Interactive Geometric Constraint Satisfaction

Remco C. Veltkamp* Farhad Arbab‡

* Utrecht University, Department of Computing Science
Padualaan 14, 3584 CH Utrecht, The Netherlands
Remco.Veltkamp@cs.ruu.nl

‡ CWI, Department of Interactive Systems
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Farhad.Arbab@cwi.nl

Abstract

This paper presents a new incremental approach to geometric constraint satisfaction that is tailored to interactive applications. Our approach categorizes solutions into geometric primitives representing a range of solutions with uniform geometric characteristics. This scheme keeps intermediate solutions in the geometric domain, providing geometrically meaningful feedback to the user and the ability to interlace the interpretation of previous and new geometric constraints on the same high level of abstraction. This approach preserves the declarative semantics of constraints and leads to a number of advantages, including graceful handling of underconstrained specifications, the natural processing of expressions of both conjunctive and disjunctive constraints, the ability to perform satisfaction locally and incrementally, and support for constraint inference and geometric reasoning.

1991 Computing Reviews Classification:
I.2 [Artificial Intelligence] Problem solving.
I.3 [Computer Graphics] Graphics utilities, Computational geometry and object modeling.
J.6 [Computer Aided Engineering] Computer aided design.

Key Words and Phrases: geometric constraints, incremental constraint satisfaction, computer aided design.

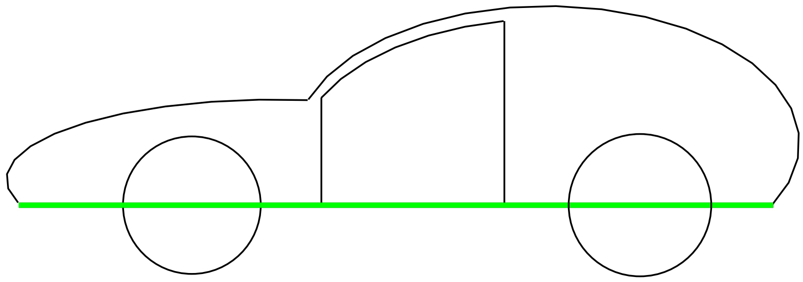

Figure 1: Solution of on(front-wheel,chassis) *and* on(rear-wheel,chassis).

1. Introduction

Constraints specify dependency relations between objects which must be satisfied and maintained by some constraint management system. Constraint systems are used in a wide range of applications, including user interface design [1], animation [2], geometric modeling [3], and artificial intelligence [4]. *Geometric* constraints can fix one or more degrees of freedom for positioning, orientation and dimensioning of the constraint variables. For example, when a circle of fixed radius is constrained to be tangent to a fixed line segment, the position of the circle center is restricted to two line segments parallel to the given one, at a distance equal to the radius.

A constraint satisfaction system relieves some of the burden of its users: it is easier to state constraints than to satisfy them. Problems can be solved by specifying constraints, and the user need not specify how to solve the constraints. But even if a system cannot satisfy all constraints that can occur in a given domain, it can free its users from the error-prone process of solving the many little, but time-consuming simpler problems. Applications of geometric constraints are typically found in graphical editors, page layout programs, and CAD tools.

Consider the following example of geometric constraints, specified in order to position the wheels of a car:

1. anchor(chassis)
2. on(front-wheel, chassis)
3. on(rear-wheel, chassis)
4. distance(front-wheel, rear-wheel, 32)
5. distance(front-wheel, door-vertex, 5)

where the chassis is a line segment, and the wheels are circles. The first constraint anchors the chassis. The second and third constraints allow the centers of the wheels to be positioned anywhere on the green line in figure 1. The fourth constraint allows the front wheel to be positioned anywhere on a circle concentric with the rear wheel, and vice versa, as illustrated in figure 2 for a particular admissible position of the wheels. The fifth constraint allows the front wheel to be placed anywhere on the green circle shown in figure 3. Figure 4 shows the position of both wheels satisfying all constraints, where the position of the front wheel is the intersection of the green line in figure 1 and the green

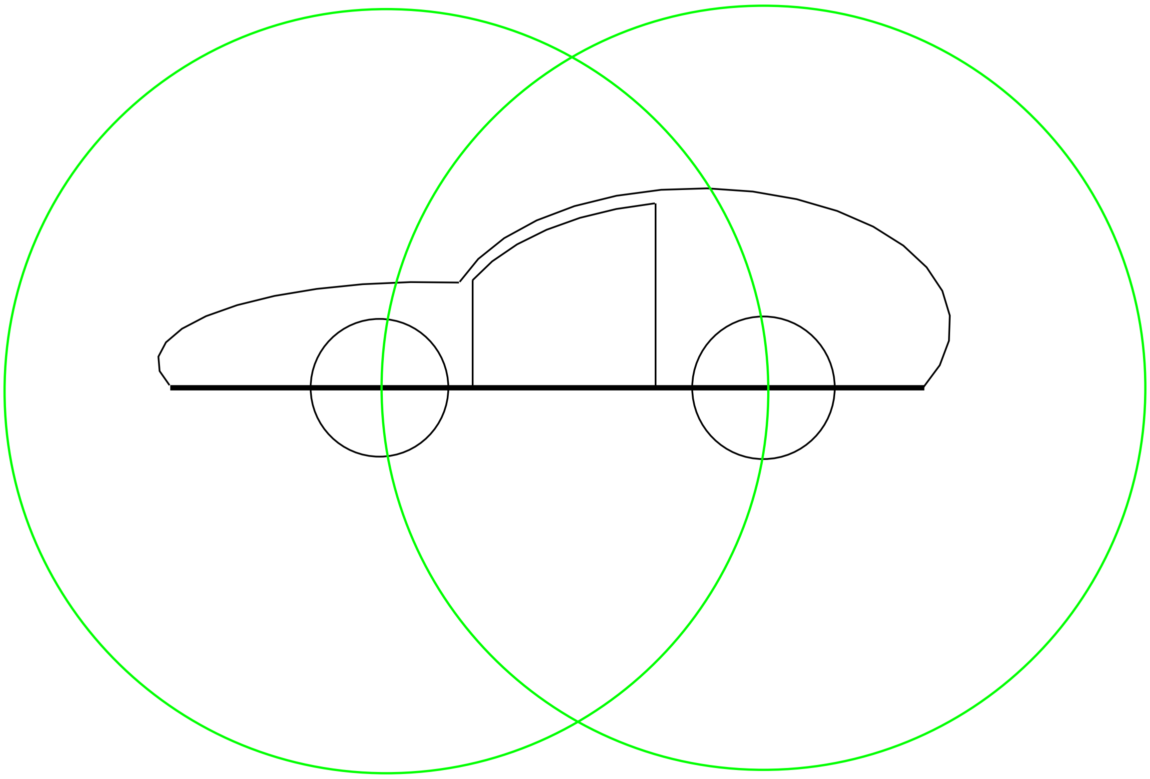

Figure 2: Solution of distance(front-wheel, rear-wheel, 32).

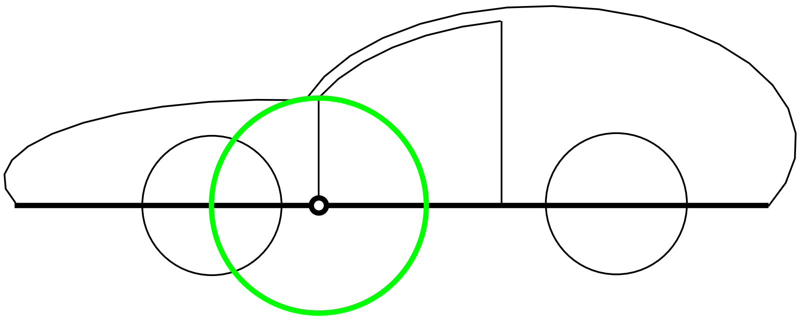

Figure 3: Solution of distance(front-wheel, door-vertex, 5).

circle in figure 3, and the position of the rear wheel is the intersection of the green line in figure 1 and the right circle in figure 2, after this circle is repositioned to be concentric with the front wheel (by delayed propagation, as will be explained later). However, before sending this specification in terms of constraints to the manufacturing floor, it might be useful to realize that the configuration in figure 5 also satisfies the constraints, with the understanding that the rear wheel is now in front of the front wheel.

The point about this example is that interactive applications typically give rise to underconstrained problems, and that it is non-trivial to see the effect of a set of constraints

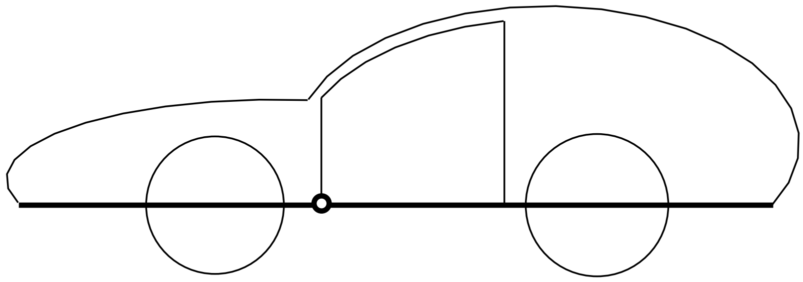

Figure 4: Final solution.

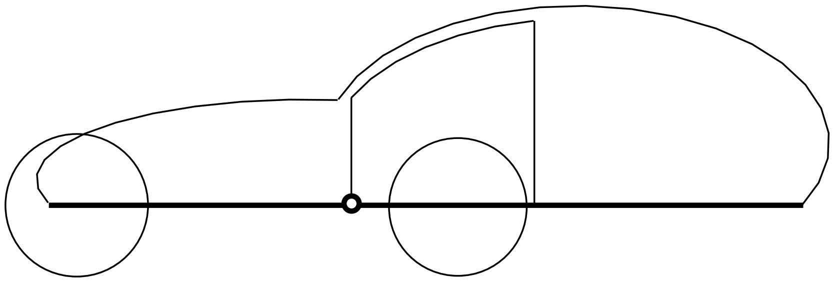

Figure 5: Alternative final solution.

beforehand. Therefore, systems that converge on one solution are not satisfactory. In this paper we introduce a new approach to geometric constraint satisfaction in interactive applications. Some features of this approach, illustrated above and explained in the rest of this paper, are:

- in underconstrained situations all solutions are provided,
- the solutions are in a geometrical form,
- temporary inability to propagate these solutions is handled by delayed evaluation.

Essentially, we represent the solution set of a variable with respect to a constraint as the union of geometric primitives, and combine this partial solution with the variable's previous solution. In section 2 we present some background on constraint satisfaction and the motivation for our new approach. The approach itself is introduced in section 3. In section 4 we elaborate the algorithmic aspects of our method; this is followed by some discussion in section 5. In section 6 we discuss a prototype implementation of our method. Our approach is compared with other methods and systems in section 7. Section 8 contains some concluding remarks.

2. Constraint satisfaction

Formally, a constraint satisfaction problem (CSP) can be specified by a finite set of variables $v_1, \ldots, v_n$, a domain D_i of possible values for each v_i, and a set of constraints $C_1, \ldots, C_m$. Solving a CSP is finding a valuation $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$ of the variables for which all constraints are satisfied. Possible variants of the CSP are: find a single solution (valuation), find all solutions, find a best solution, find the number of solutions, determine satisfiability (whether or not a solution exists).

The satisfiability variant is the simplest one (indeed, it follows from all the others), and even this one is in NP: it is in the class of Nondeterministic Polynomial-bounded problems, i.e. it can be solved by a nondeterministic algorithm in polynomial time, but there is no deterministic algorithm known to solve the problem in polynomial time. It is even NP-complete, i.e. if there were a polynomial-bounded algorithm to solve the problem, then there would be a polynomial-bounded algorithm for each problem in NP. This can be deduced from the fact that the CNF-satisfiability problem can be converted to it in polynomial time, and the CNF-satisfiability problem is NP-complete [5]. The CNF-satisfiability problem is to determine if there is a truth assignment (a way to assign the values true and false) for the variables in a logical expression in conjunctive normal form (CNF) such that the value of the expression is true. A logical expression in conjunctive normal form is a sequence of clauses separated by conjunctions ($\wedge$), where a clause is a sequence of variables and negations of variables separated by disjunctions ($\vee$).

Because the general CSP is a hard problem, satisfaction algorithms are often slow. In order to obtain faster algorithms, one can confine the domain of the constraints and the variables and exploit some of the specific knowledge of the domain. The time complexity of such specialized constraint satisfaction depends on both the domain and the kind of constraints. Symbolic solution of the algebraic expressions describing geometric constraints is known to be NP-hard; tree structured constraint networks can be solved in linear time; linear constraints over real numbers can be solved in polynomial time; a single polynomial constraint equation of degree higher than four does not even have an analytical solution; and the complexity of solving integer polynomials of degree greater than two is still unknown.

In this paper we deal with constraints and variables in Euclidean geometry, and we derive all solutions to the constraints. Although the general problem is NP-complete, we will see that the time complexity of our algorithm is linear in the number of constraints because the satisfaction is performed locally and possible loops are broken.

2.1. Satisfaction techniques

Constraint satisfaction techniques can be classified as structured or unstructured. Structured methods group dependent constraints into independent sets that can be solved separately; unstructured methods do not. Solving the *overall set of equations* comprising all constraints is perhaps the simplest approach to constraint satisfaction. One way to solve the overall set of equations is by relaxation. Numerical relaxation makes an initial guess at the values of the variables in an equation, and estimates the error by some heuristic. In view of this error, the guesses are adjusted accordingly and a new error is estimated. This repeats until the error is minimized. A disadvantage of this method is that it converges to only one of the roots of an equation. Moreover, *which* root is found depends on the initial value of the variable. This makes the solution unpredictable in underconstrained situations. Numerical relaxation is also computationally expensive, and can be used only in continuous numeric domains. Other unstructured methods include algebraic manipulation [6] and augmented term rewriting [7].

Structured methods impose a structure on the set of constraints by grouping them into sets of dependent constraints. A set of dependent constraints can be represented as a *network* of constraints. These groups are satisfied independently. Numerical computation of a group of constraints is often done by relaxation. Deductive systems infer information about the admissible values of variables and use some method to assimilate the results of the inference process throughout the network. We mention two such methods: propagation of known states and propagation of degrees of freedom.

- *Propagation of known states*, or just local propagation, can be performed when there are parts in the network whose states are completely known (have no degrees of freedom). The satisfaction system looks for one-step deductions that will allow the states of other parts to be known. This is repeated until all constraints are satisfied or no more known states can be propagated. If not all constraints can be satisfied, the remaining constraints must be resolved by, for example, numerical relaxation. Many constraint satisfaction systems use some form of local propagation.

- *Propagating degrees of freedom* amounts to discarding all parts of the network that can be satisfied easily and solving the rest by some other method. This method identifies a part in the constraint network with enough degrees of freedom so that it can be changed to satisfy all of its constraints. That part and all the constraints that apply to it are then removed from the network. Deletion of these constraints may give another part sufficient degrees of freedom to satisfy all of its constraints. This continues until no more degrees of freedom can be propagated. The part of the network that is left is then satisfied by some other method, if necessary, and the result is propagated towards the discarded parts, which are successively satisfied (propagation of known states).

The above techniques are two methods of propagation, independent of *what* is actually propagated. We distinguish the following types of information to be inferred and propagated: a whole solution set, a single solution, algebraic expressions, and constraints.

- *Solution set* inference makes deductions on the set of possible solutions, which are restricted by the constraints. Traditionally this is a finite domain [8], but also continuous intervals of numerical values have been studied [9]. In *single solution* inference, constraint variables get assigned a single value, often numeric.

- The *operational approach* [10], [11] performs single *geometric* solution inference. It satisfies constraints sequentially by performing operations (translation, rotation, etc.) on the geometric objects involved. An already satisfied constraint either tolerates an operation on one of its operands, or must propose a transformation to satisfy the constraint again. In this way operations can be propagated through a constraint network until all operations are tolerated.

- In *constraint* inference, implied constraints are derived and explicitly added to the network. Implied constraints can be recognized by a unification mechanism or by the use of multiple redundant views [12]. Finding implied constraints can be used to avoid extensive manipulations in cases where local propagation does not suffice. It can also be used to restate the constraints in a different way with the same meaning, which can help the system to solve constraints locally, instead of resorting to more costly techniques such as relaxation.

2.2. Motivation

Incremental satisfaction of geometric constraints arises naturally in many interactive applications in graphics and geometric modeling. A good deal of work has already been done on constraint satisfaction in general [7] [4], and constraint satisfaction in the geometric domain in particular [13] [14] [15] [10] [11] [16] [17] [3].

Many systems detect overconstrained situations, but cannot satisfactory handle underconstrained cases. For example, many numerical methods behave unpredictably when changes are made to an underconstraint set of constraint equations. On the other hand, a Logic Programming system like Prolog for solving numeric constraints can handle underdetermined equations and return a set of deduced equations that constitutes the solution. However, in a geometric context we prefer a geometrical form for the solution. A possible geometric approach is the planning of transformations of the variables so as to satisfy constraints [10] [11] [17].

This may give rise to two problems:

- The proposed solution is as intended, but the user is not aware of the ambiguity of the specification. The existence of alternative solutions may cause problems for post-processing.
- The solution is not as intended. Now the user must interfere, but he may not know the alternative solutions.

We distinguish several types of alternative solutions:

- A constraint can have several discrete solutions. For example a circle through two points, with a fixed diameter greater than the distance between the points, can have two positions.
- A constraint can have a continuous range of infinitely many solutions. For example, the admissible locus of a point having a fixed distance to a fixed point forms a circle.

Simultaneous evaluation of mutually constrained objects (multi-dependency) is generally a difficult problem. Consider the distance constraint between the two wheels in the

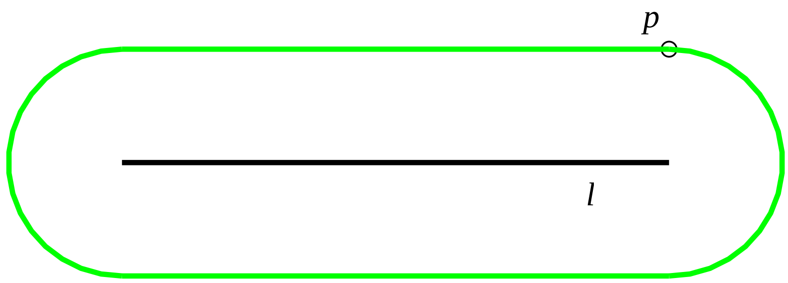

Figure 6: Locus units resulting from distance(p, l, 15)*: two line segments and two half circles.*

example in the introduction, where neither of the two has a fixed position. Many systems cannot cope with such a situation, and require that only one of the constraint operands is variable (single-dependency). It is often possible, though, to determine the structure of a solution. In our example, the circle center of either of the two wheels must lie on a circle concentric with the other wheel, wherever its position. Thus we can delay propagation until the position of one of the wheels is actually known (as happens in the example).

3. A new geometric approach

Our approach to geometric constraint satisfaction is based on solution set inference, and performs propagation of known states. Our geometric satisfaction method provides ranges of solutions and discrete alternative solutions in underconstrained situations by means of geometrical locus units. In combination with delayed satisfaction, this gives a more powerful problem solving capability than usually provided by geometric constraint systems.

The units of the solution domain are geometric primitives that represent the locus units. They describe the parts of the solution set with uniform geometrical characteristics. This notion of locus unit captures the cases of a single solution, an interval, a region, etc.

Essentially, we represent the solution set of a variable with respect to a constraint as the union of geometric primitives, and combine this partial solution with the variable's previous solution. For example, the complete locus of a point p having a fixed distance to a line segment l consists of the union of four locus units (geometric primitives): two line segments and two half circles (see figure 6). These units represent sets of alternative solutions. When p is further constrained to lie on another line segment, say m, this gives another locus unit, coincident with m. The final solution is the intersection of m with all the alternative solutions so far. In calculating these intersections we take advantage of the partitioning of solutions into geometric primitives, which can be simply intersected.

Conceptually, the constraints and the variables form a network, where the variables are nodes and each constraint is a hyper-edge between its operands. The network can be disconnected, in which case the variables in one connected component and the variables in other connected components are not related by constraints. The solution of each variable is a locus unit expression, i.e. the union or intersection of locus units or other locus unit expressions.

Our model of constraint satisfaction consists of a set of constraints C, a set of constraint variables V, a set of locus units Q, a locus unit generating function G, and a function Tolerate() that satisfies a constraint and performs propagation. The constraint variables and the locus units are geometric primitives.

A constraint is a relation between a number of variables from V. A constraint can be k-ary, i.e. it can involve any number of variables, depending on its type (see the constraints used in the introduction). Constraints may be multi-directional, i.e. each of the involved variables constrains the others, again depending on the constraint type. Multi-directional constraints provide a natural way to impose certain relations [1]. For example the constraint distance(point, circle, 0) is multi-directional and means the same as distance(circle, point, 0), but the constraint on(point, circle) is one-directional and means that the point must lie on the circle, whereas on(circle, point) in our implementation means that the circle *center* must lie on the point. In contrast to many other systems, more than one variable may be undetermined (like in the constraint distance(front-wheel, rear-wheel, 32) in the example). Constraints are specified incrementally, so that every variable in V is involved in an ordered set of constraints. Each constraint can be either *conjunctive* or *disjunctive*. A conjunctive constraint further restricts the degrees of freedom of the involved variables. A disjunctive constraint provides alternatives to the solution of *previously* specified constraints on that variable. Considering a single variable and only constraints on that variable, the constraints thus form an expression of the form

$$(\dots(c_1 \ \mathsf{AND/OR} \ c_2) \ \mathsf{AND/OR} \ \dots) \ \mathsf{AND/OR} \ c_m.$$

This means that a constraint expression like $(c_1 \mathsf{OR}\ c_2)\mathsf{AND}\ (c_3 \mathsf{OR}\ c_4)$, for example, is not possible in this scheme.

The locus unit generating function $G : (c, v_i) \mapsto (q_1, \dots, q_n)$ generates the set of locus units for v_i that are consistent with the constraint $c(v_1, \dots, v_k)$ and all the variables v_j, $j = 1, \dots, k$, $i \neq j$, with their current locus units, regardless of the current units of v_i. Note that not all combinations of types of constraints and variables need be geometrically meaningful; G may be defined only for a limited number of combinations. The generation of the locus units can involve geometric reasoning [18], taking into account the nature of the constraint and the current locus units of the variables. For instance, consider again the example in the introduction. At first the constraint distance(front-wheel, rear-wheel, 32) has the effect that the locus unit generating function assigns to the rear wheel a circle that has no fixed position because the front wheel is not fixed. But after the constraint distance(front-wheel, door-vertex, 5), the front wheel has two locus units (two different points on the chassis, say p_1 and p_2). In the propagation of this new solution, the locus unit generating function assigns to the rear wheel two circles centered at p_1 and p_2 (which are subsequently intersected with the chassis).

When a new constraint is added to the network, the locus unit generating function is applied to each of its operands. For a single variable, the union of the resulting units, $\cup(q_1, \dots, q_n)$, is the solution to that one constraint. If the constraint is conjunctive, $q_1, \dots, q_n$ must be intersected with the variable's current solution, say locus unit expression qe: $\cap(qe, \cup(q_1, \dots, q_n)$. If it is a disjunctive constraint, we take the union of the new and the current solution: $\cup(qe, \cup(q_1, \dots, q_n)$. In this way we get an expression of locus units, which is the solution set of the constraints so far for that single variable. In general, this expression is of the form $B(qe_1, \ldots, qe_m)$, where B is one of the Boolean set operators union or intersection, and each qe_i is a locus unit expression.

After the addition of a new constraint to the network, let $q_1, \ldots, q_n$ be the new locus units generated for one of the variables involved in that constraint. If $B(qe_1, \ldots, qe_m)$ is the locus unit expression representing the solution set of this variable prior to the new constraint, the new solution set of the variable becomes $B'(B(qe_1, \ldots, qe_m), \cup(q_1, \ldots, q_n))$. If the new constraint is conjunctive, then B' is $\cap$, and if it is disjunctive, B' is $\cup$. The resulting expression is transformed into a union of locus unit expressions, and as much of its intersections as possible are evaluated. The aim of this transformation is to represent the solution as the union of simple units. The transformation rules are as follows:

1. $\bigcup(\cup(q_1, \ldots, q_n)) = \cup(q_1, \ldots, q_n)$
2. $\bigcap(\cup(q_1, \ldots, q_n)) = \cup(q_1, \ldots, q_n)$
3. $\bigcup(\cup(qe_1, \ldots, qe_m), \cup(q_1, \ldots, q_n)) = \cup(qe_1, \ldots, qe_m, q_1, \ldots, q_n)$
4. $\bigcup(\cap(qe_1, \ldots, qe_m), \cup(q_1, \ldots, q_n)) = \bigcup(\cap(qe_1, \ldots, qe_m), q_1, \ldots, q_n)$
5. $\bigcap(\cup(qe_1, \ldots, qe_m), \cup(q_1, \ldots, q_n)) = \bigcup_{i,j}(q_i \cap qe_j)$
6. $\bigcap(\cap(qe_1, \ldots, qe_m), \cup(q_1, \ldots, q_n)) = \bigcup_{i=1}^{n} \cap(qe_1, \ldots, qe_m, q_i)$

Note that there is no rule for the intersection of two intersections, because the new units $q_1, \ldots, q_n$ always form a union. Each locus unit expression qe_j is either a single unit or an intersection of other locus unit expressions. Any intersection $q_i \cap qe_j$ on the right-hand sides of equations 5 and 6 that can be computed, is evaluated into a union of units: $q_i \cap qe_j = \cup(q'_1, \ldots, q'_k)$.

It may be difficult to compute some intersections, for example if one of the locus units involved is a circle that can float around, as in figure 2. In such cases we leave the intersection unevaluated, as in figure 7. This is why a qe_j can be an intersection of units or of other locus unit expressions: $qe_j = \cap(qe'_1, \ldots, qe'_k)$. The right-hand side of equation 6 is split into intersections that are computed and intersections that remain unevaluated. In this way we implement delayed satisfaction. A subsequent intersection may resolve the problem. Consider a locus unit that is a circle whose center is allowed to move over a line segment, say circle c. If some future constraints on the same variable give two line segments l_1 and l_2 as locus units, and the intersection of l_1 with the swept circle c is left unevaluated, the subsequent intersection $(c \cap l_1) \cap l_2$ is transformed into $(c \cap l_2) \cap (l_1 \cap l_2)$, by transformation rule 6. If $l_1 \cap l_2 = \phi$, then so is the final solution. If $l_1 \cap l_2$ is a point, it is easier to test if it is included in $(c \cap l_2)$ than to compute $(c \cap l_2)$ itself. In this way, delayed satisfaction of constraints is handled uniformly, thereby enhancing the satisfaction capabilities of the system.

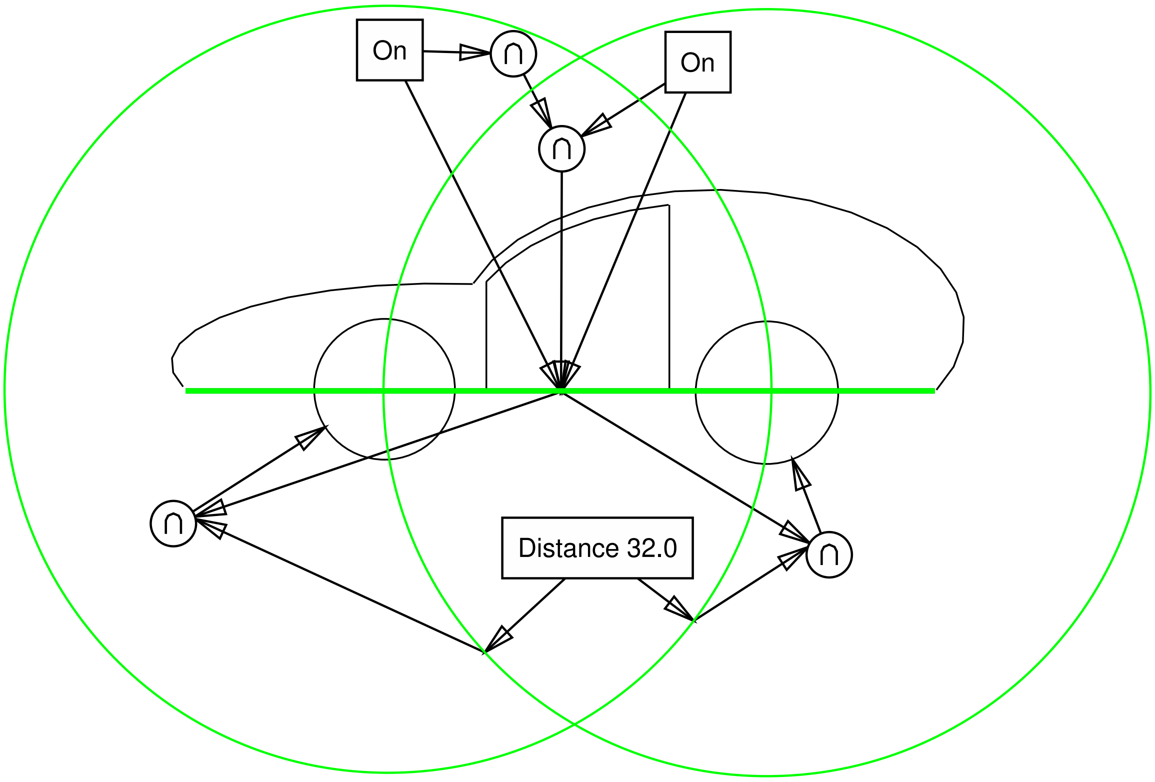

Figure 7: Network with unevaluated intersections.

4. Algorithmic aspects

The algorithm we use is essentially a standard local propagation algorithm, but is given here to show the several aspects that were introduced in the previous section. We also describe how these aspects affect issues such as termination and time complexity.

Let S_i be the set of locus units generated for an operand v_i with respect to a constraint c: $S_i = G(c, v_i)$. The procedure $\mathsf{Evaluate}(v_i, S_i)$ computes intersections and transforms the locus unit expression as described in the previous section. The returned value of $\mathsf{Evaluate}()$ is EMPTY if the resulting solution set of v_i is empty, CHANGED if it has been changed, and UNCHANGED if it is unchanged. We can invoke $G(c, v_i)$ and apply $\mathsf{Evaluate}(v_i, S_i)$ for each variable v_i of c. If one of the variables is assigned an empty solution set, the constraint expression is inconsistent — the specification is overconstrained.

4.1. Propagation

If the solution set of a variable v_i has been changed and v_i is involved in another constraint c', the solutions of other variables of c' must also be updated. These variables must therefore be recorded. The following procedure $\mathsf{Revise}(c)$ updates all variables of c, and returns the set of variables whose solutions are changed:

```
Revise (c)
{
  Changed = φ
  for each v_i of c
    S_i = G(c, v_i)
  for each v_i of c
    case Evaluate (v_i, S_i)
    {
      EMPTY: Report (OVER-CONSTRAINED)
      CHANGED: add v_i to Changed                    // propagate
      UNCHANGED:                                     // do nothing
    }
  return Changed
}
```

When a new constraint is specified, it must be added to the network and the resulting changes must be propagated to related constraints. This may cause new changes which in turn must be propagated, and so on. This is performed by the following algorithm:

```
Tolerate(c)
{
  Fifo = [c]                                   // first-in-first-out queue
  while Fifo ≠ φ
  {
   c' = FirstElementOf (Fifo)
   remove c' from Fifo
   Changed = Revise (c)
   for each v ∈ Changed
     for each c'' ≠ c' having v as parameter
       add c'' to Fifo
  }
}
```

Revise(c) makes the solution set of every variable of c consistent with c and its other variables using their current locus units. That is, the solution is made locally consistent. When Tolerate(c) terminates (see next section), and an overconstrained case has not been detected, the solution sets of all variables are locally consistent (this is equivalent to arc consistency in [8]). When only conjunctive constraints are added to a network, Tolerate(c) amounts to the filtering algorithm of [19] for the labeling of edges in an image, the paradigm example of an arc consistency algorithm. Because constraints that have variables with changed locus units are placed in a first-in-first-out queue (as opposed to an unordered list), they are guaranteed to be revised again, whether the propagation is finite or infinite. This is called a fair propagation in [20].

This algorithm performs incremental satisfaction of a new constraint c. In order to satisfy a set of new constraints $\{c_0, \ldots, c_n\}$, the FIFO queue can be initialized with Fifo $= [c_0, \ldots, c_n]$.

4.2. Termination

This section discusses termination of the propagation in Tolerate(c). We treat the cases where c is a conjunctive or a disjunctive constraint separately.

4.2.1. Conjunctive constraint

When a conjunctive constraint $c(v_1, \ldots, v_k)$ is added to the network, the solution sets of variables do not get larger. More formally, Evaluate($v_i, G(c, v_i)$) is then monotonic in the following sense. Let $\{[S_1, \ldots, S_k]\mathsf{Evaluate}(v_i, G(c, v_i))\}$ denote the solution set of v_i after applying Evaluate($v_i, G(c, v_i)$), and S_j the current solution set of v_j, $j = 1, \ldots, k$. Observe that:

1. $\{[S_1, \ldots, S_k]\mathsf{Evaluate}(v_i, G(c, v_i))\} \subseteq S_i$
2. if $S'_j \subseteq S_j$, $j = 1, \ldots, k$, then $\{[S'_1, \ldots, S'_k]\mathsf{Evaluate}(v_i, G(c, v_i))\} \subseteq \{[S_1, \ldots, S_k]\mathsf{Evaluate}(v_i, G(c, v_i))\}$.

Because of the monotonicity of Evaluate($v_i, G(c, v_i)$), and the fairness of propagation in Tolerate(c), the following holds (see [20]): if there exists some terminating propagation resulting in a locally consistent solution, then Tolerate(c) will find it. Moreover, the solution is unique (though its representation in terms of locus units is not necessarily unique).

Still, the propagation in Tolerate(c) need not terminate. A simple example is where two line segments are related by a constraint that requires their lengths to be equal, and a constraint that sets one length to half the other (a pure numeric analogue is the case of two constraints $x = y$ and $x = 2y$). This can lead to an infinite loop halving each line segment in turn.

A loop occurs when a constraint is revised more than once, in one invocation of Tolerate(). When this happens, it is not clear whether this is the prefix of an infinite loop or not, but we must break the loop after a finite number of times. One possibility is to terminate propagation by force when a constraint is addressed a fixed number of times. A more sophisticated approach is to break a loop when there are no more substantial changes made to the solution set. Specifically, we can break a loop when none of the locus units of a variable change type (for example change from a line segment into a point), and no locus unit changes more than a certain 'geometric margin'. After a loop has been cut-off, the current locus units of the variables in the loop form a super-set of the real solution. There are several things we can do with such a super-set:

1. give the super-set to the user as an approximate solution,
2. compute the solution set numerically, using the super-set to obtain initial values and bounds, if necessary,

3. compute a single solution from the solution set,
4. test whether a solution exists instead of computing a solution (set).

In our implementation we have chosen the first option, which is still better than many graphics systems can provide, see e.g. [21].

4.2.2. Disjunctive constraint

If a disjunctive constraint is added to the network, the solution sets of variables may get larger. This can more easily lead to an infinite loop than when the solution is restricted. When a variable involved in a circular chain of constraints gets assigned alternative locus units, these constraints may produce alternative units for their variables ad infinitum. Again, a loop need not be the prefix of an infinite loop, but propagation must be stopped. A loop can be stopped after a constraint is addressed a fixed number of times, or when the increase of the solution set exceeds a certain 'geometric margin'.

A (possibly infinite) increase of the solution may not have been the intention of the user, and the system can ask whether the alternative must be withdrawn. In any case, after a loop has been cut-off, the current locus units of the variables in the loop form a sub-set of the real solution. We can use this result to:

1. give the sub-set to the user as an approximate solution, which is still better than many graphics systems can provide,
2. try to generate a super-set of the solution, and proceed with propagation in order to restrict the solution.

A super-set can be generated by generalizing locus units, e.g. a point to a line segment. In our implementation we have chosen the first option.

4.3. Time complexity

In this section we analyze the time complexity of our algorithm, assuming that a loop in the iteration is broken if it addresses a constraint more than a fixed number of times. Let ℓ be the maximum number of loops in the iteration, a the maximum arity of the constraint types, q the maximum number of locus units returned by the locus unit generating function, e the number of constraints in the network, v the number of variables in the network, d_i the number of constraints on the ith variable, and d the maximum d_i over all variables. After each call $\mathsf{Revise}(c)$, the number of elements put into the queue by each variable v_i constrained by c is $d_i - 1$. Summing over all variables ℓ times, the total number of revisions is

$$\ell \sum_{i=1}^{v} (d_i - 1) = \mathcal{O}(\ell(ae - v)).$$

If the constraint network is not connected, each of the connected components may be treated independently, so we assume that the network is connected and thus $(a-1)e \geq v-1$. The total number of revisions is then $\mathcal{O}(a\ell e)$.

A call to Revise(c) results in the generation of locus units for each of the variables of c. The generation of locus units may involve geometric reasoning, but in any particular implementation this involves a fixed number of constraint types, geometric primitive types, and geometric configurations that are recognized, e.g. organized in a lookup table, thus taking a constant time. The at most q locus units generated for each variable v_i must be combined with its previous solution. In the worst case the locus units must be intersected with a locus unit expression which is itself an intersection of locus unit expressions, etc. The depth of this expression of unevaluated locus unit intersections is at most d_i, i.e. the number of constraints on variable v_i. So, in the worst case the number of intersections will be q^{d_i} for that variable, and at most $\mathcal{O}(aq^d)$ for one call to Revise(c). Summing over the total number of revisions gives a total time complexity of $\mathcal{O}(a^2 \ell e q^d)$. In this expression, a, ℓ, and q are constants. In the theoretically worst case, the maximum number of constraints on each variable is equal to the total number of constraints, e. However, for large systems of constraints and variables it is unrealistic to assume that a variable is subject to all constraints. In practice there will be a constant upper bound on the number of constrains applied to each variable, i.e. d is constant. This makes the time complexity of our Tolerate algorithm linear in the number of constraints. Moreover, the algorithm is local in the sense that propagation to other constraints stops when the solution of these constraints are not affected. No constraints are unnecessarily revised, and so the actual time complexity is typically sub-linear in the total number of constraints.

4.4. Completeness

A constraint satisfaction system is called complete if it always finds a solution to the constraints, if one exists. Completeness of any implementation of our approach depends on the set of geometric primitives, the allowable constraints, and the power of the locus unit generating function. In general, a solution set may not be representable with a fixed set of geometric primitives. In that sense the set of constraints and geometric primitives should agree with each other, so that the locus unit generating function is able to generate locus units for all admissible combinations of constraints and operands. Another aspect is the system's competence to intersect locus units. Consider a locus unit that is a circle whose center is allowed to move over a line segment, thus sweeping out a whole area of solutions. Although the area has a simple shape (a rectangle plus two half discs), its definition in terms of a swept circle makes intersection with subsequent locus units non-trivial. In this example the intersection is clearly computable, but a particular implementation may not be able to perform this intersection.

The locus unit generating function may use geometric reasoning [18]. Geometric reasoning can also facilitate constraint inferencing, the derivation of implied constraints. For example, given the constraints parallel(line1, line2) and parallel(line2, line3), the implied constraint parallel(line1, line3) can be derived. Constraints can also be restated, which can sometimes help the locus unit generating function. For example the constraint on(point,line) can be restated as distance(point, line, 0). The declarative representation of information in terms of constraints not only supports the integrity of supplied information, but also facilitates geometric reasoning, which prevents accumulation of errors and inconsistency of construction.

A more modest requirement than completeness of the whole system is completeness of propagation. A propagation scheme is called complete if the solution it assigns to a variable represents accurately the set of values it can attain given the constraints, i.e. after propagation the network is arc consistent in the sense of [8]. In a constraint system whose propagation scheme is complete, we can consistently assign to any variable any value within its solution, and pick values for all the other variables so that all constraints are satisfied. Provided that the locus unit generating function generates the right units for all implemented combinations of constraint types and operand types, and provided that no loops need to be broken, our propagation algorithm is complete.

5. Discussion

As mentioned before, a solution set may not be representable as the union of a number of geometric primitives (locus units) from a given set. In that case, the solution is represented implicitly as the intersection of several locus unit expressions. Conversely, a solution set need not be uniquely representable as a locus unit expression. This is no problem, however, since the different representations form the same solution.

Constraint satisfaction as performed in this paper yields a locally consistent solution for all variables, or equivalently, has the property of arc consistency: we can consistently assign to any variable any value within its solution, and pick values for all the other variables so that all constraints are satisfied. However, all these whole solution sets do not necessarily satisfy all constraints simultaneously. All constraints together constitute a global constraint which does specify solution sets for each variable that wholy satisfy all constraints simultaneously. Derivation of these solution sets is done, for example, by [22].

Operations on a variable, such as translate, scale, mirror, and project, can be handled by performing proper corresponding operations on its locus units. The resulting new solution must then be propagated through the network. This is precisely what constraint maintenance is about: maintaining satisfied constraints when the variables change. For interactive CAD purposes, a constraint must also be removable. After deletion of a constraint, the solution of its variables must be resynthesized, and then propagated through the network, see also section 6.

So far we have seen zero and one-dimensional locus units. A two-dimensional unit (region) results from a constraint like inside(point, circle), yielding a disc. The same approach to geometric constraint satisfaction works in three-dimensional space [23]. However, it becomes more likely that a solution cannot be represented as the union of geometric primitives and must be represented as an unevaluated intersection of locus units, in which case the locus unit expressions are akin to CSG-trees.

6. Implementation

We have implemented our constraint satisfaction method and incorporated it into a drawing editor using GoPATH [24] [25]. All figures in this paper were made by this prototype

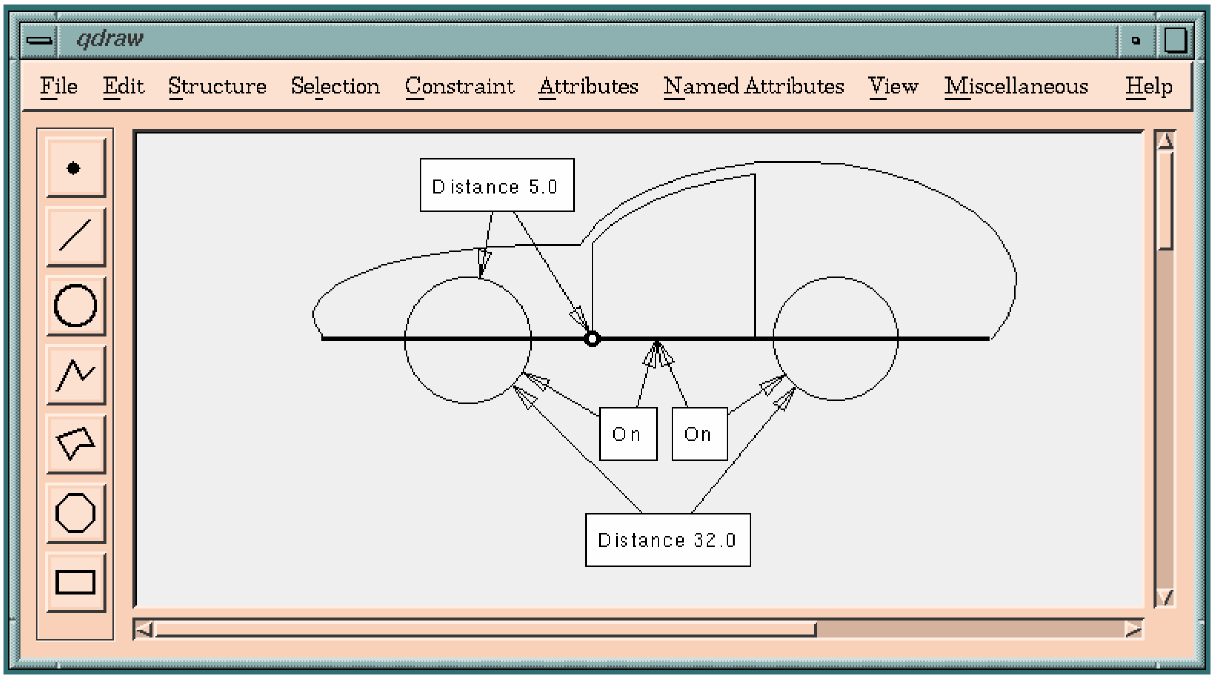

Figure 8: Snapshot of the drawing program showing constraint network without locus units.

system. Apart from the constraints anchor, on, and distance which were used in section 1, we implemented constraints tangent and angle. For example, angle(point1, point2, ampl, start) yields a locus unit that is a circle segment with amplitude ampl starting at angle start, and whose radius may result from a distance constraint. Constraints and their operands can be interactively generated and removed. Our straightforward prototype implementation is efficient enough for interactive use of the editor.

The constraint network is represented as a directed multi-partite graph. A constraint, conceptually a hyper-edge between its operands, is represented by a rectangular box and arrows towards its variables (see figure 8). Constraint nodes have only outgoing edges, while the constraint operands have only incoming edges. In our implementation, the anchor constraint is an exception: it is not displayed as a node in the graph but is visualized by drawing its operand in bold style, see for example the chassis in figure 8. Operands that are not anchored are drawn in normal style, as the wheels, and so are the elements that are not subject to constraints.

The locus units are drawn green. If a locus unit is fixed, it is drawn in bold style, otherwise in normal style. When locus units are generated, they are placed between the variable it belongs to and the constraint it results from. In order to represent unions and intersections of locus units, Boolean set operation nodes $\cup$ and $\cap$ are used. Between a constraint and an associated variable is a sequence of (zero or more) locus units interleaved with Boolean set operation nodes. Figure 9 depicts the network corresponding to figure 6, showing the union of the four locus units. Such a network can be read starting from the variables back to the constraints. The intersection node between the line segment

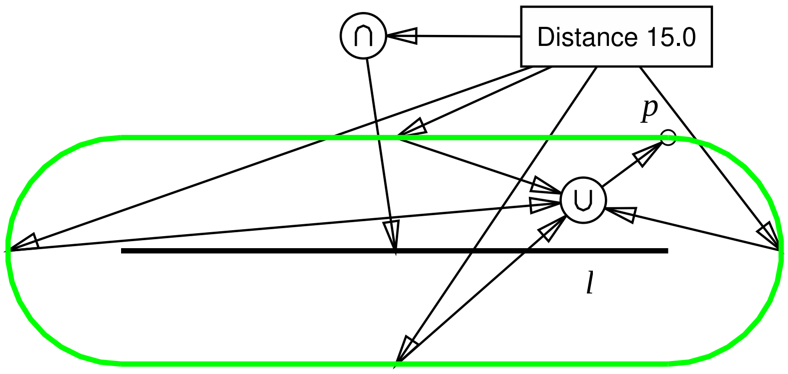

Figure 9: Network associated with distance(p, l, 15)

and the constraint node results from the fact that the constraint has been specified as a conjunctive one. When we cannot compute an intersection of two locus units, we leave the intersection unevaluated, and represent it with a $\cap$ -node in the network. Figure 7 shows a network with some intersection nodes. This network corresponds to the example in the introduction, just after the fourth constraint. At that point, the solution for each wheel consists of the intersection of the green line segment and the proper green circle. The intersections are not yet computed but explicitly represented in the network, because the green circles have no fixed position (and are therefore not drawn in bold style).

Because displaying the whole constraint network clutters the screen for more than only a few constraints, and because the network is hard to read, this is not the normal display mode. Users can select a number of display modes. One option is displaying the whole network. Another choice is to display a simplified network showing only variables and constraints but no locus units (see figure 8). Using constraint icons like in [26] would be still better. A user can also choose to display all variables and only the locus units of selected variables, as done in figures 1-6.

The usual way to use constraints in a drawing program is to put constraints on the objects in the drawing in order to specify their position and orientation. Our locus unit approach provides an alternative way to make drawings: not the constraint operands but the primitives representing the locus units constitute the drawing. An example is given in figure 10, showing Roman letters constructed after [27], using all constraint types implemented in our prototype system.

For interactive CAD purposes, a constraint must also be removable. In our system, deletion of a constraint from the network involves removal of the constraint node and the sequence of locus units and Boolean set operation nodes towards its involved variables. Sequences from other constraints to the same variables must be 'repaired' by intersecting the proper locus units. In general, this re-evaluation is necessary because dropping locus units from a sequence of intersections may yield completely different results. The resulting new locus units must be propagated through the new network.

A useful interaction mode (not implemented in our prototype system) would be to drag geometric primitives along their locus units (a constrained translation) towards a specific

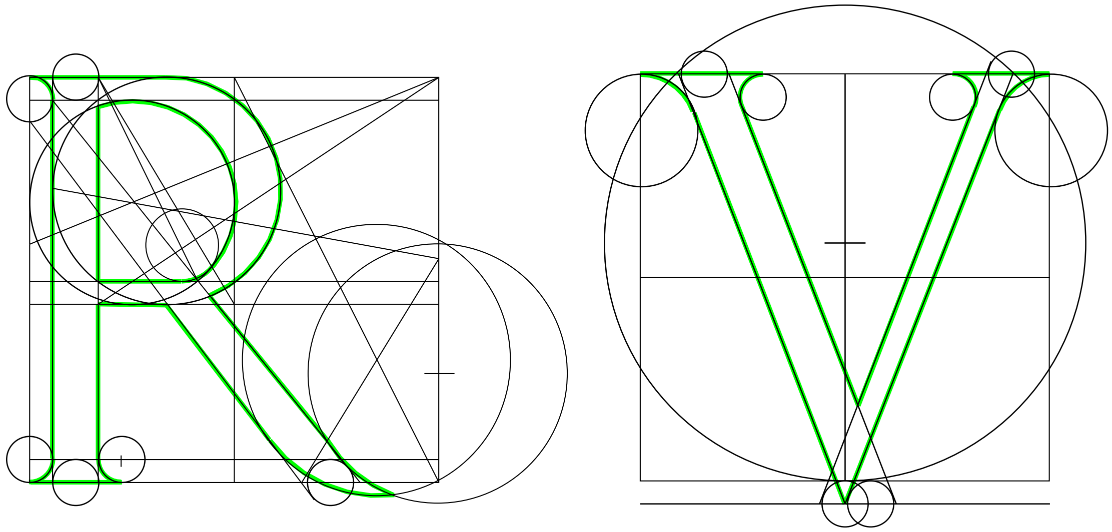

Figure 10: Roman letters, constructed after [27].

position, and to anchor the primitive there.

7. Comparison

Real general purpose languages for constraint logic programming (CLP) can be used in a wide range of applications, but are usually limited in their satisfaction power in each specific domain. Most constraint languages are biased to a more specific domain $\mathbb{D}$: CLP($\mathbb{D}$) [28]. For numeric constraints this yields CLP($\mathbb{R}$) over the domain of real numbers [29]. CLP is often described as logic programming with unification (pattern matching) replaced by constraint solving over a computation domain (e.g. Booleans or reals). When a CSP is embedded into logic programming, a constraint can be defined in the program as a set of facts and rules. To solve CSP in traditional logic programming, backtrack search is used and the constraints are used passively as posteriori tests. However, propagation techniques can have a dramatic effect in cutting down the size of the search space [30]. A general purpose language for constraint imperative programming is Kaleidoscope [31].

In the rest of this section we specifically consider geometric constraint systems. Obvious drawbacks of unstructured techniques are their computational complexity and their potential inefficiency for interactive applications: each single change leads to re-solving the whole set of constraints. An example is variational geometry [32] which translates dimensional constraints into an overall system of equations, which is solved numerically by the Newton-Raphson method. The dimensional constraints are defined by equations on coordinates of characteristic points. Each time a dimensional value is changed, the whole system of equations must be solved. Another system that turns all constraints into numerical equations is Juno [15]. Juno is a simple system based on one geometric primitive: the point. It uses a Newton-Raphson iteration technique to solve constraints. The user must supply an initial value to start the iteration. All these systems yield a single numerical solution.

Sketchpad [13] was the first constraint-based drawing system. It satisfies constraints using propagation of degrees of freedom. When this fails, it resorts to relaxation. ThingLab [14] enlarges the possibilities of Sketchpad with extensibility and object-oriented techniques, so that new classes of objects and constraints can be defined. It uses both propagation of degrees of freedom and propagation of known states. Both systems provide a single solution to constraints. The work on ThingLab evolved into the SkyBlue constraint solver [1], which makes an analysis of the constraint network and identifies loops, before performing local propagation. In the constraint-based geometric modeler Converge [26] the constraint network is partitioned, and the parts are solved numerically. In Converge, a locus can be specified to define a constraint, whereas our loci (geometric primitives) result from constraints.

[10] presents an operational interpretation of constraints in CSG modeling. Constraints are specified by users in terms of relations between boundary features, and are transformed by the system into rigid motions of parts of the CSG tree. An underconstrained situation can simply not occur. Users must specify the order of evaluation, and are responsible for solving conflicts. OTP (Operational Transformation Planning) [11] also provides an operational interpretation of constraints. It infers a single solution to the constraints. The satisfaction process is planned through symbolic reasoning on the geometric level, that is, by geometric reasoning [18]. [17] performs a degrees of freedom analysis on markers (local coordinate frames) on a geometric object, determining its translational and rotational degrees of freedom. If needed, a locus analysis is done akin to, but simpler than, our locus unit approach. Transformations are then generated so as to satisfy constraints. Coupled degrees of freedom, however, cannot be neatly divided into translational and rotational degrees of freedom. A branch variable must choose from multiple discrete solutions. The value of each branch variable must be set by users, affecting the rest of the solution derivation.

In [16] constraints relate coordinate systems. Constraints between degrees of freedom (for example between the x- and y-coordinate because of a distance constraint) are evaluated after lower-order constraints (for example one that uniquely determines the x-coordinate). This is a form of delayed satisfaction. Selection among alternative solutions to constraints (single solution inference) is based on 'minimal resulting disturbance'. [33] presents a system with single numeric solution inference, in which the numerical solutions are derived by algebraic methods. The propagation mechanism employed is propagation of degrees of freedom.

Our approach differs from all systems above in that solutions are represented, carried forward, and shown to the user in geometrical form. In many underconstrained cases, all solutions are derived (see section 4.2.2). The geometrical representation of the solution also avoids time complexity problems, since a single locus unit can represent a range of infinitely many solutions.

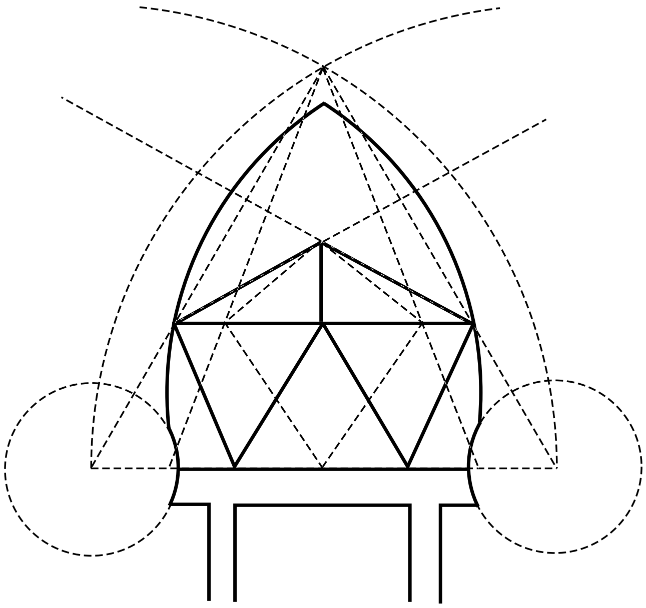

Figure 11: Construction drawing, after the construction drawing of the spire of the porter's lodge of Park Güell, Barcelona, Spain, by Antoni Gaudí [34].

8. Conclusions

In this paper we have presented a new incremental approach to geometric constraint satisfaction, representing solutions in terms of locus units, which are geometric primitives. Expressions of unions and intersections of locus units represent the whole solution to the constraints, so that underconstrained situations, which are so typical in interactive applications, are easily handled. The representation in terms of locus units also provides geometrically meaningful feedback to the user and supports the ability to combine the interpretation of previous and new geometric constraints on the same high level of abstraction, thus allowing geometric reasoning and constraint inference. Another advantage of our approach is the natural processing of expressions of both conjunctive and disjunctive constraints. Because the whole solution to the constraints is represented, the final solution does not depend on the order in which the constraints are solved, so that the declarative semantics of constraints is preserved.

The set of geometric primitives used as constraint operands in our prototype system consists of point, circle, and line segment; the set of constraints consists of anchor, on, distance, and tangent and angle. On the one hand this allows only a limited number of constraints in the realm of geometry; on the other hand several other constraint systems offer only a single primitive, e.g. the point. Moreover, this limited set of constraints already covers a large amount of constraints typical in more specific fields such as mechanical engineering. Our approach, even with this limited set of primitives and constraints, can relieve much of the burden of repeatedly satisfying mundane constraints that are dynamically changing, a situation so typical in interactive applications. Figure 11 gives a final example of a picture made with the prototype system, using all the constraint types available.

In order to extend the capabilities of our prototype system, not only larger sets of geometric primitives and constraints may be needed, but also it may be necessary to define constraints on sub-objects (e.g. the vertices of a polyline), or on compound objects (e.g. polylines). Some useful enhancements to the interface of our system include the ability to infer constraints, like in [35] and [36], the ability to copy parts of the current constraint network, and to define macros of constraint expressions, such as middle(M,A,B) $\equiv$ on(M, linesegment(A,B)) AND distance(M, A, $\|AB\|/2$).

Acknowledgements

This work was supported by NWO (Dutch Organization for Scientific Research) under Grant NF-51/62-514.

References

[1] M. Sannella. Constraint satisfaction and debugging for Interactive User Interfaces. *(Ph.D. dissertation), Technical Report 94-09-10*, Dept. Computer Science and Engineering, University of Washington, Washington, 1993.

[2] J.-F. Balaguer and E. Gobetti. Supporting Interactive Animation Using Multi-way Constraints. In [37], 49 – 66.

[3] M. Dohmen. A survey of constraint satisfaction techniques for geometric modeling. *Computers & Graphics*, 19(6), 1995, 831-845.

[4] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[5] S. Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1978.

[6] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra — systems and algorithms for algebraic computation.* Academic Press, 1988.

[7] W. Leler. *Constraint Programming Languages, Their Specification and Generation.* Addison-Wesley, 1988.

[8] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8, 1977, 99 – 118.

[9] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32, 1987, 281 – 331.

[10] J. R. Rossignac. Constraints in constructive solid geometry. In F. Crow and S. M. Pizer (editors), *Proceedings of the 1986 ACM Workshop on Interactive 3D Graphics*, ACM Press, 1986, 93 – 110.

[11] F. Arbab and B. Wang. A geometric constraint management system in Oar. In P. J. W. ten Hagen and P. Veerkamp (editors), *Intelligent CAD Systems III – Practical Experience and Evaluation*, Springer-Verlag, 1991, 231 – 252.

[12] G. L. Steele Jr. and G. J. Sussman. CONSTRAINTS – A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 1980, 1 –39.

[13] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, AFIPS Press, 1963, 329 – 345.

[14] A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4), 1981, 353 – 387.

[15] G. Nelson. Juno, a constraint-based graphics system. *Computer Graphics*, 19(3), 1985, 235 – 243.

[16] M. J. G. M. v. Emmerik. A system for interactive graphical modeling with 3D constraints. In T. Chua and T. Kunii (editors), *CG International '90*, Springer-Verlag, 1990, 361 – 376.

[17] G. A. Kramer. *Solving Geometric Constraint Systems.* MIT Press, 1992.

[18] F. Arbab and J. M. Wing. Geometric reasoning: A new paradigm for processing geometric information. In H. Yoshikawa and E. A. Warman (editors), *Design Theory for CAD*, Elsevier Science Publishers, 1985, 145 – 165.

[19] D. L. Waltz. *Generating Semantic Descriptions from Drawings of Scenes with Shadows.* PhD thesis, MIT, 1972.

[20] H.-W. Güsgen and J. Hertzberg. Some fundamental properties of local constraint propagation. *Artificial Intelligence*, 36, 1988, 237 – 247.

[21] B. N. Freeman-Benson, J. Maloney, and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1), 1990, 54 – 63.

[22] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11), 1978, 958 – 966.

[23] R. C. Veltkamp. Geometric constraint management with quanta. In D. C. Brown, M. B. Waldron, and H. Yoshikawa (editors), *Intelligent Computer Aided Design*, North-Holland, 1992, 409 – 423.

[24] *GoPATH 1.2.0 — A Path To Object Oriented Graphics, a public domain environment for graphical and interactive application development.* Bull – Imaging and Office Solutions, 1993.

[25] J. Davy. Go, a graphical and interactive C++ toolkit for application data presentation and editing. In *Proceedings 5th Annual Technical Conference on the X Window System*, 1991.

[26] S. Sistare. Graphical interaction techniques in constraint-based geometric modeling. In *Proceedings of Graphics Interface'91*, 1991, 85 – 92.

[27] D. L. Goines. *A Constructed Roman alphabet: a geometric analysis of the Greek and Roman capitals and of the Arabic numerals.* David R. Gordine, Boston, 1982.

[28] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7), 1990, 52 – 68.

[29] N. C. Heintze, S. Michaylov, and P. J. Stuckey. CLP($\mathbb{R}$) and some problems in electrical engineering. In J.-L. Lassez (editor), *Proceedings of the 4th International Conference on Logic Programming*, MIT Press, 1987.

[30] T. L. Provost and M. Wallace. Generalized constraint propagation over the CLP scheme. *The Journal of Logic Programming*, 16, 1993, 319 – 359.

[31] B. N. Freeman-Benson. Kaleidoscope: Mixing objects, constraints, and imperative programming. *SIGPLAN Notices*, 25(10), 1990, 77 – 88.

[32] V. C. Lin, D. C. Gossard, and R. A. Light. Variational geometry in computer-aided design. *Computer Graphics*, 15(3), 1981, 171 – 177.

[33] J. C. Owen. Algebraic solution for geometry from dimensional constraints. In *Proceedings of the ACM Conference on Solid Modeling*, ACM Press, 1991, 397 – 407.

[34] R. Zerbst. *Antoni Gaudí*. Taschen, 1987.

[35] M. Gleicher and A. Witkin. Creating and manipulating constrained models. Technical Report CMU-CS-91-125, Carnegie Mellon University, School of Computer Science, 1991.

[36] S. R. Alpert. Graceful interaction with graphical constraints. *IEEE Computer Graphics & Applications*, 13(2), 1993, 82 – 91.

[37] R. C. Veltkamp and E. H. Blake (editors). *Programming Paradigms in Graphics*, Springer-Verlag, ISBN 3-211-82788-9, 1995.