A New Cubic B-Splines Design Method for Pen Input Environment

Dae Hyun KIM†, Nonmember and Myoung-Jun KIM††∗, Member

SUMMARY  Pen-input is not a new means for CAD designers, in particular, in the concept design phase. Meanwhile, B-Splines are well known curve and surface design tool in 3D shape modeling in the final modeling stages in which neat curves and surfaces should be produced. In this paper, an intuitive B-Spline design method that can be used for the CAD systems both in conceptual modeling phase and in later design phases is proposed. Unlike the control point based interactive modification schemes for the B-spline curves and surfaces, we extend what has been called the “touch-and-replace” method used for poly-line modification in the late 1980s to B-Splines; our approach uses successive pen strokes to modify the final shape of the existing B-Spline curves and surfaces. We also show related user test results in this paper as an empirical proof.

key words: pen-input, B-Splines, concept design, computer aided design, pen strokes

1. Introduction

A significant achievement for one-handed freeform drawings was made by Baudel[5]. He proposed a brand new method by which users can freely modify their drawings with successive pen markings; for example, the user gives a pen marking over an existing curve \( C(u) \), and the system sculpts the curve considering the shape of the pen marking—for a comprehensive overview, see Fig. 1. One restriction of this approach is that it uses off-line spline curves, which means that the edited curves are represented by piecewise linear curves; the program generates \( G^1 \) continuous Bézier curves only after an explicit user request (e.g. by pushing a button). Adobe Illustrator uses this scheme but is limited to Bézier curves [1].

There exist different curve drawing methods by which the user can use both hands in drawing. Singh proposed an interactive method to convert a B-Spline curve into stylish French curve segments by iteratively breaking the curve, fitting them with elliptic curves, and merging them together [17]. Balakrishnan et al. implemented tape drawing method which has been used to draw smooth curves in car design [3].

Sachs et al. proposed a 3D curve sketching environment, 3-Draw, for a virtual environment [7]. In 3-Draw, the user holds a stylus in one hand and a tablet in the other.

These tools serve to draw 3D curves but they offered no appropriate curve representation of the drawn curve and intuitive modification of the drawn curve. Krueger suggested similar environment using computer vision techniques to manipulate four control points along a spline curve [13]. Poston et al. and Cutler et al. also suggested two handed input for virtual environment, however, they are more concerned about fluent user interactions rather than inventing a curve drawing method [6], [16].

We now focus on the B-spline curve editing. Fowler et al. [9] found a new way to directly manipulate B-Spline curves, using constrained minimization techniques. This method is inherently dependent on the structure of the knot vector. For example, when the knot vector is dense, the locality of the modification becomes small, leaving the burden to have a good knowledge over the knot structures.

Fig. 1  Top: The target B-Spline curve \( C(u) \), and the user-drawn modification pen marking \( h \). Middle: Computing the shape change considering both the knot vector of the existing curve and the given pen marking. Bottom: Local modification to the existing curve.
Banks et al. [4] proposed a new method to edit B-Spline curves by cutting and sketching control polygons. It imposes an additional burden on the user to understand what control points are, rather than curve shapes. However, their examples revealed that editing the control polygon by a pen marking is quite intuitive, although it is not as intuitive as a direct curve manipulation. At the end of Sect. 2, we show that the our algorithm generates a control polygon of the curves in a similar way as the user sketched a control polygon in [4].

Zheng et al. proposed a new method to deform a curve by matching it with another input curve [19]. The authors used knot removal techniques [15] which increase computation time and sacrifice smooth shape changes in the portion of the transition between the original curve and the input curve.

Recently, Grossman et al. proposed a method to draw nice curves using traditional tape drawing; their achievement is about allowing the user to draw aesthetic curves such as car body line. For this, large display wall is needed and special interaction devices have been invented too [11]. Compared to this approach, ours are more general and targeted for small-scale applications such as Adobe Illustrators and plug-ins for existing 3D modeling systems.

Applying Baudel’s approach, which seems to us most fit for pen-input display, to the B-Spline curves brings about two problems. First, each pen input needs more knots to be inserted into the existing knot vector, in particular, around the transition part between the original curve and the sketched curve. Therefore, iterative sketches on the existing curve will end up with many unnecessary knots; considering that multiple knots can reduce the continuity of the final curve, it is desirable to have only necessary knots to keep the desired shape. Second, similar to the first problem, often user may want to simplify the existing curve by giving a new pen marking, as shown in Fig. 7. In this case, the user should know about knot structure.

Michaelik et al. [14] suggested a method to sketch B-Spline curves but did not detail how to address the aforementioned problems. Our approach here combines two methods from Fowler et al. [9] and Baudel [5]; by which it resolves the locality constraints inherent from B-Spline knot structure in the pen-input sketching environment. The former modifies B-Spline curve based on the underlying knot structure of the curve and is very restricted in determining the locality of modification; for example, if there are dense knots for the B-Spline curve, moving a curve point will change only very small region of the curve regardless user’s intention. Meanwhile, the latter works only on Bézier curves. Our approach takes the two approaches such that the smooth B-Spline curves can be sketched through pen-input; given a pen marking drawn over a B-Spline curve, as shown in Fig. 1, the algorithm defines how to modify the curve considering both the pen marking and the existing curve, resulting in an implicitly defined shape change. The main contributions of our work can be summarized as follows:

**Direct control** In this method, the user’s pen markings are directly applied to a cubic B-Spline curve. **Locality** means that part of the curve for sculpting is implied by the shape of the input pen marking.

**On-the-fly C² curve** The cubic B-Spline curve provides C² continuity all over the curve unless it has redundant knots inside. Therefore, the pen marking, which locally and directly operates on the curve, will produce another C² continuous B-Spline curve.

**Fewer knots** The algorithm incrementally adds new knots. Thus only necessary knots are introduced in the transition area between the original curve and the pen marking. For example, when the original curve has a complicated shape (and thus many control points) a simplifying stroke may modify the curve to contain fewer control points.

This paper also demonstrates how this approach can be used in deforming surfaces intuitively as well. First, we show how to re-sketch a parameter line of a B-Spline surface, resulting in a surface deformation. Second, we re-sketch a wire (B-Spline curve) that is bound to a surface, resulting in an implicit surface deformation described in [18].

### 2. Preliminaries

Let us first assume that we have the following cubic B-Spline curve:

\[
C(u) = \sum_{i=0}^{n} B_i P_i
\]

where \(P_i\) is a control point and \(B_i\) is a B-Spline basis functions over a knot vector \(T\):

\[
T = \{t_0, t_1, t_2, t_3, \ldots, t_{n+1}, t_{n+2}, t_{n+3}, t_{n+4}\}
\]

Figure 1 shows the overall procedure to be taken to modify the given B-Spline curve \(C(u)\):

- **STEP1:** The user gives a pen marking, \(h = \{h_i | i = 0, \ldots, m\}\).
- **STEP2:** The algorithm computes the shape change in a digitized point sequence that consists of the following (See also Fig. 1):
  - \(h\): The input pen marking.
  - \(\bar{B}, \bar{B}':\) Smooth transitions from \(h\) to the existing curve (dotted curve in the middle of Fig. 1).
  - \(\bar{E}, \bar{E}':\) The remaining curve portions within the knot interval which contains each end of the transitions.
- **STEP3:** The algorithm locally modifies the \(C(u)\) according to the digitized shape change.

Here, we assume that the user is iteratively modifying the existing curve but not drawing a new curve, as shown in Fig. 8; when the user wants to draw a new curve, the user has to explicitly change the mode, for example, by pushing a button. Also, since \(C^0\) continuity can be achieved by simply
connecting two $C^2$ curves and simply adding another user interface, this paper focuses only on $C^2$ continuous sculpting; any user pen marking placed near an existing curve results in a $C^2$ continuous local modification to the curve.

The remainder of this paper is organized as follows. Section 3 explains how to compute the shape change intended by the user pen marking. Section 4 explains constrained curve modification to the cubic B-Spline curve. Section 5 demonstrates how to use this B-Spline curve modification scheme to the B-Spline surfaces.

### 3. Digitizing the Shape Change

This section details STEP 2 in Sect. 2. To properly extract the curve portions that we will piece together with the input pen marking $h$ to form the shape change, we need to locate several break points that divide the existing curve. In the following, we detect the breaking points in the parameter domain of $C(u)$, $[t_k, u_{ps}, u_e, u_{pe}, t_1]$, as shown on the top of Fig. 1.

We first establish approximate correspondence between $C(u)$ and $h$; from the parameter domain of $C(u)$, locate an interval $[u_s, u_e]$, which approximately corresponds to the pen marking by simply searching for the closest points $C(u)$ from two endpoints of the pen marking. As a result, as shown in Fig. 1, $C(u)$ of $u \in [u_s, u_e]$ constitutes the approximate correspondence.

To determine where $B'(t)(B')$ starts or ends in the parameter domain of $C(u)$, we define two terms, $u_{ps} = u_s - d_1$ and $u_{pe} = u_e + d_2$; where $d_1 = 3 ||C(u_s) - h_0||$ and $d_2 = 3 ||C(u_e) - h_m||$. For a broader blending area, we could choose a value larger than three. Here notable is that $u_{ps}$ and $u_{pe}$ are inferred from the real 3D coordinate space: distance between the end point of the pen input and a curve point. Therefore, it is recommended that the existing B-Spline curve to be sculpted has arc length parametrization. Nevertheless this is not mandatory; after modifying the non-arc-length parameterized curve several times with our algorithm, this curve will be locally closer to arc-length parametrization. Blending area becomes larger when the constant "3" becomes large because, thereby, the local curve (later defined as $C_{loc}$) that embeds the shape change covers larger area of the existing curve. This constant "3" has been chosen through our experiments.

The two curve portions corresponding to the two intervals, $[u_{ps}, u_s]$ and $[u_e, u_{pe}]$, will be used to make a continuous geometric transition from $C(u)$ to the pen marking $h$. Now from the knot vector $T$, look for the $t_k$ and $t_l$: $t_k = \text{max}(t_i \leq t_p, t_i \in T)$ and $t_l = \min(t_i \geq t_p, t_i \in T)$. We now have all the breaking points we need to decompose the curve $C(u)$. Subsequently, only the left part of the curve will be explained, since the right part functions analogously.

- Call the curve portion $C(u)$ of $u \in [t_k, u_{ps}]$, $E'(u)$.
- Convert $C(u)$ of $u \in [u_{ps}, u_s]$ into a piecewise linear curve, $B'(t)$, which has chord-length parametrization, $t \in [u_{ps}, u_s]$.

- Blend $B'(t)$ with a line segment $L$, which is tangent at $h_0$ and has length $d_1$, and keep it into $B'(t)$ itself [5].

We now know how the original curve should be changed by the shape change embodied by several curve portions, $E'(u)$, $B'(t)$, $h$, $B'(u)$, and $E'(u)$ (See also Fig. 1). Piece them together into one piecewise linear curve $\bar{h}$ with the parameter domain $[t_k, t_1]$. In the next section, $\bar{h}$ will be approximated with $C_{loc}(u)$, which will replace part of $C(u)$, $u \in [t_k, t_1]$, resulting in the local modification. At a first glance, $E'(u)$ seems unnecessary. To modify $C(u)$ only within the domain $[u_{ps}, u_{pe}]$, inserting multiple knots at $u_{ps}$ is required. This will introduce unnecessarily many knots; several times of sketching over the $C(u)$ will introduce uncontrollably many knots, which at some point will need running knot removal algorithm. Therefore, we let the B-Spline curve $C_{loc}(u)$ approximate the piecewise linear curve over knot interval $[t_k, t_1]$, iteratively minimizing pointwise errors and adding necessary knots.

Also the case in which $u_s - d_1$ exceeds the parameter boundary of $C(u)$ should be considered, as shown in Fig. 2. There are two possibilities: The first one is to let $B'(t) (or h)$ replace the left part of $C(u)$. The second one lets the final curve interpolate the end point of $C(u)$.

4. Updating the B-Spline Curve $C(u)$

This section explains how to iteratively approximate the shape change, $\bar{h}$, with a cubic B-Spline curve $C_{loc}(u)$ and replace part of $C(u)$ with that; conceptual overview is shown in Fig. 3. Therefore, the final curve is obtained by locally modifying $C(u)$ through a user pen marking. The new curve $C_{loc}(u)$ is constrained to follow the target curve $C(u)$ by sharing the control points—which are called constrained control points—near the transition part with those of $C(u)$; for example, two control polygons coincide in part as notified in Fig. 3. The iterative approximation process for $C_{loc}(u)$ starts with a new knot vector with the knot interval $[t_k, t_1]$ empty; the algorithm iteratively adds new knots in $[t_k, t_1]$ and computes the unknown control points using constrained least squares. Changing the knot vector $T$ into $T_{loc}$ and later adding a new knot into $T_{loc}$ causes changes in
the constrained control points as well; the algorithm checks whether the constrained control points from $C(u)$ are influenced (i.e., refined) by the knot insertion algorithm [12]. Usually, inserting a new knot around $t_k$ would refine the control polygon and shorten (or extend) the edge $P_{k-3}P_{k-2}$. This influence can be checked by looking at knot span $[t_k,t_{k+2}]$ (and $[t_{k-2},t_k]$) of $T$; to simplify the description of the algorithm, in the following, we define mark $z_k$.

First, we need to initialize a new knot vector $T_{loc}$ for $C_{loc}(u)$.

- Collect knots from the knot vector, $T$, of $C(u)$ and make a new knot vector for $C_{loc}(u)$,

$$T_{loc} = \{t_{k-3},t_{k-2},t_{k-1},t_k,t_{k+1},t_{k+2},t_{k+3}\}. \tag{3}$$

- If there are more than one knot in $(t_k, t_i)$ of $T$, create two marks to watch any change to the constrained control points:

$$z_k = t_{k+1}, z_l = t_{l-1}. \tag{4}$$

Let us note that, constructing the knot vector $T_{loc}$ from $T$, all the knots in $(t_k, t_i)$ have been ignored since they do not reflect the parametrization of $h$ any longer. Instead, new knots will be inserted while approximating $h$ with $C_{loc}$. The marks simplifies the algorithm to keep track of the knots, newly inserted, at the intervals $[t_k,t_{l-2}]$ (and $[t_{k-2},t_k]$), so that we could notice when and how the constrained control points from $C(u)$ are affected.

Overall algorithm can be summarized as the following procedure:

**Algorithm 1-1: LocalCurveApproximation**

**INPUT:** $C(u)$, $T_{loc}$, tolerance $\varepsilon$, and $h$

**CHANGE:** $C_{loc}(u)$ with $T_{loc}$ updated

**(S1)** Set up the constraints $\mathcal{Z}$ = the constrained control points $(P_{k-3}, P_{k-2})$ and $(P_{l-2}, P_{l-1})$

**(S2)** $C_{loc}(u) = \text{ConstrainedLeastSquares} \ (\text{constraints}, T_{loc}, h)$;

**(S3)** Error = $\text{FindMaxError} (C_{loc}(u), h)$;

**(S4)** if Error $< \varepsilon$, return CONVERGENCE

**(S5)** $i = \text{TraceMostDeviatedKnotInterval}(C_{loc}(u), h, T_{loc})$;

**(S6)** $\text{SubdivideKnotVector} \ (\text{constraints}, \frac{t_i + t_{i+1}}{2}, T_{loc}, C_{loc}(u))$;

In step (S1), the constraints for the new curve $C_{loc}(u)$ are set. The control points of $C(u)$ that influence the curve segment within $[t_k,t_{k+1}]$ and $[t_{l-1}, t_l]$ are $(P_{k-3}, P_{k-2}, P_{k-1}, P_k)$ on the left side and $(P_{l-4}, P_{l-3}, P_{l-2}, P_{l-1})$ on the right side (See also Fig. 3). For the first iteration, therefore, $(P_{k-3}, P_{k-2})$ and $(P_{l-2}, P_{l-1})$ are used as constraints for the ConstrainedLeastSquares in step (S2) (for more details on the constrained optimization, we refer the readers to [10]). However, in a certain condition, which may occur from the next iterations inserting a new knot, $P_{k-2}$ and $P_{l-2}$ are to be modified in step (S6); we discuss how to set up the constraints later.

In step (S3), $\text{FindMaxError}$ measures the maximum error: suppose the piecewise linear curve $\tilde{h}$ has the following chord-length parameterization:

$$\{u_0, \cdots, u_m\}, u_0 = t_k, u_m = t_l. \tag{5}$$

Then the maximum error $M$ is defined as follows:

$$M = \max_{i=0}^{m} ||C_{loc}(u_i) - \tilde{h}_i||. \tag{6}$$

If the maximum error is less than the input tolerance $\varepsilon$, the output curve $C_{loc}(u)$ is considered converged and the algorithm stops. $\text{TraceMostDeviatedKnotInterval}$ of the step (S5) searches for the $i$-th knot span of $T_{loc}$, which has the most deviated error, and determines a new knot:

$$t^* = \frac{t_i + t_{i+1}}{2}. \tag{7}$$

In step (S6), $\text{SubdivideKnotVector}$ inserts the new knot $t^*$ into $T_{loc}$. The above algorithm takes $(P_{k-3}, P_{k-2})$ for the constraints, which come from $C(u)$. When the new knot is inserted near a mark, $(P_{k-3}, P_{k-2})$ of $C(u)$ are also influenced by a B-Spline knot insertion algorithm [12]; thus $C_{loc}(u)$ takes the constrained control points accordingly modified by the knot insertion algorithm. The marks are used to see when these constrained control points are affected by the knot insertion.

To explain the influence we need some new definitions. Suppose two marks, $z_k$ and $z_l$ are inside the $T_{loc}$.

**Definition 4.1** (Multiplicity of a mark $z$: $\mu(z)$):

The number of the knots of the $T_{loc}$ that are located in $(z - \varepsilon, z + \varepsilon)$, counting in the mark $z$ itself, is called the multiplicity of the mark; denote it by $\mu(z)$.

**Definition 4.2** (Neighborhood of a mark $z$: $N(z)$):

The neighborhood of the mark is an open interval $(t_{\text{prev}}, t_{\text{next}})$, where $t_{\text{prev}}$ (or $t_{\text{next}}$) is the closest knot to $z$, approaching from the left (right) side of $z$. Denote the neighborhood of $z$ by $N(z)$.

Whenever a new knot inserted into the neighborhood of a mark, the constraints, $(P_{k-3}, P_{k-2})$ and $(P_{l-2}, P_{l-1})$, should
be modified, according to the B-Spline knot insertion algorithm [12]; Figure 4 illustrates the knot insertion principle. From this principle, for example, when a new knot \( t^* \) is inserted into the neighborhood of a mark, \( z_k \), the control point \( P_{k-2} \) is modified as follows:

\[
P_{k-2} = \left( 1 - \frac{t^* - t_{k-2}}{z_k - t_{k-2}} \right) P_{k-3} + \left( \frac{t^* - t_{k-2}}{z_k - t_{k-2}} \right) P_{k-2}.
\]

Let us note that when \( t^* \) goes beyond the \( z_k \) to the right the resulting \( P_{k-2} \) is an extrapolation point of the line segment \( P_{k-3}P_{k-2} \); otherwise, it is an interpolation point. It is notable that our algorithm generalizes both the interpolation and extrapolation cases, thereby, avoiding unnecessary knots. In the same way to Eq. (8), when the new knot is inserted into the neighborhood of \( z_l \), the control point \( P_{l-2} \) is modified as follows:

\[
P_{l-2} = \left( 1 - \frac{t^* - z_l}{t_{l+2} - z_l} \right) P_{l-3} + \left( \frac{t^* - z_l}{t_{l+2} - z_l} \right) P_{l-2}.
\]

We now define the step (S6), \textit{SubdivideKnotVector}, using the above definitions:

\textbf{Algorithm 1-2: SubdivideKnotVector}

\textbf{INPUT:} constraints, \( t^* \), \( T_{loc} \), and \( C_{loc}(u) \)

\textbf{CHANGE:} \( T_{loc} \) and the constraints updated

\textbf{if} no marks are defined

insert \( t^* \) into \( T_{loc} \)

\( z_k = z_l = t^* \)

Update the constraints according to Eq. (8) and Eq. (9)

\textbf{else}

\textbf{if} \( \mu(z_k) \geq 1 \) and \( t^* \in N(z_k) \)

Update the constraints according to Eq. (8)

\( z_k = t^* \), insert \( t^* \) into \( T_{loc} \)

\textbf{if} \( \mu(z_l) \geq 1 \) and \( t^* \in N(z_l) \)

Update the constraints according to Eq. (9)

\( z_l = t^* \), insert \( t^* \) into \( T_{loc} \)

This approach using interpolation and extrapolation of control points in accord with the newly inserted knots enhances the quality of local modification because it properly

sets up the constraints. Therefore, it can internally manipulate control polygon like Banks’ software [4] that lets the user manually sculpt the control polygon itself; such sculpting effect is well shown at the bottom of Fig. 5, 6, and 7.

An example of extrapolating control points can be found in Fig. 7 where the user intends to simplify the target curve.
5. Results and Applications

Now we discuss the implementation issues of our algorithm and also present more results in 2D curve sketching as well as 3D curve sketching that leads to surface sketching. We also made a user test where the selected users (experienced designers) draw curves using our curve drawing tool and a commercial modeling S/W MAYA [2].

5.1 Implementation

We implemented the sketching algorithm described using the Microsoft Visual C programming language and tested on a Fujitsu 1.4 GHz Tablet PC running under the Windows XP Tablet PC Edition. We chose OpenGL as a graphics library and used a public domain library, IRIT [8] to represent and manipulate B-Spline curves and Surfaces. The algorithm runs interactively for 2D curve sketching, but shows a certain response time to apply this to 3D surface editing; this latency does come from deforming 3D surface.

Figure 8 shows that the user successively sketches the existing curve (left column) and the algorithm locally modifies the B-spline curve. The remainder of the figures demonstrates the B-Spline surface sketching applying the curve sketching metaphor.

5.2 Extension to Surface Sketching

We now show how to incrementally sketch B-Spline surfaces using the algorithm developed in Sect. 3 and Sect. 4. In this section two methods to sketch B-Spline surfaces, applying the curve sketching method, will be demonstrated:

- Surface sketching by re-sketching a \( u \)-curve (or a \( v \)-curve) of the existing surface.
- Surface sketching by re-sketching an arbitrary B-Spline curve bound to the existing surface, which has been proposed in [18].

For the first method, a pen marking is used to extract a
KIM and KIM: A NEW CUBIC B-SPLINES DESIGN METHOD FOR PEN INPUT ENVIRONMENT

Fig. 9  Sketching B-spline surfaces. Top Left: A default auxiliary surface, which is developable (red transparent surface), is constructed perpendicular to the u-curve (or the v-curve). Top right: The user draws a curve onto the auxiliary surface to sculpt the actual surface. Bottom: The resulting surface is shown.

To extract the u-curve from a bicubic B-Spline surface, the user draws a pen marking. This pen marking is projected onto the uv parameter domain of the surface. Least squares fitting is applied to determine a u-curve (or v-curve, subsequently u-curve) which is closest to the pen marking (See Fig. 9 top left). A further user interaction comes to setup an auxiliary surface; auxiliary surface is, in this case, defined to be a ruled surface that passes though the u-curve along a user defined direction. Then the user re-sketches the u-curve to deform the surface (See Fig. 9 top right). The re-sketched pen marking will be projected onto the auxiliary surface; the locally modified resulting curve is a 3D curve since the auxiliary surface is not necessarily to be a plane. To deform the surface as well, we need to solve a system of linear equations as defined in [9].

The step (S5) of Alg. 1-1, knot selection, needs a slight modification to consider the surface knot space. Newly added knots for the u-curve re-sketched would increase the number of control points of the target surface after each sketch interaction, and the number of knots will eventually explode after several steps of deformation. Therefore, a knot vector of the surface before the deformation is reused for the knot selection. Two knot vectors from the u-curve and from the surface must be compatible with each other. As noted in Sect. 4, when the pen marking simplifies the parameter line, a knot removal algorithm [15] can be used to simplify the knot vector of the target surface as well. Another example of surface deformation by re-sketching a parameter line of a B-Spline surface is demonstrated in Fig. 10.

The second method, re-sketching an arbitrary curve on a target B-Spline surface and deforming the surface to exactly follow the surface curve involves a singular matrix to be solved [14]; moreover, this singular system is mostly ill-conditioned and causes unexpected results. Such an exact method, however, is not necessary for interactive sketching applications. Therefore, to demonstrate more complex examples of sketching B-Spline surfaces, we apply our curve sketching to the WIRES of [18]. In this method, deforming the curve bound to the target surface leads to implicitly deforming the surface as well. In Fig. 11, an intersection curve between the target surface and the auxiliary surface—in this case, the auxiliary surface is an arbitrary surface—is re-sketched and the target surface is deformed following the re-sketched B-Spline curve projected onto the auxiliary surface.

5.3 User Evaluations

For more quantitative evaluation, we conducted a user test where we selected ten subjects to draw four objects (silhouette as well as feature curves) shown in Fig. 12 using both our system and a commercial 3D modeling tool MAYA [2]. In this test, the observer watched each subject drawing the target curves on each object image embedded into each sys-
Fig. 12 Drawing curves on four different objects.

Fig. 13 Test results for the car object: left part shows times taken to draw the curves with our tool for each subject. Right part shows times with the commercial tool.

Fig. 14 For each object average drawing time is shown.

6. Conclusion and Future Work

This paper proposed and implemented the constrained cubic B-spline curve modification based on pen-input, which has been proven to be a useful tool for sketching smooth curves and surfaces. As a future work, we suggest to bring this into a virtual 3D environment where the user can draw 3D curve more freely with 3D pointing devices; which has been otherwise quite awkward for the users using pen-input display (i.e., by projecting 2D sketching onto an auxiliary surface), as demonstrated so in Sect. 5.

References

Dae Hyun Kim  received his BS degree in 1994 in computer science from Seoul National University of Technology, Korea, his MS in 1996 in computer science from Korea University, his PhD in 2004 from Bremen University, Germany. Since his PhD, he had been a post-doctoral researcher under the supervision of Prof. Myoung-Jun Kim, Ewha Womans University, Korea. From 2004 to 2006 he worked for the Institute for Graphic Interfaces as a senior researcher and the leader of the Tiled Display Team. Since 2006 he has been working for FXCode Corp. He also worked for the Korea Electronics and Telecommunications Research Institute (ETRI) from 1996 to 1999 as a researcher for the computer graphics team. His research focuses on geometric modeling, sketch-based modeling, and, recently, appearance modeling.

Myoung-Jun Kim  received his BS degree in 1989 in computer science from the Korea Institute of Technology, Korea, his MS in 1992 and his PhD in 1996, both in computer science, from the Korea Advanced Institute of Science and Technology (KAIST). Since his PhD, he has been a postdoctoral researcher at the University of Washington in 1996. He has also worked for the Korea Electronics and Telecommunications Research Institute (ETRI) from 1997 to 2000 as a senior researcher for the Computer Graphics Team. From 2001, he has been an assistant professor at Ewha Womans University. His technical research and experience include mathematical modeling, computer graphics, and, recently, digital appearance modeling.