The geometric calculator

Luis A. Pineda
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas (IIMAS)
Universidad Nacional Autónoma de México (UNAM)
Ciudad Universitaria, México, D. F., México
e-mail: <luis@leibniz.iimas.unam.mx>

Abstract

In this paper a geometric calculator is presented. A language of geometrical expressions, that are evaluated by the calculator, and the design and implementation of its interpreter is also presented. Geometrical expressions correspond to abstract data-types that can be defined dynamically through composition operations. Diagrams can be represented by a set of geometrical expressions, and the geometrical properties of a diagram or a diagram sequence can be computed through the calculator. The expressive power of the tool is illustrated with two applications: the generation of design patterns, and a simple but interesting case of geometrical constraint satisfaction.

Keywords: geometric calculator, geometrical languages, geometric abstract data-types, geometrical interpretation, diagrammatic representation, shape grammars, geometric constraint satisfaction.

1. The geometric calculator

In the same way that arithmetic calculators return the value of arithmetic expressions, a geometric calculator is a program that given a geometrical operation over a number of geometrical objects, returns an object of an appropriate geometrical type as its value. Also, in the same way that composite arithmetic expression result from the combination of a number of operators with their corresponding arguments, basic geometrical symbols and operators can be combined in the definition of composite geometrical expressions. Consider the drawing in Figure 1:

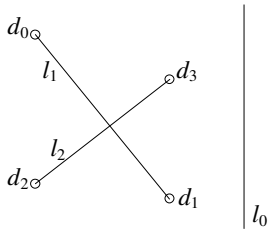


Figure 1

Examples of geometrical expressions that can be evaluated by the geometric calculator in relation to this drawing are:

(1) *is_line(line(*d_0, d_1*))*

(2) *perpendicular*(l_1, l_2)

(3) *parallel*(l_1, l_2)

(4) *intersect*(l_1, l_2)

(5) *symmetrical*(*triangle*(d_1, d_3, *intersect*(l_1, l_2)), l_0)

(6) d_i = *intersect*(l_1, l_2)

Expression (1) is a type predicate that verifies that a geometrical object is well-formed (i.e. a line must have a length); expression (2) is true and (3) false in relation to the diagram; expressions (4) and (5) are functional constructors that have as their values the dot at the intersection between lines l_1 and l_2 and the triangle that

results from computing the axial symmetry of the triangle formed by d_1, d_3 and the intersection dot in relation the axis l_0, as shown in Figure 2; expression (6) predicates the geometrical equality between two objects of same type, and can be true or false depending on the positions of the dots in question.

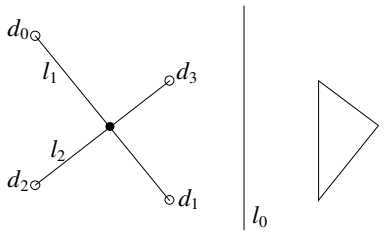


Figure 2

The geometrical calculator evaluates a geometrical expression in relation to a geometrical environment constituted by a set of geometrical symbols, in the same way that programming languages interpreters evaluate expressions in relation to a programming environment, as illustrated in Figure 3. In the example, the geometrical environment is the set $\{d_0, d_1, d_2, d_3, l_0, l_1, l_2\}$, and the symmetrical triangle and intersection dot are constructed in relation to this set.

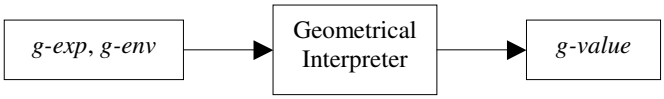


Figure 3. Geometrical interpretation process

The geometrical environment can be empty; in this case, the meaning or semantic value of a geometrical expression is simply the description of a geometrical object in the geometric domain, with no reference to any particular geometrical configuration or its graphical realization in a diagram.

Geometrical expressions have extensional values (i.e. the values of properties and relations of geometrical objects in the evaluation state), and the evaluation process returns these values; consequently, geometrical objects can be represented or depicted through graphical symbols in a rendering process. However, the identity of geometrical objects does not depend on the values of their properties in any particular state: if the position of d_3 in Figure 2 is altered as shown in Figure 4 (e.g. by dragging it to a new position with a graphical cursor in a typical interactive manipulation) the identity of the dot is preserved; in the change process, objects defined in terms of other objects can also be altered, like the line l_2 whose definition includes the identifier of the modified dot, and the intersection dot and symmetrical triangle, whose descriptions include references to other objects; more generally, definitions (1) to (6) are *intensional* as the properties and relations of the objects represented through these expressions depend not only of geometrical algorithms but also on properties of the basic objects (i.e. defined extensionally) in the environment in evaluation state.

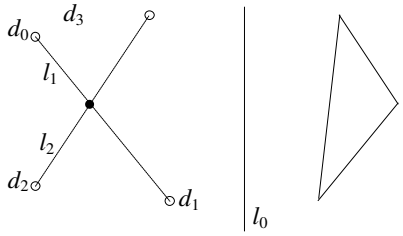


Figure 4

The expressive power of a geometric calculator depends on the set geometrical sorts (e.g. dot, line, path, right-triangle, triangle, square, rectangle, polygon), and in the richness and variety of predicates and function symbols defined in the language. Our present implementation includes, in addition to the basic constructors for every geometrical type, type predicates, equality predicates, geometrical predicates (e.g. horizontal, vertical, parallel,

perpendicular, etc.), property selectors (e.g. position of a dot, length or angle of a line, dot at the intersection between two lines, or the dot at the extreme or "t" joint between two lines), rotations, translations and symmetrical operations over an object with respect to a given reference, etc. We also include a facility for defining geometrical functions that can be applied to geometrical objects on demand.

Interactive graphics editing operations, graphics design, diagrammatic problem-solving and theorem-proving require the representation of diagrammatic sequences; in these kinds of applications each state in a sequence may be represented by an environment. Environments should only include the geometrical objects relevant for the task at hand, and can be thought of as the objects that are attended to by the interpreter in the problem-solving task. Also, a diagram sequence is normally spatially coherent, and only the values of a few properties of a subset of objects in the environment differ from one state to the next in the sequence. For instance, the difference between the environment for Figures 2 and 4 is the coordinate position of d_3; the length of l_2 and the properties of the intersection dot and the symmetrical triangle are also altered, but the values of these expressions are implicit in their intensional definitions, which remain the same along the change process, and are only evaluated on demand. For all these reasons, the geometric calculator, with its ability to deal with intensional representations of geometrical objects, is a tool suitable for the definition of intelligent graphics, diagrammatic problem-solvers and theorem-provers in IA.

2. The geometrical Language

In this section, the syntax and semantics of the language evaluated by geometric calculator is presented. The definition includes a strategy to handle geometrical expressions that have no well-defined referent in an interpretation state; we also show the definition of the interpretation environment, and how geometrical objects can be given extensional and also intensional definitions. Finally, we present the interpretation strategy.

2.1 Syntax

We first define an ordered and sorted set of geometrical objects; the geometrical sorts are *dot*, *line*, *path*, *right-triangle*, *triangle*, *square*, *rectangle, parallelogram* and *polygon*. Geometrical sorts are ordered: *line* < *path*, *right-triangle* < *triangle* < *polygon* and *square* < *rectangle* < *parallelogram* < *polygon*. In addition, we define the following sorts that have no graphical realization: *real*, *real-pair* and *bool*.

With this set of sorts, we define a term-grammar constituted by a set of geometrical constructor and selector operators of type $s_1,\ldots,s_n \rightarrow s_m$; the syntax is given by the following term composition rule: if f is an operator of type $s_1,\ldots,s_n \rightarrow s_m$, and $x_1,\ldots,x_n$ are terms of sorts $s_1,\ldots,s_n$, the expression $f(x_1,\ldots,x_n)$ is a well-formed term of sort s_m. We also provide a denumerable set of constant and variable symbols for every sort. The set of geometrical operators constitute a *signature* for the language (Goguen et. al., 1978; Pineda, 1989); for clarity we represent the elements of the operator's set with the following notation:

(23) *g-op*(*operator-name*, *argument-list*, *sort-of-term*)

We define a basic constructor operator for geometrical objects of every sort as follows:

(23) g-op(dot, [bool, real-pair], dot)
g-op(line, [bool, dot,dot], line)
g-op(path, [bool, dots-list], path)
g-op(right-triang, [bool, dot, dot, dot], righ-triang)
g-op(triang, [bool, dot, dot, dot], triang)
g-op(square, [bool, dot, dot, dot, dot], square)
g-op(rectangle, [bool,dot, dot, dot, dot], rectangle)
g-op(parallelogram, [bool,dot, dot, dot, dot], parallelogram)
g-op(polygon, [bool, dots-list], polygon)

We also define type and equality predicates for symbols of every geometrical sort:

(23) g-op(*is-sort*, [*bool*,*sort*,*sort*], *bool*)
g-op(*eq*, [*bool*,*sort,sort*], *bool*)

The signature includes also a large number of selectors for computing the geometrical properties of symbols of every kind (e.g. position of a dot, length and angle of a line, hypotenuse, right-angle, right-sides, angles and area

of a right-triangle, etc.), verifying whether symbols have a geometrical property (e.g. vertical, horizontal), or stand in geometrical relation (e.g. parallel, perpendicular, a polygon within a polygon, a dot on a line or path, etc.), and for constructing a new object (e.g. intersection dot between two lines, the object that results of rotating or translating a given object, or the symmetrical object of a reference object in relation to an axis, etc.). The full signature of our current implementation is listed in Appendix 1. Also, in our interpretation scheme, the list of operators can be augmented by listing new operators in the *g-op* list, and by providing the corresponding geometric algorithm in the semantics, as will be shown below. Likewise, for the definition of 3-D objects it is only required to state the corresponding operators, and the 3-D algorithms for computing the corresponding properties and relations, as shown in Garza and Pineda (1998).

2.2 Undefined values

The first argument in the argument of every operator in the signature is always of sort *bool* (e.g. (7), (8) and (9)). We define the value of this argument to be true in a given term if all its geometrical arguments have a well-defined value in the evaluation state, and false otherwise. In the same way that division by cero has an undefined value in arithmetic expressions, a geometrical term can have an undefined value in some states, although its can be well-defined in others. For instance, the boolean variable B_0 in (10) is true unless one or both of the lines are not well-defined in the evaluation state (i.e. have no length, or their origin and extreme dots are not well-defined themselves):

(10) *intersect*(B_0, l_1, l_2)

Likewise, the value of (10) is of an object of sort *dot* and its interpretation produces (11); the variable B_1 in this latter expression is true if the dot in question is well-defined, and false otherwise (e.g. the lines are parallel). The term x_i:y_i is an object of sort *real-pair* and x_i and y_i stand for the actual real values of the coordinate position of the intersection dot.

(11) *dot*(B_1, x_i:y_i)

We this device, we handle the interpretation of geometrical expressions taken into account whether or not all arguments of a term, and the term itself, have a well-defined value. If this is the case, the interpreter returns the description of a geometrical object of the proper type; otherwise, the interpreter returns the input term with the first argument set to false. In case the lines in (10) have not a well-defined value or are parallel in the evaluation state, for example, the interpreter returns (12) as its value:

(12) *intersect*(*false*, l_1, l_2)

2.3 Semantics

We turn now to the semantics of the geometrical language; this is constituted by two main objects: the geometrical interpreter and a set of geometrical algorithms; every algorithm in this set is associated to a constructor or a selector operator in the signature, and computes the basic geometrical property or relation named by the corresponding operator. The interpreter is a program that given a term and a geometrical environment applies the operator to the value of its arguments in relation to the environment and returns a term of the corresponding sort: the extensional value of the input term in the evaluation state.

The evaluation environment is defined as a set of geometrical definitions and descriptions (in our implementation, the environment is represented as a prolog list, but the evaluation process is independent of the order of the definitions in the list); each definition consists of an identifier associated to a well-defined expressions that has as its functor a constructor operator of the corresponding sort; descriptions can also be stated directly in the environment, as will be shown below in Section 3.1. Definitions can be extensional or intensional; in the former case, the description term associated to the identifier has no geometrical identifiers within its body; if the definition is intensional, on the other hand, the object being defined is a function of the objects denoted by the identifiers included in its description. The environment of the diagram in Figure 1, for instance, is represented as follows:

(23) *g_env* = [*g_def*(d_0, *dot*(*true*, x_0:y_0)),
g_def(d_1, *dot*(*true*, x_1:y_1)),
g_def(d_2, *dot*(*true*, x_2:y_1)),
g_def(d_3, *dot*(*true*, x_3:y_3)),
g_def(l_0, *line*(*true*, *dot*(*true*, x_4:y_4), *dot*(*true*, x_5:y_5)))

$g\_def(l_1, line(B_1, d_0, d_1))$
$g\_def(l_2, line(B_2, d_2, d_3))$
].

The first four definitions in (13) are extensional: these associate a constant with the extensional definition of an object of sort dot; expressions of the form $x_i{:}y_i$ stand for the actual coordinate values of the dot in the environment's state. The fifth definition is also extensional: although the description associated to l_0 contains two dots as its parts, it is already in a reduced form (i.e. the parameter dots are represented as basic expressions of sort dot, in the basic constructor's format, and have no name). The definition of l_1 and l_2, on the other hand, are intensional as their values can only be found by evaluating the parameter dots in relation to the environment. In the example, the value of the first Boolean argument of extensional definitions is set to "true", as the coordinate values of the dots included in these definitions are given directly; however, the value of this argument depends on the evaluation state and it is usually left under specified in definitions and descriptions.

The value of expressions (1) to (5) in relation (13) is illustrated in (14) to (18):

(14) $g\_eval(is\_line(line(B, d_0, d_1)), g\_env) \rightarrow true$

(15) $g\_eval(perpendicular(B, l_1, l_2) , g\_env) \rightarrow true$

(16) $g\_eval(parallel(B, l_1, l_2) , g\_env) \rightarrow false$

(17) $g\_eval(intersect(B, l_1, l_2), g\_env) \rightarrow dot(true, x_i{:}y_i)$

(18) $g\_eval(symmetrical(B_1, triangle(B_2, d_1, d_3, intersect(l_1, l_2)), l_0), g\_env) \rightarrow$

$triangle(true, dot(true, x_i{:}y_i), dot(true, x_j{:}y_j), dot(true, x_k{:}y_k))$

The expressions at the right-side of the arrow are produced by the interpreter and denote a truth value or the extensional description of a geometrical object in the evaluations state; in this latter case, the expression is a description in the object's basic constructor format, as defined in the signature. So in (18), for instance, the interpreter returns the description of a triangle formed by the dots at positions *i*, *j* and *k*. As the coordinate positions of the three vertices of the triangle are well-defined, the boolean parameter of the triangle's constructor is true.

The value of expression (6) can be obtained by evaluating the expression in relation of an environment *g_env'* as shown in (19); the environment *g_env´* is like *g_env* except that the value of (17) in relation to *g_env* (i.e. $dot(true, x_i{:}y_i)$) is named as d_i, and *g_env´* extends *g_env* with the definition $d_i = dot(true, x_i{:}y_i)$.

(19) $g\_eval(eq(B, d_i, intersect(B, l_1, l_2)), g\_env') \rightarrow true$

Now we give the interpreter's evaluation strategy: if the expression is a boolean constant, a real or a real-pair, the value is represented by the same expression. If the expression is an identifier in a definition, either extensional or intensional, its value is the value of the right-side of the definition (i.e., the *g_def* relation in (13)). Otherwise, the interpreter evaluates the arguments of the term left to right, and applies the reduced arguments to the expression's main operator; if there is an argument without a well-defined value, or the term as a whole has not a well-defined value, the interpreter returns the input expression but with the first boolean argument set to false.

Finally, once the arguments of a term have been fully reduced, the geometrical algorithm associated to the operator's term is applied to the arguments, and this computation results in a geometrical object of the operator's sorts. The specification of the algorithms for all the operators defined in the signature constitutes the semantics of the language and the interpreter applies these algorithms compositionally. The Prolog´s code of the current interpreter is simple enough and is given fully in Appendix 2.

The application of the geometrical semantics of the operator's signature is performed by the *g_value* clause in *reduce_gterm*, the main evaluation function of the interpreter. The variable *BASIC_EXP* in the clause *g_value*(*BASIC_EXP*, *VALUE*) is an expression whose functor is an operator in the signature, and its arguments are reduced already; the *g_value* clause applies the geometric algorithm associated with the operator to its arguments directly, and returns the value of the expression in *VALUE*. In appendix 3 we show the semantics of the *dot* constructor. The first instance of the *g_value* clause gives the semantic value of a constant in a definition, which is the value of its associated description. This value is given in the basic constructor format of the

symbol's sort. The second entry of the *g_eval* clause codifies the constructor operation properly; the geometric arguments of the constructor are the three dots defining the triangle, which must have a different position and must not be aligned; if these two condition hold, the three dots "construct" a well-defined triangle, and the first boolean argument of the constructor operator is set to true; otherwise, the object is not well-defined and the value of this argument is set to false. In the appendix 3 it is also shown the semantics of the type and equality predicates for objects of sort *triang*, and the selector operators for objects of this kind; through these operators, the properties of a triangle, like the description of their sides, the value of its angles, the base, height and area, can be provided by the interpreter directly. The values of some of these properties are extracted directly from the object's basic definition, but the value of some others, like the base, height and area, are computed in terms of other geometrical expressions, by invoking the interpreter recursively (i.e. through the *eval_gterm* function). The geometrical semantics for all the operators in the signature has the same format. In order to enrich the functionality of the geometric calculator with a new operator it is only required to list the operator with its arguments and sort in the signature (i.e. as in Appendix 1) and provide its geometrical semantics through the corresponding *g_eval* definitions (i.e. as in Appendix 3).

We also include the sort *func* as a special sort in the signature; this sort permits to construct geometric functions dynamically; this is illustrated by the operator *class_right_triang* that given the lengths of the right sides and the hypotenuse of a right triangle returns a function of type *real-pair* × *real* → *right_triang*; this latter function represents the class of right triangles of the given dimensions, and when it is applied to a position and an angle, returns a right triangle of the given sides at the corresponding position and angle.

More generally, the system allows the definition and evaluation of geometric functions with the following format:

(20) *lambda*(*var-list*, *g-exp*)

In (20) *var-list* is a list of variables, *g-exp* is a geometrical expression containing all the variables in *var-list* as free variables; the interpreter can apply functions of this form to a list of arguments (i.e. a list of geometric expressions of a proper geometrical sort); in the functional application process, the interpreter binds the arguments with their corresponding variables in *var-list* first, and evaluates the resulting expression in relation to the environment, as shown in the corresponding code in Appendix 2. With this we conclude the description of the functionality of the current implementation of the geometric calculator.

3. Applications

We conclude this paper with two applications to illustrate the functionality of the calculator, and the expressive power of the geometrical language and its interpreter. The first is a design application in which geometrical expressions are used to construct design patterns in a simple and general way, using extensional definitions only; in the second we use intensional definitions, and we show how the calculator can be used to solve a simple but interesting class of geometric constraint satisfaction problems.

3.1 Shape grammars

The production of design patterns made as compositions of a given basic "tile" or shape form by transformation operations (i.e. translations, rotations, symmetrical operations) based on a reference or "pivot object" within the pattern, has a long tradition in design (e.g. Bronowski, 1973); a computational development of this intuition for the definition of design families or styles was introduced in the shapes grammar formalism (Stiny, 1975); in this framework, shape patterns forming particular designs were modeled through design rules, implemented with production systems, and the set of productions were thought of as the characterization of design families. A simple but interesting grammar of right-triangles was develop by Weissman-Knight (1981); the shapes produced by this grammar are not only interesting design patterns, but also appear in one of the more compelling proofs of the Theorem of Pythagoras (Pineda, 2004). This diagrammatic sequence, generated out of an arbitrary right triangle, is shown in Figure 5.

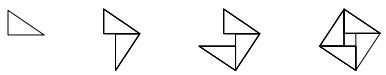


Figura 5. A grammar for right triangles

Next, we show the generation of this shape sequence with the geometric calculator. We define an initial environment in (21) including only the extensional description of an arbitrary right triangle: the *seed* for generation process; the main and secondary orientation of the seed (up, down, right and left) determine how the chain of shapes is developed; in figure 5, for instance, the main and secondary orientation of the seed are right and up respectively, but the family of shapes can be generated out of a seed with an arbitrary orientation.

(21) $g\_env_0 = [\ right\_triangle(true,\ dot(true,x_i{:}y_i),\ dot(true,x_j{:}y_j),\ dot(true,x_k{:}y_k))]$

The generation pattern is characterized by the geometrical expression in (22), which describes the next triangle in the generation sequence as a function of an arbitrary right triangle:

(22) *lambda*([*FOCUS*, *ANGLE*, *DELTA*],
translate(*true*, *rotate*(*true*, *FOCUS*, *ANGLE*, *right_vertex*(*true*, *FOCUS*)), *DELTA*)).

This function has two parameters in addition to the focus, which is the object generated last in the sequence; these are the rotation angle and the displacement (positive or negative along both coordinates) that the new object has in relation to the focus. The fourth parameter of the operation *rotate* is the rotation pivot, which in this cased is the right vertex of the focus right triangle itself. The angle and delta parameters are also a function of the focus, and correspond to $\pi/2$ or $-\pi/2$ and the difference of the lengths between the larger and shorter right sides of the right triangle respectively, and depend on the main and secondary orientation of the focus itself (e.g. if the main orientation of the focus is right and the secondary up, the rotation angle is $-\pi/2$ and the displacement is to the right). The function determining the values of these parameters is also easily defined with the geometric calculator.

The generation sequence is defined with a single production rule (i.e. function (22) applied to the current focus) that extends the environment env_i with a new object, defining the new environment env_{i+1}. The shape generation process has as its input parameters the initial environment and the focus, and returns a new environment with a reference to the new object, which becomes the focus for the next generation step; this is illustrated in Figure 6. The process is applied iteratively for the generation of the whole chain; the termination function is also defined as a geometrical condition computed by the geometric calculator (e.g. the new object is already in the input environment). For the example, the final environment env_3 contains the extensional description of the four triangles (but not of the two squares that emerge in this diagrams; for a discussion of the representation and interpretation of these emerging objects see Pineda, 2004). It is also possible to define more than one generation rule, and produce a non-deterministic design space for the shape family.

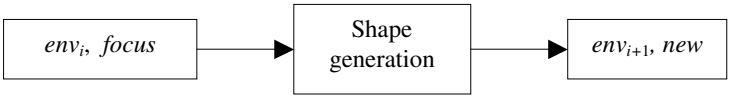


Figure 6. Shape generation process

The present generation process is an illustration of design applications in which extending a geometrical configuration is a process that depends on the local properties of a geometrical object that is currently taken as the focus of the generation process.

3.2 Theorem of the inscribed parallelogram

This theorem states that if the vertices of a four sides polygon lay on the middle of the segments of a quadrilateral, it is a parallelogram. If there are nested figures satisfying the same condition, all of these are parallelograms too; the theorem is illustrated in Figure 7.

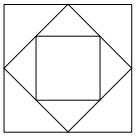
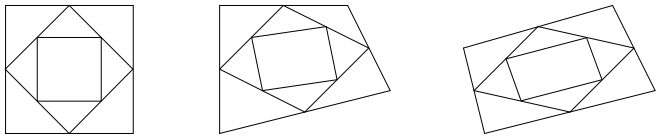


Figure 7. Theorem of the inscribed parallelogram

The problem is how to determine the coordinates of the vertices of the inscribed figures as a function of the inscribing ones if the positions of one or more vertices of the outermost control figure are modified. This is illustrated in Figure 8 where the bottom-right vertex of the control polygon in (8.a) is moved right and upwards as shown in (8.b); from (8.b) to (8,c) the upper-left vertex is moved left and downwards; as can be seen, both of the inscribed figures in all three diagrams are parallelograms, but the inscribing one needs not be.



a) b) c)

Figure 8. Modifying the control polygon

This problem has a long tradition in geometric constraint satisfaction, and was used to illustrate local propagation in ThingLab (Borning, 1981), and also constraint satisfaction through constructive procedures in object oriented programming paradigms (Alberti et. al, 1995); in our line of work, the theorem has been used to illustrate constraint satisfaction through reference relations and intensional representations (Massé, 1994). Here we present the solution of this problem through intensional representations and the geometric calculator. The initial environment env_0 for the diagram in Figure 8.a contains the following definitions (we use normalized space from 0.0 to 1:1 for the example):

(23)

%Control dots
g_def(d_1, *dot*(*true*, 0.2:0.2)).
g_def(d_2, *dot*(*true*, 0.8:0.2)).
g_def(d_3, *dot*(*true*, 0.8:0.8)).
g_def(d_4, *dot*(*true,* 0.2:0.8)).

%Control lines
g_def(l_1, *line*(*true*, d_1, d_2)).
g_def(l_2, *line*(*true*, d_2, d_3)).
g_def(l_3, *line*(*true*, d_3, d_4)).
g_def(l_4, *line*(*true*, d_4, d_1)).

%parameters parallelogram 1
g_def(d_5, *midle_dot*(*true*, l_1)).
g_def(d_6, *midle_dot*(*true*, l_2)).
g_def(d_7, *midle_dot*(*true*, l_3)).
g_def(d_8, *midle_dot*(*true*, l_4)).

%parameters parallelogram 2
g_def(d_9, *midle_dot*(*true*, *line*(*true*, d_5, d_6))).
g_def(d_{10}, *midle_dot*(*true*, *line*(*true*, d_6, d_7))).
g_def(d_{11}, *midle_dot*(*true*, *line*(*true*, d_7, d_8))).
g_def(d_{12}, *midle_dot*(*true*, *line*(*true*, d_8, d_5))).

%Nested parallelograms
g_def(plg_1, *parallelogram*(_, d_5, d_6, d_7, d_8)).
g_*def*(plg_2, *parallelogram*(_, d_9, d_{10}, d_{11}, d_{12})).

The initial diagram in Figure 8.a is produced directly by evaluating and rendering the lines of the control polygon, and the inscribed parallelograms plg_1 and plg_2 in env_0; the environment env_1 is like env_0 except for the definition of the control dot d_2; accordingly, changing the diagram from 8.a to 8.b only requires to create env_1 by a move operation defined as follows:

(24) $env_1 = move\_to(d_2, dot(true,\ 1.0{:}0.4), env_0)$

Figure 8.b is obtained by evaluating and rendering the definitions of the control lines and the parallelograms in env_1 as before; Figure 8.c is the graphical representation of the environment env_2 in which the position of d_4 is updated by a similar process:

(25) $env_2 = move\_to(d_4, dot(true,\ 0.1{:}0.6), env_1)$

Current object oriented graphical editors, as AutocCAD, PowerPoint and even Word support an approximate behavior: in these editors it is possible to run this example by creating the initial three objects, and grouping them; the group can then be modified by dragging the control dots to new positions and the nested figures will be parallelograms; however, the positions of other control dots, different from the one explicitly modified, can

be also altered as a side effect of the move operation, as the definitions of geometrical abstract data-types in object-oriented graphics is not fully compositional.

A feature of our representational scheme is that most of the definitions in (23) are intensional, and remain constant along every state in the sequence of transformations, and only the definition of the four control dots is extensional. Accordingly, there is no need to keep a copy of the intensional definitions for each state of the sequence, and a single record of these definitions, in addition to the extensional definitions for each state, is enough to record the change information of the diagrammatic sequence. This suggest to structure the environment in a two levels hierarchy: the intensional definitions *i_env* are stored in the upper level (together with extensional definitions that undergo no change along the change process), and the extensional environments e_env_i, corresponding to each state in the diagrammatic sequence, are stored in the lower one, as illustrated in Figure 9.

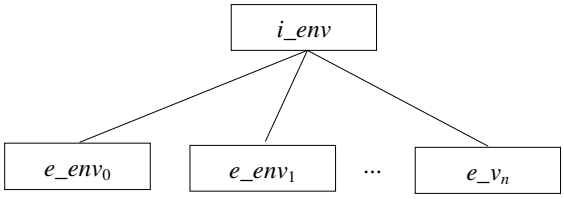


Figure 9. Geometrical environments hierarchy

The geometric calculator, with extensional and intensional definitions organized in this hierarchy, provides a flexible and economical environment for the definition of intelligent graphics applications, diagrammatic problem-solvers and theorem-provers, where transitions may be non-deterministic and the change information along the diagrammatic sequence may be required for further inference (e.g. for contrasting the geometrical properties of the same objects in different states of a design sequence or, more generally, for following the consequences of the change process).

4. Conclusions and some related work

The geometric calculator is a device that supports the definition of geometrical abstract-data types in a fully compositional fashion, and this device is useful for the representation and computation of graphical and diagrammatic information; in this respect, although geometrical and graphical abstract data types have been the object of a considerable number of studies, previous work has been concerned, for instance, with the definition of abstract types for design applications and methodologies (e.g. Lenart et. al, 1994) and for the definition and visualization of geometrical objects and relations through constraint satisfaction (e.g. Satoshi et. al, 1992; Kamada and Kawai, 1991). Abstract geometrical data-types have also been used for geometric modeling within the object-oriented paradigm; an interesting example in this line of work is the GEObject system (Alberti et. al, 1995), where geometrical knowledge is encapsulated in geometric classes and objects, and new classes or methods can be constructed dynamically; also messages exchanged between objects specify geometrical relations, and form a relationship network which captures mutual dependencies between objects; in this respect, primitives and constructive procedures of GEObject parallel the constructs that can be defined compositionally through our signature, and the graphs defined in that approach correspond to composite expressions in ours; however, unlike the declarative representation with a well-defined semantics of the geometric calculator, the representational object in the object-oriented approach is a dependency network, which can be much more difficult the maintain and modify, specially in complex scenarios where change information needs to be kept.

We conclude with a comment on the relation between abstract geometrical information and its graphical representation. In the geometric calculator we deal with geometrical abstract objects that are independent of the attributes of their graphical representations (e.g. the color of a symbol, whether a dot is filled, or a line is depicted dotted or continuous, its width, or even whether it is shown or hidden on the drawing), and also of their labels, which can also be thought of as attributes, and advocate for a strict modularity of the geometrical knowledge, its conceptual interpretation, the properties of its external graphical representation, and the interactive and rendering processes; for all these reasons, computations performed by the geometric calculator abstract over interface and applications issues; this position has also been advocated in recent work in relation to intelligent diagrammatic systems (Chandrasekaran et. al, 2004). However, the use of logical representations can be a source of some misunderstanding; for instance, our logical approach (as presented in Pineda, 1992) has been qualified as limited (in Alberti et. al, 1995), on the grounds that explicit predicates are used. However, the

use of logical expressions does not requires necessarily that predicates have to be typed in advance by the user: these can also be specified through graphical interaction, as has been the case in most of our prototype implementations. Furthermore, diagrammatic representations can be used in applications that may require no graphical interaction at all (e.g. some forms of diagrammatic problem-solving or diagrammatic representations for situated agents) and the form of presentation or visualizing of geometrical information depends on conventions and on the nature of applications; also, the conventions embedded in the definition and use of graphical interfaces are also application dependent; for all this, to keep apart geometrical knowledge from graphical representations and interactive issues is, in our perspective, a healthy design discipline.

5. References

Alberti, M. A., Bastioli, E & Marini, D. (1995). Towards object-oriented modeling of Euclidean geometry. Visual Computer 11: 378–389.

Bronowski, J. (1973). The Music of the Spheres, in The Ascent of Man. BBC Corporation, London.

Borning, A. (1981). The programming language aspects of ThingLab: A constraint-oriented laboratory, ACM Transactions in Programming Languages and Systems 3(4), pp. 353–387.

B. Chandrasekaran, Unmesh Kurup, Bonny Banerjee, John R. Josephson and Robert Winkler, "An Architecture for Problem Solving with Diagrams," in Diagrammatic Reasoning and Inference, Alan Blackwell, Kim Marriott and Atsushi Shomojima, Editors, Lecture Notes in Artificial Intelligence 2980, Berlin: Springer-Verlag, 2004, pp. 151-165.

Goguen, J. A., Thatcher, J. W. and Wagner, E. G. (1978). An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, in Current Trends in Programming Methodology. R. T. Yeh (ed.). Prentice-Hall.

Garza and L. A. Pineda (1998). Synthesis of Solid Models of Polyhedra from their Orthogonal Views using Logical Representations, *Expert Systems with Applications,* 14 (1). Pergamon, pp. 91–108.

Kamada, T. and Kawai, S. (1991). A General Framework for Visualizaing Abstract Objects and Relations. *ACM Transactions on Graphics*, Vol. 10, No. 1 : 1–39.

Lenart, M., Padawitz, P. & Pasztor, A. (1994). Formal Specification for Design Automaton, in Formal Design Methods for CAD (B-18), J. S. Gero and E. Tyugu (eds.) Elsevier Science B. V. (North-Holland).

Matsuoka, S. Takahashi, S. & Yonezawa, A (1992). A General Framework for Bidirectional Translation Between Abstract and Pictorical Data. *ACM Transactions on Information Systems*, 10 (4): 408–437.

Massé, A. (1994). Satisfacción de Restricciones por Referencia simbólica en dibujos geométricos, BSc. Thesis, ENEP Aragón, National Autonomous University of México (In Spanish).

Pineda, L. A. (1989). Graflog: a Theory of Semantics for Graphics with Applications to Human-Computer Interaction and CAD Systems. PhD Thesis, Centre for Cognitive Science, University of Edinburgh.

Pineda, L. A. (1992). Reference, Synthesis and Constraint Satisfaction, *Computer Graphics Forum*, 11 (3), pp. 334–344.

Pineda, L. A. 2004. Discovery and Proof of diagrammatic theorems: the case of the Theorem of Pythagoras, 1er. Congreso Internacional de Bioinformática, 13 y 14 de mayo de 2004, Palacio de las Convenciones de La Habana, Cuba (in Spanish).

Stiny, G. (1975). Pictorical and Formal Aspects of Shape and Shape Grammars, Basel, Birkhauser Verlag.

Weissman-Knight, T. (1981). Languages of designs: from known to new. *Environment and Planning B*, 8:213–238.