

A simple construction method for sequentially tidying up 2D online freehand sketches

Shengfeng Qin
School of Engineering and Design
Brunel University, Uxbridge
Middlesex, UB8 3PH, UK
Sheng.feng.qin@brunel.ac.uk

David K Wright
School of Engineering and Design
Brunel University, Uxbridge
Middlesex, UB8 3PH, UK
David.wright@brunel.ac.uk

Ivan Jordanov
School of Computing
University of Portsmouth
Portsmouth PO1 3HE, UK
Ivan.Jordanov@port.ac.uk

ABSTRACT

This paper presents a novel constructive approach to sequentially tidying up 2D online freehand sketches for further 3D interpretation in a conceptual design system. Upon receiving a sketch stroke, the system first identifies it as a 2D primitive and then automatically infers its 2D geometric constraints related to previous 2D geometry (if any). Based on recognized 2D constraints, the identified geometry will be modified accordingly to meet its constraints. The modification is realized in one or two sequent geometric constructions in consistence with its degrees of freedom. This method can produce 2D configurations without iterative procedures to solve constraint equations. It is simple and easy to use for a real-time application. Several examples are tested and discussed.

Keywords

Sketch-based design, Geometric constraints solver, Computer-human interface.

1. INTRODUCTION

In a conceptual design stage, product design and development often takes the form of the artist's sketches. In order to reduce product lead-time, transition directly from the stylist's sketches to a computer model is desirable [Zel96]. To meet this need, research has been carried out to develop a sketch-based user interface, recognize 2D primitives through a 2D sketch segmentation, classification and identification process and infer 3D objects [Qin00, Qin01]. Recognized 2D primitives include straight lines, circles, circular arcs, ellipses and elliptical arcs, or B-spline curves. These 2D entities are fitted with least square algorithms, but in general, they are not connected properly to reflect the user's intention. Modification in 2D is therefore required in order to give them proper position, direction and connections among them. Identification of various 2D constraints such as connectivity, parallelism and perpendicularity, is prerequisite for the 2D

modification and further 3D interpretation.

This paper presents a novel and simple constructive approach to beautifying 2D geometry based on freehand sketches. It includes three parts: (1) inferring 2D geometric constraints from rough sketches; (2) finding a solution to satisfy the constraints wherever possible; and (3) finally modifying drawing to a desired 2D geometry. The approach is based on constructive principles and degrees of freedom analysis.

This paper is organized as follows. Section 2 reviews related works and Section 3 shows constraints classification and capturing. Sections 3 and 4 describe the analysis of degrees of freedom for objects and constraints. Section 5 discusses the constructive rules. Finally, examples and conclusion are given in Section 6 and 7.

2. RELATED WORKS

Shpitalni and Lipson [Shp97] presented an approach for classifying pen strokes in an on-line sketching system and an adaptive method for clustering disconnected end-points. The following steps are applied by the clustering algorithm: (1) creating raw vertices at all endpoints of entities in the drawings, (2) determining the radius of the tolerance circle around each raw vertex, (3) identifying and grouping pairs of raw vertices when each member of the pair falls within the other members tolerance circle, (4) iteratively grouping chains of pairs into clusters and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Conference proceedings ISBN 80-86943-03-8
WSCG'2006, January 30-February 3, 2006
Plzen, Czech Republic.
Copyright UNION Agency – Science Press*

finally (5) placing a vertex node at the centroid of each cluster and adjusting lines and arcs accordingly. This method is only concerned about coincidence constraint among endpoints. It needs to wait for a completion of sketch input and then starts to tidy up. The inaccuracy of interpretation may increase. Furthermore, taking the centroid of each cluster and adjusting geometry accordingly may change some relations such as parallelism.

An automatic beautifier for drawings and illustrations was studied by Pavlidis and Van Wyk [Pav85]. A method was developed for inferring constraints that are desirable for a given (rough) drawing and then modifying the drawing to satisfy the constraints wherever possible. Drawings here were polygon-oriented. The relations (constraints) examined are: approximate equality of slope or length of sides (line segments), collinearity of sides, and vertical and horizontal alignment of points. The system restricted the number of constraints to avoid an explosion in processing time. The solution of the constraints is not always guaranteed.

A similar system Easel [Jen92] was developed by Jenkins and Martin. It behaves in as nearly an automatic manner to infer constraints and then tidy up the drawing. Geometric entities include straight lines, circular arcs and composite Bezier curves. Relations consist of unary relations such as close to a point and pairwise relations. The constraints are satisfied with multiple enforcements based on scenario analysis. Easel's performance is simply not good enough for practical sketches consisting of perhaps hundred of elements because of a time delay.

In general, geometric constraints can be topological (structural) ones, such as incidence, tangency, parallelism, perpendicularity, etc., or metric (dimensional) ones, such as distance or angle. When solving geometric constraints, a solver must produce an instance of given topology (structure) that exactly satisfies given constraints. The main geometric constraint solvers can be divided into two categories: and constructive solvers. The equational solvers translate geometric constraints into a system of algebraic equations, which is then solved using different iterative techniques. These solvers are based on numerical methods [Jen92] and symbolic methods [Gao98]. The shortcomings of numerical methods include slow runtimes, numerical instabilities and difficulties in handling redundant constraints. The disadvantage of symbolic solvers is that they are still too slow for real-time computation [Li02].

The constructive solvers [Bou95] make use of the fact that most configurations in an engineering drawing are solvable by ruler, compass or protractor. A planning phase is carried out to transform a

constraint problem into a step-by-step constructive form that is easy to compute, and then the constraint system can be solved efficiently. Generally speaking, the above solvers rely heavily on the user interaction to produce constraints either by stating relations, or by adding dimensions. These systems also focus on *rc-configuration* (ruler and compass) problems in which primitives such as ellipses and elliptic arcs are excluded.

Our system can automatically infer constraints during sketching with its inference engine, and then modify drawing in one pass to give one of the possible solutions to the constraints. Therefore, it is simple and easy to use for on-line applications.

3. CONSTRAINT INFERENCE ENGINE

In our system, once a stroke has been sketched out, it will be segmented into a series of head-to-end connected sub-strokes if necessary. Each sub-stroke will then be classified and fitted with one of 2D primitives: straight lines, circles, ellipses, circular arcs, elliptic arcs and B-spline curves [Qin01]. After the closest fitting has been found, the system constraint inference engine will infer certain geometric constraints. They can be classified into three categories: *unary*, *pairwise*, and *connection* constraints [Gao98]. The engine will first search for unary constraints and then establish pairwise and connection constraints by checking its relations to previous strokes (or sub-strokes) backwards sequentially. Once the current stroke becomes over-constrained, the inference process will be stopped and then the constraints solver will generate construction steps to solve the identified constraints.

Unary Constraints

The unary constraints are properties of a single primitive. They are directional constraints. The unary constraints apply to straight lines, ellipses, and elliptical arcs. For a straight line, the engine examines its directional angle to see whether it is roughly horizontal, or vertical, or parallel to isometric projection axes (Fig.1). If so, the straight line will be assigned corresponding unary constraint code: HOR, VER (or ISO-Y), ISO-X or ISO-Z. Similarly, for an ellipse, the system checks its major axis. For an elliptical arc, the system still checks its direction of the major axis, as for an ellipse. The rule for determining a unary constraint is that the directional angle (α) of a primitive is within a range of $(\beta-\delta)$ and $(\beta+\delta)$, where β is a constraint angle in degree (0 for HOR, 90 for VER, 30 for ISO-X and -30 for ISO-Z) and δ is an adaptive tolerance angle for the primitive. That is, $(\beta-\delta) < \alpha < (\beta+\delta)$. The parameter δ varies

with drawing speed and sizes of primitives such as lengths of lines or major axis of an ellipse. The bigger the sizes are the bigger δ . The higher the speed is the bigger δ .

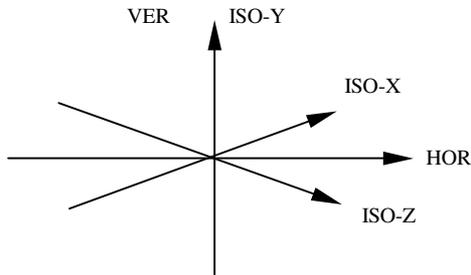


Figure 1. Directional Constraints

Pairwise Constraints

The pairwise constraints are geometric relations shared by two primitives [Gao98]. Currently, the system supports parallelism and perpendicularity between pairs of lines, ellipses, or elliptical arcs. Each line, or ellipse, can have one of the pairwise relations: either parallelism, or perpendicularity (it may have both, but the system only takes one of them). In essence, these two relations are directional constraints as well. The system searches these relations backwards for the current primitive (the latest input) by comparing directions of the current primitive and one previous primitive. If they are quite close, a parallelism relation will be found. If the difference between their directions is close to 90 degree, a perpendicularity relation will be determined. Once a relation is found, the system will stop a further search, otherwise, the search will continue until the first primitive is reached. This will reduce the number of constraints and avoid over-constrained cases. For example, a line A is parallel to lines B and C. If a parallelism relation between the lines A and B is found, then the relation between the lines A and C will not be further checked because the lines B and C should be parallel to each other. Similarly, if a line D is parallel to a line E and perpendicular to a line F, the system will only take the first found constraint because that the two constraints should be consistent, one is enough for constraining a direction.

Connection Constraints

The connection constraints are classified into three categories, namely *type-1*, 2, and 3 according to typical application scenarios.

Type-1 constraints

From the current primitive to a previous one, the inference engine searches for connectivity relations. For a *type-1* constraint, two primitives intend to join together at their end points. An example is shown in Fig. 2. The engine will first search for a pair of end points between two primitives and then check whether their end tolerance circles have intersections. If so, the two primitives will be connected at related end points. The radius of an end tolerance circle for a line is adaptive to its length and drawing speed. The longer the length is the bigger the radius. The higher the speed is, the greater the radius. Similarly, for an arc, the tolerance varies with its arc length and drawing speed. If an end point is constrained with a *type-1* relation, the relation code 1 is assigned to it (default is 0, meaning free end) and the corresponding constraint information will be recorded. Here, an adaptive tolerance is applied, since a simple fixed value may be too large or too small, resulting in either eliminating fine details in connections, or leaving adjacent ends unlinked. Indeed, different tolerances are needed for different parts of sketches, and certainly for different users. Using the adaptive tolerance can roughly satisfy this requirement.



Figure 2. Type-1 constraints

Type-2 Constraints

A *type-2* constraint is a touching relation, in which an end point of a primitive falls on the path of another (Fig. 3). This constraint is only applied to lines and arcs. That is, a primitive joins another with its one end touching on another in the middle. The constraint code for this relation is 2. To detect a *type-2* relation, the following procedure is conducted:

Step 1: Compute an adaptive tolerance value for the current primitive;

Step 2: Check if the corresponding tolerance circles at ends are intersected with a candidate primitive; if not, search for another primitives;

Step 3: If so, a *type-2* constraint is found and a constraint code 2 will be assigned to the corresponding end and related constraint information will be stored.

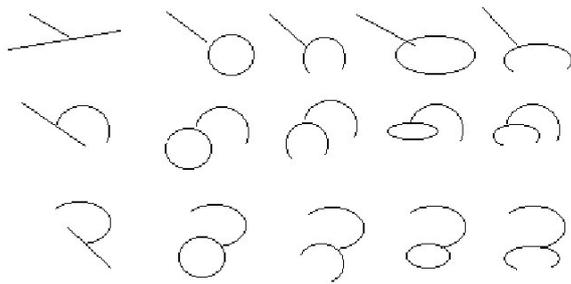


Figure3. Type-2 constraints

Type-3 constraints

The third type constraint (relation code 3) deals with a tangent connection, as shown in Fig.4, in which one end of a primitive is tangent to another primitive. Such a constraint is only concerned with lines and arcs as well. This connection can be regarded as a special case of a type-2 constraint. The current primitive not only joins the other with one end but also is tangent to it.

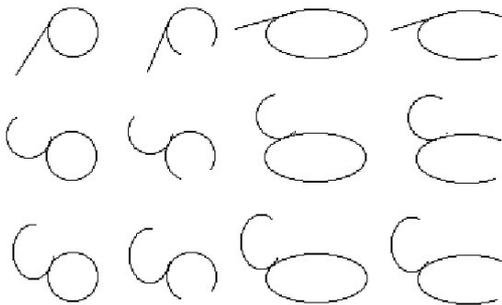
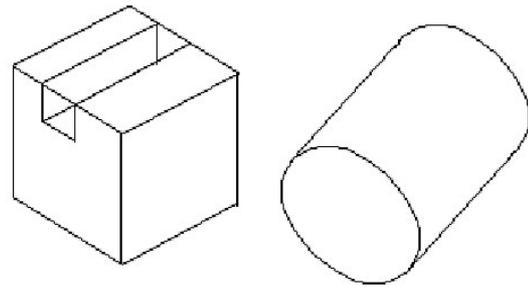


Figure 4 Type-3 constraints

To determine a *type-3* constraint, two steps are applied. First step is to check if the current primitive has a *type-2* constraint. If so, the next is to further determine whether the connection meets a tangent condition. Taking a pair of a line and a circular arc as an example, we can recognize that the connection is tangent, if the distance between the line and the centre of the arc is close to the radius of the arc. Once a *type-3* constraint is found, the former relation code 2 for a *type-2* constraint will be updated to a relation code 3. The corresponding constraint information will be recorded. Fig.5 gives some examples of different connections. When sketching a slot feature from a box, users will meet first and second type constraints (Fig.5 (a)). When silhouette lines are drawn to express a cylindrical object (Fig.5 (b)), the third type constraint will occur.



(a) type 2

(b) type 3

Figure5. Examples of different constraints

4. DEGREES OF FREEDOM ANALYSIS

Once a variety of constraints (relations) are obtained, the next is to modify individual primitives to satisfy all constraints, or to find a satisfactory solution. The system first analyses the degrees of freedom (DF) of a primitive and then determines construction rules for the primitive under certain constraints, using the degrees of freedom analysis.

Definitions

Informally, the number of degrees of freedom of a primitive object (object degrees of freedom, ODF) is the number of independent parameters required to allow the primitive to vary in location and shape. For example, in 2D space, a rigid body has two translational and one-rotational degrees of freedom to change its location. But, for ODF, extra degrees of freedom are allowed to vary its shape as well. For instance, a 2D arc may have extra 3 object degrees of freedom in terms of starting angle, subtended angle and radius to change its shape. Note that only the subtended angle and radius cannot define the starting point on the arc.

The number of constrained degrees of freedom (CDF) from a constraint is the number of degrees of freedom eliminated by the constraint. For instance, in 2D space, a position constraint limits two translational degrees of freedom of a primitive. Under given constraints, a geometric constraint solver may configure a primitive in limited ways. This is regarded as configuration degrees of freedom (CF), which is the difference between the number of ODF and the sum of its corresponding CDF. The relationship among ODF, CDF and CF can be addressed as

$$CF = ODF - \sum_{i=1}^n CDF_i,$$

where n is the number of constraints. We consider a primitive as well defined (or well-constrained), if and only if CF is equal to zero. It is under-constrained, when $CF > 0$, and over-constrained while $CF < 0$.

Object Degrees of Freedom

In our system, there are no explicit dimensional constraints. Thus, each primitive may vary in location or shape. This means that primitive geometry in the system is not a rigid-object. Different primitives have different object degrees of freedom, in accordance with different construction limitations. The degrees of freedom (DF) for each primitive are shown in Table 1. For example, a circle has no rotational DF because it is a perfect symmetry; it also has no dimensional DF, which means that its radius is fixed during construction processing. This assumption will make the construction task simple. Similarly, this dimensional restriction is applied on ellipses. However, for an ellipse, a rotational degree is given to allow a rotation of its major axis about its centre for meeting a directional constraint. Although an ellipse can be constructed by its four correspond circular arcs using *rc*-configurations, its rotational degree of freedom is unique comparing to a circle. For a circular or an elliptical arc, a dimensional DF is given to allow the system to change its extended angles, but not for changing its radius (or radii). Here the system simply treats a B-spline curve like a straight line. Note that lines, arcs and B-spline curves have the same structure of object degrees of freedom. If they are under the same constraints, their constructions rules will be similar.

Primitives	Translationa 1 DOF	Rotational DOF	Dimensiona 1 DOF	Total ODF
Line	2	1	1	4
Arc	2	1	1	4
Elliptical arc	2	1	1	4
Circle	2	0	0	2
Ellipse	2	1	0	3
B-spline	2	1	1	4

Table 1. Object degrees of freedom

Constrained Degrees of Freedom

Various constraints restrict different degrees of freedom. The constrained degree of freedom (CDF) for each type of constraints is given in Table 2. A pairwise or unary constraint will restrict a rotational degree of freedom. For a *type-1* relation, it is an incidence constraint, which restricts two translational degrees of freedom. For a *type-2* relation, it requires that one end point of a primitive to be extended onto a constrained primitive. So, this type constraint

eliminates a dimensional degree. A *type-3* constraint will remove a dimensional degree of freedom as a *type-2* one, and further restrict a rotational degree of freedom by requiring a tangency relation.

Constraints	CDF
Pairwise	1
Unary	1
Type 1	2
Type 2	1
Type 3	2

Table 2 Constrained degrees of freedom

5. CONSTRUCTION RULES

To configure sketched 2D primitives with identified constraints, the system calculates configuration degrees of freedom and then produces construction rules (or steps) according to several general construction strategies.

General Construction Strategies

When solving constraints, the following general construction strategies are applied to all types of primitives to generate construction steps:

- (1) If a primitive is free from any constraints, default constraints for fixing its position will be applied. In this case, its CF is zero.
- (2) If a primitive of lines, ellipses or elliptical arcs has a unary constraint and it is well-constrained or under-constrained, *Minimal movement policy* will be applied on it. That is, if the current element is required to change its direction, the system should try its best to keep movements minimal, since original position and size of the geometry represents users' initial intent. This policy attempts to capture users' intent more accurately. Fig.6 gives an example of this minimal movement strategy. In Fig.6 (a), a straight line (dashed line) needs to be modified to a vertical line. In accordance with the current policy, the solver rotates it about its mid-point, to a vertical line (solid line). The system does not take the second solution (Fig.6 (b)), which rotates the line about its one end to form a vertical line, because the resulting line will be far from the original one.



Figure 6. Minimal movement

(3) *One-side policy*: when dealing with the current primitive, the constraint solver ignores all constraints between the current primitive and those generated after it. This means that only constraints between the current primitive and previous ones (on one-side of it) will be solved. This strategy reduces the number of constraints to be treated, and focuses on a local configuration problem. This policy respects the fact that when sketching a current stroke, the user mainly takes previous drawing as references to form new constraints, although some intentions might be born at this moment. If the user stops drawing after the current stroke, the system should still give a possible solution.

(4) *Background propagation*: if a current primitive has some constraints with previous ones, the solver first tries to modify it to satisfy the constraints, and to keep previous ones unchanged, although the constraints could be met by changing the previous, either. Otherwise, once a new stroke inputs, some new constraints may be added in a constraint chain from the current primitive to the first one and all previous geometry will be changed wavelike. This will not only lead to heavy computation and instability, but will also harm the minimal movement policy. Actually, the backward propagation strategy can be introduced from the minimal movement policy.

(5) *Clustering policy*: if any two primitives meet at their end points by a *type-1* constraint, the common position will be figured out and fixed for ever. If none of them has a directional constraint, the common position will be a mid-point of related two ends. If any of them has a directional constraint, their geometric intersection point will be the common position. Once a common position is found, it will become fixed.

Generation of Construction Steps

Before considering how to configure a new primitive, analysis of its constraints and degrees of freedom is performed. Then the decision on how to construct the geometry in sequential steps is made accordingly. In general, if geometry is under-constrained, a set of default constraints will be applied to make it well-constrained. Afterwards, any default constraints can be further modified such as free ends. The most common default constraint is a joint restricting two translational degrees of freedom. When the geometry becomes well-constrained, it can be constructed against typical application scenarios. If it is over-

constrained, typically, extra constraints such as directional ones will be removed to make it well connected. In our system, connection constraints have a higher priority than directional ones because they contain more important topological information for 3D interpretation.

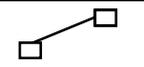
Scenario	No.	Constraints U-unary, P-pairwise	OD F	CD F	CF	Add Default constraints □ J- Joint D- Dimension
	1	0	4	0	4	2 J
	2	1 U or 1 P	4	1	3	1 J 1 D (explicit)
	3	1 type-2	4	1	3	1 J 1 D (implicit)
	4	1 type-2 and 1 U or 1 P	4	2	2	1 J 1 D (implicit)
	5	1 type-1	4	2	2	1 J
	6	1 type-3	4	2	2	1 J
	7	1 type-1 and 1 U or 1 P	4	3	1	1 D (explicit)
	8	1 type-3 and 1 U or 1 P	4	3	1	1 D (explicit)

Table 3. General construction analysis-under constrained cases

Tables 3 and 4 illustrate a case study for a line configuration. In general, there are 13 combinations of different constraints. Most of them (nine cases) are under-constrained (see Table 3). Only three of them are well constrained in Table 4 (No. 9-11) and the last two cases are over-constrained. This means that for most of the cases, a possible solution can be found easily under the current solving strategies. The main concern is how to add default constraint(s) and solve constraint equations. After choosing the default constraints the construction steps will become well defined. If the constraints include a directional one (unary or parallelism or perpendicularity), in general, two construction steps are needed. They can be performed separately by firstly modifying the current primitive to meet the directional constraint and then simply focusing on the predefined a one-step construction. For example, in the case No.4 (Table 3), a line is constrained with a unary relation (VER)

and a type-2 relation. We can solve the problem in two steps. First rotating the line around its mid-point to meet the unary relation. This operation is the same as in the case No.2 with a default explicit dimensional constraint. After that, the problem will be similar to the case No.3. The second step is to extend the line by finding its intersection point to meet the type-2 constraint, which can be regarded as an implicit dimensional constraint. In these three cases, program routines for changing direction and obtaining intersections are separate. They are reusable and combinable. This can not only save developing time, but also reduce the number of constructive steps. Case No. 1 simply takes two default ends and Case No. 5 takes one default end and a constrained end with clustering. Case No.6 can be solved by finding a tangent line from the default end. For cases No. 7 and 8, after a rotation, the next is to move the line to an incident point or tangent point. For three well-constrained cases (Table 4), they need only one step to solve their constraint equations, which depends on the types of involved primitives. The last two cases (Table 4) are over constrained, they can be first modified into well-constrained cases and then solved in a similar way to Case No. 9.

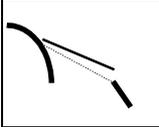
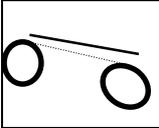
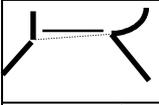
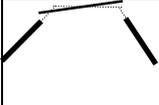
Scenario	No	Constraints U- P- pairwise	O D F	C D F	C F	Remove Default constraints
	9	2 type-1	4	4	0	0
	10	1 type-1 and 1 type-3	4	4	0	0
	11	2 type-3	4	4	4	0
	12	2 type-1 and 1 U or 1 P	4	5	- 1	1 U or 1 P
	13	2 type-1 and 1 U or 1 P.	4	5	- 1	Modify free ends or remove 1U or 1P

Table 4 General construction analysis -well and over constrained cases

B-spline curves have been restricted to have only type-1 constraints. They can be just regarded as special cases of lines, as in cases No. 1, No. 5 and No. 9. The solver simply assigns incident points to

their end points. A circle can only move in 2D with a constant radius. So, its construction is always to find a displacement of its centre point. An ellipse direction can be changed under a unary or a pairwise constraint, and also its centre points can be shifted in a similar way to a circle.

Circular and elliptical arcs are open curve sections with two ends, and have 4 object degrees of freedom as lines. They also have the same types of constraints to be applied as lines. Topologically speaking, they are within the same class of line objects for tidying up. Therefore, construction rules for arcs are similar to those for lines. Each line case has a corresponding case for arcs. Taking the case No. 3 as an example, the construction method for a line is to find its intersection point between two lines. For an arc (circular or elliptical), the construction method is still to find intersection point, but between an arc and a line. The difference is the use of different equations to obtain an intersection point. But, the construction method is the same.

5. EXAMPLES AND DISCUSSION

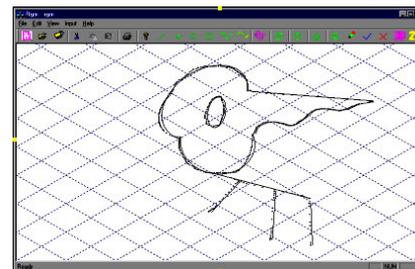


Figure 7. Input of sketches

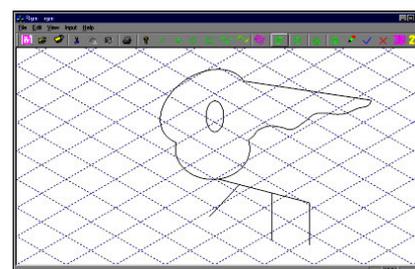


Figure 8. Tidying up

With our system, 2D online sketches can be rapidly transferred into 2D primitives and further can be beautified with right connections. The tidying up processing is based on our construction rules and degrees of freedom analysis. This method lets users to work on their design ideas with a real-time system in a more natural way. Fig. 7 illustrates original input of sketches, which consists of several lines, arcs and a B-spline curve. Constraints involved in this case include type-1 and type-2 connections, e.g., a line touching an arc, and unary relations, e.g. vertical

lines. These constraints are detected successfully by the inference engine, and then are solved properly by the constraint solver. Fig. 8 gives the result of this 2D configuration. It can be seen that the ellipse in the middle and the two vertical lines are changed under unary constraints. Three lines are modified to touch on other primitives under *type-2* constraints. All *type-1* constraints are solved correctly.

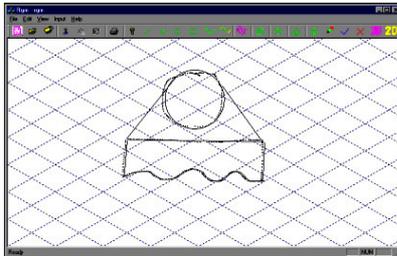


Figure 9. Sketched input

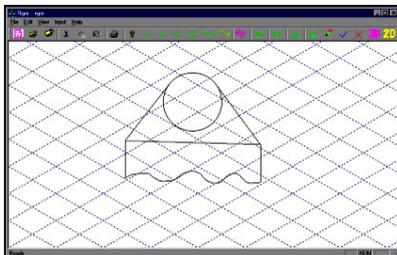


Figure 10. Result of beautification

Figure 9 shows an example of sketches with *type-3* constraints. Two lines tangent to a circle are inputted. The result of the tidying up is given in Figure 10. The last stroke for the horizontal line is an over-constrained case. Its unary constraint (HOR) is removed off because its two ends become fixed points already. It can be seen that the system can capture *type-3* constraints (tangency), and the solver works properly.

7. CONCLUSION

The constraint solver based on the construction rules and degrees of freedom analysis can quickly and properly give one of the possible solutions. It determines primitives one by one, and does not involve solving a system of simultaneous non-linear equations. The inference engine and the constraint solver can deal with elliptical primitives and free-form curves.

The System works directly on sketches. No dimensional schema is required and users are not asked to add dimensions to the sketches, as in most commercial parametric CAD systems. The constraint solver treats 2D primitives as semi-rigid-objects with a dimensional degree of freedom. For example, a line in 2D space has 4 object degrees of freedom instead

of 3 for a rigid-body. In this way, the solver treats the dimension information either as default constraint (changeable constraint), depending on the object's configuration degrees of freedom. In contrast, most geometric constraint solvers [Gao98,Bou95], regarded dimensional constraints as rigid constraints.

The system is performed fast enough for a real-time sketch-based application. This is important for conceptual design, especially for distributed design systems [Qin03]. The solver will require more pre-coded construction rules, if the number of constraint types is increased. This drawback will be balanced with its speedy performance.

8. REFERENCES

- [Zel96] Zeleznik, RC, Herndon, KP, and Hugnes, JF, SKETCH: an interface for sketching 2D scenes, *SIGGRAPH*, pp.163-170, 1996.
- [Qin00] Qin, SF, Wright, DK and Jordanov, IN, From on-line sketching to 2D and 3D geometry: a system based on fuzzy knowledge, *Computer-Aided Design* 32, pp.851-866, 2000.
- [Qin01] Qin, SF, Wright, DK and Jordanov, IN, On-line segmentation of freehand sketches by knowledge-based nonlinear thresholding operations, *Pattern Recognition* 34, pp.1885-1893, 2001.
- [Shp97] Shpitalni M and Lipson H, Classification of sketch strokes and corner detection using conic sections and adaptive clustering, *Journal of Mechanical Design* 119, pp.131-135, 1997.
- [Pav85] Pavlidis T, and Van Wyk CJ, An automatic beautifier for drawings and illustration, *ACM Computer Graphics* 19 (3), pp. 225-234, 1985.
- [Jen92] Jenkins DL Martin RR, Applying Constraints to enforce users' intentions in free-hand 2D sketches. *Intelligent Systems Engineering*, Vol. 1, 31-49, 1992.
- [Gao98] Gao XS, Chou SC, Solving geometric constraint systems II. A symbolic approach and decision of re-constructibility, *Computer-Aided Design* 30(2), pp.115-22, 1998.
- [Li02] Li YT, Hu SM, and Sun JG, A constructive approach to solving 3-D geometric constraint systems using dependence analysis, *Computer-Aided Design* 34, pp. 97-108, 2002.
- [Bou95] Bouma W Fudos I, Hoffmann C, Cai J, Paige R, Geometric constraint solver. *Computer – Aided design* 27(6), pp. 487-501,1995.
- [Qin03] Qin SF, Harrison R, West AA , Jordanov and Wright DK, A framework of web-based conceptual design, *Computers In Industry* 50, pp.153-164, 2000.