REPRESENTING GEOMETRIC CONSTRUCTIONS AS PROGRAMS: A BRIEF EXPLORATION

BRUCE SHERIN

School of Education and Social Policy
Northwestern University
2115 North Campus Drive
Evanston, IL 60208, U.S.A.
e-mail: bsherin@northwestern.edu

This column will publish short (ranging from just a few paragraphs to ten or so pages), lively and intriguing computer-related mathematics vignettes. These vignettes or snapshots should illustrate ways in which computer environments have transformed the practice of mathematics or mathematics pedagogy. They could also include puzzles or brain teasers involving the use of computers or computational theory. Snapshots are subject to peer review.

In this issue's snapshot, Bruce Sherin presents a series of geometric constructions made in a Boxer programming environment enriched with a modest set of primitives. In reflecting on the resultant blended turtle geometric and dynamic geometry environment, he discusses the role and potential of programmable applications and the affordances of a programming representation of geometric concepts and other sub-domains of mathematics.

Computer Math Snapshots Editor: Uri Wilensky
Center for Connected Learning and Computer-Based Modeling
Northwestern University, U.S.A.
e-mail: uri@northwestern.edu

INTRODUCTION

In this snapshot, I pick up two research strands that have been prominent in the community that contributes to IJCML. The first strand is research around instructional uses of *turtle geometry*, which had its earliest and most familiar incarnations in the Logo programming language (Abelson and diSessa, 1980; Papert, 1980), and which continues to live on in Logo and its descendents. The second strand, which has appeared somewhat more recently, relates to instructional uses of *dynamic geometry* environments. Prominent instances of this type of environment include Cabri (Laborde and Laborde, 1995) and Geometer's Sketchpad (Jackiw, 2001).

Because these two strands of work pertain to instruction in geometry they are, in principle, related. However, in practice, they have remained relatively separate worlds. Different computer tools are employed in each strand, and the associated activities are, at least superficially, quite different. In turtle geometry activities, students write their own programs, and the constructions they create are often comparatively simple. In contrast, in dynamic geometry environments, students create and interact with constructions primarily though a point-and-click interface. Furthermore, because these environments are specifically created for geometry, they are extremely powerful within this domain, and they allow students to make complex constructions with relative ease.

The purpose of this snapshot is to explore briefly the merging of these two strands of work. Unlike other snapshots, I will not be exploring mathematics that is particularly novel for readers. Rather, my purpose is to look at how, by merging programming and dynamic geometry, we can create *new ways of representing* familiar mathematics. Indeed, I believe that my points can be most forcefully made if the mathematics used for illustration is very familiar to the reader. Thus, that is the tactic I will adopt; for illustration, I will draw on some of the canonical examples from turtle geometry and dynamic geometry.

A central feature of this exploration is that I take seriously the possibility that it is productive to represent geometric constructions as programs. These programming representations of constructions are the "new ways of representing" that are the focus of this snapshot. To proceed, I will start with a fully developed programming language (rather than adding a macro language to a dynamic geometry environment), and I will enrich this programming environment with new primitives that support the creation of geometric constructions. I will not systematically compare this technique to what is possible with the scripting features that are built into some dynamic geometry environments. I only hope to illustrate that, if we set out to enrich a Logo-like programming environment, it is relatively straightforward to create a powerful and elegant language for representing constructions.

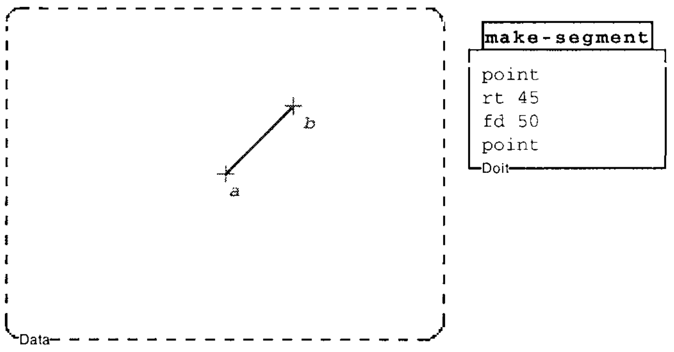
I should mention that, in setting out on this exploration, I am following a path suggested by Michael Eisenberg (Eisenberg, 1995). In previous work, Eisenberg argued for ‘programmable applications’, applications in which an enriched programming language is integrated with a direct-manipulation interface. Along with his colleagues, Eisenberg developed a number of prototype applications, with the Scheme dialect of Lisp as the base language (e.g., Blough and Eisenberg, 1995). Although I will be working with a Logo-like programming language, the approach suggested here is clearly in the same spirit as Eisenberg’s work.

In the remainder of this introduction, I will give a brief history of the geometry tools that are at the heart of this work. Then, in the second main section of this snapshot, I will introduce the enriched programming environment by describing how it can be used to construct the perpendicular bisector of a line segment. In the third section, I then present examples in order to show how it is possible to make very compact and elegant representations of relatively complex constructions. This will first be illustrated with the bisector construction, as well as with the construction of the centroid of a triangle. Finally, in the last main part of the snapshot, I present one final example, the construction of a circle inscribed in a polygon. The purpose of this example is to illustrate how, through the merging of dynamic geometry and turtle geometry, we may be able to create constructions that would be difficult to construct while working solely with one of the component approaches.

The Tool and Its History

In the discussion that follows, I will be describing a set of tools that were constructed to enrich a particular programming environment called *Boxer* (diSessa et al., 1991). In many respects, Boxer is a direct descendent of Logo. As we will see, it is possible in Boxer to write turtle geometry procedures using Logo’s familiar turtle graphics commands. However, Boxer also includes some modern amenities, including a programming interface that is hierarchically structured as boxes within boxes (hence the name).

The history of the geometry tools in Boxer merits a brief recounting. The origins of the tools can be traced, first, to the work of a teacher-researcher-designer named Henri Picciotto, who developed a set of programming tools for use with Logo. In these tools, Picciotto essentially added a set of new primitives to Logo. Picciotto created a manual and curricular activities, which were shipped as a companion to a commercial Logo product (Picciotto, 1990). At Picciotto’s request, I ported his tools to Boxer. In doing so, new design elements were added, in part to take advantage of the new capabilities offered by Boxer. Somewhat later, Andrea diSessa added the capability to click and drag points to dynam-



```
point
rt 45
fd 50
point
```

Figure 1. Program that draws a segment.

ically modify constructions. Throughout this time, Picciotto frequently tinkered with the Boxer geometry tools, adding features and modifying the tools to support his particular needs.

I have related this brief history in order to give the reader a feel for the scope of effort that was involved in creating the tools that will be described below. Unlike most dynamic geometry environments, the amount of effort involved here was modest. Indeed, there was no systematic design effort, or even any organized collaboration. Instead, over the course of years, the tools were picked up and refined by a few Boxer users, working in their spare time.

A First Example

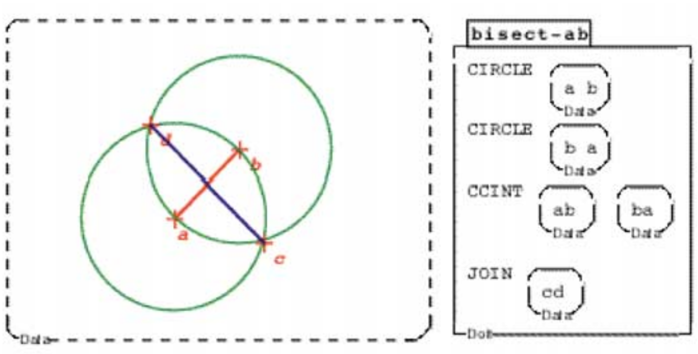
I start with an absolutely prototypical example from the world of straightedge-and-compass geometry: the construction of the perpendicular bisector of a line segment. Figure 1 shows a simple Boxer program that constructs the segment we will bisect. This program first draws a point. It then tells the turtle to turn right 45° and go forward 50 steps before, finally, making a second point. We see in this program the first of our special-purpose primitives, the **point** command. This command draws a point at the current location of the turtle and then labels it with the next available label from an internal list. A description of a selection of the Boxer geometry commands is given in Table I. In most cases, these commands have names and functions that follow those in Picciotto's original Logo tools.

Figure 2 shows a second a program, named **bisect-ab**, that constructs a perpendicular bisector of our segment $\overline{ab}$. This program is made entirely using the new geometry primitives. It first draws two circles, one centered on point a and through point *b*, and the other centered on *b* and through *a*. Then, in the third line of the program, the **ccint** command is used to find

TABLE I

Geometer primitives and their functions

Primitive	Function
point	Draws and labels a point at current turtle location.
join (p1 p2)	Draws a line connecting points p2 and p2.
circle (p1 p2)	Draws a circle centered at point p1 and through p2.
ccint c1 c2	Finds the intersection points of circles c1 and c2.
llint l1 l2	Finds the intersection of the two lines, l1 and l2.
goto p1	Moves the turtle to point p1.
aimto p1	Rotates the turtle so that is headed toward point p1.
ml (p1 p2)	Outputs the length of the segment with endpoints p1 and p2.



```
bisect-ab
CIRCLE  a b
CIRCLE  b a
CCINT   ab   ba
JOIN    cd
```

Figure 2. A program that constructs a perpendicular bisector.

the intersection of the two circles. Finally, the last line of the procedure draws a line that connects these two intersection points.

To this point, all we have done is to associate steps in a construction with programming commands. Certainly the resulting functionality is much less than any dynamic geometry environment. However, it is worth noting that we already have some of the important features of dynamic geometry. Notice that, as shown in Figure 3, we can change the location of points a and b, and then rerun the construction. In an important sense, this is the essence of dynamic geometry, the ability to apply 'the same construction' across different circumstances.

Furthermore, representing the construction in this way has done more than give us some of the basic functionality of a dynamic geometry environment. It is not only the case that we can apply 'the same construction' in multiple circumstances; we also have given this construction a concrete

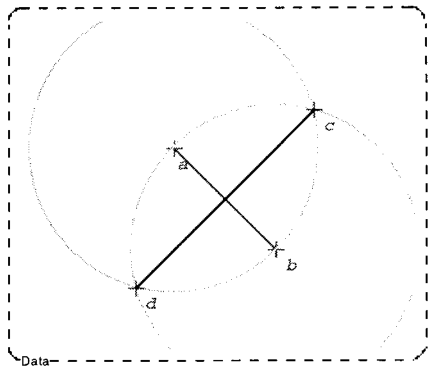


Figure 3. Bisect-ab applied to a different line segment.

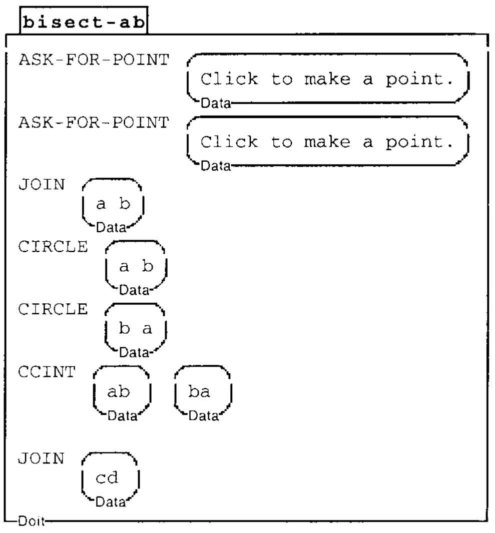
manifestation. And this all works quite naturally. After all, this is what programming languages are for, the representation of procedures.

Toward a More Interactive Environment

One obvious absence in the preceding discussion was any mention of a point-and-click interface. For instance, in the examples described above, it was not possible for a user to click on an object in the graphical display of the construction and delete it. Fully adding this feature to the interface would involve a significant investment of time by a programmer. However, note that it is possible to get the same effect by deleting a line in the program and then rerunning the program.

Similar observations hold for other desirable features of a point-and-click interface. In many cases, operations on the programming representation can substitute for point-and-click interactivity in the graphical display; a user can operate on the programming representation, rather than on the graphical representation of the construction. These observations suggest a more general question: When can operations on a programming representation substitute for point-and-click interactivity in the graphical display? This is a topic for a longer paper; here, I will just make a couple of points.

First, consider the case in which we simply want to add or delete objects from a graphical display by operating on a programming representation. The ease with which this can be accomplished depends on the nature of the correspondence between lines of programming and relevant objects in the display. In the preceding examples, the mapping was straightforward. But we can imagine that, in other integrated programming environments, this will not always be the case. Multiple lines of programming will sometimes be required to construct an object. And, in some cases, the lines of



```
bisect-ab
ASK-FOR-POINT  Click to make a point.
               Data
ASK-FOR-POINT  Click to make a point.
               Data
JOIN  a b
      Data
CIRCLE  a b
        Data
CIRCLE  b a
        Data
CCINT  ab     ba
       Data   Data
JOIN  cd
      Data
Doit
```

Figure 4. Bisect-ab revised to use ask-for-point.

programming that correspond to objects in a display may be interspersed. In such instances, it may be non-trivial to add or delete objects by operating on a programming representation.

Point-and-click interaction with a graphical display can allow other types of operations on the objects in a graphical display, beyond simple addition and deletion. For example, it is possible to change continuous parameters that are associated with a display, such as the location or size of an object, by moving or dragging the mouse pointer. Generally, these operations can be accomplished, by making changes to a programming representation. However, these changes will only be straightforward when the parameter in question appears in a localized manner within a program (e.g., when the parameter appears as a single number in the program).

Although we recognized that much can be accomplished through operations on the programming representation, we did choose to add some point-and-click interactivity. These particular additions were easy enough to add that the benefits were worth the small amount of programming effort required. We made two specific additions that are relevant here. First, we created a command called **ask-for-point**. When this command is executed, the user is prompted to click somewhere in the graphics display, and a new point is then created at the indicated location. Figure 4 shows a new version of **bisect-ab** that uses **ask-for-point**. When this program is run, the user is prompted to select locations for the two initial points. The program then joins these two points, and then constructs the bisector, just as in previous versions of **bisect-ab**.

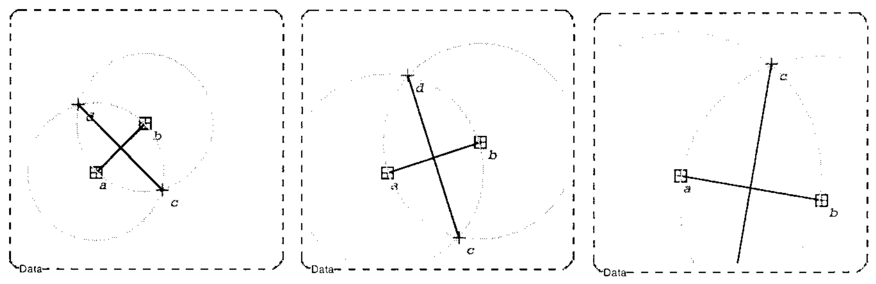


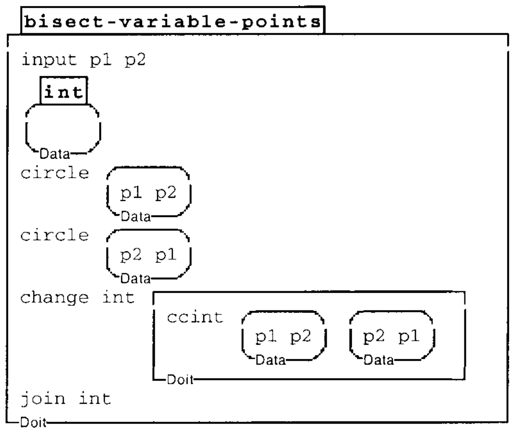
Figure 5. The construction changes as point b is dragged.

Second, we also added the capability for some true dynamic interaction with the graphical display. When this feature is turned on, the user can drag point *a* or *b* in our above constructions. Figure 5 shows a sequence where point *b* has been dragged down and to the right. What is most interesting about this feature is the relative ease with which it was added. Because a programming representation of the construction was available, the underlying code must simply rerun this programming representation as the point is dragged. This is another instance in which the availability of a programming representation of the construction has paid off.

FURTHER EXPLORATIONS OF THE PARADIGM

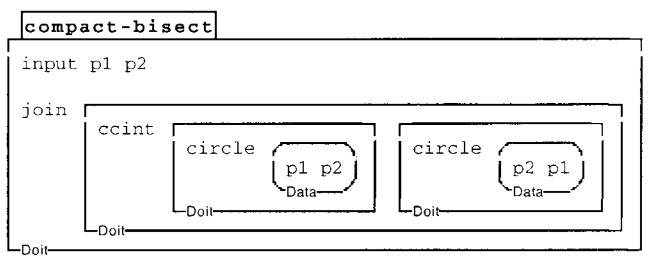
To restate: our goal here is to explore what happens if we take seriously the idea that we can have programming representations of geometric constructions. In this section, we take this proposal to some of its more interesting extremes, and we will begin to see some more dramatic payoffs in terms of power and elegance of representation. To begin, notice that our **bisect-ab** procedure was written so that it operates on points named 'a' and 'b'. This allows for some flexibility, since the points a and b can be at any location. However, there are some respects in which this is not as flexible as one might want. For example, it would not be possible to use **bisect-ab** to construct bisectors of two segments within a single construction.

This situation is ameliorated if we make use of variables in the place of specific point names, as in Figure 6. This new procedure takes two inputs, named **p1** and **p2**, as specified at the top of the procedure. The procedure begins by drawing the two circles, as before. Then it finds the intersections of these two circles and stores them in the local variable named 'int'. (This local variable appears in Figure 6 as a box with the name 'int'.) Finally, **int** is passed as an argument to **join**, and the bisector segment is drawn.



```
bisect-variable-points
input p1 p2
int
  Data
circle  p1 p2  Data
circle  p2 p1  Data
change int  ccint  p1 p2 Data   p2 p1 Data
            Doit
join int
Doit
```

Figure 6. A bisect program using variables.



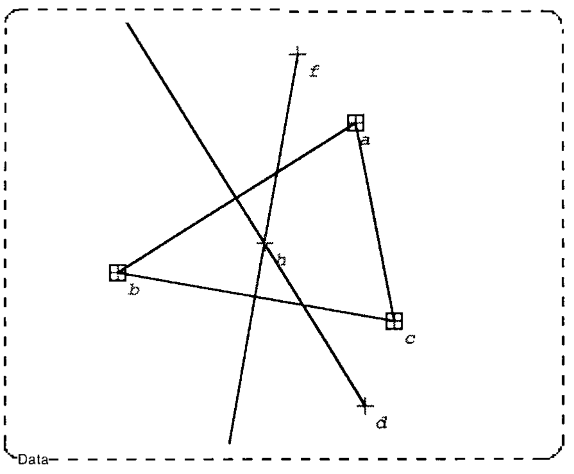
```
compact-bisect
input p1 p2
join  ccint  circle  p1 p2 Data    circle  p2 p1 Data
             Doit                  Doit
      Doit
Doit
```

Figure 7. A very compact version of the segment bisector program.

In addition to using variables, the program in Figure 6 makes use of the fact that our geometry primitives output convenient values. In particular, in the third line of programming, I am making use of the fact that the **ccint** command not only marks the two points of intersection in the graphical display, it also outputs these two points.

In providing outputs in this manner, we are taking a big step toward creating a language that can support useful representations of geometric constructions. Through the use of these output values, we can begin to make compact, yet powerful expressions. For illustration, Figure 7 shows how we can capitalize on outputs to create a compact, one-line version of our segment bisector. In this short procedure, each of the calls to **circle** produce outputs that serve as the inputs to **ccint**. The output of **ccint** (the intersection points) then, in turn, serves as the input to **join**.

Using any of the versions of our bisect program that use variables, we can construct bisectors at multiple points within a more complex construction. Figure 8 shows an example in which the **compact-bisect** program has been used in a prototypical application from dynamic geometry, the

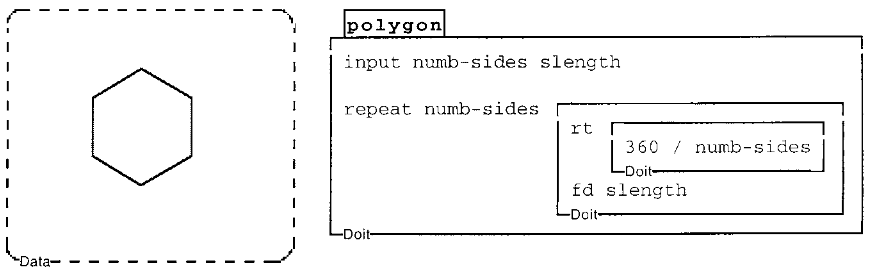


```
centroid
input p1 p2 p3

llint [compact-bisect p1 p2] [compact-bisect p2 p3]
```

Figure 8. Program that constructs the centroid of a triangle.

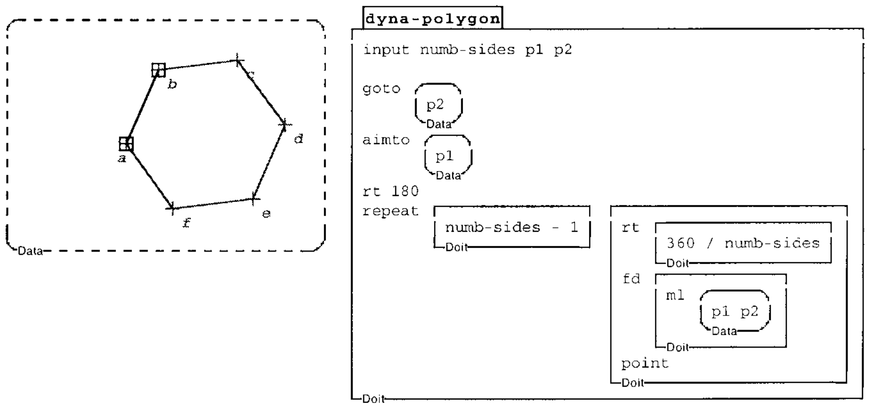
construction of the centroid of a triangle. The result is, once again, a compact one-line program. This program constructs bisectors of two sides of the triangle. The resulting segments are then passed as inputs to the **llint** command, which finds the intersection of two given lines. (In Figure 8, I have suppressed the circles in the construction for clarity.)

I want to pause for a moment to reflect on what we have just seen. In the programs in Figure 7 and Figure 8, we have two extremely compact representations, each embodying quite a bit of mathematics. Is this a good thing? Certainly, we must expect some of the usual pitfalls associated with the use of highly compact representations in mathematics instruction. In particular, it is possible to envision situations in which students make use of representations of this sort, without being able to unpack them in order to understand what they say. However, we should also expect some of the usual benefits of compact representations. It is easier to treat a compact representation as a single entity, capable of consideration and manipulation in its own right. Furthermore, while compact representations of this sort hide some structure, they also make certain kinds of structure more evident.



```
polygon
input numb-sides slength

repeat numb-sides
    rt
        360 / numb-sides
    fd slength
```

Figure 9. Procedure to draw a polygon.



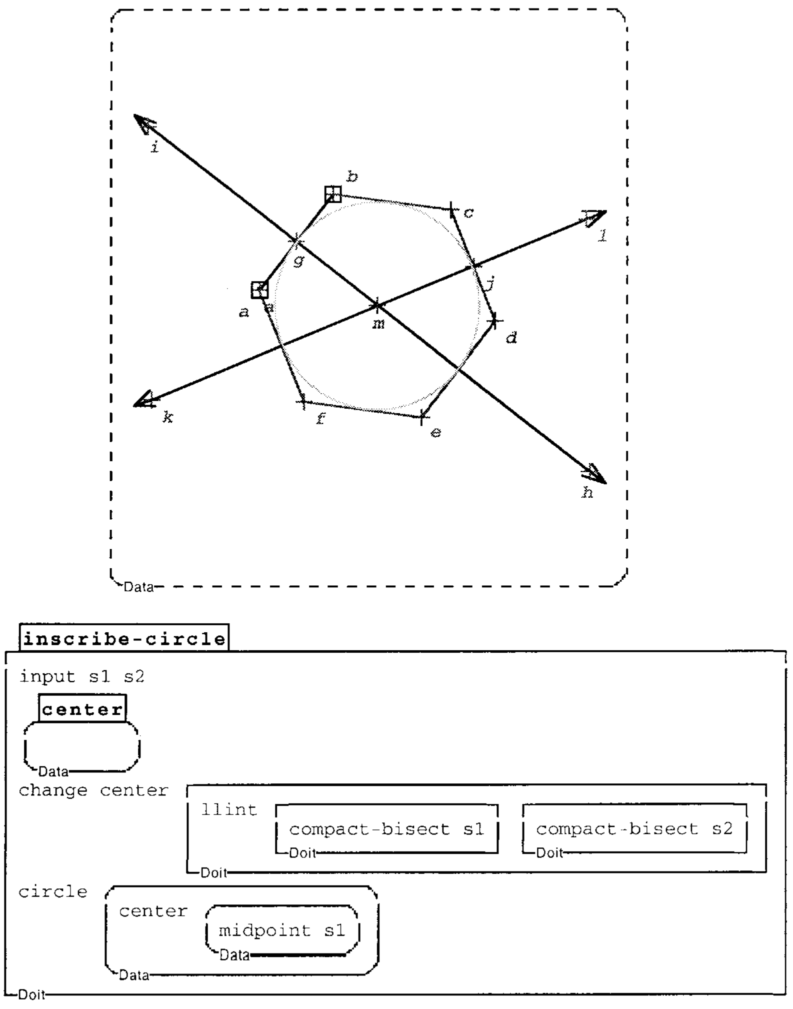
```
dyna-polygon
input numb-sides p1 p2

goto
    p2
aimto
    p1
rt 180
repeat
    numb-sides - 1
        rt
            360 / numb-sides
        fd
            ml
                p1 p2
        point
```

Figure 10. A new version of the polygon procedure.

A FINAL EXAMPLE

Before concluding, I want to present a final example in order to illustrate the power that is derived from synthesizing the techniques of turtle geometry and dynamic geometry. Here we start with the turtle geometry procedure shown in Figure 9. This procedure draws a regular polygon given two inputs, the number of sides (**number-sides**) and the length of a side (**slength**). Readers familiar with turtle geometry will recognize this as one of the standard exercises that is given to students.

In our dynamic geometry sub-environment, it makes sense to revise this procedure slightly. The revised procedure, which is shown in Figure 10 takes the number of sides as one of its inputs, but the other two inputs are points that will be the endpoints of one side of the polygon. Using these three inputs, the procedure can construct a regular polygon in which all of the vertices are labeled points. In addition to the standard turtle primitives and the **point** command, this procedure makes use of three geometry commands that I have not yet discussed. The **goto** and **aimto** commands

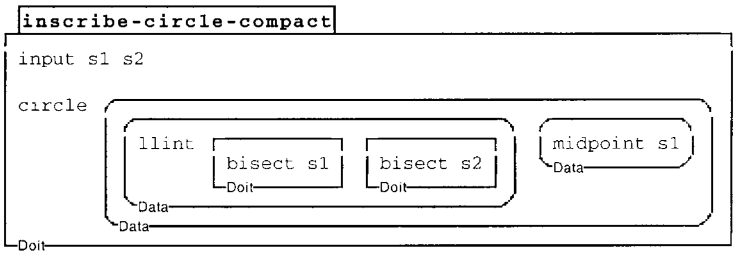


```
inscribe-circle
input s1 s2
center
  Data
change center
  llint  compact-bisect s1   compact-bisect s2
         Doit                Doit
  Doit
circle
  center  midpoint s1
          Data
  Data
Doit
```

Figure 11. Procedure that inscribes a circle in a polygon.

position and orient the turtle so that it is prepared to draw the remainder of the polygon, and the **ml** command outputs the length of the given line segment (refer to Table I).

When the polygon is constructed in this manner, it can interact productively with other features of our simple geometry environment. For example, it is now possible to drag points *a* and *b*. More importantly, the constructed polygon can serve as the basis for the sort of constructions that are easily accomplished with compass and straightedge. For example, as shown in Figure 11, we can write a procedure to inscribe a circle in a given regular polygon. This new procedure takes, as inputs, segments



```
inscribe-circle-compact
input s1 s2
circle
  llint  bisect s1   bisect s2      midpoint s1
         Doit        Doit           Data
  Data
  Data
Doit
```

Figure 12. More compact procedure that inscribes a circle in a polygon.

corresponding to any two sides of the polygon (**s1** and **s2**). It first constructs the perpendicular bisectors of these two segments. Then it marks the point of intersection of these bisectors and stores the result in a variable named ‘center’. Finally, it draws a circle using the midpoint of **s1** as a point on the circle. As with our other examples, this procedure can be rewritten in a compact form, as shown in Figure 12.

In constructing these inscribed circles, I am capitalizing on the virtues of both types of environments. Drawing a polygon with a variable number of sides is relatively easy in turtle geometry, but comparatively hard with the limited tools of straightedge and compass. Of course, we could include a special purpose command for drawing polygons in a dynamic geometry environment. But, in the present case, we get this capability largely for free, simply because we have turtle geometry commands available. Conversely, inscribing a circle in a polygon is moderately difficult in turtle geometry, and comparatively easy with dynamic geometry-like tools. Because we have commands that mimic some of the capabilities of dynamic geometry, we can use these features in combination with our turtle geometry commands.

CONCLUSION

The purpose of this snapshot was to explore the merging of turtle geometry with dynamic geometry. To do this, I proceeded by describing how a few users enriched a programming environment so that it was possible to create programming representations of geometric constructions. To conclude, I will now draw together some of the main observations made during this exploration.

I begin with the observation that the brief examples presented here provide something of an existence proof that Eisenberg’s (1995) integrated programming paradigm can be applied to dynamic geometry. I proposed to take seriously the notion that we could employ programming representations of geometric constructions. In following this proposal we saw that, within the examples considered, the mapping from features of programming to geometric constructions can be made very comfortably. The creation and use of sub-procedures maps cleanly onto reused components in geometric constructions. And the notion of *variable* has useful counterparts in constructions. Furthermore, it is possible, in at least some cases, to create programming representations that are compact, while still being powerful.

Second, through the merging of these two approaches, we accrued some of the benefits of both turtle geometry and dynamic geometry. Some of what is hard in dynamic geometry environments is comparatively straightforward in turtle geometry. Conversely, much of what is hard in turtle geometry is relatively easy with dynamic geometry-like tools.

As an aside, note that some of these benefits may be amplified if we can make use of a programming language that has a range of uses, and that students already know. It is possible to add macro languages to dynamic geometry environments. But using a language that users already know could potentially have many advantages. Users would not need to learn a new language. Furthermore, they may make use of the additional power afforded by the programming environment, in ways that we cannot anticipate. Whether these benefits actually accrue is a matter for future research.

My third observation is that a few users were able to build a relatively powerful environment with comparatively little programming work. Certainly, the resulting tools are not as powerful as the excellent dynamic geometry environments that are commercially available. However, given the limited amount of effort that was required, it is striking that we could obtain such an important subset of the desired functionality. Furthermore, although the geometry tools described in this snapshot are not yet powerful enough to supplant existing applications, I believe that the examples presented suggest that it may be worth following this avenue further, to create a more powerful and feature-rich environment. Such an environment may then present us with a real alternative to existing environments.

Finally, the exploration presented here can be taken as an example of what may be possible across a range of educational applications in mathematics. Because of the significant effort that was required to create dynamic geometry applications, developers of educational software may have been hesitant to tackle new domains; at the least, developers would have to think carefully about which few domains were deserving of their effort. However, if it is possible to get significant functionality by enriching programming environments, it may be possible to apply a dynamic geometry-like approach to more sub-domains within mathematics. For example, we could imagine, a programming language that has been enriched for number theory – for example, with primitives that find the factors of a number, or determine whether it is prime – or a programming language that has been enriched for probability theory, or for the manipulation of vectors and matrices. Such environments might allow teachers and students to harness the power of computation across the range of mathematical sub-disciplines.

REFERENCES

Abelson, H. and diSessa, A.A. (1980). *Turtle Geometry*. Cambridge, MA: MIT Press.

Blough, E. and Eisenberg, M. (1995). Combining programming languages and direct manipulation in environments for computational science. In G. Olson and S. Schuon (Eds), *Proceedings of Designing Interactive Systems* (pp. 123–130). Ann Arbor, MI.

diSessa, A.A., Abelson, H. and Ploger, D. (1991). An overview of Boxer. *Journal of Mathematical Behavior* 10(1): 3–15.

Eisenberg, M. (April, 1995). Programmable applications: Interpreter meets interface. *SIGCHI Bulletin* 27(2).

Jackiw, N. (2001). The Geometer’s Sketchpad (Version 4.0) [Computer Software]. Emeryville, CA: KCP Technologies, Inc.

Laborde, C. and Laborde J.M. (1995). The Case of Cabri-géomètre: learning geometry in a computer-based environment. In D. Watson and D. Tinsley (Eds), *Integrating Information Technology into Education* (pp. 95–106). London: Chapman & Hall.

Papert, S. (1980). *Mindstorms*. New York: Basic Books.

Picciotto, H. (1990). *Logo Math: Tools and Games, a Comprehensive Computer Environment to Enhance the Discovery-based Learning of Secondary School Mathematics.* Cambridge, MA: Terrapin, Inc.