PLATO: A Language for Geometric Illustrations*

Boyko B. Bantchev

We present a descriptive language in which planimetric illustrations can be expressed. The language is suitable for specifying a wide class of figures that typically appear in textbooks on geometry. It is easy to use and the resulting specifications are small in volume and clear. An interpretive implementation of the language translates specifications into PostScript, and thus geometric figures can be reproduced independently or incorporated in a document in a number of document-processing environments.

Introduction

Desktop publishing becomes more and more an everyday activity for an increasingly large number of people. High-resolution printers are widely available, and page layout and design software that manipulates scalable fonts and makes use of picture inclusion is now commonplace.

An important part of the publishing process is the preparation of diagrams and figures that have to be incorporated in a document. A large number of general-purpose programs with sophisticated GUIs that aid in this kind of work are on the market or in the public domain.

Specific kinds of pictures, however, require specific means for their creation. Often, a well designed descriptive language is preferrable to using GUI programs. We believe that this is the case with drawing planimetric figures, and propose a language that eases the creation of such figures.

In this language, named PLATO, a planimetric illustration is drawn with a series of commands for construction of geometric objects, as well as for placing denotational material to accompany the drawing. The commands are very natural in use and the resulting description – a specification of the illustration – is expressive and terse. An interpreter translates this specification into PostScript – the internationally recognized as standard and massively used page description language. PostScript provides a device independent high quality representation of a picture that can be reproduced in virtually all operating environments or sent directly to a printer.

PLATO can be used by the authors or illustrators of textbooks and reference books in geometry, as well as teachers, students and anyone interested in the creation of such kind of drawings.

Why a language?

It is the popular practice nowadays that geometric illustrations (as well as many others) are created using some general-purpose GUI program, such as CorelDraw. Using a general-purpose program, however, is not satisfying for the following two major reasons. Firstly, such a program, because of its general orientedness, is not well suited for the particular

*The paper has been published in *Mathematics and Education in Mathematics, 2000.* Proc. 29^{th} Spring Conf. of the Union of Bulgarian Mathematicians, 2000, pp. 241-247

need: it lacks a number of desireful methods of construction that a geometer would think of as necessary ones; it is also far from being well equipped with tools for providing auxiliary denotations in the drawing, while in almost any drawing these are very important. As a result, the illustrations created using a general-purpose GUI program, are not of very good quality.

Secondly, it is a general principle that the more detail one needs to put in the specification of some formal object, the less likely it is that an interactive program is the proper tool for doing this; instead, an adequately designed (text) language should be used. For example, for serious programming, a visual programming language is not a match to a 'text' one. Or, in the world of document preparation, MS Word is very inefficient in terms of flexibility, to compare it with TEX. The same argument applies to our object of consideration.

So, a specifically designed language for describing geometric illustrations is the tool we should be seeking for.

This conclusion is supported by the successful application of PIC [2], IDEAL [3] and the more recent MetaPost [1]. These are descriptive languages for drawing illustrations of another sort, and they have proved to be adequate tools for the purpose for which they have been created.

Essentials of the PLATO approach to constructing figures

A figure description in PLATO is a sequence of commands, some of which construct geometric objects, others place accompanying material – such as text, arrowheads, angular marks and hatch marks, yet others change modes or serve similar purposes.

To construct a geometric object, one of the available construction commands must be used. In the act of construction, the object itself and/or its constituting parts may receive names, depending on the user's choice. They also may be drawn or not, also selectively, and independently of whether any naming takes place. Thus, a construction command has the following functions:

- it constructs a geometric object;
- it gives the constructed object a name (optionally);
- it draws the constructed object (this is the default but it may be suppressed);
- it (optionally) decomposes the object, so that (some of) its constituents
  - — receive names (optionally), and/or
  - — are themselves drawn (which is the default if decomposition takes place, but may be suppressed).

The basic kinds of geometric objects that PLATO can construct are point, line segment, line, circle, triangle and quadrilateral. For each type of object, there is a construction command named after that oject, which provides several construction methods, distinguishable among each other by the sequence of their input arguments. Thus, in the current version of the language there are two variants of each of the `point`, `segment` and `line` commands, three variants of the `circle` command, six variants of the `triangle` command, and one variant of the `quad` command.

Other construction commands serve to create objects 'in sub-ordination' to other ones (bisectors, medians, altitudes in a triangle, inscribed and circumscribed circles etc.), or to produce a copy of an already constructed object by means of a congruence transformation, or simply to construct a particular variant of a basic kind of object (trapeziums, parallelograms, rectangles and squares as specific kinds of quadrilaterals).

For each object type, there is a fixed set of *attribute names* that allow characteristic values of that object to be refered to. Some attributes are numeric, others are themselves

geometric objects.

For example, if `s` is a segment, `s.p1` is its start point, and `s.p1.x` is the x-coordinate of that point. Other attributes' names for a segment are `p2`, `len` and `slp` – its end point, length and slope correspondingly.

For each type of object there is a specific kind of *constructor* that distinguishes it from the others. A constructor is used for an ad hoc creation of objects of the corresponding type. It is a list of values that stand for some of the attributes for that object: by assigning each value to the attribute it corresponds to, an object is created. A constructor itself represents a value – the constructed geometric object, which allows constructors to be nested.

For example, a constructor of a point is a list of two values that represent its coordinates; a constructor of a line segment is a list of two values that represent the start and the end points of the segment. So, if `a` and `b` name two points, `[a,b]` would be a segment constructor. The attribute with the name `p1` of the resulting segment will have the value of `a`, while that with the name `p2` takes the value of `b`.

Similarly, a point can be created by the constructor `[x+5,y*2]`, where `x` and `y` are numeric values, and so `[[x+5,y*2],b]` is also a valid segment constructor.

Constructors are not meant to be used for direct construction of geometric objects. Rather, they serve as input arguments to PLATO commands – those that construct objects or others. As an example, consider the command

```
segment([x+5,y*2],b) -> p
```

which constructs a segment by specifying an (ad hoc created) start point `[x+5,y*2]` and a (named) end point `b`.

The general form of a construction command is

name`(`*arguments*`) -> `*pattern*

where *pattern* is another important concept in PLATO. A pattern must correspond to the type of object that is being constructed with the particular command. The latter, however, does not play a rôle here, only the type of the object that it constructs.

The pattern has a similar list structure to that of the constructor for the same type of object. This includes the possibility of nesting patterns in the same way that constructors are nested. The following differences, however, distinguish a pattern from a constructor.

- All the elements within a pattern are *names*, not values.
- A pattern, or its sub-pattern, or both, may themselves have names. If so, the name immediately precedes the pattern.
  Also, the name alone can be specified, with no (sub-)pattern at all; i. e., either the pattern or the name may be omitted, but not both.
- Wherever a name may appear in the pattern or its sub-patterns, the *dummy name* `.` (a dot) may appear in its place.
- As a result of the construction, each name in the pattern and in its sub-patterns *receives the value* of a corresponding attribute of the constructed object or of its sub-objects (unlike constructors, where the elements *transmit* values).
  In this respect, a construction command may be viewed as a multiple-assignment action. Dummy-named elements do not participate in the assignment.
- By default, the constructed object is drawn, unless a minus sign (`-`) precedes its name or the pattern on the right side of the construction command. The same applies to all sub-patterns or the corresponding names: for each sub-pattern, the corresponding sub-object is drawn by default, or this is suppressed by inserting a `-`. The drawing/suppressing activity is independent of the use of dummies for naming.

In summary, the use of patterns amounts to the following. A pattern serves to decompose a geometric object value into parts, or to decompose those parts, etc. Decomposition enables name-value bindings to occur, as well as drawing sub-parts to take place. Wherever just a name suffices, a sub-patterning is not needed. Wherever a binding or drawing a sub-object is not desired, it can be suppressed.

To extend the above segment construction example by use of patterns, consider

```
segment([x+5,y*2],b) -> p[A,[.,y1]]
```

where the segment itself receives the name `p`, its start point then is named `A`, the y-coordinate of the end point is named `y1`, and both the segment and the two points are drawn; also consider

```
segment([x+5,y*2],b) -> -[-A[x1,.],[.,y1]]
```

where the start point now is not drawn, but is both named and decomposed, so the name `x1` is bound to that point's x-coordinate. The constructed segment is neither named nor drawn.

Other features

PLATO has a rich set of construction commands that incorporate the power of planimetric construction methods most likely to suffice in a number of illustration applications. Clearly, though, however rich that set is, not all possible drawings will be directly obtainable by use of construction commands.

Therefore, PLATO provides the possibility of doing arithmetic and some special functions' computations and storing the result by means of binding a name to it. The so obtained values can then participate in other computations, or they can feed a construction directly by supplying the required input for it. The use of numerical values, readily available as the attributes of the already constructed geometric objects, is also important here.

The command `produce` in the form

`produce(`*arithmetic expression*`) ->` *name*

binds tne specified name to the computed value from the left, and is thus analogous to an assignment statement in a programming language.

Another form of `produce`, namely

`produce(`*object name*`) ->` *pattern*

may be used with an *already constructed object* and a proper pattern for it to obtain the side effects (name binding, drawing) that otherwise may have been obtained at the time the object had been constructed.

Such a delayed treatment is always possible with no loss of generality, provided that the object has a name by which it can be referred to. This 'trick' is sometimes helpful for reasons of dealing with visualization subtleties (planned effects of putting one part of the resulting image on top of another; see the example at the end of this paper).

PLATO also has a rich set of commands that provide auxiliary (purely denotational) elements in the drawing. These include, besides placement of text with various fonts and sub-/superscripts, as well as with Greek letters and other special characters, the specification and placement of the following:

- arrowhead marks (in six possible styles)
- hatch marks (for line segments, in four possible styles)
- angular marks (in eight possible styles)

- hatch-filling a contour (in three possible 'basic' styles, with variations of each one)

In addition, commands exist for selecting a colour, a line style and thickness, and a clipping region. All these can be varied as necessary within the same drawing.

The set of commands for drawing auxiliary elements is considered very significant for ensuring the overall quality of a geometric illustration. It has been carefully designed in order to make it possible to observe a variety of denotational styles and standards.

An example

Here follows an example of how PLATO can be used. Fig. 1 shows a description, the output of which can be seen on fig. 2. The line numbers in fig. 1 are not part of the commands but are only given for reference.

```
 1: limits(0,0,50,70)
 2: triangle(20,d:130,14,[30,30],0) - > -t[-A,-B,-C]
 3: altitude(t,A) -> -p[-.,-A1]
 4: altitude(t,B) -> -q[-.,-B1]
 5: intersection(p,q) -> -H
 6: segment(A,H) -> -p
 7: segment(B,H) -> -q
 8: colour(gray);  linestyle(_,T)
 9: circumscribed(A,B,A1) -> k1
10: circumscribed(H,C,A1) -> k2
11: colour(black);  linestyle(_,N)
12: anglemark(A,A1,C,*))
13: anglemark(C,B1,B,*))
14: produce(t) -> .;  produce(p) -> .; produce(q) -> .
15: linestyle(-,N);  segment(C,A1) -> .;  segment(C,B1) -> .
16: pointmark(o)
17: produce(A) -> .;  produce(B) -> .;  produce(C) -> .
18: letterstyle("Times-Italic",4)
19: label("A",A,WSW,6)
20: label("B",B,SE,3)
21: label("C",C,NW,3)
22: label("A_{1}",A1,W,8)
23: label("B_{1}",B1,ESE,4)
24: label("H",H,NW,3)
25: label("&{kappa}_{1}",k1.cen,SE,k1.rad+4)
26: label("&{kappa}_{2}",k2.cen,NE,k2.rad+2)
```

Fig. 1

Here is a short explanation of how the description works. Line 1 defines a rectangular area that will contain the entire drawing. Line 2 constructs a triangle, two sides of which are 20 mm and 14 mm long, correspondingly, with an angle of 130° between them. One vertex of the triangle has coordinates (30,30).

Once the triangle `t` is created, its three vertices are obtainable by the names `t.A`, `t.B` and `t.C` (`A`, `B` and `C` are attribute names for a triangle), but, for convenience, we introduce *direct names* to access those vertices: `A` for `t.A` etc.

The two altitudes `p` and `q` are found to intersect at point `H`. Note that `p` and `q` first get bound in lines 3,4 (AA_1 and BB_1 on fig. 2), and later (lines 6,7) are bound again to AH and BH.

The circles κ_1 and κ_2 are displayed in gray colour, using a thin pen. They are drawn first (although constructed last!), so as not to 'erase' any other part of the drawing. Similarly, the (empty) circle pointmarks at points A, B and C are drawn last, so that the marks remain clear.

At the very end (lines 18-26) the text denotations are placed. Each invocation of the command `label` takes a labeling string, a reference point, a direction (such as `N` for North, `SE` for South-East etc.) and a distance from the reference point in that direction to place the label.

It is important to make difference between the identifiers that are used to name objects within the description, and the labels placed in the drawing. Although they may look the same (this is intentionally done here for convenience), they are nevertheless independent from each other.

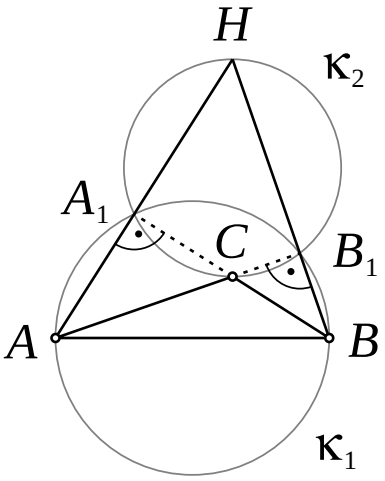

Fig. 2

References

[1] Hobby J.D. *A user's manual for MetaPost.* Bell Labs, Murray Hill, N.J. (1998)

[2] Kernighan B.W. *PIC – a language for typesetting graphics.* Software—Practice and Experience, Vol.12 (1982), No.1, 1-21.

[3] Van Wyk Ch.J. *A graphics typesetting language.* Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, 1981.