1145/3025453.3025941

[52] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. 2018. $Q: A Super-quick, Articulation-invariant Stroke-gesture Recognizer for Low-resource Devices. In *Proceedings of the 20th ACM International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI '18)*. ACM, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/3229434.3229465

[53] Julie R. Williamson, Stephen Brewster, and Rama Vennelakanti. 2013. Mo!Games: Evaluating Mobile Gestures in the Wild. In *Proceedings of the 15th ACM on International Conference on Multimodal Interaction (ICMI '13)*. ACM, New York, NY, USA, 173–180. https://doi.org/10.1145/2522848.2522874

[54] Atsushi Yamada and Fujio Yamaguchi. 1996. Homogeneous Bounding Boxes as Tools for Intersection Algorithms of Rational Bézier Curves and Surfaces. *The Visual Computer* 12, 4 (01 Apr 1996), 202–214. https://doi.org/10.1007/BF01782323

[55] Chee K. Yap. 2006. Complete Subdivision Algorithms, I: Intersection of Bézier Curves. In *Proceedings of the 22nd ACM Symposium on Computational Geometry (SCG '06)*. ACM, New York, NY, USA, 217–226. https://doi.org/10.1145/1137856.1137890

[56] Shumin Zhai, Per Kristensson, Caroline Appert, Tue Andersen, and Xiang Cao. 2012. Foundational Issues in Touch-Surface Stroke Gesture Design: An Integrative Review. *Foundations of Trends in Human-Computer Interaction* 5, 2 (feb 2012), 97–205. https://doi.org/10.1561/1100000012

[57] Lejun Shao; Hao Zhou. 1996. Curve Fitting with Bézier Cubics. *Graphical Models and Image Processing* 58 (1996). Issue 3. https://doi.org/10.1006/gmip.1996.0019

Received February 14th, 2020