The Visual Computer

Applications of Euclidean constructions to computer graphics

Norma Fuller and Przemyslaw Prusinkiewicz

Department of Computer Science,
University of Regina,
Regina, Saskatchewan, S4S 0A2 Canada

This paper presents an interactive graphics system called L.E.G.O. The purpose of L.E.G.O. is to model two- and three-dimensional objects using Euclidean geometry constructions. L.E.G.O. has a layered structure which makes it convenient to use, both for the experienced programmer and the novice. The programs may be written in a compiled language (C), written interactively in an interpreted language (LISP) or developed using a graphical interface in a multiple-window environment. Applications of L.E.G.O. include computer-assisted instruction of geometry and computer graphics, geometric modeling, and kinematic analysis. The use of imperative constructions and the powerful interface based on the idea of graphical programming are the most distinctive features of the system.

Key words: Constraint-based graphics systems – Geometric constructions – Graphical programming – Iconic interfaces – Computer-assisted instruction

1 Introduction

Computer-assisted instruction of geometry has been largely influenced by the LOGO project (Abelson and diSessa 1980; Papert 1980). However, its central concept, turtle geometry, makes LOGO of little use in teaching classical Euclidean ideas. Consequently, with educational applications in mind, we have created a computer graphics system called L.E.G.O., based on the Euclidean notions and operations. The educational aspects of L.E.G.O., a survey of applications and the visual interface are described in previous papers by Fuller et al. (1985), and Fuller and Prusinkiewicz (1986, 1988). In this paper we focus on the layered structural design of the L.E.G.O. system. Because of this design, L.E.G.O. meets the needs of both the experienced programmer and the novice.

The L.E.G.O. graphics system was created using the layered approach (Fig. 1). The bottom layer in the current implementation is the IRIS Graphics Library (User's Guide for IRIS 1986). Geometric objects in Level 0 are expressed in Cartesian coordinates, for example, circ (500.0, 400.0, 100.0). Level 1, called C-LEGO, is a library of C functions which allow the user to specify objects in terms of Euclidean geometry, for example, line (A, B). Programs using C-LEGO functions must be compiled. Level 2 or LISP-LEGO is a LISP interface for C-LEGO. Programs can be built interactively statement by statement. Functions can be defined interactively and used at the same session without compiling. The user of this layer is not required to define objects prior to their use. Finally, the visual interface in Level 3 allows for building LISP-LEGO programs using graphical operations based on the direct manipulation approach.

The paper is organized as follows. Section 2 introduces the concept of construction-based modeling. In Sect. 3 constraint-based systems known from the literature are surveyed and contrasted with L.E.G.O. The essential aspects of L.E.G.O. are described in the next three sections, devoted to C-

Level 3:	Visual interface
Level 2:	LISP-LEGO
Level 1:	C-LEGO
Level 0:	Graphics library

Fig. 1. The layered structure of the L.E.G.O. graphics system

© Springer-Verlag 1989

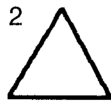


Fig. 2. An object

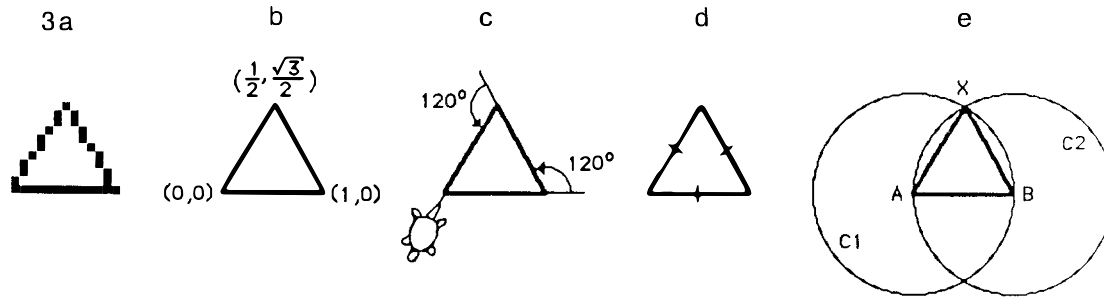


Fig. 3. Some interpretations of the object from Fig. 2

LEGO, LISP-LEGO and the graphics interface respectively. Selected applications of L.E.G.O. are outlined in Sect. 7. Finally, in Sect. 8 we summarize the obtained results.

C-LEGO is written in C. LISP-LEGO and the visual interface are written in Franz LISP (Foderaro 1979; Wilensky 1984). Both interface styles can be used concurrently. The version of L.E.G.O. described in this paper runs on an IRIS 3130 workstation. The mex window management system (User's Guide for IRIS 1986) provides the rudiments for man-machine interaction.

2 Construction-based modeling

Modeling using geometric constructions is distinctively different from other modeling techniques. This difference can be best presented by referring to an example. Thus, consider the object shown in Fig. 2. Some methods for describing it are illustrated in Fig. 3, and can be expressed as follows:

(a) Fig. 2 is a set of pixels.

(b) Fig. 2 is a plot of three lines, with the Cartesian coordinates of the endpoints equal to $(0,0)$, $(1,0)$ and $\left(\frac{1}{2}, \frac{\sqrt{3}}{2}\right)$.

(c) Fig. 2 is the trace of a turtle moving forward a unit distance, turning left by 120°, moving forward a unit distance, turning left by 120°, and moving forward a unit distance again.

(d) Fig. 2 is an equilateral triangle.

(e) Fig. 2 is the result of the following construction:
- Given line AB, draw circles $C1$ and $C2$ with radius AB and centers A and B, respectively.
- Intersect circles $C1$ and $C2$. Denote a point of intersection by X.
- Draw lines AX and BX.

Descriptions of type (a) and (b) are widely used in computer graphics. In case (a) a picture is thought of as a bitmap, i.e., a set of pixels defined in a system of coordinates. In case (b) the graphics primitives are not limited to pixels; they include lines and, possibly, also other simple figures, such as circles, ellipses and filled polygons. These primitives are defined with respect to a system of coordinates. A description of type (c), made popular by LOGO (Abelson and diSessa 1980; Papert 1980) differs from type (b) in that polar coordinates (relative to the current position of the turtle) are used instead of Cartesian coordinates. All three cases, however, reflect the analytic approach to geometry. Geometric figures are denied existence independent of a system of coordinates. They are conceptualized as plots of analytic relations.

Descriptions (d) and (e) belong to a totally different family. They describe Fig. 2 directly in geometric terms. In case (d) the figure is specified using a set of declarative constraints, or geometric *relations* between elements of the picture. These constraints impose equal length on all sides of the triangle. In case (e), the figure is described using a sequence of imperative constraints or Euclidean geometry

constructions (with a straightedge and a compass). This is the type of object description used in L.E.G.O. (Throughout this paper, the term "object" is used to denote both two-dimensional figures and three-dimensional objects.)
The idea of applying geometric constructions to object modeling is a relatively new one in computer graphics. This is rather surprising, given the fundamental role of constructions in Euclidean geometry. One system, other than L.E.G.O., which does use geometric constructions is a two-dimensional illustrator Gargoyle (Bier and Stone 1986). In contrast to L.E.G.O., however, the Gargoyle constructions are forgotten as soon as they are used, rather than becoming a part of the data structure. Consequently, the behavior of L.E.G.O. and Gargoyle is essentially different. Recently, a graphics system closer in spirit to L.E.G.O. has been proposed (Noma et al. 1988). Other constraint-based systems reported in the literature use declarative constraints.

3 Construction-based modeling and constraint-based systems

This section places L.E.G.O. in the context of previous work in the area of constraint-based graphics systems. Attention is focused on the techniques for constraint solving and the design of man-machine graphical interfaces.

3.1 Constraint solving techniques

Constraint-based graphics systems described in the literature (Borning 1981; Borning and Duisberg 1986; Knuth 1979; Nelsen 1985; Prusinkiewicz and Streibel 1986; Sutherland 1963; Van Wyk 1982) accept object definitions expressed in terms of geometric relations between object elements. These definitions are subsequently transformed into analytical descriptions of graphical primitives to be displayed or plotted. Many fundamental relations lead to nonlinear equations. For example, relations "lines $L1$ and $L2$ are congruent" or "lines $L1$ and $L2$ are perpendicular to each other" are represented by quadratic equations.
Various techniques for solving systems of nonlinear equations have been used to satisfy sets of constraints. The most straightforward approach relies on general-purpose numerical methods. For example, Sketchpad (Sutherland 1963) used the relaxation method, and Juno (Nelsen 1985) used the Newton-Raphson method. Unfortunately, these methods are adequate only when applied to relatively small sets of constraints (a few tens of equations). For larger sets it is difficult to find initial conditions which let the system of equations converge to the desired solution. In addition, numerical methods tend to be too slow for interactive applications.
The difficulties related to the use of general-purpose numerical methods can be overcome using two approaches. One technique, implemented in METAFONT (Knuth 1979) and IDEAL (Van Wyk 1982) restricts admissible constraints to those which can be expressed by linear equations. For example, specifying a distance between two arbitrary points is not allowed, since it leads to a quadratic equation. On the other hand, a vertical or horizontal displacement is expressed by a linear equation and therefore can be specified.
Another technique relies on dividing the set of constraints into smaller subsets, which can be solved in succession. The constraint solver may attempt to perform this partitioning automatically, using heuristic algorithms (Borning 1981). This leads to a purely declarative type of object description, since the user lists constraints without indicating how to solve them. Alternatively, the user may be required to explicitly partition the set of constraints, by describing the object as a hierarchy of components (Nelsen 1985; Prusinkiewicz and Streibel 1986; Sutherland 1963). By this means, he indicates the order in which the constraints should be solved. This introduces an imperative element to the object definition.
Construction-based object descriptions used in L.E.G.O. are purely imperative. They specify geometric objects by algorithms expressed in geometric terms. These algorithms never require solving more than two quadratic equations simultaneously. The solutions can always be found analytically without ever resorting to numerical methods.

3.2 Graphical interfaces

The objects modeled in constraint-based systems can be described textually, in an appropriate programming language (Knuth 1979; Prusinkiewicz and Streibel 1986; Van Wyk 1982) or visually, using a graphical interface. In the case of declarative

constraints, the interface is straightforward. The type of constraint is determined by selecting an icon or menu item, and the arguments are picked directly on the graphics screen. This type of interface originates from Sketchpad (Sutherland 1963). Unfortunately, it cannot be easily extended to geometric constructions, for they require a mechanism for defining complete algorithms. A systematic approach to designing a graphical interface for this purpose can be based on the idea of graphical programming, i.e., creating programs by manipulating graphical objects on the screen.
Two approaches to graphical programming can be distinguished (Glinert and Tanimoto 1984). In the explicit programming case, the user manipulates a graphical representation of the program, for example, a flowchart or a Nassi-Shneiderman diagram (Pong and Ng 1983). In the case of implicit programming, images on the screen represent an example of the solution of the given problem. The program is created, to some extent, as a "side effect" of constructing the object on the screen. Implicit graphical programming is the cornerstone of the graphical interface of L.E.G.O. It was also fundamental to the design of the graphical interface in Juno (Nelsen 1985).

4 The C-LEGO graphics library

This section describes Level 1 of the L.E.G.O. graphics system. Basic objects of Euclidean geometry (points, lines, circles, planes and spheres) are represented as predefined structures on which geometric operations can be performed. From the user's perspective, these structures are defined as follows:

```
struct primitive {
  char kind; /* point, line, etc. */
  union {
    struct point {
      float x;
      float y;
      float z;
    } point;
    struct line {
      struct point start;
      struct point end;
    } line;
    struct plane {
      struct coefficients {
        float a;
        float b;
        float c;
        float d;
      };
    } plane;
    struct circle {
      struct point center;
      float radius;
      struct plane circleplane;
    } circle;
    struct sphere {
      struct point center;
      float radius;
    } sphere;
  } type;
};
typedef struct primitive POINT;
typedef struct primitive LINE;
typedef struct primitive CIRCLE;
typedef struct primitive PLANE;
typedef struct primitive SPHERE;
typedef struct primitive PRIMITIVE;
```

The complete definition of the PRIMITIVE structure also contains information used internally for intersection calculations, for example, direction cosines of a line.
The following *modeling* functions are essential for developing two-dimensional constructions:

POINT point2 (*x*, *y*)
float *x*, *y*;
Returns a POINT structure given the coordinates *x* and *y*.

LINE line (*point*1, *point*2)
POINT *point*1, *point*2;
Returns a LINE structure given previously defined *point*1 and *point*2.

CIRCLE circle2 (*point*, *radius*)
POINT *point*;
float *radius*;
Returns a CIRCLE structure given a previously defined *point* and a *radius*.

PRIMITIVE intersection (*primitive*1, *primitive*2, *code*)
PRIMITIVE *primitive*1, *primitive*2;
int code;
Returns a PRIMITIVE structure for the intersection between previously defined *primitive*1 and *primitive*2. The *code* indicates which point

should be returned in the case where there are two points of intersection between the primitives.

The C-LEGO library also contains functions returning a numerical value, for example:

float distance (*point* 1, *point* 2)
POINT *point* 1, *point* 2;
Returns the distance between previously defined *point* 1 and *point* 2.

float length (*line*)
LINE *line*;
Returns the length of previously defined *line*.

The modeling functions do not display the primitives. The following functions can be used to produce the graphical output:

display (*primitive*)
PRIMITIVE *primitive*;
Displays any previously defined *primitive*.

display_filled (*circle*)
CIRCLE *circle*;
Displays a filled *circle*.

display_label (*primitive*, *string*)
PRIMITIVE *primitive*;
char *string* [ ];
Displays a *string* of characters at a place appropriate for the defined *primitive*.

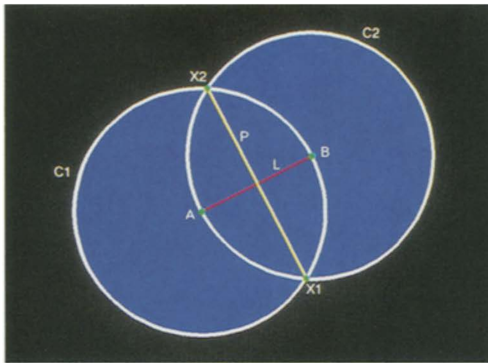
In practice, the C-LEGO functions are often interleaved with direct calls to the underlying graphics library (Level 0 of L.E.G.O.), for example, to change the current color or line width.
In order to illustrate key features of C-LEGO, let us consider some simple programs. The first program creates line *L* defined by points *A* and *B*, and bisects *L* with a line *P* perpendicular to *L* (Fig. 4).

Program 1.

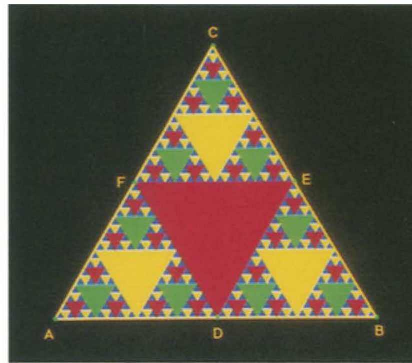
```
main ( )
{
/* Definition of variables */
POINT A, B, X1, X2;
LINE L, P;
CIRCLE C1, C2;
float r;

/* Modeling of object */
A = point2 (400.0, 370.0);
B = point2 (600.0, 470.0);
L = line (A, B);
r = length (L);
C1 = circle2 (A, r);
C2 = circle2 (B, r);
X1 = intersection (C1, C2, FIRST);
X2 = intersection (C1, C2, SECOND);
P = line (X1, X2);

/* Display of object */
... /* initialize machine's graphics system */
... /* clear background */
color (BLUE); display_filled (C1);
display_filled (C2);
color (WHITE); display (C1); display (C2);
color (RED); display (L);
color (YELLOW); display (P);
color (GREEN); display (A); display (B);
display (X1); display (X2);

/* Display of labels */
color (WHITE);
displaylabel (C1, "C1"); displaylabel (C2, "C2");
displaylabel (A, "A");
displaylabel (B, "B"); displaylabel (L, "L");
displaylabel (P, "P");
displaylabel (X1, "X1"); displaylabel (X2, "X2");

... /* terminate machine's graphics system */
}
```

The idea of reducing new problems to the problems previously solved is essential to the Euclidean approach to geometry. For example, once the construction for bisecting a line had been described in Proposition 10, Book 1 of Euclid's *Elements*, (Todhunter 1933), Euclid referred to it in Proposition 12, Book 1, by a simple statement "bisect *FG* at *H*" without further explanations. In programming languages a similar reduction is made possible by the mechanism of function definition. For example, function **bisect** can be defined as follows:

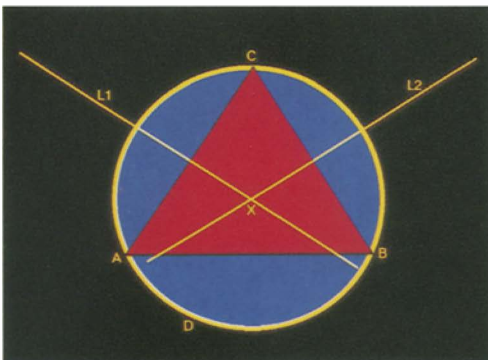
```
LINE bisect (A, B)
POINT A, B;
{
  POINT X1, X2;
  LINE L, P;
  CIRCLE C1, C2;
  float r;

  ... /* lines 11 thru 17 from Program 1 */
  return (P);
}
```



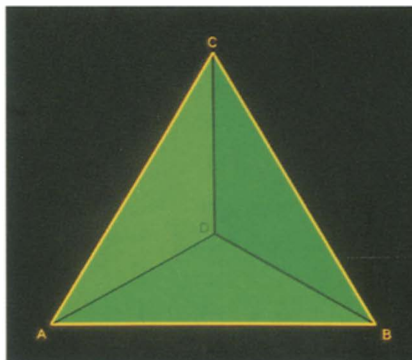
4



6



5



7

Fig. 4. Bisecting a line in C-LEGO

Fig. 5. Construction of the circumcircle of a triangle

Fig. 6. The Sierpiński gasket

Fig. 7. Construction of a regular tetrahedron

This function can then be used, for example, to construct the circumcircle of a given triangle *ABC* (Fig. 5).

Program 2.

```
main ( )
{
.../* definition of variables */
.../* assign coordinates to A, B, C */
   L1 = bisect (A, C);
   L2 = bisect (B, C);
   X = intersection (L1, L2, ANY);
   D = circle2 (X, distance (X, A));
.../* display the primitives desired */
}
```

The mechanism of recursion is also useful while defining geometric objects. For example, consider a user-defined function **midtriangle** (A, B, C, DPtr, EPtr, FPtr). Given the vertices of a triangle *ABC*, **midtriangle** returns the midpoints of the edges: *D*, *E*, *F*; and draws the triangle *DEF*. Using **midtriangle**, the Sierpiński gasket (Mandelbrot 1982) (Fig. 6) can be defined as follows:

Program 3.

```
gasket (A, B, C)
POINT A, B, C;
{
  POINT D, E, F;
  midtriangle (A, B, C, &D, &E, &F);
  if (distance (A, B) > 40.0) {
    gasket (A, D, F);
    gasket (B, E, D);
    gasket (C, F, E);
  }
}
```

C-LEGO primitives are all handled internally as three-dimensional. The functions **point2** and **circle2** are special cases of the three-dimensional **point** and **circle** and create the objects in the z = 0 plane. The expanded set of C-LEGO functions for three-dimensional constructions includes:

POINT point (*x*, *y*, *z*)
float *x*, *y*, *z*;
Returns a POINT structure given coordinates *x*, *y* and *z*.

PLANE plane (*point* 1, *point* 2, *point* 3)
POINT *point* 1, *point* 2, *point* 3;
Returns a PLANE structure given three previously defined non-collinear points: *point* 1, *point* 2 and *point* 3.

CIRCLE circle (*center*, *radius*, *plane*)
POINT *center*;
PLANE *plane*;
float *radius*;
Returns a CIRCLE structure given a previously defined *plane*, *center* point and *radius*.

SPHERE sphere (*center*, *radius*)
POINT *center*;
float *radius*;
Returns a SPHERE structure given a previously defined *center* and *radius*.

An example of a three-dimensional construction is given by Program 4. It creates a regular tetrahedron, given an equilateral triangle *ABC* (Fig. 7).

Program 4.

```
main ( )
{
   .../* definition of variables */
   .../* assign coordinates to A, B, C */
   r = distance (A, B);
   S1 = sphere (A, r);
   S2 = sphere (B, r);
   S3 = sphere (C, r);
   C1 = intersection (S1, S2, ANY);
   C2 = intersection (S2, S3, ANY);
   D = intersection (C1, C2, FIRST);
   L1 = line (D, A);
   L2 = line (D, B);
   L3 = line (D, C);
   .../* display desired primitives */
}
```

5 The LISP-LEGO language

Level 2 of the L.E.G.O. graphics system is a LISP textual interface to C-LEGO. LISP-LEGO offers a user the convenience of the interpreted language Franz LISP (Foderaro 1979; Wilensky 1984). Two distinctive features of LISP-LEGO are:

– Functions can be built interactively and used in the same session without compiling.
– Geometric primitives need not be defined prior to their use.

The philosophy of LISP-LEGO is to provide the user with immediate visual feedback. Consequently, in contrast to C-LEGO, the display of geometric primitives is incorporated into the modeling functions. LISP-LEGO is an extension to LISP rather than a library of LISP functions since it maintains its own symbol table which contains information about the graphical primitives.
Modeling functions of LISP-LEGO are similar to those of C-LEGO except that they use the syntax of LISP, for example:

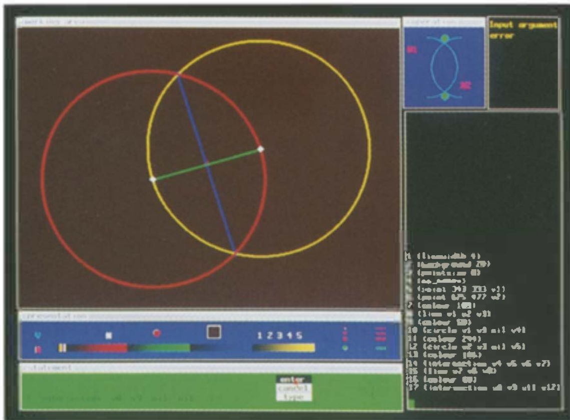
(**point** *x y* [*z*] *new_name*)
(**line** *point* 1 *point* 2 *new_name*)
(**circle** *center radius* [*plane*] *new_name*)
(**plane** *point* 1 *point* 2 *point* 3 *new_name*)
(**sphere** *center radius new_name*)
(**intersection** *primitive* 1 *primitive* 2 *new_name* 1 [*new_name* 2])

The radius used to define the circle and sphere in LISP-LEGO is a LINE structure rather than a numerical value as in C-LEGO. This is motivated by the needs of the visual interface in which the arguments are picked using the mouse (Sect. 6).
As an example of geometric construction specification in LISP-LEGO, consider the following program:

Program 5.

```
(color WHITE)
(point 400 370 A)
(point 600 470 B)
(color GREEN)
(line A B L)
(color RED)
(circle A L C1)
(color YELLOW)
(circle B L C2)
(intersection C1 C2 X1 X2)
(color BLUE)
(line X1 X2 P)
```

The resulting figure is shown in the "working area" window in Fig. 8.
Within the LISP-LEGO symbol table, two- and



8

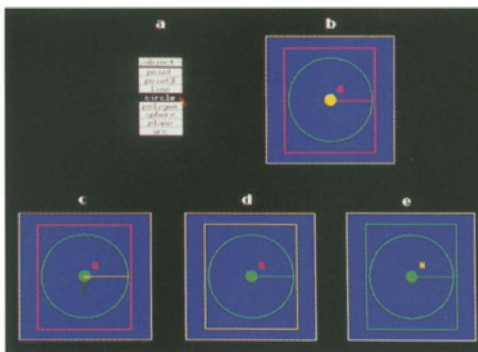
Fig. 8. Example of the screen in graphical programming mode

Fig. 9. Menu and icon states while creating a circle

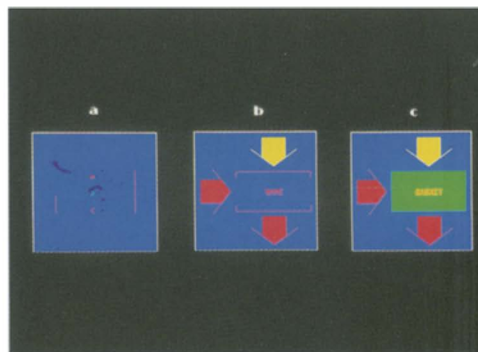
Fig. 10. Examples of icons used in the graphics interface

Fig. 11. Examples of icons used in the graphics interface

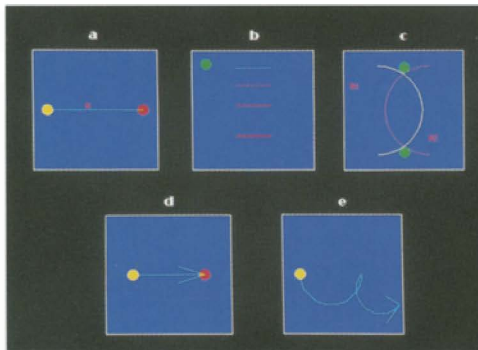
Fig. 12. Representation of a plane clipped by a box. The image belongs to a series of educational slides developed using L.E.G.O. to illustrate viewing in three dimensions



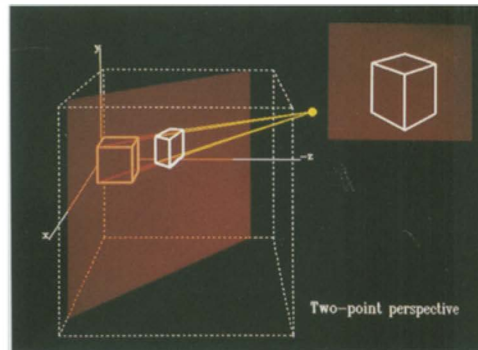
9



11



10



12

three-dimensional objects are represented in the same way. In order to place two-dimensional primitives (specifically, circles and points given by only two coordinates) in three dimensional space, the notion of *current plane* is introduced. By default it is the $z=0$ plane. The current plane can be changed using the **current_plane** function. The arguments to this function are three non-collinear points, or two parallel or intersecting lines. The current plane can be also forced to align with any previously specified plane.

In total, LISP-LEGO has approximately 100 predefined functions (Fuller 1985), which can be grouped into nine classes.

1. **Object definition functions** are used to create graphics objects: points, lines, circles, planes and spheres. Functions **point**, **line**, **sphere**, **intersection**, etc. belong to this category. The object definition functions are the fundamental tools for modeling geometric objects.

2. **Query functions** provide information about graphical primitives. Two subclasses can be distinguished:
   – *Functions which return a numerical value* (e.g. coordinate of a point, distance between points, length of a line). They are used primarily in conditional statements.
   – *Functions which return a graphic primitive* (e.g. endpoint of a line, center of a sphere, plane containing a circle). They are useful when arguments other than points are passed to functions.

3. **Drawing functions** are used to display text and simple figures such as arcs of circles and filled polygons. These figures are not considered as primitives and, consequently, cannot be passed as arguments to the function **intersection**.

4. **Presentation definition functions** are used to control the appearance of graphical objects on the screen. Examples of controlled features are listed below:
   – *Visibility of primitives.* Auxiliary construction lines may be removed from the final picture.
   – *Display of primitive names.* In some applications, such as the presentation of geometric constructions for educational use, primitives should be labeled. In other cases, such as the modeling of realistic scenes, display of names should be suppressed.
   – *Color, width and style of lines.*
   – *Color, size and type of fonts.*

5. **Function definition functions** allow the user to define a construction as a function with geometric primitives as input and output.

6. **Viewing functions** are used to define parameters of the projection, rotate objects in space, etc.

7. **Interaction supporting functions** make it possible to remove or modify previously defined primitives. These functions are particularly useful when developing constructions interactively, since each statement entered to the system is immediately executed and it cannot be subsequently altered by editing.

8. **System functions** are used for file manipulation (such as function loading), to configure the system for a particular type of graphics output device (such as a plotter), etc.

9. **Debugging functions** provide information about primitives stored in the symbol table, actual viewing parameters, etc.

LISP-LEGO functions in classes 1, 2 and 3 have corresponding functions in C-LEGO. Most of the functions in classes 4 and 6 have related functions in Level 0.

6 Visual programming

6.1 General principles

The visual interface is built on top of the textual interface. This means that it is used to build a textual LISP-LEGO program statement by statement. However, it supersedes typing by more intuitive operations. This is achieved using several graphics input techniques. Thus, function names are selected from a menu. Coordinates are located using the mouse. Graphical objects (passed as arguments to functions) are identified by picking using the mouse rather than by referring to them by name. (Usually the user need not even know object names.) The appearance of objects (color, line width, point size) and the viewing parameters are controlled by manipulating the appropriate icons. In order to make the visual interface complete (i.e. superseding *all* functions of the textural interface) a method for defining functions and condition statements in the graphics mode is also available. Using graphical operations we build a LISP-LEGO statement, then execute and append it to the program.

6.2 Implementation

The screen is partitioned into windows (Fig. 8). The actual construction is created in a window called the *working area.* The second window displays the icon associated with the current geometric *operation.* The third graphics window contains icons which change the *presentation* of objects such as the color, line width, etc. Three textual windows contain system and error *messages,* the textual form of the LISP-LEGO *statement* being built and the listing of the *program.*
In order to build a LISP-LEGO statement, the user first selects a function name from a pop-up menu. The icon associated with this choice will appear in the operation window and a skeleton of the statement will appear in the statement window.

The fields of the statement are filled by selecting primitives in the working area, by locating a position in the working area, by selecting parts of the icon in the operation window or by selecting parts of the icons in the presentation window. The statement window is updated as each field is filled. Once a complete statement has been built, it can be executed by selecting the **enter** item from the pop-up menu. If the statement is successfully executed it is appended to the program built and displayed in the program window. Some statements are automatically executed and added to the program by selecting an item in the presentation window (e.g. changing the current drawing color). The **cancel** item aborts the definition of the statement.

Let us consider the building of a LISP-LEGO statement which draws a circle. It is assumed that the point which will become the center of the circle and a line equal to its radius are already on the screen. For this example, let us assume that their names are *A* and *B* respectively. The user starts the construction by selecting the word **circle** in a pop-up menu (Fig. 9a). "(circle nil nil nil nil)" appears in the statement window. This indicates that there are four arguments to the circle function. The circle icon is drawn in the operation window and its central point changes color from red to yellow (Fig. 9b). This informs the user that the system expects him to pick (by pointing with the cursor) the point to become the circle center. After this has been done, "(circle A nil nil nil)" appears in the statement window and the colors of the icon change to indicate that a center has now been established and that a line defining the circle radius should be picked next (Fig. 9c). The primitive *B* is selected and "(circle A B nil nil)" in the statement window indicates that the radius will be the length of line *B*. At this point the icon no longer prompts the user for there is now sufficient information in order to draw a circle.

The third argument to the circle function is the plane on which it lies. If this is not given, by default it is the **current_plane**. The fourth argument is the name that will be assigned to the circle. If the user does not specify the name, it will be automatically assigned by the system. In order for the user to enter the name of the plane, he clicks into the square shown in the icon. The square will change color to yellow indicating that the plane can now be selected (Fig. 9d). Similarly the name of the circle can be entered by selecting the "N" in the circle icon and typing the name (Fig. 9e).

The use can alter the default sequence of actions by clicking into an appropriate element of the icon. For example, by clicking into the line depicting the circle radius the user brings the icon to the state shown in Fig. 9c. This allows for defining the radius before choosing the center of the circle, or for redefining the radius in the case of a mistake.

In principle, icons represent functions of LISP-LEGO. However, this is not a one-to-one correspondence and calls to different LISP-LEGO functions may result from manipulating the same icon. In the terminology of Lodding (1983), most icons are representational. They depict an instance of the general class of objects they refer to. Thus, the icon show in Fig. 10a is used to create a line given its endpoints, and the icon shown in Fig. 10b is used to select line width. The representational icons become slightly less intuitive when they refer to large classes of objects. The icon shown in Fig. 10c presents the points of intersection between two circles, but it can also be applied to intersect other primitives, such as a sphere and a plane. The icons used to specify actions such as moving a point (Fig. 10d) or removing a primitive (Fig. 10e) are abstract. Their design emphasizes an action to be performed rather than a concrete graphical object.

Three of the icons are less intuitive and require further explanations. The icon shown in Fig. 11a is used to define conditional statements. The only test available in the graphical programming mode compares the lengths of two lines. The user picks these lines and specifies, in succession, the alternative constructions to be performed depending on the result of the comparison. After the definition of the conditional statement is completed, the construction displayed on the screen corresponds to the actual relation between the compared lines.

The above method for defining conditional statements may seem slightly strange. What is the point of comparing lines which are already on the screen and the result of the comparison is known in advance? The answer to this question is that the result of the comparison may change. Specifically, this may happen if the starting points of the construction are moved on the screen, or if the conditional statement is defined within a function which can be called with different arguments.

In order to define functions, a section of the program must be selected as a function body. One way of doing this in the graphical programming mode is to rerun the program and click into elements ⊓ and ⊔ in the function-definition icon

(Fig. 11 b) when the relevant portion of the construction starts and terminates. Items in a pop-up menu make it possible to step through the program (forwards and backwards), thus facilitating this task. Alternatively, the user can enter the numbers of the first and the last lines of the program section to become the function body. Referring to the program in the textual form, while inconsistent with the graphical programming approach, is convenient for users familiar with LISP-LEGO. Newly created functions can be added to the menu of user-defined functions. They can be used as the predefined functions with the exception that they are all represented by the same generic icon (Fig. 11 c).

The name of the function being defined is entered from the keyboard. The list of its input primitives is specified by selecting the top arrow in the function-definition icon (Fig. 11 b) and picking the desired primitives in the working area of the screen. The list of primitives to be returned is formed in an analogous way, after selecting the bottom arrow in the icon. The left arrow can be optionally used to declare numerical parameters.

Two problems that arose in using the graphical interface to develop three-dimensional constructions were the location of three-dimensional points and the picking of planes. Firstly, at the present, three-dimensional points may be specified in two ways. The coordinates of the point may be typed by choosing the **type** option in the menu or the point may be located on the **current_plane**. The location selected by the cursor defines a line of points that are projected onto the screen at that location. The point of intersection between that line and the current_plane will be the three-dimensional point. Secondly, some way was needed to clip planes since if they were represented in their entirety they would generally cover the entire working area. They are intersected with the sides of a box and the resulting lines of intersection can be picked to select the plane (Fig. 12).

A user may prefer the graphical interface for some operations, and the textual interface (as described in Sect. 5) for others. Consequently, while building a LISP-LEGO program, both interface types can be used concurrently in a multiple-window environment provided by mex (User's Guide for IRIS 1986). Mixing graphical and textual operations within a single statement is possible.

LISP-LEGO programs can also be considered as text files and edited outside the L.E.G.O. system. This mode of operation is particularly useful when making substantial changes to an existing program. For the purpose of file editing, it is convenient to run a text editor concurrently with L.E.G.O., in a separate window. The modified program can be then quickly loaded into the L.E.G.O. system and rerun.

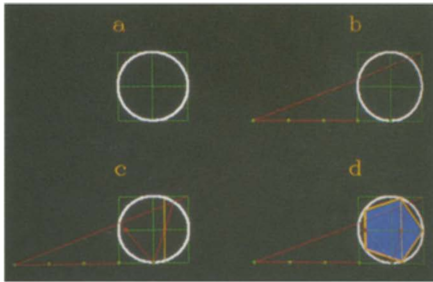
7 Applications of L.E.G.O.

This section presents selected applications of L.E.G.O.

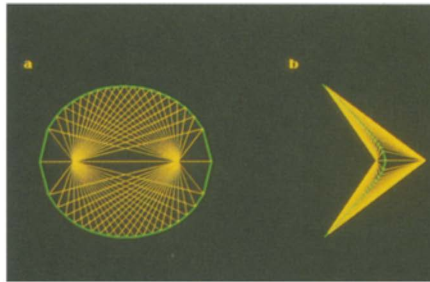
7.1 Computer-assisted instruction of Euclidean geometry

The fundamental concept of L.E.G.O., mimicry of constructions with straightedge and compass, makes the system suitable for teaching Euclidean geometry (Fuller et al. 1985). The educational applications of L.E.G.O. fall roughly in two categories: "the computer is a blackboard" and "the computer is a virtual laboratory". In the first case, L.E.G.O. is used by the instructor to prepare illustrations of geometric objects and constructions. Such illustrations are more precise and visually more attractive than those drafted on a traditional blackboard. Pictures created using L.E.G.O. can be captured using a camera and presented as slides, distributed to students in the form of plots or prints, or shown directly on the computer screen. Stepping through a LISP-LEGO program makes it possible to present a construction in progress. For example, Fig. 13 illustrates von Staudt's construction of a regular pentagon (Behnke et al. 1983).

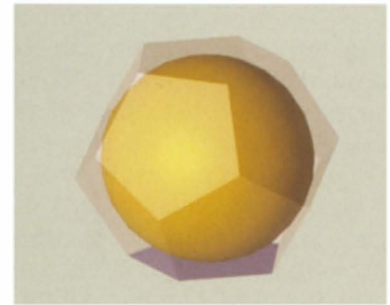
Use of L.E.G.O. as a virtual laboratory assumes interaction between the system and the students. Specifically, they can look at the L.E.G.O. objects from different angles and change data for constructions. Object manipulation reveals the general properties of the objects and constructions, and helps formulate them in the form of hypotheses. For example, moving vertices of a quadrangle (Fig. 14) brings to the student's attention that the figure created by connecting the midpoints of the edges of an arbitrary quadrangle is always a paral-



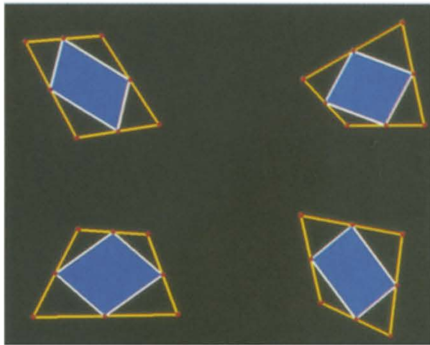
13



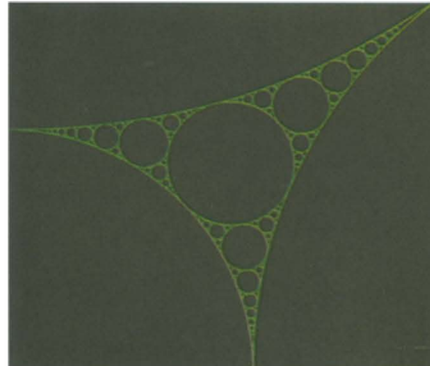
15



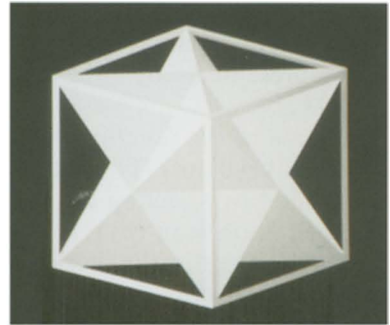
18



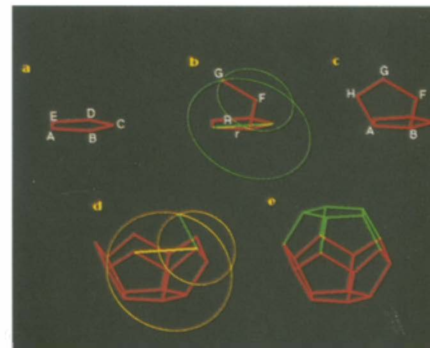
14



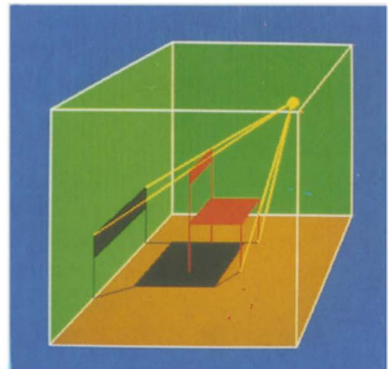
16



19



17



20

Fig. 13. Von Staudt's construction of a regular pentagon

Fig. 14. Manipulating an object reveals its general properties

Fig. 15. An ellipse and a hyperbole defined as loci

Fig. 16. The Appollonian gasket

Fig. 17. Construction of a dodecahedron

Fig. 18. A sphere inscribed in a dodecahedron

Fig. 19. A stellated octahedron

Fig. 20. Construction of a shadow

lelogram (Varignon's theorem – see Coxeter and Greitzer 1967).

The notion of a geometric locus is fundamental in geometry, yet many loci are difficult to visualize using traditional methods. An approximation of a locus which is neither a straight line nor a circle involves a repetitive construction of its subsequent points which is tedious if a "real" straightedge and compass are used. On the other hand repetitive constructions can be easily programmed in L.E.G.O. Figure 15 shows an ellipse and a hyperbole which were constructed using the following definitions:

– An ellipse is a locus of points A such that the sum of the distances of A from two fixed points, called foci, is constant.
– An hyperbole is a locus of points A such that the difference of the distances of A from two fixed foci is constant.

Another concept involving repetitive constructions is the recursive definition of geometric objects. A simple example of a recursive construction in L.E.G.O. was presented in Fig. 6. Figure 16 shows a more complex example – the Appollonian gasket (Mandelbrot 1982). This figure is obtained by recursively constructing the circle tangent to three

given circles using the method described by Coxeter and Greitzer (1967).
L.E.G.O. is particularly useful when illustrating three-dimensional objects and constructions. Figure 17 shows a dodecahedron and illustrates its construction. (a, b) Let r and R denote an edge and a diagonal of the regular pentagon $ABCDE$. Intersect three spheres with centers at points A, B, C and radii R, r and R, respectively, and call the point of intersection F. (c) Given three vertices A, B, F construct pentagon $ABFGH$. (d) Continue constructing pentagons given three vertices until the entire dodecahedron is formed (e). If the von Staudt's construction for a pentagon is defined as a function, pentagon $ABCDE$ in this example can be constructed using a single function call. Figure 18 shows a sphere inscribed in a dodecahedron.

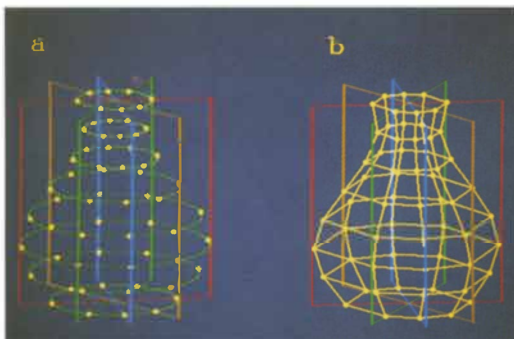
Figure 19 shows a mathematical model of a stellated octahedron. It is constructed by choosing four vertices of a cube to be the vertices of a regular tetrahedron. This can be done in two ways and the resulting two tetrahedra together form the model.
L.E.G.O. is also a convenient vehicle for illustrating concepts related to projections. Since the projection of a point onto a plane is defined by the intersection of the projector with the projection plane, it can be easily constructed in L.E.G.O. Figure 12 illustrates two-point perspective projection (Foley and Van Dam 1982). Figure 20 applies a similar approach to construct a shadow.

7.2 Geometric modeling

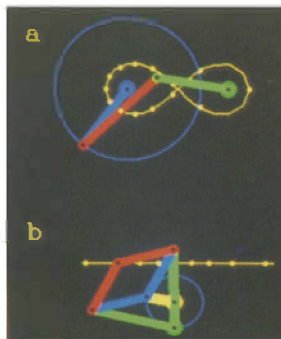
Repetitive (recursive or iterative) geometric constructions can be used to model curved surfaces. Figure 21 illustrates the L.E.G.O. construction of a polygon mesh of a vase. The vertices of the vase lie at the intersections of four vertical planes with a sequence of horizontal circles.

7.3 Modeling of mechanisms and kinematic analysis

Mechanisms consist of movable elements (links) connected together in kinematic pairs which put constraints on the motion of the links. The essential problem of kinematic analysis is to determine the relationship between the input and the output motion of a mechanism. This relationship can be very complex and difficult to grasp. Consequently, working models of mechanisms are often necessary to gain a full understanding of the motion (Hain 1967). Alternatively, mechanisms can be represented as computer models. The possibility of modeling mechanisms using constraint-based graphics systems was recognized by Sutherland and described as the most interesting application of Sketchpad (Sutherland 1963). Various types of mechanisms can also be modeled using L.E.G.O. They can be interactively manipulated by the user, or put in motion by a "virtual motor", i.e., a function which moves the input links without user intervention. As an example of a two-dimensional mechanism consider James Watt's linkage (Cundy and Rollet 1961) (Fig. 22a). If it is put in motion by rotating the left link, the midpoint of the middle link traces a Bernoulli's lemniscate. A "stroboscopic picture" of the linkage reveals that the velocity of the midpoint of the middle (red) link varies while the left (blue) link rotates at a constant speed (Fig. 23). Another mechanism, called Peaucelier's linkage, is shown in Figs. 22b and 24. It is interesting from the historical perspective, as it is the first exact solution to the straight-line motion problem. (This problem consists of converting a circular mo-



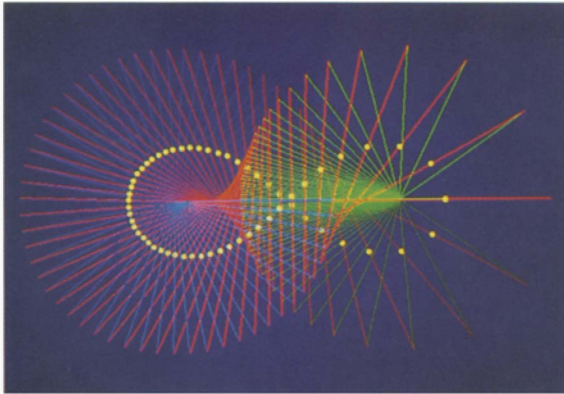
21



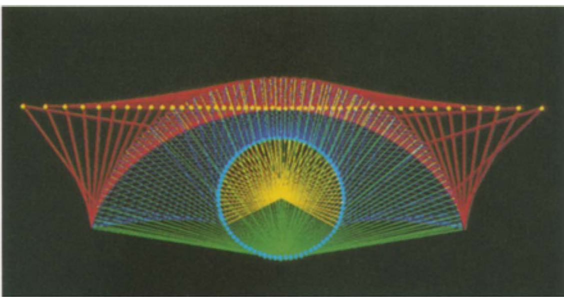
22

Fig. 21. Construction of the polygon mesh of a vase

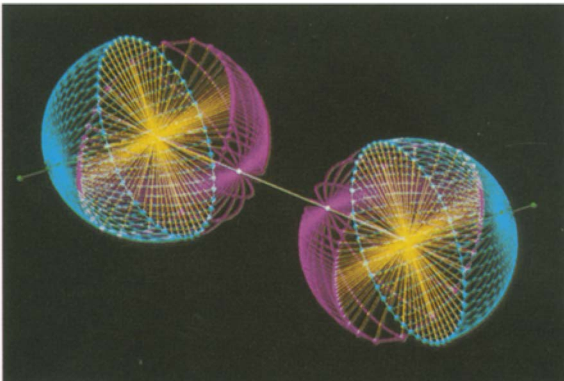
Fig. 22. Examples of linkages: (**a**) James Watt's linkage, (**b**) Peaucelier's linkage



23



24



25

Fig. 23. A stroboscopic view of James Watt's linkage

Fig. 24. A stroboscopic view of Peaucelier's linkage

Fig. 25. A stroboscopic view of two Hooke's joints

tion at the input into a linear motion at the output of the linkage (Cundy and Rollet 1961).)
As an example of a three-dimensional mechanism consider Hooke's joint (Cundy and Rollet 1961). It is used to connect two co-planar, non-parallel shafts. Each shaft has a semicircular forked end which is connected by pin-joints to a central cross member. When two Hooke's joints are connected by means of an intermediate shaft then the driven shaft can be made to revolve at the same speed as the driver at all instants provided that the forks at the ends of the intermediate shaft are co-planar and the other forks are equally inclined to the intermediate shaft. Figure 25 shows the stoboscopic view of such a mechanism.

8 Conclusions

L.E.G.O. is a modeling system based on geometric constructions. Objects are described in terms of Euclidean geometry. The system has a layered structure. C-LEGO is convenient to use for an experienced programmer, while the graphical interface makes the system useful for a novice. The graphical interface can be used concurrently with the textual interface (LISP-LEGO), and the user is allowed to change the type of interface even when specifying a single statement.
Construction-based modeling is ideally suited for some applications, such as the teaching of Euclidean geometry and the analysis of mechanisms. The construction-based approach can also be used in less obvious applications, such as the modeling of curved surfaces.
Since the beginning of 1985, various versions of L.E.G.O. have been available to computer graphics students at the University of Regina. They found the system very attractive, easy to use, and applicable to many practical problems. Although these opinions were not formally surveyed, they reinforce our conclusion that geometric constructions and implicit graphical programming provide a viable basis for an interactive computer graphics system.

Acknowledgement. This research was supported in part by grant No. A0324 and a scholarship from the Natural Sciences and Engineering Research Council of Canada.

References

Abelson H, diSessa A (1980) Turtle geometry: the computer as a medium for exploring mathematics. MIT Press, Cambridge

Behnke H, Bachmann F, Fladt K, Kunle H (eds) (1983) Fundamentals of Mathematics, vol. 2. Geometry. MIT Press, Cambridge

Bier A, Stone M (1986) Snap-dragging. Comput Graph 20(4):233–240

Borning A (1981) The programming language aspects of Thinglab, a constraint-oriented simulation laboratory. ACM Trans Program Lang Syst 3(4):353–387

Borning A, Duisberg R (1986) Constraint-based tools for building user interfaces. ACM Trans Graph 5(4):345–374
Coxeter HSM, Greitzer SL (1967) Geometry revisited. Random House, New York
Cundy HM, Rollet AP (1961) Mathematical models. Oxford Univ Press, London
Foderaro J (1979) The Franz LISP manual. Univ California, Berkeley
Foley JD, Van Dam A (1982) Fundamentals of interactive computer graphics. Addison-Wesley, Reading
Fuller N (1985) User's Guide to L.E.G.O. – Version 1.0. Tech Rep CS-85-19, Dep Comput Sci, Univ Regina
Fuller N, Prusinkiewicz P (1986) L.E.G.O. – An interactive graphics system for teaching geometry and computer graphics. Proc CIPS Edmonton '86, pp 75–84
Fuller N, Prusinkiewicz P (1988) Geometric modeling with Euclidean constructions. In: Magnenat-Thalmann N, Thalmann D (eds) New trends in computer graphics. Proc CG International '88, Springer, Berlin Heidelberg New York, pp 379–392
Fuller N, Prusinkiewicz P, Rambally G (1985) L.E.G.O. – An interactive computer graphics system for teaching geometry. Proc 4th World Conf Comput Educat, pp 359–364
Glinert EP, Tanimoto SL (1984) Pict: An interactive graphical programming environment. Computer 17(11):7–25
Hain K (1967) Applied kinematics. McGraw-Hill, New York
Knuth DE (1979) TEX and METAFONT. Digital Press and American Mathematical Society, Bedford
Lodding KN (1983) Iconic interfacing. IEEE Comput Graph Appl 3(2):11–20
Mandelbrot BB (1982) The fractal geometry of nature. WH Freeman, San Francisco
Nelsen G (1985) Juno, a constraint-based graphics system. Comput Graph 19(33):235–243
Noma T, Kunii TL, Kin N, Enomoto H, Aso E, Yamamoto T (1988) Drawing input through geometrical constructions: specification and applications. In: Magnenat-Thalmann N, Thalmann D (eds) New Trends in Computer Graphics. Proc CG International '88, Springer, Berlin Heidelberg New York, pp 403–415
Papert S (1980) Mindstroms: children, computers, and powerful ideas. Basic Books, New York
Pong MC, Ng N (1983) PIGS – A system for programming with interactive graphical support. Software Pract Exper 13(9):847–855
Prusinkiewicz P, Streibel D (1986) Constraint-based modeling of three-dimensional shapes. Proc Graph Interface '86 – Vision Interface '86, pp 158–163
Sutherland IE (1963) Sketchpad: a man-machine graphical communication system. In: 1963 Spring Joint Comput Conf. Reprinted in: Freeman H (ed) Interactive Computer Graphics. IEEE Comput Soc 1980, pp 1–19
Todhunter I (ed) (1933) Elements of Euclid. JM Dent & Sons, London
User's Guide for IRIS Graphics Programming (Version 3.0) (1986) Silicon Graphics, Mountain View, California
Van Wyk CJ (1982) A high-level language for specifying pictures. ACM Trans Graph 1(2):163–182
Wilensky R (1984) LISP craft. WW Norton, New York

NORMA FULLER is a Ph.D. candidate at the University of Regina. She received her B. Math from the University of Waterloo.

PRZEMYSLAW PRUSINKIEWICZ is an associate professor of Computer Science at the University of Regina, Canada. His research interests include computer graphics, interactive techniques, and computer music. Prusinkiewicz received his MS in 1974 and Ph.D. in 1978, both in computer science, from the Technical University of Warsaw.