Geometer Reference Manual

Tom Davis

May 17, 2006

Contents

Chapter 1

Introduction

See Appendix 9 for the Quickstart Guide.

If you just want start playing with the program, go ahead—that's a perfectly reasonable approach. If you decide to do so, your best approach is to begin by working through the first couple of examples in the tutorial (Chapter 2). The files used in the tutorial are in the `Demos` subdirectory of the installation directory. See Section 1.1 for details.

You can always come back here later.

1.1 Finding Geometer Files

All the sample files can be found in the installation directory of **Geometer**. During installation on a Windows machine, **Geometer** suggests that it be installed in the directory:

```
C:/Program Files/Geometer
```

but the person who installed it could have put it anywhere. On a Macintosh it could also be anywhere.

Double-click on the Geometer icon and once **Geometer** is running, and if you issue the *Open File* command from **Geometer**, the file chooser will start in that installation directory. (If you double-click on a **Geometer** diagram, the file chooser will start in the directory of the file that you double-clicked.) For details on the use of the file chooser see Section 4.9.

Here are the usual subdirectories of the **Geometer** installation directory:

- All the files used in the online tutorial (which is very similar to the tutorial chapter in this manual—see Chapter 2) are in the subdirectory called `Demos`.
- The files used in this reference manual are in the subdirectory called `Reference`. That directory has various subdirectories corresponding to the chapters in this manual.

- If you purchased the Book called *Geometry with Computers*, the files used there are in the `GeomBook` subdirectory. If you did not purchase the book, those files will not be present.
- The `HTML` subdirectory does not contain **Geometer** diagram files; it contains the help files that are viewed from the **Geometer** *Help* menu. See below. On a Windows or LINUX system, set your environment variable `GEOMDOC` to refer to this directory. `GEOMDOC` should probably be something like this: `C:`
  `Program Files`
  `Geometer`
  `HTML`.
- There may be other directories with additional files in them.

Example: If you are going through the tutorial (either here or online) and it tells you to open the file called `Circ.T`, start the file chooser using the *Open* command under the *File* pulldown menu. Double click on the directory entry `Demos`, and then double click on the file name called `Circ.T`.

Example: If you are reading this reference manual and you would like to experiment with the file referred to as `Ref1/Orthocenter.T`, start the file chooser with the *Open* command in the *File* pulldown menu, double click on the `Reference` directory (that's where all the reference manual files are found), then double-click on the `Ref1` subdirectory, and finally on the file called `Orthocenter.T`.

If you own the book, *Geometry with Computers*, its files are organized exactly like those in the reference manual except that you will first click on the `GeomBook` directory instead of the `Reference` directory.

1.1.1 Help Files

The **Geometer** help files in HTML format are found in the `HTML` subdirectory of the installation directory. If you would prefer to use your normal web browser on those files instead of the simple-minded **Geometer** built-in browser, double-click on the file called `index.html` in the `HTML` subdirectory outside of the **Geometer** program.

1.2 Notational Conventions

The notation used in this manual is fairly standard, but there are a few notational conventions:

- Command names are displayed as follows: *Open* or *PP=>L*.
- File names, or the contents of **Geometer**'s text files `are written like this`.
- Keyboard commands are written like this: **Ctrl-C** (which means to hold down the **Ctrl** key while pressing the **C** key).
- Sections that are difficult or very difficult are preceded by the symbols: ♦ or ♦♦.

1.3 What is Geometer?

Geometer is a tool to visualize Euclidean geometry dynamically. It is for students or teachers who wish to gain a more intuitive understanding of geometry. It can be used for many things:

1. It can test thousands of configurations of geometric conjectures, at least to the accuracy of the screen resolution of your computer.
2. Students can step through pre-packaged proofs, constructions, or derivations of geometric facts.
3. Teachers can to construct those pre-packaged proofs and demonstrations.
4. It can be used as an exploratory tool to search for geometric relationships.
5. It can be used to generate high quality drawings for use on class handouts, or for publication.
6. It can be used to write and run demonstration scripts that run like canned video demonstrations.

More specifically, **Geometer** is a program for drawing, editing, and displaying figures, (or "diagrams", as they will be called here), of plane Euclidean geometry. It is a constraint solving system in the sense that the user specifies constraints, and **Geometer** calculates the diagram from those constraints.

At first glance, it seems like a simple drawing program with a geometric flavor. Imagine the following sequence of commands:

1. The user begins with a blank window, and begins by issuing a command to draw some points. Three mouse clicks on different places in the window result in three points labeled A, B, and C.
2. Next, the user issues the **Geometer** command to draw the circle centered at one point and passing through another. After identifying A as the center and B as the other, a circle will be added to the drawing centered at A, and passing through B.
3. Finally, the user issues commands to draw the lines through C that are tangent to ("tangent to" = "touching") the circle. Depending on exactly where A, B, and C are located, the result will be a drawing something like one of those that appear in Figure 1.1. (Notice that for the configuration in the lower left of the figure, tangent lines through the point C do not exist, so **Geometer** doesn't draw them.)

On the screen, the "obvious" diagram appears, but internally, only the locations of the points are stored, together with the constraints that the circle and two lines must satisfy. What makes **Geometer** dynamic is that at any point in the display or editing of the diagram, the user may move one or more of the unconstrained points (A, B, and

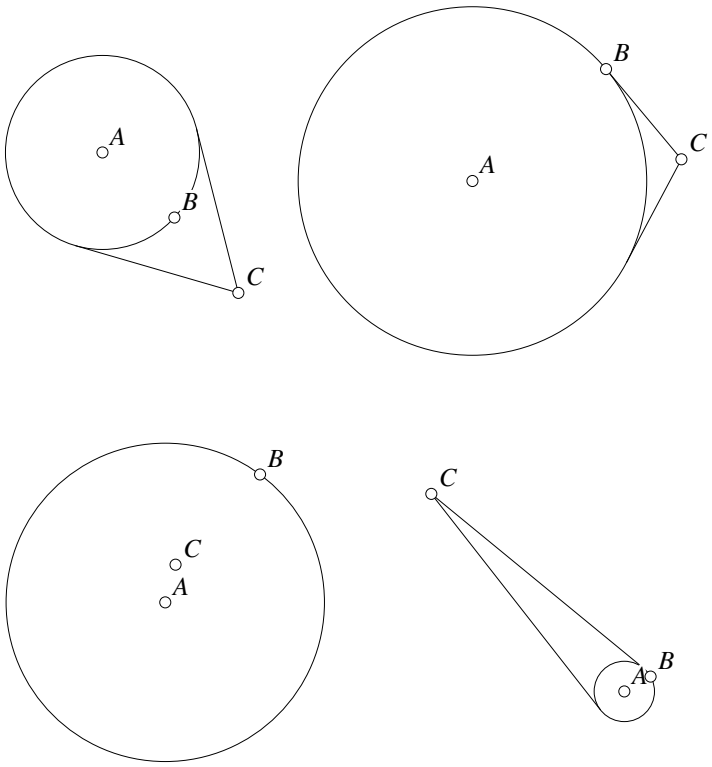

Figure 1.1: **Geometer** As Constraint Solver
`Ref1/Circtan.D` [D]

C in this particular example) and the diagram will be altered so that the constraints continue to be satisfied.

Figure 1.1 illustrates four possible configurations of the situation described in the previous example. After the initial commands to specify the figure, the user can simply use the mouse to drag around the points A, B, and C and the drawing will change dynamically in such a way that the circle remains centered at A, passing through B, and so that the lines through C remain tangent to that circle. All four examples in the figure can be obtained from any of the others by a suitable movement of the points A, B, and C.

In fact, the user doesn't even need to wait for all the constraints to be satisfied before modifying the figure. Imagine that the original points were arranged as in the lower left example, and after drawing the circle, the user realizes that the tangent lines won't make sense. So all that's required is to drag C outside the circle, and then to issue the commands to draw the tangent lines.

In the vast majority of cases, geometric diagrams can be manipulated in **Geometer** using a graphical user interface (GUI). The mouse is used to select and drag points, and as they are moved, the constraints are continuously re-solved, and with each solution, an updated diagram is displayed.

1.4 What is a Geometer Diagram?

A typical **Geometer** diagram illustrates a geometric concept or theorem as a drawing on a computer screen. **Geometer** diagrams are far better than figures in a textbook, however, because they can be modified by the user, and as the modification occurs, **Geometer** continuously re-solves the constraint equations and redisplays the diagram. It's as if you were reading a textbook that contains thousands of figures, or at least thousands of variations on each figure.

Geometer can step through a proof or demonstration, emphasizing the important parts of each step, but still allowing you to manipulate the figure at any stage, and to go back to previous steps if you get confused or need to check on something.

In addition to being dynamic, the diagrams have features that can be emphasized by means of different colors, labels, marks, and blinking. Text can be associated with each stage of a demonstration.

Geometer also has a script mode where a diagram can be driven automatically through a programmed animated script.

The best thing about **Geometer**, however, is not the collection of packaged examples and illustrations, but the fact that you, as a student or teacher of geometry, can build and manipulate your own diagrams, making geometric discovery much easier and more fun.

The word "geometry", of course, is used here in a very loose sense. **Geometer** can work with trigonometry, projective geometry, and can even be used to generate computer art. If you're willing to go to the effort, it can be used like a miniature graphical programming language with a user interface that's primarily graphical.

1.5 Geometer Features

In addition to the points, lines and circles in the example, **Geometer** supports other types of geometric primitives. The complete list includes points, lines, circles, conic sections, angles, arcs, polygons and Bézier curves. There are dozens and dozens of constraints among the geometric primitives that can be specified. Finally, **Geometer** supports a few non-geometric primitives, including, but not limited to: floating point numbers, text, and projective transformations.

Also included is machinery to define "macros"—complex operations that are constructed from simpler ones which can be repeated many times. It supports machinery to run scripts to put **Geometer** into "auto-pilot" mode, and still other machinery that

allows users to step through a complex proof or construction and to have various features appear and disappear as necessary. At every stage, the user can manually alter the geometry (and the constraints will be continuously re-solved) to see what happens.

Finally, and this is perhaps one of its most powerful features, **Geometer** can present both a textual and a geometric representation of any diagram. In other words, you can edit geometric diagrams using either visually or textually.

To illustrate, here is **Geometer**'s textual description of the example above with the circle and the tangent lines:

```
.geometry "version 0.2";
v1 = .free(0.2275, 0.515, "A");
v2 = .free(0.52, 0.76, "B");
v3 = .free(0.685, 0.5625, "C");
c1 = .c.vv(v1, v2);
l1 = .l.vc(v3, c1, 1);
l2 = .l.vc(v3, c1, 2);
```

Don't worry about the exact details, but here's how to read that textual description.

The first three lines create the three free points (the "`.free`" means they can be freely moved with a mouse)*. Internally, they are named `v1`, `v2` and `v3`; on the screen, they are called A, B, and C. The two numbers in each line are the current coordinates of the point. The next line defines a circle, internally called `c1`, which satisfies the `.c.vv` constraint. That constraint is that the circle created must use the first point as its center and must pass through the second. In this example, the first point is `v1` (labeled A) and the second point is `v2` (labeled B)—exactly what we see in the diagram. The final lines of text define the tangent lines with similar constraints.

So far there's nothing too special, but (again without worrying too much about the details, take a look at the following **Geometer** code that draws the actual figure displayed in the text (Figure 1.1). Remember that the actual figure has four copies of the figure above, where each contains points, circles and lines, but in all cases the constraints satisfied are the same. By using a macro, all the constraints are specified once, and then those are applied to four different sets of points to draw the four examples you see in the figure. The macro consists of the six lines beginning with `.macro` and the three lines between the curly braces are the macro body.

```
.geometry "version 0.2";
.macro circtans(.vertex v1, .vertex v2, .vertex v3)
{
    c1 = .c.vv(v1, v2);
    l1 = .l.vc(v3, c1, 1);
    l2 = .l.vc(v3, c1, 2);
}
v1 = .free(0.2275, 0.515, "A");
v2 = .free(0.52, 0.76, "B");
v3 = .free(0.685, 0.5625, "C");
w1 = .free(-0.445, -0.4075, "A");
w2 = .free(-0.2375, -0.1275, "B");
w3 = .free(-0.4225, -0.325, "C");
```

*In the internal format points were originally called "vertices", and that is why you see all of the `v`s. Not only do the points have names like `v1`, `v2` and so on, but the commands like `.c.vv` stand for "create a circle from two vertices". If the program were rewritten today, the command would be `.c.pp`.

```
x1 = .free(0.56, -0.6025, "A");
x2 = .free(0.6175, -0.57, "B");
x3 = .free(0.1375, -0.17, "C");
y1 = .free(-0.5825, 0.5775, "A");
y2 = .free(-0.425, 0.435, "B");
y3 = .free(-0.285, 0.27, "C");
circtans(v1, v2, v3);
circtans(w1, w2, w3);
circtans(x1, x2, x3);
circtans(y1, y2, y3);
```

Since there's both a textual and a graphical version of every diagram and you can edit it using the graphical user interface or a text editor, you have the best of all possible worlds and you can edit your diagram using the most appropriate method.

(Because of the existence of a textual version of each diagram, if you are computer-literate, it is quite easy to write computer programs that generate **Geometer** diagrams. This can be quite a powerful feature for complex diagrams. Some of the more complicated diagrams on this CD were generated that way.)

1.6 ♦Geometer as a Research Tool

If you don't know high school geometry, it's probably best to skip this section.

As a final example in this introduction, here is how **Geometer** might be used to help solve a real problem. (The problem here would not be considered "research" by a professional mathematician, but if you don't know how to solve it, it's research for you.)

Here's the problem: Given the lengths of the three medians of a triangle, find (in other words, construct) the triangle itself. Let's see how **Geometer** might help. Recall that a median of a triangle is the line segment connecting a vertex with the midpoint of the opposite side.

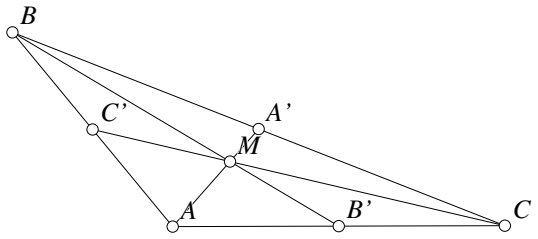

Figure 1.2: Triangle with its medians
`Ref1/Meds.T` [M]

First, look at Figure 1.2. We are given the three lengths AA', BB', and CC' and we wish to find the triangle. We know that the point M where the three medians meet is $2/3$ of the way from each vertex to the opposite side—in other words, $AM = 2A'M$,

$BM = 2B'M$, and $CM = 2C'M$. So if we start with, say, AA', we can find a point M on it that is $1/3$ of the way between A' and A (this is a standard construction).

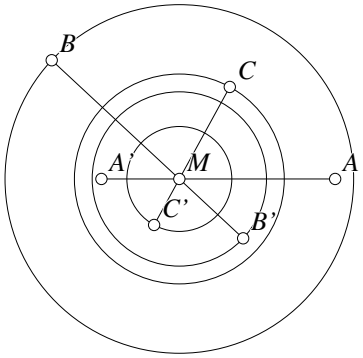

Figure 1.3: Possible median locations
`Ref1/Meds1.D` [D]

Now look at Figure 1.3. If you know the location of segment AA', you can find where M is, and since you know that all the medians meet at M, and that they are all $2/3$ of the distance between the vertex of the triangle and the opposite edge, the points B, B', C, and C' must all lie somewhere on the circles shown in the figure.

So if we leave AA' fixed and if we could move B around its circle to all of its possible positions, we know that the median property will be satisfied if the midpoint between B and A is at C'. (If this isn't clear, look back at Figure 1.2 and read it again.)

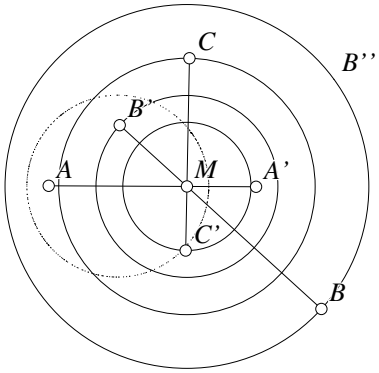

Figure 1.4: Possible midpoint locations
`Ref1/Meds2.T` [M]

So as B moves around its circle to all of its possible positions, what path does the midpoint between B and A follow? If you're not sure, the best bet is to make that midpoint, draw it in a smearing color, and then move B'' once around its circle (leave

B fixed so the circle size doesn't change, and then move B'' around to try all the positions on that circle) to see what you get. What you get is Figure 1.4.

The path of the midpoint of AB appeared to be a circle! Suspecting that this path is, in fact, a circle (and knowing some geometry), it is easy to show that the path of the midpoints is a circle, and to work out the center and radius of that circle. If you construct that circle and the circle of possible locations for C', the point C' has to be at the intersection of those two circles (See Figure 1.4). Once you know where C' goes, it's easy to find C (opposite M from C', and twice as far away), and then B will be on the line CA', and as far from A' as C is.

You could probably have figured this out without **Geometer**, but it is certainly a lot easier to play with the drawing for a few seconds and notice the critical property for the construction.

Chapter 2

Tutorial

> Geometry is the science of correct reasoning on incorrect figures.
> *George Pólya*

2.1 Let's Get Going

Geometer is easy to use, and since a picture is worth a thousand words, let's just start with a picture. Install **Geometer** on your computer, double-click the **Geometer** icon and you'll be ready to go.

If you own the book, *Geometry Using Computers*, it contains many more examples that are completely explained and it thus will serve as a more advanced tutorial. There is also a teacher's tutorial later in this document. See Chapter 6.

If you have problems starting the program (the window is too big for your screen, the colors are strange, the font is too small to read, et cetera) there are preferences that can be changed. See Section 5.12.5.

The first example will illustrate a standard theorem from high school geometry—the fact that the three medians of any triangle all meet at a point, or as a geometer would say, the three medians are "concurrent".

(In what follows, we assume that you know how to use a mouse and keyboard, how to use pulldown menus, how to start programs, how to use a simple cut-and-paste text editor, et cetera. If not, get your kid to help you.)

Run the **Geometer** program by double-clicking the icon. The window that appears has three sections: a large black work area, a pulldown menu across the top, and a command area on the right side of the screen that's filled with buttons and other controls. See Figure 2.1.

If you have just installed Geometer, there will also be a "Tip of the day" that you can dismiss for now by clicking on its "Close" button. This tip will appear every time you

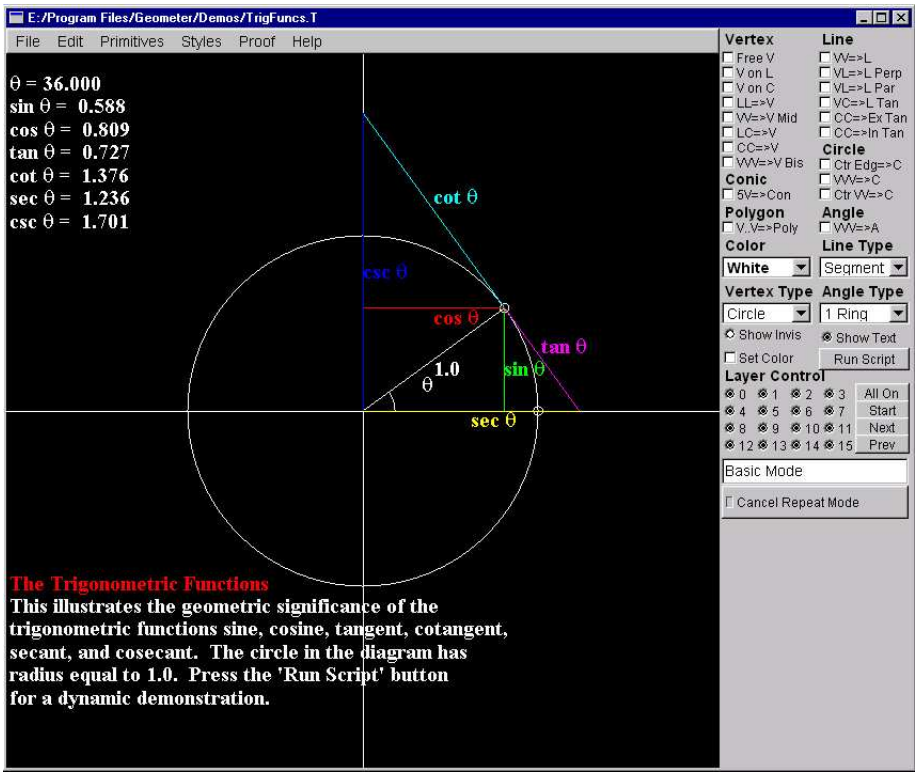

Figure 2.1: Geometer Window

start **Geometer** until you turn it off by clicking in the little box at the bottom. It can be turned back on with the *Edit Preferences* command in the *Edit* pulldown menu.

Under the **File** entry in the pulldown menu, select **Open**. A file selection dialog box will appear, and use that to double click on the directory **Demos** and then double click on **Medians.T** to open that file.

You should see a large triangle ABC as shown in Figure 2.2. Use the mouse to move the cursor over one of the points labeled A, B, or C, press the left mouse button, and hold it down as you move the mouse around. The point will be dragged by the mouse, and the whole figure will change in response. Note that the three lines inside the triangle continue to cross at the same point.

So what's going on?

This example illustrates an important theorem in Euclidean geometry—that the three medians of a triangle meet at a point. A median is a line that connects a vertex of a triangle to the midpoint of the opposite side. The medians in this example are the line segments connecting A to A_m, B to B_m, and C to C_m. (A_m is halfway between B and C, B_m is halfway between A and C, and so on.)

Now try dragging a vertex again and watch what happens a bit more closely—as you

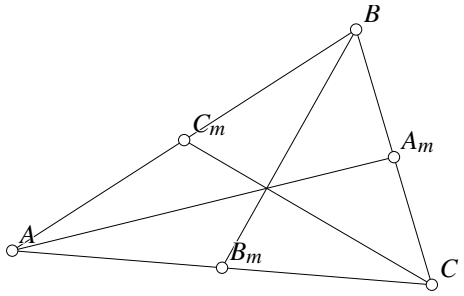

Figure 2.2: Medians
`Ref1/Medians.T` [P]

move the vertex A, B, or C, the points A_m, B_m, and C_m move in such a way that they remain exactly in the middle of each side.

As you drag the vertices of the triangle around, you are effectively testing the theorem for thousands of possible triangles, and although this is not a proof of the theorem, at least it provides a lot of evidence that the theorem may be true*. What we see is clearly not a proof for many reasons:

1. The drawing is only accurate to the resolution of your computer screen which is far less accurate than $1/100$ centimeter—even on the best monitors available today. Maybe the lines just come close to meeting at a point—to within a billionth of a millimeter, for example.

2. You only tried a few thousand examples. Maybe you missed the one bad one.

3. The window on your computer screen is unlikely to be more than about 2000 pixels by 2000 pixels, and it's probably much smaller. What if the theorem starts to fail for triangles that are long and skinny—say $1,000,000$ pixels long†?

But the thousands of **Geometer** diagrams are certainly better than what you get in a typical textbook or what you can do with paper and pencil. If you only look at a single illustration in a textbook, how do you know that the person who drew it wasn't simply lucky, and for the particular triangle drawn, the three medians do happen to coincide? After all, if you only look at a diagram of an equilateral triangle, a "theorem" that states that all medians of a triangle are also angle bisectors of that triangle appears to be true.

*When the author was showing a very early version of **Geometer** to a friend, she asked if it were a "theorem prover". The answer, of course, is no. It's a "theorem convincer".

†This reminds me of an exchange that occurred 30 years ago in a logic class taught by Fred Thompson at Caltech. A student was arguing with him about linguistics and (foolishly) said something about "a random English sentence". Professor Thompson said, "What do you mean by 'a random sentence'?" The student grabbed a history text and said, "The fifteenth sentence on page 241 of this book." (or something like that). Thompson replied, "How on earth can you call that a random sentence? That book doesn't have even a single sentence with more than a million words in it, and almost all sentences are longer than that."

Finally notice that the only points you can move are A, B, and C; you can't move the medians A_m, B_m or C_m (although they do get highlighted when you click on them—more on that later), or the point at the intersection of all three medians. Some points are free, others are constrained, and as we'll see later, some are partially constrained.

All the diagrams that appear in this release were created with **Geometer** and the vast majority of them can be manipulated in exactly the same way you manipulated the medians in the example above. Some of them allow even more manipulation options.

2.2 Viewing Prepared Files

Before trying to create your own **Geometer** diagrams, you may wish to view a few of the prepackaged demonstrations. In fact, for many people, the best way to learn what **Geometer** does will be to try lots and lots of the diagrams. When you see something that you don't understand, you can look at the text version, or if that doesn't work, you can break down and look at the reference manual. We'll look at one canned demonstration, but you can get a better idea of **Geometer**'s capabilities by loading a bunch of other diagrams and playing with them.

2.2.1 A Geometer Proof: The Nine-Point Circle

Let's use **Geometer** to illustrate a proof of a very interesting (and surprising) theorem. Using the *Open* command in the *File* pulldown menu, this time open the file called `Ninepoint.T` (which should be in the `Demos` directory). If you restarted **Geometer** after you viewed the medians example, you will have to do the same thing as before: double-click on the `Demos` directory and then on the file `Ninepoint.T`. If you are still in the same **Geometer** session, the file browser will remember that you are in the `Demos` directory and all you will need to do is double-click on the new **Geometer** file name.

The text on the screen tells you that for any triangle, the midpoints of the sides, the feet of the altitudes, and the points half-way between the orthocenter and the vertices of the triangle all lie on a single circle. (The orthocenter is the point where the three altitudes of the triangle meet.)

It would be tough to visualize this without some sort of drawing, and every geometry book in which the theorem appears will have a drawing showing you what's going on.

Similarly, **Geometer**'s diagram (Figure 2.3) illustrates one example. The altitudes are the perpendiculars to the sides that pass through the vertices opposite. In the **Geometer** diagram, they are drawn in red. The points E, F, and G at the feet of the altitudes are labeled in red as well. The midpoints of the sides are labeled in yellow, and the points midway between the intersection of the altitudes at H and the vertices are labeled in green. All nine points lie on the magenta circle.

Mouse down on the points A, B, or C, and move them around. The triangle can change shape, perhaps drastically, and the magenta circle will change size and position as well,

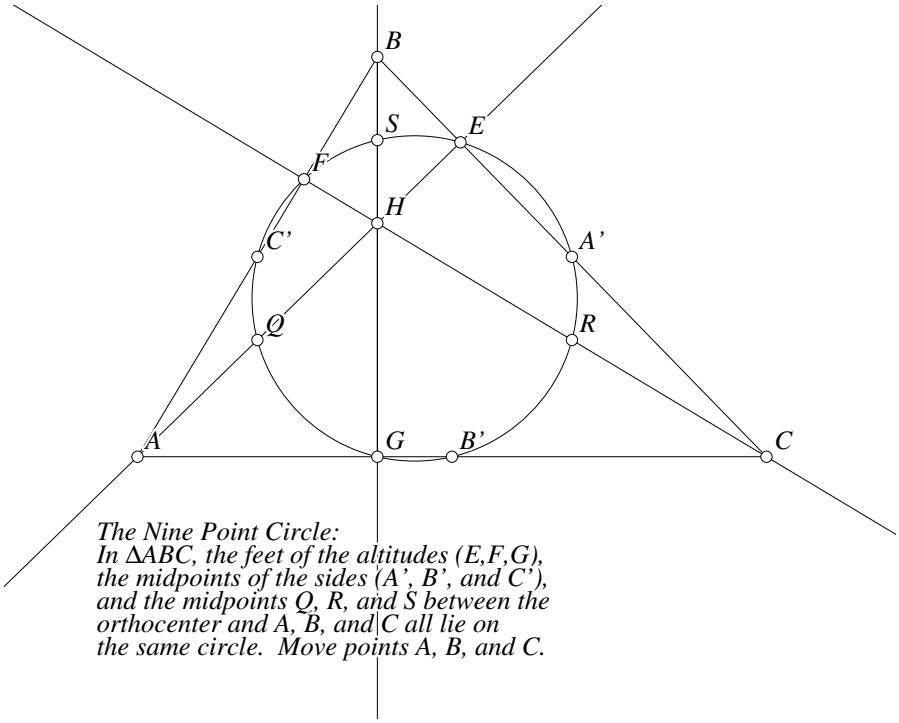

Figure 2.3: The Nine Point Circle
`Ref1/Ninepoint.T` [P]

but by golly, those nine points will stay on that circle. They may change their ordering on the circle, and the altitudes may meet outside the triangle and their feet may lie outside as well, but no matter how you move the triangle's vertices, those nine points remain locked on a circle.

This is a nice demonstration, and the more playing you do with the diagram, the more convinced you will become that the theorem is true. But you have not seen a proof. For all you know, you didn't manage to find some configuration of the points that cause the points not to lie on a circle, and in reality, your eyes (and computer calculations) are only so good. The true points may lie a millionth of a centimeter off the true circle.

Warning: if you're a bit rusty at geometry (which is a likely reason that you're using **Geometer**), don't worry if you can't follow the following proof—it depends on knowing a few facts from elementary geometry. This theorem was chosen, however, because it is pretty amazing, and the proof requires a few steps (but not too many). Even if you aren't rusty, you'll still have to think—just not quite as hard as you'd have to if you were staring at a single static figure in a high school geometry textbook.

To step through a proof of the theorem, move your mouse to the lower right part of the command area on the right side of the window, and click on the button that says: *Next.* This advances one step into the proof.

At each stage in the proof, there's usually a bit of text at the bottom of the **Geometer** window to explain what's going on, and the diagram changes somewhat. In the first stage of the proof of the nine-point property, you will simply be convinced that a particular pair of lines is parallel and that the lines are the same length. The lines in question are emphasized with a blinking color, and the text gives the geometric reasons why they must be the same length.

As you step through the proof (by pressing the *Next* key repeatedly), at any point you can manipulate the figure. While you're looking at this first step of the proof, try moving the points A, B, and C, and notice that although they move around, the blinking lines remain parallel, and that they remain the same length. (You'll also notice that not only do the lines stay parallel, but their endpoints seem to form a rectangle—a fact that will be proven later on.) It is interesting to note that one blinking line is not necessarily always above the other, as you can verify by moving the triangle's vertices. Thus if the proof had depended on the fact that $\overline{C'A'}$ was above $\overline{QR}$, it would obviously be incorrect.

Once you're convinced of the first step of the proof, press *Next* again to continue. Read the statement that appears, and convince yourself that it is also true. Then simply repeat this process all the way to the end. You can back up to a previous step in the proof using the *Prev* button, or you can start over at the beginning of the proof with the *Start* button.

Feel free to poke around and look at the other files in the various subdirectories to get an idea of the sorts of things that **Geometer** can do. Remember that all of them were created with **Geometer** itself, so you can learn to do the same.

To get a good overview of the features, here is a recommended set of diagram files to examine. Don't worry if you don't understand exactly what's going on—just observe the sorts of things that **Geometer** can do for you. The rest of this manual is to help you understand why these things work.

- `Demos/Heptadecagon.T`. This is a construction of a regular 17-sided figure (the heptadecagon) using only straight-edge and compass methods. As with the ninepoint theorem above, after loading the file, press the *Next* button to step through the construction.

- `Demos/Spirograph.T`. This example demonstrates both **Geometer**'s transformation commands and scripting commands. After loading the file, press the *Run Script* button to see a sort of spirograph drawing appear. (The **S** key is a shortcut for *Run Script*.)

  If you get bored watching the script run, click with the mouse anywhere in the drawing area. That will stop the script. If you wish to erase the junk that's been drawn so far, just click again in the drawing area.

- `Demos/Fagnano.T`. Here is an interesting proof that a certain triangle is the solution to a minimization problem. Step through the proof as before with the *Next* button, but then go back to the beginning of the proof (with the *Start* button), modify the figure so that the initial triangle has an obtuse angle (an angle larger than $90°$), and step through the proof again to see why the proof is invalid in

this case. As you step through the proof, see which statements are false because there is an obtuse angle.

- `Demos/Circ.T`. Finally, load this file and press the *Run Script* button to convince yourself that the area of a circle is given by the formula $A = \pi r^2$.

2.3 Making A Simple Drawing

OK, now that you know what **Geometer** can do, let's try to create a theorem ourselves. A nice exercise is to recreate the first example we saw in the introduction illustrating the fact that the three medians of a triangle meet in a point.

Get rid of anything on your screen using the *New* command under the *File* pulldown menu. Depending on what you did, you may have to agree to throw away your changes. Agree to anything, and after you do, you've got a clear drawing area.

First, create the three vertices of the triangle. To do this, find the *Free P* button near the upper left of the command area. Click on the little square to the left of the name, and then move your cursor into the drawing area an click down where you want the first point to appear. Do the same thing twice more (click on the *Free P* button, then click in the drawing area to place a point), and you'll have the three vertices of your triangle. By default, **Geometer** labels free points that you make as A, B, C, ... If you don't like these names, they can be changed later. For now, just leave them the way they are. The points are called "Free" in that they don't depend on anything. You can freely move them with your mouse.

Note: 99% of the time, the first thing you will do when you begin a new Geometer diagram is to create some points. All the other primitives (lines, circles, et cetera) are defined in terms of points, so if you don't have a point or two, there is almost nothing you can do.

Now if you don't like the positions where you put them down, you can move them with the mouse, exactly as you did in the previous canned examples. Move them around until they form a triangle that you find satisfying.

Next you need to make the edges of the triangle. At the upper right of the command menu is a button labeled *PP=>L*. It's under the *Line* area, so it'll create a new line, and the name indicates how: two points will determine a line. Click on that button, and then click on two different points to make the first edge of the triangle.

Do exactly the same thing to make the other two edges, but as you make them, notice what's going on in the little text window near the bottom of the command menu. When you start, you're in "Manipulation Mode" (meaning that you can manipulate the diagram by clicking down on points and dragging them), but as soon as you click on the *PP=>L* button, it changes to say "Pick Point 1". After you've successfully clicked on the first point, it'll change again to "Pick Point 2". When the line has been successfully completed, the message changes back to "Manipulation Mode". It's obvious what's going on in this simple example, but in more complicated situations, it's nice to be able to look there and see exactly what **Geometer** expects. It can even be a help in simple

situations. Suppose you want to make a point where a circle intersects a line, so you issue the appropriate command, and click on the circle, but then **Geometer** won't seem to let you click on the line. If you look in the little box, suppose it still says "Pick Circle". This might happen because you got sloppy and didn't click close enough to the circle for **Geometer** to know which one you meant.

After you've got the three points and three edges connecting them (or at any point in between), you can move the points around as long as you're not in the middle of specifying a line. (In other words, when you're in "Manipulation Mode" you can always manipulate the points.

Now we need to construct the midpoints of the sides. This can be done with a standard straight-edge and compass construction, but it's such a common operation that **Geometer** provides a shortcut; in this case, the *PP=>P Mid* command. This makes a new point that's the midpoint between two other points.

Since you're probably lazy (but probably not as lazy as the author), you're already tired of going to the command menu, clicking on a command, and then clicking to enter a new geometric object. **Geometer** provides a short cut, called "Repeat Mode" where you can repeatedly add the same type of object. Let's try it to make all three midpoints.

Instead of simply clicking on the *PP=>P Mid* button, either double-click on it or hold down the **Ctrl** key on your keyboard before clicking on the button. Note that the little yellow light goes on in the "Cancel Repeat Mode" button at the bottom of the command menu. In repeat mode, click on all three pairs of points, and each time to do so, you'll get another midpoint. After the third one, you don't want to continue making midpoints, so click on the "Cancel Repeat Mode" button to get yourself out of repeat mode. The keyboard shortcut for "Cancel Repeat Mode" is **Ctrl-g** (in other words, hold down the **Ctrl** key while you type **g**).

Note: You can also get out of repeat mode simply by clicking on another command button. If you just click on a new button, you'll be able to make one of those objects. If you double-click the new button (or hold down the **Ctrl** key when you press the new button), you'll be in repeat mode for the new command.

Now finish the theorem illustration by putting in the medians—the lines connecting the vertices of the original triangle with the midpoints of the opposite sides. For this you'll use the *PP=>L* command again, and the lazy ones among us will double-click (or use the **Ctrl** button) to do it in repeat mode.

Get out of repeat mode if you're in it, and you can illustrate the theorem by moving the vertices of the newly created triangle and noting that the three medians always seem to meet in a point.

Finally, let's add some text to the diagram to indicate what it shows. Under the *Primitives* pulldown menu, select the *Text->Descriptive Text* command. A little window appears that supports a very simple editor. Type in a description, something like: `Three medians meet at the centroid`, and then click on the *Save* button. Your text will appear in the lower left of the drawing area. You can edit this text if you wish by clicking on it (it will be underlined when selected), and then use the *Edit Text* command in the *Edit* pulldown menu. The text can have multiple lines, but can only be 1000 characters long.

2.4 A New Theorem

OK, the medians theorem is pretty easy, and you've probably seen it before, so let's get a little more practice with **Geometer** and at the same time learn an interesting fact about Euclidean geometry.

In the *File* pulldown menu, select *New* and your **Geometer** work area will be cleared (again, after you agree to throw away your changes).

Geometer's most important editing commands appear in the command area on the right side of your screen. At the top of that area are most of the geometric creation commands. Each command has a little square to its left, and clicking on that square will allow you to create a geometric object. The commands are arranged by what they produce—all the point creation commands are in one group, and all the line creation commands in another.

Make two free points in the drawing area, and try to put the first of them roughly in the center of the drawing area and the other toward the middle of the right edge. You're going to use the centered one as the center of a circle and the other as a point on its edge.

Now click the square button next to *Ctr PP=>C* under the label *Circle*. This command works like a physical compass in that you need to set the distance between the point and the pencil and identify where the point goes before you draw a circle. You're going to make a circle by choosing a point for its center and then setting the radius by identifying two points whose separation is the radius that you desire. Note that in the status window near the bottom of the command area, the text reads "Pick Center Point"—remember that you can always look in this status box to get some idea of what **Geometer** wants you to do next. In this case, you need to click on the point that you want to use as the center of the circle, so click on the point labeled "A" near the center of your window.

Now you need to pick two points whose distance apart is the radius. These will be A and B. The status window now reads "Pick Reference Point 1", so click on the point labeled A and then on point B and a circle will appear. Try moving the two free points and note that the circle changes in such a way that its center is always point A and point B always lies on its edge.

The *Ctr PP=>C* command is the most useful one for making circles. Quite often, however, the situation is what we had above—one point (A) is the center and the other (B) is on the edge of the circle. To use the command, you need to pick A, then A, then B. There is a shortcut: *Ctr Edg=>C* that allows you to pick the center and then a point on the edge so that drawing this common type of circle is a bit faster, but be sure to remember the most powerful method: *Ctr PP=>C*.

Now use the *P on C* (point on circle) command under the label *Point* and click anywhere on the edge of your circle. A new point will be created there. Try to move this point with the mouse, and note that it is constrained to stay on the edge of the circle, but otherwise can move freely. Finally, use the command *PP=>P Mid* (also under *Point*) and pick point B and then the point constrained to be on the edge of the circle. This

command creates a point that will always be the midpoint between the two points. (You may want to draw the line segment between B and C to see that the midpoint (D) is on the line between them, but do what you like.)

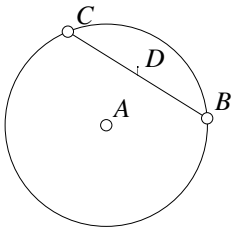

Figure 2.4: Midpoint
`Ref1/Midpoint.T` [M]

At this point, you should have a drawing that looks something like what is illustrated in Figure 2.4.

Now let's learn some geometry. Click down on the point that's constrained to be on the edge of the circle (the one labeled C—not the one labeled B), and move the mouse around and around the circle, watching what happens to the midpoint D. Notice that the midpoint seems to go around also, and seems to sweep out a circle of its own.

We can make this even more obvious by clicking on the midpoint D to select it (notice that when selected, it is displayed with bolder lines). Then, using the *Color* selector button in the command area, change the color to *Smear* and the midpoint's color will change to a medium shade of gray.

Click down on the point labeled C constrained to be on the edge and move it around the circle again. This time, the midpoint will smear and you can see that all its valid positions seem to lie on a smaller circle passing through points A and B. You should get a picture something like that in Figure 2.5.

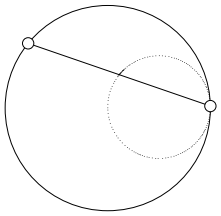

Figure 2.5: Circle-Point Midpoint Theorem
`Ref1/Smearmidpoint.T` [M]

Let's try a couple of other things—click on the *PP=>P Mid* command and create a new midpoint between your smearing midpoint and the point constrained to the circle's edge. Make that have the smear color as well, and again move the circle-constrained

point around. Both midpoints sweep out circles, but the new one is $3/4$ the size of the original circle.

Does it matter that the fixed point is on the circle? Can you construct a **Geometer** diagram where the other end of the segment is not B on the circle's boundary, but is somewhere else, either inside or outside the circle? What is the general situation? Can you figure out why?

2.5 Modifying Your Simple Drawing

OK. Your drawing is pretty boring—except for the smearing, it's all one color, and you only used a few different commands to make it. Feel free to play around with the other commands to see what you can figure out. You can't get into serious trouble, and the only recommendation is to avoid the *Edit Geometry* command in the *Edit* pulldown menu. The *Rpt Set Color* button may be a little mysterious as well.

One of the most useful commands is found under the *Edit* pulldown menu—*Edit Name* (This is such a commonly used command that you can also access it with **Ctrl-n**—hold down the **Ctrl** key while you type an "n"). Select a point or a line in your drawing by clicking on it and issue the *Edit Name* command. A little box pops up with the old name (if it had one), and you can edit it or type in a completely new name. (If it has a name that you want to get rid of, just backspace away the name when the dialog box comes up. The point or line will then be unlabeled.)

Although only certain primitives display their names on the screen (points, lines, and angles[‡]), you can name anything, and this is sometimes useful because the *Describe Geometry* command will use this name when it tells you what's selected. If you don't give something a name, **Geometer** will make up a name for you, and **Geometer** is not very imaginative. Its point names all look like `A`, `B`, ..., `Z`, `a`, `b`, ..., `z`, `1`, `2`, `3`, `4`, ...

Click on things to select them, and try changing their colors, point type, and so on. Obviously, you can't change the point type of a line, or the angle type of a point, but most commands make sense.

Basically the name on every button in the upper part of the command area includes an indication of what it makes, and what's required to make it. The basic primitives that you can make include points, lines, circles, polygons, angles, and conic sections. The command *CC=>P* says that two circles are required to make a point at their intersection. A lot of the commands do the "obvious" thing: *PP=>L* gives a line passing through two points, *LC=>P* gives the point at the intersection of a line and a circle (which you must click on in that order—line first, then circle), et cetera.

Sometimes there's additional information. We saw the "*Mid*" in *PP=>P Mid* that found a point that's the midpoint of two others; *PL=>L Perp* generates a line that's perpendicular to a line and passes through a point, and *CC=>Ext Tan* yields a line (it's in the *Line* section) which is the external tangent to two circles.

[‡] And "flts" which will be discussed later

Let's take a slightly closer look at the *CC=>P* command which makes a point at the intersection of two circles. But which one? Circles usually intersect in either two places or none. How does this command work? Try it out, or look ahead in the next section where all the commands are described in detail to get the details.

Keep in mind that while you're fiddling with commands, you can always look in the little command feedback window to see what **Geometer** needs next. The command feedback window is just above the *Cancel Repeat Mode* button. (Even then, you can get into trouble. For example, if you've got four points in your drawing and you click on the *5P=>Con* command, you will then need to select 5 *different* points. You can click on the first four with no problems, but **Geometer** will keep asking you to "Pick Point 5" no matter what you do, since it insists on having different points for that particular command. **Geometer** is currently too stupid to look ahead to make sure there's enough stuff defined to allow a command to succeed, so if you get stuck, you can simply click on a new command in the middle of trying to execute the impossible command to abort it.)

If you start a new drawing using the *New* command in the *File* pulldown menu, it's almost certain that the first commands you'll issue will be a few *Free P* commands. All the others require some geometry to be there already (a line needs two points to connect, a circle needs a center and an edge, or three points on its outline, et cetera). Similarly, before you can issue the *LC=>P* command (that makes a new point at the intersection of a line and a circle), you'd better have a line and a circle on the screen. If they don't intersect, that's fine—try making a circle and a line that doesn't go through it, issue the *LC=>P* command and click on the line and circle as you are asked for them. No point appears, but as you move the line across the circle (or the circle across the line, of course), the new point will suddenly appear. Now, for fun, connect that "ephemeral" point with another to make a line. As you move the original circle and line to make them intersect or not, both the ephemeral point and the line based on it will appear and disappear. This may seem strange, but trust me—this is a good thing!

Another "good thing" that may seem disturbing at first is that **Geometer** considers every line to be infinitely long, even though it may be displayed as a short segment (the line type can be "segment", "ray", or "line"). If the extension of the segment hits the circle, you'll get the point at the intersection. **Geometer** does try to avoid selecting a line except when you click on a drawn part of it, however. Make a line between two points that's of line-type "Segment" and click near it to see where it's sensitive to selection. Then change it to line-type "line" and notice that you no longer need to click between the two points to select it.

If you're short on ideas to try, here's something. Make a demonstration of the fact that the three altitudes of a triangle also meet in a point. An altitude is a line that drops from a point perpendicularly to the opposite side of the triangle. You'll probably want to use the command, *PL=>L Perp*. It takes a point and a line, and constructs the line through the point and perpendicular to the line.

Or just see what you can do in the way of generating geometric art by making some constrained linkage with some of the moving parts drawn in the "Smear" color.

2.6 ♦Using The Text Editor

Before we go on to cover more of the common commands, here's one more thing you should try. Make some sort of simple drawing that includes at least a few lines and points, and for fun, change the colors of a couple of them—in fact, make the first point you create red (it should have the name "A" if you started with the *New* command). Don't make things too complicated either—include at most a dozen objects so it's easy to see what's going on.

You're going to edit the textual version of your diagram. This is more "exciting" in the sense that you'll now have an opportunity to really foul things up, but remember that if you get into trouble, you can always delete all the text in the file and save the empty file, and you'll at least be back to an empty drawing.

Under the *Edit* pulldown menu, select the *Edit Geometry* command. A small text editor window will appear that contains some strange-looking text. Take a look at it—it's just a textual description of the drawing.

Assuming you used the buttons from the command menu, every line (except for the first that contains **Geometer** version information) represents an object on the screen. Your red point named "A" will have an entry (probably on the second line of the file) that looks like this:

```
v1 = .free(-0.420784, 0.212947, .red, "A");
```

The numbers represent the current coordinates on the screen, so the ones in the example line above will be different from yours, but the rest should look about the same. "`v1`" is the internal name of the point, "`.free`" says that it is a free (unconstrained) point that you can move with your mouse, the coordinates tell where it is in the x and y directions (the initial screen runs roughly from -1.0 to 1.0 in both dimensions), the "`.red`" tells the color, and the "`"A"`" says that the point is to be displayed on the screen using the name "A".

Now, using the editor (it's a simple, mouse-based, cut-and-paste editor with nothing fancy about it), change the line to look as follows, with the "`.red`" changed to "`.magenta`", the name "`A`" changed to "`Fred`", and both of the coordinates are changed to 0:

```
v1 = .free(0,0, .magenta, "Fred");
```

Use the *Save* command in the editor's *File* pulldown menu and the results will appear on the screen. The new point will be colored magenta, it'll be labeled "Fred" instead of "A", and it will appear in the exact center of the drawing area, since you changed its coordinates to $x = 0$ and $y = 0$.

Other than that, everything is exactly as it was, and you can still click on the "Fred" point and move it around. After moving it, issue the *Edit Geometry* command and you'll see that your changes are still represented—the point will still be called "Fred" and it will still be magenta, but (since you moved it), the coordinates will probably no longer be 0 and 0.

Let's try a couple of other things with the editor before we continue with **Geometer**'s basic features. Edit the geometry again (using the *Edit Geometry* command in the *Edit*

pulldown) and add a name to one of your lines as follows. Suppose the line description originally looked like this:

```
l1 = .l.vv(v2, v1);
```

Change it to this:

```
l1 = .l.vv(v2, v1, "Line");
```

Finally, type an entirely new line like this after the end of all the other lines in the file displayed in the text editor window:

```
.text("Here's some displayed text.", .green);
```

(Be sure to get all the punctuation right, including the periods in the initial "`.text`" and the "`.green`".)

When you save the file, the line will have the name "Line" next to it, and "Here's some displayed text." will appear in green on the left of your window's drawing area.

If you make a typing error, **Geometer** is pretty stupid about figuring out exactly what's wrong, but it'll give you some clue in an alert window, and will then return you to the editor with the first bad line highlighted. Or at least what it thinks is the bad line—it may be reading the next line before it notices that you've left something out of the previous. If this happens, find the error, fix it, and save again. If you've been doing a lot of editing and have made a lot of errors, **Geometer** will only find one at a time, and will keep popping up that editor window until you've fixed them all.

If you really foul things up and you just want to bail out to the situation before you tried to edit the file, use the *Quit* command in the pulldown menu.

2.7 An Easy Example

Let's make a diagram that allows you to specify the lengths of three segments. The diagram then constructs a triangle whose sides are equal to those segment lengths.

We would like a diagram like that in Figure 2.6 but without those extra circles. The diagram contains the three segments in the upper left whose lengths we can adjust with the mouse, and the resulting triangle is displayed below. If we can get a diagram like the one displayed, we can simply make all the circles the invisible color and we will have exactly what we want.

To make the drawing in the figure, first create six points to control the lengths AB, CD and EF of the sides of the desired triangle. Connect them with line segments if you wish. Next, create an arbitrary free point in the drawing area (which is called G in the figure) to be one vertex of our final triangle. If we draw a circle around G with radius equal to AB, then we can pick any point on that circle (H in the diagram) and that point will be a distance AB from G. In other words $GH = AB$. This is a good way to copy a distance.

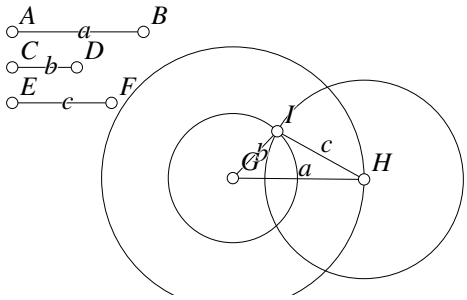

Figure 2.6: Triangle from Three Segments
`Ref1/TriConstruct.T` [M]

To draw the circle, use the *Ctr PP=>C* command. Pick G as the center and then pick A and B as the two reference points. The circle drawn will thus have center G and radius equal to the distance between the reference points, or AB in this case.

Next, use the *P on C* command and click anywhere on the circle to create H. Connect G and H with *PP=>L*.

Now create circles about G and H with radii CD and EF, exactly as we just did. At the point of intersection of these circles must lie the third vertex of the triangle, since it will thus be CD away from G and EF away from H. To find the intersection of two circles use the *CC=>P* command. This will create vertex I which we then connect to vertices G and H to complete our triangle.

We can now adjust the lengths of the segments by moving the points A, B, ..., F.

2.8 An Easier Example

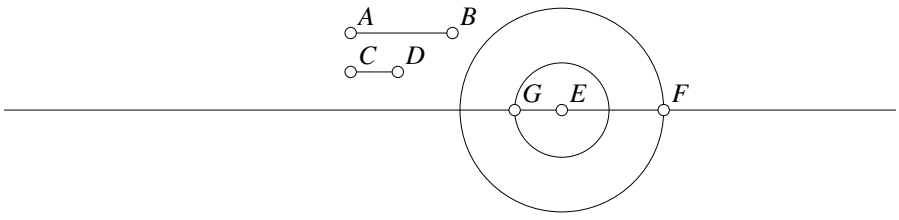

Figure 2.7: Add Two Segment Lengths
`Ref1/Addsegs.T` [M]

The second example is even easier, especially if you have just seen the previous example. The problem is this: construct a **Geometer** diagram where you can adjust the lengths of two segments and the diagram will display a new segment whose length is

the sum of the lengths of those two. It will look something like the previous example, but with only two adjustable segments above, and one final segment as the result as displayed in Figure 2.7.

The solution is quite simple. Begin as you did before by making two segments AB and CD with four free points. You will be able to adjust these lengths in the final diagram. Next, create a free point in the drawing area and make two circles around it of radii AB and CD. Choose F on one of the circles, draw a line (not a line segment—if **Geometer** draws a segment, click on the *Line Type* button in the command window and select *Line* as the type) through F and E, and (using *LC=>P*) find the intersection of that line with the other circle at the point G on the other side of E from F.

Clearly GF is the required segment. The length EF is the same as AB and EG is the same as CD. But $GF = GE + EF$, so we are done.

2.9 A More Difficult Example

he problem in this section is a bit trickier, but if you have paid attention to the last two sections, it is not that difficult.

In this section we will consider an example that is a bit more difficult. The actual construction will depend on a geometric observation, and the construction itself is also a bit more complicated.

The way to learn the most is to try to draw the example yourself without reading more. Then, whether you succeed or not, read the description here, since the problem is not only solved, but the figure is modified in such a way to make it a useful for teaching the concept in a class.

2.9.1 Tangent Circles Problem

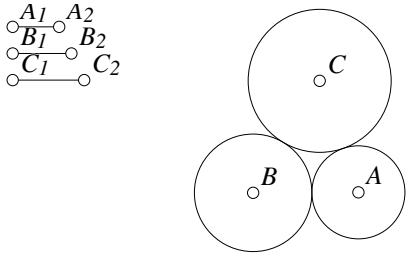

Figure 2.8: Three Tangent Circles
`Ref1/TangentCircs.T` [M]

Here is the problem: Given three lengths, construct a set of three circles that are mutually tangent and have those three lengths as their radii. In figure 2.8 you can adjust the

three lengths by moving the points labeled A_2, B_2 and C_2 at the top of the diagram. As you do so, the three circles change size appropriately to maintain their tangency.

Without reading further, try to make this drawing. As a hint, think about where the centers of those circles must be.

2.9.2 The Solution

The key observation is that if the radii of the three circles centered at A, B and C are to be a, b, and c, respectively, then the centers A and B must be a distance $a + b$ apart, B and C must be $b + c$ apart, and A and C must be $a + c$ apart.

Thus the solution is pretty easy. First, we need to construct line segments that have the lengths $d_1 = a + b$, $d_2 = b + c$ and $d_3 = c + a$. Then we need to construct a triangle having these three segments as lengths. The vertices of this triangle will serve as centers for the three circles.

It is easy to construct a line segment that is the sum of two segments—just draw a line and from a single point, draw circles having radii equal to the two lengths. The points on opposite sides of the single point where they intersect the line will be separated by a length that is the sum of the two lengths.

In this way the three lengths of the desired triangle could be constructed. Once we know the lengths d_1, d_2 and d_3, it is easy to construct the triangle. Draw circles of radii d_2 and d_3 about the two ends of a segment of length d_1. The point where those circles intersect is d_2 from one end and d_3 from the other, so we are done.

Being lazy, we would like to minimize the number of times we have to copy lengths, so here is a reasonable way to do the construction. Assume that you have the three lengths a, b and c laid out as in Figure 2.8. Draw a new line with a point "P" on it. Using P as the center, draw circles of radius a and b. (This can be done with the *Ctr PP=>C* command. This command first takes the center of a circle and then uses the distance between the two points you select as the radius of the circle. In **Geometer** you simply need to select V as the center and then the endpoints A_1 and A_2 of the length-defining segment as your radius to obtain the first circle, for example.)

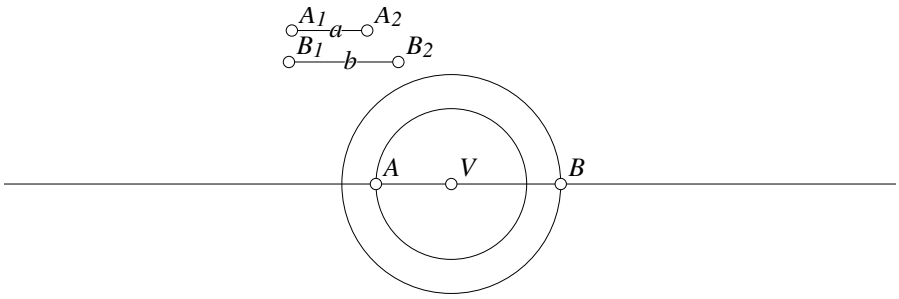

Figure 2.9: Adding Radii
`Ref1/Tan1.D` [D]

On one side of P find the intersection of one of the circles and on the other, the intersection with the other. The distance between these two intersection points will be $a+b$. See Figure 2.9.

Now we have identified A and B on our triangle, and they are clearly separated by $a + b$.

Now draw a third circle about V of radius c. It should be clear that the distance from B to the intersection of that line with the circle on the other side of V from B is $b + c$ and that the distance from A to intersection of that circle on the other side of V from A is $a + c$. Thus we can draw two circles centered at A and B having edge points at the two intersections with the line of the circle of radius c centered at V. The intersection of these circles yields the location of the point C.

To complete the diagram, draw circles of radius a, b, and c about points A, B and C, respectively. To clear up the clutter, you probably will want to change the color of all your auxiliary lines to the invisible color.

At this point the problem is solved, but suppose you would like to convert your diagram to a beautiful one that would be more suitable for a classroom presentation. Load the diagram `Ref1/TangentCircs.T` and play with it as you read these notes. Here are some things that you might do:

1. You may want to change the names of the points on the controlling segments to $A_1, A_2, \ldots, C_2$ to make it clear to the viewer that the segment A_1A_2 controls the length of the radius of the circle centered at A. You can also name the segment A_1A_2 `a`, et cetera, to make the correspondence even more clear.

2. If A_1 and A_2 are free points, as you drag them around, the figure will certainly work, but it is much more pleasing if you can just drag A_2 and it moves on a line parallel to B_1B_2 and C_1C_2. To do this in the diagram, A_1, B_1 and C_1 are pinned points. Pinned points cannot be moved. To create a pinned point, look in the *Primitives* pulldown menu under *Points*.

   Next, another point is chosen (pinned and invisible) that determines the three horizontal lines from A_1, B_1 and C_1. The points A_2, B_2 and C_2 are constrained to lie on those lines. Then visible line segments are drawn from A_1 to A_2, et cetera.

3. Color-coordinate the drawing. All the lengths and points associated with A are one color, with B another, and so on.

4. To make it completely clear that the radii of the final circles are equal to a, b and c, we need to draw those as well. AV and BV can be drawn easily, but we need to find the intersections of the lines AC with the circles and BC with the circles. Draw those lines, find the intersections using *LC=>P*, make the lines invisible, and then draw the shorter segments to those intersections in the appropriate colors.

5. Add some text to the diagram to explain what is going on with the *Text* command under the *Primitives* pulldown menu.

6. Finally, to assure that the lines A_1A_2, B_1B_2 and C_1C_2 are parallel and evenly spaced, use the text editor to modify the coordinates of the pinned points so that the lines are perfectly horizontal and evenly spaced.

Here is a complete listing of the **Geometer** code to produce the diagram:

```
.geometry "version 0.60";
v1 = .pinned(-0.78, 0.85, .red, "A\sub{1}");
v2 = .pinned(-0.78, 0.7, .green, "B\sub{1}");
v3 = .pinned(-0.78, 0.55, .cyan, "C\sub{1}");
v4 = .pinned(0.78, 0.85, .in);
v5 = .pinned(0.78, 0.7, .in);
v6 = .pinned(0.78, 0.55, .in);
l1 = .l.vv(v1, v4, .in);
l2 = .l.vv(v2, v5, .in);
l3 = .l.vv(v3, v6, .in);
v7 = .vonl(l1, -0.452096, 0.85, .red, "A\sub{2}");
v8 = .vonl(l2, -0.598802, 0.7, .green, "B\sub{2}");
v9 = .vonl(l3, -0.38024, 0.55, .cyan, "C\sub{2}");
l4 = .l.vv(v1, v7, .red);
l5 = .l.vv(v2, v8, .green);
l6 = .l.vv(v3, v9, .cyan);
v10 = .free(-0.517964, -0.0898204, .in);
v11 = .free(-0.0269461, -0.0898204, .in, "V");
l7 = .l.vv(v10, v11, .in, .longline);
c1 = .c.ctrvv(v11, v1, v7, .in);
c2 = .c.ctrvv(v11, v2, v8, .in);
v12 = .v.lc(l7, c1, 1, .red, "A");
v13 = .v.lc(l7, c2, 2, .green, "B");
c3 = .c.ctrvv(v11, v3, v9, .in);
v14 = .v.lc(l7, c3, 1, .in);
v15 = .v.lc(l7, c3, 2, .in);
c4 = .c.vv(v12, v15, .in);
c5 = .c.vv(v13, v14, .in);
v16 = .v.cc(c4, c5, 2, .cyan, "C");
c6 = .c.ctrvv(v16, v3, v9, .cyan);
c8 = .c.ctrvv(v13, v2, v8, .green);
c9 = .c.ctrvv(v12, v1, v7, .red);
l8 = .l.vv(v12, v16, .in);
l9 = .l.vv(v16, v13, .in);
v17 = .v.lc(l8, c6, 1, .in);
v18 = .v.lc(l9, c6, 2, .in);
l10 = .l.vv(v12, v11, .red);
l11 = .l.vv(v12, v17, .red);
l12 = .l.vv(v17, v16, .cyan);
l13 = .l.vv(v16, v18, .cyan);
```

```
l14 = .l.vv(v18, v13, .green);
l15 = .l.vv(v13, v11, .green);
.text("Given three lengths, construct three mutually
tangent circles having those three lengths
as their radii.");
```

Chapter 3

Geometer Usage Strategies

> Since you are now studying geometry and trigonometry, I will give you a problem. A ship sails the ocean. It left Boston with a cargo of wool. It grosses 200 tons. It is bound for Le Havre. The mainmast is broken, the cabin boy is on deck, there are 12 passengers aboard, the wind is blowing East-North-East, the clock points to a quarter past three in the afternoon. It is the month of May. How old is the captain?
>
> *Gustave Flaubert*

3.1 Using Diagrams

The use of diagrams is essential to understanding geometry, but paradoxically, they can almost never count on them for proof. The diagrams primarily provide intuition.

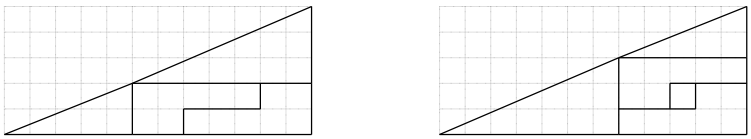

Figure 3.1: Misleading Diagram
`Ref1/Badtriangle.D` [D]

Everyone has seen examples like that in Figure 3.1, where one geometric object (in this case the 5-12-13 right triangle on the left) is cut into pieces that are rearranged (on the right) to form the same object, but missing a piece. Where did the extra area go? (Hint: There are three triangles in each of the two figures—are they all similar?)

But more disturbing things can happen, especially if the drawings are not accurate. Consider Figure 3.2, which can be used to "prove" that a right angle is greater than $90°$.

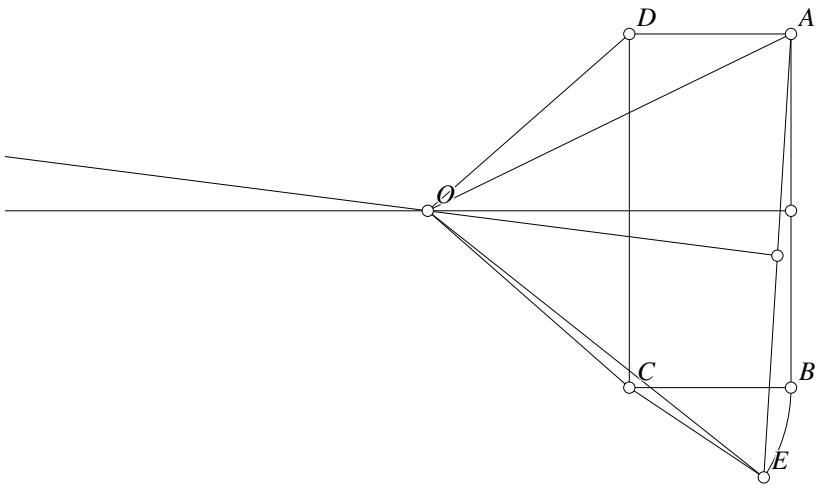

Figure 3.2: Bogus Proof
`Ref1/Incorrect.D` [D]

Start with an arbitrary rectangle $ABCD$ as in the figure, and using a compass, draw an arc of a circle centered at C, and passing through B, that goes a little outside the segment BC—to point E.

Since $ABCD$ is a rectangle, $\angle ADC = \angle BCD = 90°$. Since $\angle BCE$ is a little bigger than zero, $\angle DCE$ is a little bigger than $90°$. We will "prove" that $\angle ADC = \angle DCE$—something that is obviously not true.

Here is the bogus argument. Construct the perpendicular bisectors of the segments AB and AE. Since those perpendicular bisectors are clearly not parallel, they will meet at some point O.

Since $ABCD$ is a rectangle, the perpendicular bisector of AB is also the perpendicular bisector of DC, so all the points on it are equidistant from D and C. Therefore $DO = CO$. By similar reasoning, O is equidistant from A and E, so $AO = EO$.

Points B and E are on the same circle centered at C, so $BC = CE$, and since $ABCD$ is a rectangle, $AD = BC = CE$.

So to summarize, $DO = CO$, $AD = CE$, and $AO = EO$. Therefore the two triangles ADO and ECO are congruent since they share three equal pairs of sides (using SSS congruence). Therefore $\angle ECO = \angle ADO$, and we can subtract the equal angles $\angle CDO$ and $\angle DCO$ to obtain the result we want—that $\angle ADC = \angle ECD$.

Well, it is clearly not true, but every step seems correct. What's wrong? The answer is that the diagram is not drawn as accurately as it could be. (In fact, the author cheated and had to misuse **Geometer** to get the desired misleading effect.)

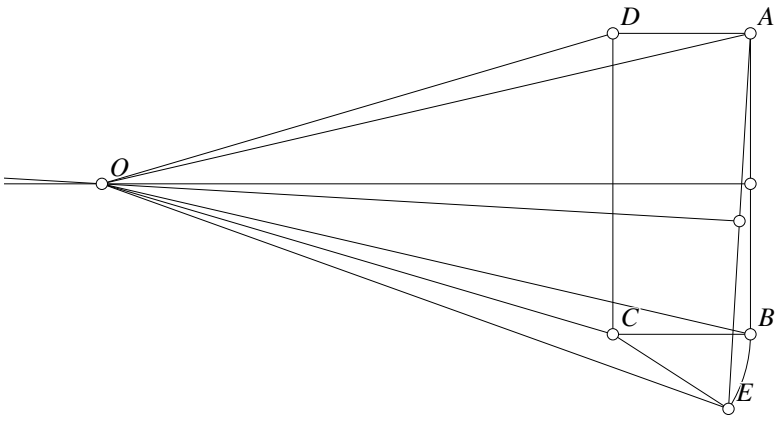

Figure 3.3: Correct Figure
`Ref1/Correct.T` [M]

Figure 3.3 shows an accurately drawn diagram of the situation, and it is instantly obvious what went wrong—the line segment EO lies on the outside of the rectangle. In fact, if you check in the correct figure, $\angle ADO = \angle ECO$, which is what we proved.

The point is that an accurate figure can correct some very mysterious problems, and computers allow you to draw extremely accurate figures without too much effort.

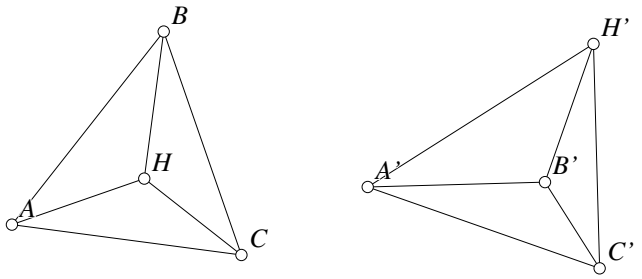

Figure 3.4: Possible Orthocenter Positions
`Ref1/Orthoerror.T` [M]

Other dangers arise because a single figure may not show all of the possibilities. A proof of some property of the orthocenter of a triangle (the orthocenter is where the three altitudes meet) must work whether the orthocenter lies inside or outside or on the triangle. If a proof depends on an interior orthocenter, it is not necessarily valid.

Figure 3.4 illustrates both possibilities for triangles $\triangle ABC$ with orthocenter H and $\triangle A'B'C'$ with orthocenter H'. Suppose that only the diagram on the left were considered, and one step in the proof were something like this:

Connect the orthocenter H to the three vertices of the triangle forming three smaller

triangles. The sum of the areas of those triangles is the area of the original:

$$\mathcal{A}(\triangle AHB) + \mathcal{A}(\triangle BHC) + \mathcal{A}(\triangle CHA) = \mathcal{A}(\triangle ABC).$$

Clearly that is not true for $\triangle A'B'C'$ on the right of Figure 3.4, so this is a nice example of an error that can be made based on an incorrect diagram*.

(On the other hand, we do see something quite interesting in the pair of diagrams in the figure. Notice on the right that B' appears to be the orthocenter of $\triangle A'H'C'$. Is that always true?)

Finally, a great example of where a theorem simply is not true with a different figure is illustrated by Fagnano's Theorem. Fagnano's theorem states something about an acute-angled triangle that is simply not true if the triangle contains an obtuse angle. If the drawing is made with a triangle containing an obtuse angle, it is clear that the theorem fails, but if the "proof" is based only on a drawing of an acute-angled triangle, the "prover" may not notice that the theorem does not always hold.

Here is another nice example where a computer can be very helpful. Consider the following "theorem":

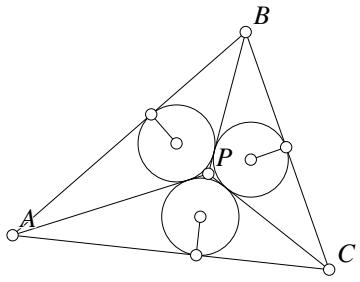

Figure 3.5: Equal Inscribed Circles
`Ref1/Badcircs.T` [M]

In $\triangle ABC$, point P is chosen in the interior of the triangle and lines PA, PB, and PC are drawn. If the inscribed circles of triangles $\triangle APB$, $\triangle BPC$, and $\triangle CPA$ have equal radii, show that P is the incenter of $\triangle ABC$ (see Figure 3.5).

It is tough to prove because it is not true! But for small triangles, it is *very* close. In Figure 3.5, when P is the incenter, the differences of the radii are around one part in a thousand. This is very difficult to see with your eyes, but a computer geometry program shows the error instantly.

These last three examples provide even more reason to use computer drawn figures, where the figures can be manipulated to see if the theorem is true in all cases, and if the proposed proof is valid for all cases.

*With signed areas (positive and negative areas), it may be possible that the proof continues to hold, depending on the rest of the proof.

3.2 Basic Techniques

Here is a list of things to try when presented with a new geometry problem:

- Draw an accurate figure. With **Geometer** or another computer geometry program, this is easy. If there are different configurations possible, draw those as well.

- Look at the figure sideways and upside-down. For example, many people are conditioned to thinking of the base of a triangle as the edge that is parallel to the bottom of the page, and is closest to the bottom of the page. That is because when the definition of the base of a triangle was presented, it was drawn that way in the textbook, and the English word "base" itself implies "bottom". But *any* side of a triangle can be its base. So if a base and an altitude of a triangle are needed to calculate its area, do not get locked into the mental frame of mind that the base is on the bottom and the altitude goes up and down—every triangle has three bases and three altitudes and any pair can be used equally well to calculate the area.

  In a similar way, right angles need not have their sides parallel to the edges of the textbook. It looks better that way, so figures in books are usually so drawn, but in real problems, triangles can be oriented in any which way. In fact, sometimes when you finally find that right angle that cracks the problem, you will swear that it was deliberately hiding itself from you.

- Do not stop looking after you find something. For example, if you are trying to solve a problem and you find a couple of triangles that you can prove congruent or similar or something, remember that there may be more than one pair. If you cannot solve the problem with the first set, maybe there is a second (or a third, or forth, ...).

- If you do not seem to be making progress, make sure that you are trying to use all the given information. For example, if a point lies on the circumcircle of a triangle, how can you use that fact? You know that the point and the vertices of the triangle are all equidistant from some point (the circumcenter). Is that useful?

- Remember to try indirect proofs. Assume the theorem is false and see if that leads to some preposterous result. Many important theorems in mathematics (geometry included) cannot be proven directly and must be proven by showing that the failure of the theorem leads to a contradiction.

- Look at simple situations. For example, if you are trying to prove some property of general polygons, it is best to check that they work on triangles and quadrilaterals. If it is a result about triangles, look at simple triangles—right triangles or equilateral triangles, or 3-4-5 triangles.

- Do a complete calculation for simple cases. For example, if you are trying to show that the medians of a triangle meet at a point that is $2/3$ of the distance

from the vertex to the opposite side, do the calculation by hand on an example where you know everything—perhaps an equilateral triangle, or a $45°–45°–90°$ triangle.

- Think about similar problems you have seen and how they were solved.
- Mentally summarize techniques that are used to do that particular sort of problem. For example, if you need to show that four points lie on a circle, think of the two or three standard ways to show this.

3.3 Using Computers Effectively

This section deals with techniques that are particularly well-suited to geometric investigations using computer geometry programs, including **Geometer**.

Assuming that you are not just fooling around and have a specific problem to work on, there are plenty of things you can do to simplify your search for a solution. Here are some of them:

- Draw an accurate diagram. Learn to use your computer program to draw the figure exactly as it appears in the problem statement.
- Use features like color and line-stippling to keep track of what you are doing. For example, if you need to draw some auxiliary lines that you know you will change to the invisible color later, color them red while you need to work with them. Similarly, if you draw all three altitudes of a triangle, color them all the same, but in a different color from the other lines in the figure.
- Move any points that are not constrained and see how the diagram moves, just to get a feel for why it is true. Or perhaps it is not true in which case your job is much simpler.
- Test the limits of the diagram to see what happens. In other words, if the theorem you are trying to prove is about all triangles, be sure to try weird shapes as well as the common ones. Make it equilateral (approximately, of course). Make it a right triangle. Make the sides as different as possible. Make the triangle long and skinny. Try triangles with all acute angles. Try triangles with an obtuse angle near $180°$.
- If possible, go beyond the limits and see why the theorem fails. For example, suppose the theorem deals with a simple quadrilateral (where "simple" means that the edges do not cross each other). First make it so the lines *almost* cross (test the limits). Then make them cross and see why (or if) the theorem fails. Surprisingly, some theorems continue to work. A great example to try for this is to prove that if you connect the midpoints of a quadrilateral in order, the resulting figure is a parallelogram. It turns out that this is true even if the edges of the original quadrilateral cross each other.

- Another way to go beyond the limits is to omit each of the conditions of the theorem and see why it fails. If the theorem says something about a right triangle, see what happens with a triangle that does not contain a 90° angle. Sometimes you may be surprised to learn that a condition is not necessary.

- As you are modifying the figure, look for other patterns. Some are obvious—do two lines seem to remain parallel? If so, maybe you can prove they are and that fact will help. A bit harder to check is to see if some pair of angles stays the same. If you know which angles you are looking for this is not hard, but if you are just groping for a way to start, there are usually a lot of possibilities.

- It is hard to check some things like "Do these four points remain on a circle?" Luckily, you can construct a circle through any three given points. Pick three of the points, draw the circle through them, and then modify the diagram to see if the fourth point stays on the circle with the other three (at least to the accuracy of your eye).

- Similarly, you can check to see if three points seem to lie on a line by drawing the line through two of them and playing with the diagram to see if the other point also seems to remain on that same line.

- If an experiment like one of the two examples above fails—the four points do not stay on a circle, or the three points do not stay on the line, be sure to delete the bad circle/line before proceeding so that you do not get confused.

- If you are a power user, you can often have the program display lengths, ratios, areas, et cetera, so you can see, for example, if a ratio of lengths seems to stay constant or not.

- ♦ If you are thinking about using inversion to solve a problem, it is easy to draw a circle and invert the objects of interest in it. Then you can try moving that circle around to see if any configurations make the inverted drawing easier to work with.

- Remember to use different colors, styles, et cetera, to keep things straight. In a complex diagram if you are looking at four points to see the interactions between them and you are afraid you will lose track of exactly which points they are, paint them all green. You can always change them back to more reasonable colors later.

- Do not feel compelled to use the computer program—it is just another tool, and not necessarily the best tool. It is often easier to doodle with pen on paper if high accuracy is not required. It is also sometimes a good idea to use the computer to create a good drawing and then to print one or more copies of high-quality versions of the problem on paper, on which you can doodle with your pen.

Chapter 4

Basic Reference

This chapter of the reference manual lists all the most common commands for **Geometer**. All of them are accessible using the GUI (graphical user interface). (Of course all of them are also accessible via the text editor.)

4.1 Geometer's Philosophy

Now that you've been led by the hand through a couple of examples, you probably have a pretty good feeling for how **Geometer** operates. Thus as new commands are introduced, you probably won't find it necessary to try every one of them, but do try a few—especially the ones that are particularly confusing or interesting.

Geometer is basically a simple constraint solving system. Each new item that appears in a **Geometer** diagram must be either an unconstrained element (like a free point), or it must be based on previously defined objects. The only geometry that's completely free is the free point, so most diagrams will be based on one or more of those.

Other unconstrained objects that can appear include text and numbers, but nothing depends on text, and numbers will be discussed in the advanced section.

In addition, there are some partially constrained points—a point that can move, but is constrained to lie on a line, a circle or a conic section, for example.

If the theorem you're trying to illustrate requires an unconstrained line or circle, just construct that line or circle from free points in one of the usual ways, and move or reshape the line or circle by moving the free points. (There are other ways to do this, but again, that's an advanced topic.)

For every redisplay of the current diagram, **Geometer** re-evaluates the entire file one line at a time, in order. (Actually, it's quite a bit cleverer than that, but the net result will always be the same as if it had done exactly this—except that the technique **Geometer** uses will get a redisplay a lot faster.)

4.2 Entering Geometry

This section is a description of all the commands you can use to enter geometric primitives. Of these, the most commonly used commands can be found as buttons in the upper part of the command menu on the right side of your window, but there are many more, all of which are listed in the pulldown menus under *Primitives*. In every case, if you click on the command's button, or use the pulldown menu, **Geometer** will produce a single instance of the primitive. If you hold down the **Ctrl** key when using the command menu or pulldown (you can't double-click these), **Geometer** will go into repeat mode for that command and you can make as many instances of it as you like. To exit repeat mode, use the *Cancel Repeat Mode* button at the bottom of the command menu (or type *Ctrl-g*), or click on another command without using the **Ctrl** button.

As soon as a new item is created, it is automatically selected. This is because the property-setting commands (like color) affect the selected item, and it's very common to create an item and decide immediately when you see it that you want it to be a different color or have a different style. Since newly-created items are selected, you merely need to click on the correct new color button to change the color of a new point, or click on the *Point Type* button to change how it looks on the screen.

This is not a list of all the constraints—there are additional constraints based on non-geometric primitives, but they are discussed in the advanced sections. All the commands below are accessed with the GUI (graphical user interface); the other constraints must be entered with a text editor.

4.2.1 Point Creation Commands

Here are the ways to make new points from the graphical interface. Remember that many of them appear only in the *Primitives* pulldown menu. When new points are created, they automatically have a name generated, starting with "A", then "B", and so on. There's a command in the *Edit* pulldown menu to toggle this on and off. The command is called *Point Names* and has a keyboard shortcut of **Ctrl-t**.

Free P. Click anywhere in the drawing area to create a completely free point that can later be moved by the mouse. It will have no constraints whatsoever on its movements.

P on L. Click on a line, and a new point will be created that is constrained to lie on the given line. If you move the point with your mouse, it will stay locked on the line. If you move the line (by moving other points upon which it depends), the partially constrained point will move in such a way that it remains on that line. In fact, it is simply projected to the new line (moved to the closest point on the line) over and over as the line changes.)

P on C. This is exactly the same as above, except that the newly created point is constrained to stay on a circle, and is projected back to the circle if the circle changes.

P on Conic. The same as above, but the point is constrained to stay on a conic section. This and the points mentioned above are the only ones you can move with a mouse. All the others, and in fact, all the other geometric primitives are completely constrained.

Pinned P. This is just like a free pinned, except that after it is placed, you will be unable to move it. (Of course you can change it or delete it using the editor, but pinned points are useful for constructing a diagram with certain points immovable.)

LL=>P. Construct the (completely constrained) point that lies at the intersection of two lines. If you know a bit about projective geometry, even if the lines are parallel, a point is created at their "intersection at infinity". If you don't know any projective geometry, don't worry about it, but this is another "good thing".

PP=>P Mid. Construct the point that's midway between the other two.

LC=>P. Construct the point that lies at the intersection of the given line and circle. Depending on the configurations of the line and circle, there may be zero, one, or two possibilities for the location of this point. To get the one you want, click on the line and circle near to where you want the point to occur, if there are two possibilities. **Note:** Sometimes as you twist the geometry, the point may jump to the other solution, although this is rare. What is going on is that to solve for the intersections, **Geometer** is basically solving a quadratic equation which may have two roots, and when you pick the intersection point, you're telling **Geometer** which root to use. As the diagram changes when you move points, the root you want may change. If this causes problems, there are almost always ways to get around them with alternate constructions.

CC=>P. The same as above, but this time pick the point that lies at the intersection of two circles. All the same instructions and warnings hold because just as with a line and a circle, two circles can intersect in zero, one, or two places.

PPP=>P Bis. This creates a point that lies on the angle bisector of the other three points and inside the angle. What's meant by "inside" the angle is a bit tricky—see the subsection on angles, below.

C=>P Ctr. Given a circle, this command generates the point that lies at its center.

LP=>P Mirror. This reflects a point across a line. The new point lies on the opposite side of the line, the same distance from the line as the old point, and so that if you connected the new point to the old one with a line, the connecting line would be perpendicular to the line of reflection.

PC=>P Inv. The new point is the inversion of the old one through the circle. It and the old point lie on the same ray from the center of the circle. If O is the center of the circle, V is the old point, W is the new one, and r is the radius of the circle, the distances of V and W from the center satisfy the following equation: $\overline{OV} \cdot \overline{OW} = r^2$.

APP=>P. A new point is constructed so that it and the two given points form the given angle.

LCon=>P. The new point lies at the intersection of the given line and conic section. There may be zero, one, or two intersections. The intersection used is closest to where you clicked on the conic section with the mouse.

LCP=>P Other. The new point is at the intersection of the given line and circle, and is guaranteed to be different from the given point. This is useful if you create one of the intersections of a line and a circle and later need the other one.

LCC=>P Other. The new point is at the intersection of the two circles, and is guaranteed to be different from the given point. This is useful if you create one of the intersections of the two circles and later need the other one.

PPP=>P Harmonic. The new point is the harmonic conjugate of the other three. In other words, if the first three points are A, B, and C, the newly-created point X will satisfy $\mathcal{H}(AC, BX)$. No attempt is made to assure that the three given points lie on a line. If they do not, the position of the new point will not make much geometric sense.

4.2.2 ♦Conic Creation

5P=>Con. This creates a conic section passing through the five given points. A conic section can be a circle, an ellipse, a parabola, or a hyperbola. There are also "degenerate" conics that consist of a pair of crossing lines that can occur if the five points are lined up exactly right (or exactly wrong, depending on what you're trying to do).

5L=>Con. This creates a conic section that is tangent to all five of the given lines.

4.2.3 New Angle

PPP=>A. Select three points where the second point you pick will be at the vertex or "corner" of the angle. The ray from the vertex to the first point forms the first side of the angle and the ray from the vertex to the third point forms the second side. The inside of the angle goes counter-clockwise from the first ray to the second. In other words, if you click on points A, B, and C to form an angle, the angle runs from the ray $\overrightarrow{BA}$ counter-clockwise to the ray $\overrightarrow{BC}$. Depending on the orientation of A, B, and C, this may be the "long way around", and "inside" the angle may not be what you expect. There is a *Flip Angle* command in the *Edit* pulldown menu that you can use if you get it wrong. After creation, the angle is automatically selected, so if you notice you got it backwards, just issue the *Flip Angle* command immediately – it operates on the selected angle.

4.2.4 Making New Lines

PP=>L. Make the line connecting the two points.

PL=>L Perp. Construct a line through the point and perpendicular to the line.

PL=>L Par. Construct a new line passing through the point and parallel to the given line.

PC=>L Tan. Construct the line through the point and tangent to the circle. Assuming the point is outside the circle, there are two possibilities for this line, so click on the side of the circle where you'd like the line to be tangent. If the point is on the circle, there's only one possibility, and if it's inside the circle, you won't get anything.

CC=>Ex Tan. Construct a line externally tangent to the two circles. There are generally two possibilities for this line, so click on the desired sides of the circles to get the correct one.

CC=>In Tan. Same as the command above, except make the internal tangent line. Again, there are generally two possibilities.

PP=>L Perp Bis. This constructs the line that is perpendicular to the line connecting the given points, and cuts that line midway between them.

PCon=>L Tan. The line created goes through the given point and is tangent to the given conic section. There may be zero, one, or two such tangent lines.

4.2.5 New Circles

Ctr Edg=>C. The first point you pick will be the center of the circle and the second will be on its edge.

PPP=>C. The three points you pick will lie on the edge of the new circle.

Ctr PP=>C. The first point you pick will be the center of the new circle, and the radius of the new circle will be the distance between the next two points you pick. This command simulates the use of a compass that you've set to be the distance between two points, and then use to create a circle with a (possibly) different point as center.

PC Rad=>C. The first point you pick will be the center of the new circle, and the radius of the circle will be exactly the same as the radius of the other circle you select. This command basically provides a "compass" for straight-edge and compass constructions. After you've drawn one circle with your compass, this allows you to use the same compass setting to make a new circle of the same size, but with a different center.

LC=>C Inv. This makes a line that's the inversion of the given line through the given circle. See the description of the inversion of a point through a circle in the point creation commands (*PC=>P Inv*) for a definition of inversion. Every point of the line is inverted to give a circle.

CC=>C Inv. The first given circle is inverted through the second in the same way as are points and lines to give a new circle.

LLL=>C. A circle is formed that is tangent to the three given lines. There are generally four possibilities—one inside the triangle formed by the three lines, and one outside each edge. To get the one you want, try to click on the edges near where the circle will be tangent to the lines. If you want a circle that's tangent outside the triangle, it's better to click far from the triangle on the lines where you want tangency outside the triangle.

4.2.6 New Polygon

P..P=>Poly. Construct a polygon by clicking on its vertices in order, beginning with the first. To complete the polygon, click again on the first vertex. The maximum number of vertices allowed in a polygon is 10, so if you click on ten vertices, the polygon will automatically be completed with those ten vertices. The polygon need not be simple—the lines may cross each other.

Polygons are not particularly useful, except that you can find their area. The polygon's edges are not lines that can be used to create new primitives, for example. (You can, of course, add lines around the polygon's perimeter if you need to treat it both as a polygon and as a collection of surrounding lines.)

4.2.7 New Arc

PPP=>Arc. Select three points where the second point you pick will be at the center of the arc. Arcs are formed in exactly the same way as angles, but the first point is used to determine the radius of the arc. Arcs are primarily used to make nice illustrations—use circles if you're going to want to find intersections with other primitives.

Arcs are not circles and you can't find the intersection of an arc with a line. Usually, arcs are simply used to make snazzy drawings, and the underlying circles are also in the diagram, but drawn in an invisible color so they don't clutter the view.

4.2.8 ♦New Bézier Curve

PPPP=>Bez. The four given points are used as the control points for a Bézier curve that's parameterized from 0 to 1. This isn't normally useful in standard Euclidean geometry, but such curves are very useful in general graphics.

4.3 Changing Properties

"Property" refers to the visual appearance of the geometric primitives. Changing these properties has no effect on their geometric features—it just changes how they look on the screen. Some properties are almost universal, like color, and some only apply to a single kind of primitive.

Whenever you change a property, you change that property for the selected geometric object and make that property the default for the next items you create (with the exception of the "invisible" color—if the current color is set to "invisible", new items are created in white). So if you want to make three red points and then 3 yellow points, set the color to red, create your three points, click somewhere else so that the last point you created is no longer selected, set the color to yellow, and make the final three points. Alternatively, you can enter the repeated creation mode by double-clicking or holding down the **Ctrl** key when you click on the "create point" command, click locations for three points (which will be red), then click another location making a red (but selected) point, click on the "yellow" color command (which will change the red point to yellow, but will leave you in the repeat-creation mode), and then create the final two points.

If you want to change some property of an object you created a while ago, click on the object to select it and then click on the new color, line style, or whatever. Sometimes it is difficult to select the object you want because there are a bunch of other nearby objects that can't be moved away (angles are notoriously bad), but you can cycle through the different selection possibilities by simply holding down the **Ctrl** key while clicking again in the same place that is over multiple selectable items.

As was the case with the geometry creation commands, the most useful of the properties can be changed directly using buttons on the command menu, but there are a few others that are less common, and all of them except color are available under the *Styles* pulldown menu entry. The item's name, which is a sort of property, is also not available from the menu—the name can only be accessed via the editor or the dialog window you get when you type **Ctrl-n**.

Let's start with the color.

4.3.1 Color

Every primitive has a color. The built-in standard colors include white, red, green, blue, yellow, magenta, and cyan. The "Color" button in the middle of the command menu shows the current default color and you can change it by clicking down on it and sliding the cursor up and down while the mouse button is down.

In addition to the colors mentioned above, **Geometer** supports a few special colors: invisible, smear, blink1, blink2 and blink3. You cannot make any color blink; there are exactly three different blinking colors, so an object can be red or yellow or blink1 or blink2, et cetera. When an object is invisible, you can't normally see it or select it. This is great for hiding auxiliary lines you used in a construction. If you need to work with invisible items, click on the *Show Invis* button in the command menu, and all

the invisible items (and the non-invisible ones as well) will be visible, selectable, and movable, if they are points that are not completely constrained. Since one of the most common color changes made in **Geometer** is to make construction lines and circles invisible, there is a keyboard shortcut to make the current selection invisible: **Ctrl-i**.

There are three different blinking colors. When an item is drawn in one of these colors, it blinks—either between black and one of three colors or between two more closely related colors if you have a good enough graphics card on your computer. Blinking colors are great for highlighting features of interest in a step of a proof or construction.

Finally, there's a smear color. When an object is the smear color, when you drag a point that item will smear itself over the screen as long as the mouse is down. This is great for showing how one point moves in relationship with another, or to show how various interesting curves can be generated based on geometric constraints.

There are keyboard shortcuts for the first 8 colors. If you hold down **Ctrl-Shift-#**, where "#" is a number between 0 and 7, you will change the color to black, red, green, yellow, blue magenta, cyan, and white, respectively.

Geometer also lets you define new colors and use them, but you have to use the text editor. See the advanced section for more information.

The "invisible" color is also special in that if the button indicates that you're making invisible things, you will still make white ones. This is very handy, because when you're constructing something that will ultimately have a bunch of hidden features, they're drawn in white so you can work with them, but when they're correct and you want to hide them, you need only click on the *Color* button to make them invisible. After all, the "invisible" choice is on top. Otherwise you've have to scroll the color choices each time to get invisible, then click to unselect the primitive, then scroll back to a visible color.

(If you change a selected item to have the invisible color, it remains selected. This is because if you accidentally make it invisible, it disappears, and if it was an accident, all you need to do is change the color again to bring it back into view.)

4.3.2 Point Type

As a mathematical ideal, each point is infinitely small, but the points can be drawn in a number of different styles. These types include Diamond, Plus, Cross, Square, Solid, Dot, Circle, and No Mark. No Mark is basically invisible, but the point can be selected and moved. This is nice to use if you've got a polygon and you want users to be able to grab and move the points, but you don't want a glob of color on each point. Another use is to put text at arbitrary locations in your drawing. Make a point (probably pinned, if you don't want it to move), and give it a name that is the text you want to display. Then set the point type to "No Mark" and you're done. The others types are just different drawing styles for a point. By default, new points are represented by little circles.

4.3.3 Line Types

There are three independent properties associated with a line—its stipple pattern, its extent, and its marking. The stipple pattern can be solid, dashed, or dotted, the extent can be a line (infinitely long in both directions) a ray (infinitely long in one direction—the first point of the ray is the origin; the second marks the direction in which it goes to infinity), and segments that begin and end on points. Some lines have no reasonable definition of where the second point should be, so they often look like rays. Finally, each line can be supplied with hash-marks—none, one, two, or three slashes across the line that can be used in a diagram to show that it's congruent to some other line.

Geometer tries to place hash marks and line names at the middle of the line, but that may not make sense for certain lines. Usually **Geometer**'s guess is pretty good, but if you really need something else, you can usually put a point on the line where you want the name, and name the point. Make the point of type "No Mark". Line names are slightly different from points in that **Geometer** tries to avoid having them drawn on top of the line, so if you really need one of these, you can put two points close together on the line where you want the name, make a line connecting them, and name *that* line. And make the points of type "No Mark" or invisible.

4.3.4 Polygon Types

Polygons can be filled in four styles—solid, outlined, and filled with three different densities of stippling.

4.3.5 Angle Types

Angles can be drawn with one, two, or three rings, and with zero, one, or two hash marks across the rings. These markings are generally used to indicate congruence of angles. They can also be unmarked. There's also an angle mark called "Right Ang" that makes the right-angle square. Of course it's up to you to make sure the angle is a right angle. If it isn't really, **Geometer** still does it's best to put in the right-angle square, sometimes with bizarre results.

4.3.6 Line Widths

Each primitive has a width, and those that are drawn with lines make use of this property. A width of 1.0 is the default, but the width can be set with the menu to 0.5, 1.0, 2.0, 3.0, 4.0, and 6.0. All figures that are drawn with lines, including lines, circles, arcs, conics, polygons, and Bézier curves are drawn with a width that is this number times the normal width. Since screen resolutions may not make these changes visible, using different colors to represent different features is often a better strategy. But if your main goal is to produce PostScript figures for publication in a black and white format, different line widths can be useful, and since printer resolutions are typically much better than screen resolutions, this works well.

There is currently a bug in the PostScript code to draw conic sections with widths other than 1.0. All conics are drawn with width 1.0.

4.4 Miscellaneous Elementary Commands

Under the pulldown menus are a variety of useful commands. Some of the more useful of them have a speed key associated with them, and if that's the case, it is indicated in the pulldown entry. For example, the speed key equivalent for *Open* is **Ctrl-o**. The case is sometimes important—**Ctrl-s** is the same as *Save*, while **Ctrl-S** (with the uppercase "S") is *Save As*.

Under the *File* pulldown menu appear most of the standard commands you'd expect to find—*New* starts a new empty diagram, *Open* opens an existing diagram, *ReOpen* repeats the previous *Open* command. *ReOpen* is useful if you're editing a **Geometer** file with your favorite (non-**Geometer**) editor, and you'd like to take a look at the results in **Geometer** after you've written them out from the other editor.

Save and *Save As* save the current file, either using the current name, or using a name that you provide. *Insert* inserts the contents of another file into the current one. The internal names of the inserted items are "munged" so that it's unlikely they'll conflict with any of the names in the current file. After you do an *Insert*, editing will be a bit ugly because of the funny new names, but sometimes it's a very useful command.

Print produces a PostScript file and tries to print it using the printing command in your preferences file, and *Save EPS* makes an encapsulated PostScript file of the current drawing in the current directory. See Section 4.7 for details.

Quit quits but asks you if you want to save any modifications you may have made, and *Quit-No Save* doesn't bother to ask—it just throws away all your work.

Under the *Edit* command, the most important command is *Edit Geometry* which will be described in much greater detail in the next couple of chapters.

Edit Name lets you change the name of the currently selected primitive. A dialog box appears containing the old name which you can edit, delete, or replace. When you exit from the dialog box, the new name is applied to the primitive. **Warning:** Don't try to put the double-quote character (`"`) in a primitive name. Right now **Geometer** just throws them away if you try. I may fix this someday.

Delete Geometry is used to delete the selected item. This may be impossible because other items depend on it. For example, if your diagram consists of two points and the line connecting them, and you select one of the points and try to delete it, **Geometer** will complain, and won't let you do it until the line is deleted. The backspace key is a keyboard shortcut for *Delete Geometry*.

It is often a bit difficult to tell exactly what depends on what by just looking at the diagram, especially if there are some advanced features in use that don't show up graphically. If you use the *Edit Geometry* command to look at the entire structure of the file, it's usually a lot easier to figure out what's going on.

Describe Geometry gives a short description of the currently selected primitive. It tells you what it is, and what other items it's constrained by.

The *Flip Angle* command that simply reverses the sense of an angle. It is extremely easy to specify an angle backwards (in other words, to get the reflex angle of the one you really wanted), and if you create one that goes the wrong way around, just type **Ctrl-a** (or use the menu command) to flip it. Since you just created the angle, it will be selected, so you don't even have to select it—just issue the command.

Normally when you create a new point, **Geometer** supplies a name, like A, B, ... You can toggle this feature on and off with the *Point Names* command. Of course if you later decide that an unnamed point needs a name, you can select the point and change its name from the null name to whatever you want. If you are just experimenting, **Geometer**'s names are usually fine, but if you want to make a diagram with specific names, the easiest way is usually to toggle off the automatic name generation, and as soon as you create a point, edit the name with the *Edit Name* command.

Display Value toggles on or off whether the size of an angle, polygon or segment is displayed on the screen. If, in your diagram, two angles seem to be equal but you can't tell for sure, make **Geometer** angles of both of them, select each one, and use this command so that their value will be displayed on the screen. The other values that can be displayed are the length of a line segment and the area of a polygon. If you've given the angle, polygon or segment a name, that name will be displayed; otherwise, **Geometer** will display the value with the internal name. This command only applies to the selected segment, polygon or angle.

The *Edit Preferences* command brings up a little dialog box where you can change your default preferences. You can change such thing as the display size, the font sizes and types, the point size, and whether or not to be in various modes by default. You can also change the size of the text used in the **Geometer** user interface in this dialog, but it will not take effect until **Geometer** is restarted.

4.5 Proof Commands

Assuming you've loaded a geometric proof prepared by someone else, the commands in the *Proof* pulldown menu (also accessible via buttons in the lower right of the command menu) can be used to step through it. Constructing a diagram that displays your own proof is an advanced topic, covered later.

But assuming you're just looking at a proof, there are basically three things you'll want to do—go to the beginning of the proof (which is where it should be when you load it), step forward to the next stage of the proof, or go back if you need to look at a previous step.

The three commands that do this are *Start Proof*, *Next Step*, and *Previous Step*. On the buttons, the names are shortened to *Start*, *Next*, and *Prev*, but they do the same thing. Since going to the next step is the most common thing you'll do, it has a keyboard shortcut—just type the "**n**" character. Typing "**p**" is the shortcut for *Prev*

4.6 Finding Proofs: Testing Diagrams

Geometer has an extremely powerful feature that can help you find a proof by testing a diagram. The basic idea is this: If you draw a diagram with various constraints, other relationships may hold. For example, if you draw a triangle and its three medians, all that you've required is that lines be drawn from each vertex of the triangle to the midpoints of the opposite sides. Those three medians meet at a single point called the centroid and they always will, even though you did not require it in the original construction of the diagram.

In fact, almost every interesting theorem is a result of something similar—if you have a diagram with certain constraints, other constraints are required to hold. For something like the concurrence of three medians, the result is obvious to the eye, but other relationships may not be—that two segments are equal, for example, or that four points happen to lie in a harmonic relationship.

Geometer includes a mechanism to help you search for such relationships in a semi-automatic fashion. Here is how to use it:

1. Draw a **Geometer** diagram in the usual way, or load one from a prepared file.
2. Click on the *Test Diagram* command in the *Proof* pulldown menu.
3. Drag around various free points in the diagram to test a lot of different configurations.
4. Click on the *End Test* command in the *Proof* pulldown menu.
5. Examine the list of relationships that appears in the window that pops up.
6. Dismiss the window, and continue using **Geometer**.

It works as follows. When you begin the test, **Geometer** looks at a very large collection of possible relationships and makes a list of all that seem to be true of the diagram in its initial configuration. Then, as you adjust the size and shape of the diagram, **Geometer** checks and rechecks the list to see which relationships continue to hold. If a relationship fails, it is dropped from the list. Finally, when you finish the test, **Geometer** displays all the relationships that held throughout the test.

If a relationship "obviously" holds, **Geometer** does not bother to list it. For example, if points A, B, C, and D are drawn, and also lines AB, AC, and AD, it is "obvious" that those three lines intersect at a point (they are defined to do so, after all), so **Geometer** will not report on this coincidence. But in the example above where the three medians meet at a point, the coincidence is not at all obvious, so **Geometer** will report it.

As an exercise, try drawing a triangle and its three medians in **Geometer**, and then test the diagram as described above. It should report the coincidence of those three lines. Now, add to the same diagram the three altitudes of the triangle and be sure to have points at the bases of the altitudes. Run the test again, and you will get a large set of interesting relations. Among other things, **Geometer** will find that the three medians

and the three feet of the altitudes all lie on a circle (this is the famous nine-point circle). It will find some other circles as well—do you see why?

Unless they lie in a line, three points always lie on a circle, so obviously it is pointless to indicate that, but if four points lie on a circle, that is interesting. In the example above, there are six points on a circle, so **Geometer** goes nuts. It finds every set of four points that work. For example, if the six points are A, B, C, D, E, and F, **Geometer** will find that all 15 combinations: $ABCD$, $ABCE$, $ABCF$, $ABDE$, $ABDF$, ..., $CDEF$ lie on a circle. Before presenting the information to you, however, it condenses it, reporting only that A, B, C, D, E, and F lie on a single circle.

Geometer, of course, does not guarantee that these relations hold exactly in every case—it simply checks to see that they hold to within a certain numerical tolerance for every configuration that you try. It may miss some relationships as well.

At present, these are the relationships that are checked:

- Three points lie on a line.
- Four points lie on a circle.
- Two lines are parallel or perpendicular.
- An angle is a right angle.
- Two angles are equal, complementary, or supplementary.
- Three lines meet at a point.
- Three circles meet at a point.
- Four points lie in a harmonic relationship.
- Two points are inverses of each other relative to a circle.
- Two segments have equal lengths.
- Two ratios of lengths are equal.
- Two triangle areas are the same.
- Ratios of triangle areas are equal.

The reason that you need to adjust the diagram between the beginning and end of the test rather than having **Geometer** simply wiggle the points around a bit is that there may be other relations you want to maintain but that are difficult to express using **Geometer**'s features. For example, you may be looking for a theorem that holds only for acute-angled triangles (like Fagnano's Theorem), or you may want to look at sets of circles that all intersect each other. If you simply let **Geometer** wiggle the points, the triangle could be wiggled to make a non-acute-angled triangle, or so that some pair of the circles don't intersect.

It is good to do at least a little wiggling yourself—by chance, you may have drawn your diagram so that some relation happens to hold for that particular configuration, but would fail immediately with only a tiny movement of one of your points. **Geometer** is a bit conservative about throwing out possible relations, particularly for equalities of ratios, so it *is* a good idea to wiggle a few points. Usually only a very tiny amount of movement is necessary to get rid of these chance relationships, however.

Ratio equalities are always listed last since there can be a lot of them. For example, if you try to test the diagram for the diagram in Figure 4.1 (in which all the points except

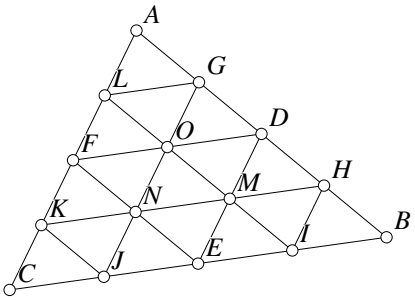

Figure 4.1: Thousands of Relations
`Ref1/Junk.T` [M]

for A, B and C are midpoints of other segments), you will find that there are thousands of ratios that are preserved.

Geometer only considers visible objects in visible colors. Thus you can construct figures with lines and change them to be an invisible color if you do not want those lines to be examined for possible relationships. This becomes important with complex figures, as there are often many relationships to consider, and if you can reduce that number, your job will be easier.

4.7 Printing

Geometer has two commands for printing: *Print* and *Save EPS*. The first prints to the printer; the second makes an Encapsulated PostScript file suitable for printing (and other things). The *Print* command relies on some information in your `.geometerrc` file which can be changed with the *Edit Preferences* window available from the *Edit* pulldown menu. (See Section 5.12.5).

Geometer can only produce a PostScript file, so you will need some method to print that file. For example, if you have a PC, you can download GhostScript for free from `http://www.cs.wisc.edu/~ghost/` and use that as a printing engine.

Your system may have other methods to print PostScript files, and any of those should also work.

If you are having trouble printing, here are some details that may help you:

- Although it only produces an Encapsulated PostScript (EPS) file, you have the most control using the *Save EPS* command. It produces a file that looks almost exactly like your current **Geometer** window with one screen pixel per printer's point. In other words, if you need a file that would exactly fill an 8.5×11 inch page, since there are 72 points per inch, you would want a **Geometer** window that is $612 = 8.5 \times 72$ pixels wide and 792 pixels high. This can be set in the *Edit Preferences* dialog available under the *Edit* pulldown menu.

Then if you know how to print EPS files, just print this one outside the **Geometer** program.

- When you issue the *Print* command **Geometer** scales your page to 550 pixels wide by 730 high and produces an EPS file that centers this information in the center of an 8.5×11 inch page. It then calls the system, using the print command you specified in your preferences file together with the name of the EPS file it just created.

- Geometer scales the current window so that it is as large as possible and still fits on an 8.5×11 inch page.

- The best way to get printing working is to have **Geometer** produce an EPS file that you can experiment with and then test possible printing commands in a command shell with that file. When you finally get exactly what you want, save it into your preferences file.

  For example, on a Windows NT 4.0 system with GhostScript installed, one possible way to print the file called `test.eps` with an attached HP LaserJet printer from a command shell is:

  ```
  gswin32 -sDEVICE=laserjet -dBATCH -dNOPAUSE test.eps
  ```

  On a Macintosh running OS X, the command is much simpler:

  ```
  lpr test.eps
  ```

  To make **Geometer** use this command to print, place this text in the *Print cmd:* input area in the *Edit Preferences* dialog:

  ```
  gswin32 -sDEVICE=laserjet -dBATCH -dNOPAUSE %s
  ```

  or

  ```
  lpr %s
  ```

  The "`%s`" informs **Geometer** that the file name is to be inserted at this point in the command.

  This is, of course, an extreme case. Your print command may be something as simple as `print %s`.

4.8 Odds And Ends

There are still a few items in the menus that haven't been covered yet, so they're all tossed together in this section.

The commands under *Help* give various sorts of help. Three of them display documentation: *Documentation. Tutorial* and *Reference Manual.* You can also view a series of *Usage Tips* that give general hints on the usage of various **Geometer** features, and in addition, there is a *Tool Tips* command that causes **Geometer** to draw a little window over any button in the command area that explains its use if you stop the mouse over it.

The feature is normally on. If you want to turn it on, click on the toggle button in the tips display.

The *Show Text* button in the command menu turns on and off the display of text. Sometimes a figure is complicated and the text overwrites some of the lines, and it's just easier to see what's going on without the added clutter of the text.

The *Rpt Set Color* button in the command menu is a quick way to change a lot of items to the same color. Before you press it, set the color button to indicate the new color you want. Then press *Rpt Set Color*, and you will be in repeat mode, where every item you click on will have its color changed to the set color. Use the *Cancel Repeat Mode* button at the bottom of the command menu to get out of this mode.

Some diagrams have a built-in script. If you want to run the script (and a well-designed diagram will tell you that a script exists), just press the *Run Script* button and the script will do its thing. The keyboard shortcut is **S**. Building a script is an advanced topic.

Finally, there may be a bunch of "Layer Control" buttons which are described in detail later. (They may not be visible—their display is controlled by an option.) Layers control stepping through proofs and constructions and lots of other things. For now, it's safer not to mess with those attractive red buttons.

4.9 The File Chooser

When you issue an *Open* command in **Geometer**, a file chooser window pops up. It is pretty obvious how to use it, but there are some hidden features that may be helpful.

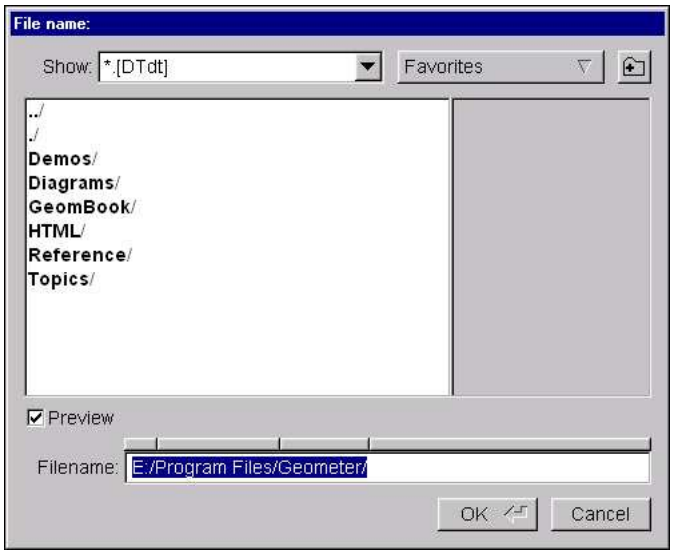

Figure 4.2: File Chooser

Figure 4.2 is a screen-shot of the file chooser. By default, it will only display **Geometer** files (with a `.T` or `.D` suffix).

The files displayed in the text area on the left are in the current directory. The current directory is set when you start a **Geometer** session. If you started **Geometer** by double-clicking on a file, the current directory will be the directory containing that file. If you started **Geometer** by double-clicking on its icon, the current directory is initially set to **Geometer**'s installation directory.

If you see the file you would like to open, in the text area, simply double-click on it and you are done. If you would like to go to one of the subdirectories that are displayed in bold-faced type, double click on the name. If you would like to go to the parent directory, double-click on the "../" entry at the top of the text area.

You can also just type your pathnames into the area called "Filename:" at the bottom of the chooser.

There is a *Favorites* button at the top of the chooser that allows you to go to certain favorite directories. By default, this includes the root of the file system, your own home directory, and **Geometer**'s installation directory. You can add or delete your own preferred directories if you wish.

Finally, above the pathname in the "Filename:" text entry area, there are thin buttons above each segment of the path. You can click on those to go up one or more directories in a single step.

Chapter 5

♦Advanced Features

> This is by far the coolest thing in existence.
> *David Tristram, speaking about* **Geometer**

For the majority of users, the topics in this chapter are not important. If all you want to do is experiment with basic geometric relationships, the information in the previous chapters should be sufficient.

Most of the topics here are not particularly difficult—just irrelevant for casual users. Readers familiar with computer programming will find much of it trivial.

If you want to use **Geometer** to create beautiful drawings, illustrations for papers, well-designed proofs for students, or animations, the material here is important.

Similarly, to do more than basic Euclidean geometry—if you want to get your hands dirty and mess with the raw numerical coordinates, for example, this chapter will tell you how.

If you just want to learn geometry, come back here only if there's something specific you need to know.

5.1 Geometry Via Text Editor

For the most common and important commands, **Geometer** provides a user-friendly graphical user interface where you can click on buttons and drag around geometry with a mouse to create and experiment with diagrams. But **Geometer** also has a "user-hostile" interface via the text editor that provides a very efficient way to access many of the extremely powerful features that are available.

The nice thing about **Geometer** is that you can use the two in combination—do most of the work with the graphical user interface, and then submerge yourself in the textual description to fine-tune your diagrams.

Geometer has a little editor built into it, and when you issue the *Edit Geometry* command (or with speed key **Ctrl-e**), it fires up this little editor on the textual description of the currently viewed diagram. The advantage of using **Geometer**'s editor is that when you finish your editing, **Geometer** immediately tries to reload the file, and if there are errors, you will be returned to the editor with the offending line highlighted.

You're welcome to use your favorite text editor as well (just make sure you save the files in "text" or "ascii" format), but then each time you write the file, you'll have to go to **Geometer** and use the *ReOpen* command. Then if there are errors, you'll have to go back to your editor, find the right line, fix it, re-save, and so on. Most **Geometer** files are tiny—a 100 line file would be quite large—so you probably don't need the power of a full-blown Emacs or equivalent.

Geometer's editor is a basic cut-and-paste editor, but it does have a couple of Emacs-type commands that you can learn about in the reference section (Section 5.12.3) on the editor. But the files are usually so small that you won't need to worry about this.

Between the time you fire up the editor with the *Edit Geometry* command and you exit (using the *Save* command in the editor's pull-down menu or the **Ctrl-s** keyboard shortcut), you can only edit the text. The current diagram visible in the **Geometer** window is frozen.

5.1.1 File Format

Let's begin by looking at a simple example. The following is a text version of the diagram that demonstrates that the three medians of a triangle meet at a point.

```
.geometry "version 0.2";
v1 = .free(-0.404517, 0.351129, "A");
v2 = .free(0.408624, 0.441478, "B");
v3 = .free(0.0513347, -0.301848, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
v4 = .v.vvmid(v1, v2);
v5 = .v.vvmid(v2, v3);
v6 = .v.vvmid(v3, v1);
l4 = .l.vv(v1, v5);
l5 = .l.vv(v2, v6);
l6 = .l.vv(v3, v4);
// Here's a comment.
```

At present, the first line doesn't do anything – but leave it unchanged. In the future, it may be used for version control, or to signal to future versions of **Geometer** that the file is ancient and needs to be brought up to date. It can also signal to an older version of geometer that it's trying to read a more modern file, and there might be trouble.

The final line shows you how to put in a comment. The comment text will be saved and rewritten, but it has no effect on the operation of **Geometer**. The "`//`" tells **Geometer** to ignore all the text to the end of the line.

The rest of the lines in the file are more typical of **Geometer** commands. **Geometer** uses a bunch of reserved words, and every reserved word begins with a period (and may contain additional periods). Thus "`.geometer`", "`.free`", "`.l.vv`", and "`.v.vvmid`" are the reserved words used in this file.

If you're writing your own **Geometer** program, you cannot accidentally have a conflict with **Geometer**'s reserved words – even for future versions of **Geometer**. You cannot use a leading period in your variable names and **Geometer** promises always to use one. A user identifier begins with a letter (or the underscore character), and is followed by a sequence of letters, digits, or the underscore character. Here are a few valid identifiers:

```
Line1 point_A m7 M7 Ma8qWs Iris1000 __7
```

Upper and lower case are distinguished, so `m7` and `M7` represent different variables.

The vast majority of **Geometer**'s commands have the form:

```
<variable> = <command>(<required params> <optional params>);
```

For example, in the second line:

```
v1 = .free(-0.404517, 0.351129, "A");
```

the variable is `v1`, which serves as the internal name of that geometric object. The command is `.free`, the required parameters are the two floating-point numbers that serve as the point's coordinates, and the display (or external) name, `"A"`, is an optional parameter.

Since these commands will play a huge role in advanced diagrams, it's convenient to have a syntax to describe the command form. This will be covered in detail in the reference section, but a couple of examples should make the form clear:

```
[P] = .v.ll([L], [L]);
```

The example above uses the strings `[P]` and `[L]` to indicate that any internal point or line name can go in the corresponding slot. Note also that the optional parameters are not listed – after the final line in the example above, you can add colors, linestyles, external names, et cetera. In addition to `[P]` for point and `[L]` for line, circles are represented by `[C]`, angles by `[A]`, and there are a bunch of others that we'll introduce as needed.

Typically, the non-geometric properties are optional. There is a set of default values for each property, and if they are not specified in a command, the default value is used. Here's a list of the defaults (some of which won't make sense to you yet):

Property	Default Value
Color	`.white`
Name	(none)
Layer	`.l0on` (all layers)
Line type	`.segment`
Line stipple	`.solidline`
Line mark	`.line0slash`

Point type	`.circpoint`
Angle type	`.ring1`
Polygon type	`.outlinepoly`

When **Geometer** writes a text file, it checks the properties of each item, and if they have the default value, it omits them from the output. You may type them as input if you've forgotten what the default is and want to be sure, but if it was a default value, **Geometer** will not bother to write it on output.

For example, if you type in a line like this:

```
vert = .free(.123, .456, .white, .segment, .l5);
```

Geometer will cheerfully parse it and will paint the point white, and draw it as a segment but only on layer 5. When you save the file, however, it will look like this:

```
vert = .free(.123, .456, .l5);
```

There's no need to write "`.white`" or the "`.segment`" as they are defaults. **Geometer** does, however, write the non-standard layering specification "`.l5`".

So don't spend a lot of time working on the formatting of your **Geometer** files in the editor. As soon as you let **Geometer** write it out, it will do as it pleases, simplifying as many things as possible and clobbering all of your careful formatting work.

When you type in properties, the order doesn't matter. the following two lines are exactly equivalent from **Geometer**'s point of view:

```
l1 = .l.vv(v1, v2, .dashline, .green);
```

and

```
l1 = .l.vv(v1, v2, .green, .dashline);
```

When you write a file, **Geometer** puts them in some order that may be different from the order you typed them in, but the net result is the same. (The order of the required parameters—in this case `v1` and `v2`—does make a difference. Even here for a line connecting two points, if that line is changed to be displayed as a ray, the origin of the ray is the first point specified, so reversing `v1` and `v2` would flip the direction of the ray.)

Look in the reference section at the end of this chapter for a concise list of the available property types, and for a complete list of the **Geometer** commands, together with their features.

In **Geometer** files, the parameters are separated by commas, and the commands are terminated by a semicolon. There is no need for the commands to occupy a single line; newlines and white space are ignored by **Geometer** (except, of course, within text strings).

If it isn't obvious already, **Geometer** tries to use sensible command names. Unfortunately, the original name used for "point" was "vertex" so where you would expect to

see a "p" you will see a "v". Other than that the command names are logical. After the leading period, the first part of the name signifies what type of geometric object is being created. After that is another period, followed by the types of things the new item will depend upon. For example, to make a point where two lines intersect, use the command ".v.ll". To make a circle passing through three points, use ".c.vvv".

Sometimes the name includes additional information. To make the line passing through a point and parallel to another given line, use ".l.vlpar". The command to make the perpendicular line also requires a point and a line, so it's name is ".l.vlperp". This convention makes it easy to remember most of the **Geometer** commands, but if you have a bad memory, there's always the reference section.

5.1.2 Names

Most items in **Geometer** have an internal name, and some of them have external names. The internal name is what appears to the left of the "=" sign. For example, in

```
vert = .free(.123, .456, "Ogden Nash");
```

the internal name of the point is "vert", and the external name is "Ogden Nash". If you're going to use this vertexpoint as something that constrains another primitive, like a line or a circle, use the internal name, "vert". "Ogden Nash" will be the name you see attached to the point on the computer screen. If there is no external name, all you'll see will be the point's circle or diamond or whatever with no attached name.

The easiest way to think about how **Geometer** displays a file is to imagine that before each redisplay it forgets everything, and starts looking at the file from the beginning again. Thus, if you've typed in the following program (you must have typed it in, since it's got illegal semantics):

```
v1 = .free(.1, .1);
v2 = .free(.2, .2);
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
v3 = .free(.3, .3);
```

Geometer will do fine for the first 3 lines. v1 is a point at $(.1, .1)$, v2 is another one at $(.2, .2)$, and l1 is a line connecting the two of them. But when it tries to interpret the next line, it fails because it's supposed to make a line connecting v2 and v3. It knows what v2 is, but it can't read ahead, so in spite of the fact that v3 is defined on the very next line, an error is generated because when **Geometer** was trying to construct line l2, it didn't yet know what v3 was.

Every geometric primitive has an internal name, even if it isn't used in later commands. (This may change in the future – there's no reason to require a user to think of names for things that will not be used, but for now, that's how it is.) Primitives like text that can't be involved in future commands never have a name.

The use **Geometer** makes of external names depends on exactly what it is that's named. If a point or a line has an external name, that name is drawn beside the point or the

middle of the line. If an angle has a name, the name is displayed on the interior of the angle, where "interior" satisfies the usual conditions.

A number (called a "`flt`" (to remind you of a "floating point number") in **Geometer**) that has an external name is listed, together with its value, on the right side of the drawing area. Thus, if you want to display the value of an angle as the user changes its size, you'd put the angle's size into an `flt` whose name was the name of the angle, and its value will be continuously displayed.

The two most common values that you may want to display are the measure of an angle or the length of a line segment. In either case, you can use the command *Display Name* (**Ctrl-T** to make the value of the selected primitive visible on the screen. If the name display is toggled on, **Geometer** will use the name of the primitive, if it exists, and otherwise the internal name. In the **Geometer** file, if a primitive is to be displayed, `.display` will appear as one of the properties. It is not an error to include this property with items other than line segments or angles, but at present, it has no effect except in those two cases.

Other items can currently take external names which are at present unused, except in the *Describe Geometry* command which will always use the external name if it's available.

5.2 Layers

Geometer is conceptually drawing on any combination of 32 layers, numbered from 0 through 31. By default, it draws on all 32 of them. Any object can be drawn on any combination of layers.

By default, the layer control commands are not visible. You can see them by choosing the *Show Layers* command in the *Edit* pulldown menu. You can also change the setting in your options file so that they are always displayed. To set your options, use the *Edit Preferences* command in the *Edit* pulldown menu.

The user can view any combination of layers. By default, all the layers are visible, so you can see everything.

When you step through a proof or other demonstration, however, the situation changes. When you press the *Start* button, **Geometer** sets it so that you're only viewing layer number 0. Every time you press *Next* **Geometer** advances to display only the next layer, so you'll view layer 0 only, then only layer 1, then layer 2, et cetera. Similarly, *Prev* goes to the previous layer. The *Off* command paradoxically turns *on* all the layers. It has the strange name since most users will not see the layer commands displayed, and will think of the *Off* command as turning off **Geometer**'s proof mode.

In the lower part of the command area there's a little array of 16 buttons numbered 0 to 15. If they are checked, the corresponding layer is active. Active means that you are currently viewing that layer, and if you are creating new geometric items, they will only be visible on that combination of layers.

Only the state of first 16 layers are displayed in the control panel; the others are there, and function, but almost no diagrams need to use the higher numbered layers, and they can be accessed using the text editor. To change the active layers among the first 16, simply click on the buttons to toggle them on or off. There's a shortcut available too—if you hold the **Alt** key down when you click on a layer, it toggles the state of that layer and all the ones beyond. That way it's easy to turn off or on all the rest of them.

It takes a bit of practice to get good at using the layers, but the proper way to program a proof or demonstration is to make sure that just the "right" things appear on each layer. You can look at some of the sample proofs, such as `Ninepoint.T` to see how it's done.

Some things, like the basic diagram, will appear on all layers. Things that appear only for a part of the proof, like construction lines, textual descriptions, et cetera, will appear on only one or a few layers. If you want to emphasize something at one stage, but then have it go back to a normal presentation for the rest, you can enter two versions of it, perhaps in different colors. For example,

```
line1 = .l.vv(v1, v2, .l0, .l2on, .blue);
line1blink = .l.vv(v1, v2, .l1, .blink);
```

has two copies of exactly the same line which connects the points `v1` and `v2`. On layer 0 and from layer 2 through 31 (that's what `.l2on` means), the normal version is shown in blue. But on layer 1, the same line is shown in a blinking color. Presumably there's some text on layer 1 pointing out the significance of this line during that stage of the proof. If the situation is more complicated than this, however, your best bet is probably to use layer colors, discussed later.

Geometer interprets a line as follows: If there are no layer commands, **Geometer** assumes that the object appears on all layers, 0-31. If any layer command appears, the assumption is that no layers are shown except the ones listed. Thus:

```
v = .v.ll(line1, line2, .l3, .l7);
```

will appear only on layers 3 and 7.

A very common situation is that you'd like to have an object appear from a certain layer on to the end. For that reason, you can use commands like: `.l7on` which means that the object in question appears on layer 7, and then on every layer after that. You can mix these with other layer commands to get things like this:

```
line1 = .l.vv(v1, v2, .l0, .l5, .l8on);
```

This line will appear on layers 0, 5, and 8, 9, 10, . . . , 31.

Another situation that commonly occurs is that you'd like something to remain visible from the beginning for a while, and then disappear. For that the "opposite" command also exists. `.tol7` as a layer specifier means that the primitives appear on every layer up to and including layer 7.

If you type in overlapping layers, when **Geometer** writes the file, it will simplify it as much as possible. For example, if you type in:

```
vert = .v.ll(line1, line2, .l0, .l1, .l5, .l7, .l8on, .l19);
```

and then you save the file, you'll find that **Geometer** has written something like this:

```
vert = .v.ll(line1, line2, .tol1, .l5, .l7on);
```

Finally, if you're writing a proof or demonstration, you'd like **Geometer** to begin displaying only layer 0, so that the *Next* key will do the right thing without requiring the user to press the *Start* button. For that reason, there's a special layer command that consists only of layer descriptions followed by semicolons to appear in a file to indicate the starting configuration. Typically this will simply be:

```
.l0;
```

but **Geometer** can understand more complex starting configurations. For example, if your file includes:

```
.l0; .l7; .l21on;
```

Geometer will start up with layers 0, 7, and 21, 22, ..., 31 visible, although why you'd want to do that is hard to imagine.

Finally, if some strange combination of layers is visible, *Next* and *Prev* actually shift the visible layers by one in either direction. In other words, if layers 1, 3, and 7 are visible, a *Next* will make layers 2, 4, and 8 visible, and a *Prev* will make only layers 0, 2, and 6 visible. If only layer 1 is visible, *Prev* will not eliminate it. Similarly, a *Next* with only layer 31 visible will leave only layer 31 visible.

Although layers work as advertised above, it is often much easier to use layer colors as described in the next section.

For more advice on using layers, see the chapter on Programming Hints.

5.3 Line Widths

The width of the lines used to draw lines, circles, arcs, polygons, conics, and Bézier curves is set to 1.0 by default. It can be set to any positive width with an entry that looks something like this:

```
line = .l.vv(vert1, vert1, .width=3.0);
```

The "`3.0`" above can be replaced by any positive number. Although **Geometer**'s menu entries only allow for a fixed set of widths, any positive width is legal, but non-standard widths must be entered with the text editor. If the width is 1.0, that is the default, and no entry is put in the text version of that primitive.

5.4 Layer Colors

Many times you would like to have a primitive appear in different colors, or perhaps not appear at all as you advance through the steps of a proof or construction. The best way to do that is with a combination of the layer and color commands. A single example is probably enough to make the whole idea clear:

```
line = .l.vv(vert1, vert2, [3 .in, .blink, 3 .white, .in]);
```

This line that connects `vert1` with `vert2` will appear on all layers, but for the first three layers, layers 0, 1, and 2, it will have the invisible color. On layer 3 it will have the blinking color, on layers 4, 5, and 6, it will have the color white, and then from layer 7 on, it will again have the invisible color.

You don't have to use the multiples, or you can use them in combination with repeated color specifications, so the primitive above is exactly equivalent to:

```
line = .l.vv(vert1, vert2, [.in, .in, .in,
        .blink, .white, 2 .white, .in]);
```

although it will be written by **Geometer** in the previous (simplified) form.

The final color to appear is assumed to be in force for the rest of the layers, so in the case above, after layer 7, the line will always be invisible.

If, in the control panel, not all layers are active and you create a new object, the latest version of **Geometer** in fact creates the primitive on all layers, but with the layer colors set so that the object will be invisible on all layers except for those that are active. For example, if layers 0, 2, and 3 are active and you create a new point with the current color of red, that point's color specification will be:

```
[.red, .in, 2 .red, .in]
```

Similarly, if you change the color of an existing object when not all the layers are active, then the color of that object will be changed only on the active layers. Suppose that the existing object has a color specification of:

```
[.red, .in, 2 .red, .in]
```

and that the current layer specification is for layers 2 and 5, and that you select the object and click on the color yellow. Only layers 2 and 5 will be changed to yellow, and the resulting object will have a color specification of:

```
[.red, .in, .yellow, .red, .in, .yellow, .in]
```

5.5 Text

5.5.1 General Information

You've already seen plenty of examples of text entry. The most common command is the `.text` command that allows you to enter text that will appear in a diagram to tell the user what's going on.

Like other primitives, text can be restricted to certain layers, and it can be drawn in various colors. To draw a line of red text that appears only on layer 3, the following will work:

```
.text("Text only on layer 3 in red.", .l3, .red);
```

The `.text` command can extend to more than one line, and if a newline character appears in the text, a new line is started on the screen. For example, the following lines:

```
.text{"Here is a line of text.
Here is the second line

Here is the fourth line.", .red);
```

will draw four lines of red text, where the third line is blank. In general, the text is displayed on the screen in the order it appears in the file. Thus, the following strange collection of lines in your file:

```
.text("Line 1 in red", .red, .l0);
.text("Line 1 in green", .green, .l1);
.text("Line 2 in yellow", .yellow, .l0);
```

would display first a red and then a yellow line on layer 0, and on layer 1, it would draw a single line in green. Clearly, it's a good idea to keep lines of adjacent text adjacent in the file, but **Geometer** doesn't require it.

5.5.2 Special Characters

It's really useful to be able to display certain special characters in geometric descriptions. For example, you'd like to use something like $\angle ABC$ to talk about an angle, or perhaps $\overline{AB}$ to represent the segment AB. **Geometer** has a limited ability to do this.

Geometer recognizes certain escape sequences in the text stream that are interpreted to refer to special characters. For example, the line:

```
.text("The \angleABC is called \alpha.");
```

would display the following on the screen:

The $\angle ABC$ is called α.

Geometer recognizes the following control sequences that generate special characters;

`\triangle`	Δ	`\angle`	$\angle$	`\congruent`	$\cong$
`\similar`	$\sim$	`\dotmath`	$\cdot$	`\multiply`	$\times$
`\fraction`	$/$	`\degrees`	$^\circ$	`\circlemultiply`	$\otimes$
`\circleplus`	$\oplus$	`\greaterequal`	$\geq$	`\lessequal`	$\leq$
`\perp`	$\perp$	`\sqrt`	$\surd$	`\therefore`	$\therefore$
`\alpha`	α	`\beta`	β	`\gamma`	γ
`\delta`	δ	`\epsilon`	ϵ	`\zeta`	ζ
`\eta`	η	`\theta`	θ	`\iota`	ι
`\kappa`	κ	`\lambda`	λ	`\mu`	μ
`\nu`	ν	`\xi`	ξ	`\omicron`	o
`\pi`	π	`\rho`	ρ	`\sigma`	σ

\tau	τ	\upsilon	υ	\phi	ϕ
\chi	χ	\psi	ψ	\omega	ω
\Gamma	Γ	\Delta	Δ	\Theta	Θ
\Lambda	Λ	\Xi	Ξ	\Pi	Π
\Sigma	Σ	\Upsilon	Υ	\Phi	Φ
\Psi	Ψ	\Omega	Ω	\cdot	$\cdot$

If there are any special characters in the standard PostScript symbol font that you need that are not included in the set above, you can enter them using the character followed by the three-digit octal character. For example:

```
.text("Here is a copyright symbol: \323");
```

will yield:

Here is a copyright symbol: ©

Geometer can also display superscripts and subscripts as follows. The line:

```
.text("a\sup{2}=b\sub{3}");
```

will yield:

$$a^2 = b_3.$$

To get a symbol for a line segment, like $\overline{AB}$, use this:

```
\overline{AB}
```

To get an arc symbol, like $\overset{\frown}{AB}$, use this:

```
\arc{AB}
```

For now, don't try to put special characters in superscripts, subscripts, or with overlines or arcs. **Geometer** will just get confused and put some sort of garbage on the screen.

5.6 Numbers And Calculation

Sometimes it's necessary just to work with numbers rather than with geometric figures. For example, if you want to make a circle that has 1.72 the radius of another, you may be able to figure out a way to do it geometrically, but it's a heck of a lot easier if you can just use numbers. For example, here's a chunk of **Geometer** code that will make a circle centered at `v1` and passing through `v2` and will, in addition, draw a circle centered at `v3` having a radius of π times the radius of the first circle:

```
c1 = .c.vv(v1, v2);
rad = .f.vv(v1, v2);
newrad = .f.rpn(rad, 3.14159265, .mul);
c2 = .c.vf(v3, newrad);
```

The first line is standard, but the second line says to make a number called "rad" whose value is the distance between the points v1 and v2. The third line says to take the value of rad, as a number, and to multiply it by 3.14159265, and to store the value in newrad. Finally, the last line says to produce a circle of radius newrad centered at the point v3. The "f" in the commands above is short for "flt" which you can think of as a floating point number.

There is a whole series of commands that obtain, manipulate, and use floating point numbers. The only access to them is via the editor interface—i.e. there is no way to use them via the graphical user interface.

These numbers are usually used as coordinates, or as sizes in transformations, so before you can use them, you have to have some idea of **Geometer**'s coordinate system. Earlier, we said that it goes *roughly* from -1.0 to 1.0 in both dimensions. This is exactly true for a square display area, but if the display area is a little longer than wide or wider than tall, the images would appear distorted with these coordinates. So what **Geometer** does is to determine the minimum of the width and the height, and makes that minimum length run from -1 to 1. Thus if the display window size were 500 pixels wide and 400 high, the x coordinates would run from -1.25 to 1.25, and the y coordinates from -1.0 to 1.0.

The numbers can also be used to describe angles, like $90°$. Most people measure angles in degrees, mathematicians and computer programmers almost always measure them in radians, where 2π radians is equal to $360°$. **Geometer** goes both ways, but if you do nothing, it assumes you're working with degrees. You can add a line to your file like this:

```
.radianmode;
```

and if that line appears, all angles in your file will be expressed in radians. The default setting is `.degreemode;`, but since it's the default, it need never appear in your file. In some of the examples that follow, I will include "`.degreemode`" explicitly just to remind you that angles are measured that way. There's no need to include it in a normal diagram (although it is legal to include it, and **Geometer** will cheerfully eliminate it from any output file it creates).

Here is a list of the all the commands involving flts that do not involve transformations. Transformations are covered in the next section. In the command names, the appearance of an "f" is used to indicate that a number (a "f"loating point number) is used or produced. In the parameter list, a flt is represented by `[F]`, `[P]` is a polygon, and `[RPN]` stands for "rpn-expression", which we'll talk about later:

```
[P] = .v.ff([F], [F]);
[C] = .c.vf([P], [F]);
[F] = .f.vvvratio([P], [P], [P]);
[P] = .v.vvf([P], [P], [F]);
[F] = .f.vv([P], [P]);
[F] = .f.vxcoord([P]);
[F] = .f.vycoord([P]);
[A] = .a.f([F]);
[F] = .f.area([P]);
[F] = .f.rpn([RPN]);
```

Most of these are fairly straight-forward. `.v.ff` makes a point using the two numbers as its x and y coordinates. `.c.vf` produces a circle centered at the point, and having a radius given by the number.

`.f.vvvratio` makes a number that is the ratio of the distances between the three points. Typically, the points are on a line, and if they are called A, B, and C, then the number produced is the ratio $\overline{AB}/\overline{AC}$. The points don't have to be on a line, however.

Use `.v.vvf` to do the opposite – to produce a point with the given ratio of distances relative to two other points. This command will produce the new point on the line connecting the other two.

`.f.vv` produces a number that's the distance between the two points, and `.f.vxcoord` and `.f.vycoord` put the x and y coordinates into a number.

`.a.f` produces an angle of the given magnitude. The magnitude depends on whether you're in degree mode or radian mode (see above), so

```
.degreemode;
ang = .a.f(180);
```

and

```
.radianmode;
ang = .a.f(3.14159265);
```

have exactly the same net effect.

Finally, `.f.area` calculates the area of the polygon and puts it in the number. If the polygon's edges cross each other, you'll get consistent results, but perhaps difficult to interpret. **Geometer** simply treats the polygon as a series of triangles and adds those together with signed areas.

So far we've just made and used numbers, but have no way to calculate with them. And exactly what is it that you can put in place of an "`f`" in the examples above?

In every case, you can use a defined number, or a literal number. For example, if you wanted to have a point whose x coordinate was locked at 0.1, but whose y coordinate was the same as the distance between two other points, the following code would work:

```
v1 = .free(0.372188, 0.255624, "A");
v2 = .free(0.609407, 0.247444, "B");
ycoord = .f.vv(v1, v2);
v3 = .v.ff(0.100000, ycoord);
```

Geometer can also do more or less arbitrary calculations with these numbers, and all of these are achieved by means of the `.f.rpn` command. The "`rpn`" refers to "reverse Polish notation", and within that command, **Geometer** supports a tiny PostScript-like or Fourth-like language. The instructions consist of a series of command tokens, number variables, angles, and number literals. They are evaluated from left to right, as follows:

A number literal, a number variable, or an angle will simply have its value pushed on the execution stack. If it's an angle, the actual number pushed on the stack will depend on whether **Geometer** is in degree mode or in radian mode.

Every other possible entry is a **Geometer** reserved word, and each has some effect on the contents of the stack. After all the entries are evaluated, the result is whatever is left on the top of the stack.

Here's a list of all the **Geometer** stack operators, and what each one does to the contents of the stack:

`.add` The top two numbers are removed from the stack and are added. The result is returned to the stack.

`.sub` The top two numbers are removed from the stack and subtracted. The result is returned to the stack.

`.mul` The top two numbers are removed from the stack and multiplied together. The result is returned to the stack.

`.div` The top two numbers are removed from the stack and divided. The result is returned to the stack.

`.mod` The "modulo" function is applied to the top two numbers on the stack which are then replaced by the result. For example, the sequence `17, 5, .mod` will result in a 2 on top of the stack, since $17 (\text{mod } 5) = 2$.

`.neg` The top number on the stack is negated.

`.sin` The trigonometric function sine is applied to the top number on the stack. The number on the top of the stack is interpreted as an angle in degrees or radians, depending on **Geometer**'s mode.

`.cos` The trigonometric function cosine is applied to the top number on the stack. The number on the top of the stack is interpreted as an angle in degrees or radians, depending on **Geometer**'s mode.

`.tan` The trigonometric tangent is taken of the top number on the stack. The number on the top of the stack is interpreted as an angle in degrees or radians, depending on **Geometer**'s mode.

`.atan2` If the top two elements on the stack are x and y, the trigonometric arctangent function is taken of x/y if $y \neq 0$. If $y = 0$ it's like taking the arctangent of an infinite number, positive or negative, depending on the sign of x. The angle produced will be in degrees or radians, depending on **Geometer**'s mode.

`.abs` The top number on the stack is replaced by its absolute value.

`.exp` If x is the top number on the stack, it is replaced by e^x.

`.log` If x is the top number on the stack, it is replaced by $\log x$—the natural logarithm.

`.rand` A random number between 0.0 and 1.0 is placed on the top of the stack.

`.dup` The top number on the stack is duplicated.

`.clear` The entire stack is cleared to empty.

`.pop` The top element on the stack is removed.

`.roll` The top two numbers on the stack are removed and are used to determine a rotation of the stack. If the stack's top two numbers are n and j, the top n numbers remaining on the stack are rolled j positions. j can be positive or negative. For example, if the stack consists originally of $a, b, c, d, e, 4, 1$, where the "1" is on the right of the stack, then the top four elements are rotated by one position, yielding a stack that looks like this: a, e, b, c, d.

`.copy` The top number on the stack is removed and is used as the number of items to copy. Thus, an original stack that looks like this: $a, b, c, d, e, 3$ after the copy operation would look like this: a, b, c, d, e, c, d, e.

`.exch` This operation exchanges the top two items on the stack.

`.eq` The top two numbers on the stack are removed and compared for equality. If they are equal, a 1 is placed on the stack. If they're unequal, a 0 is placed there.

`.ne` Just like `.eq`, except the comparison is for inequality.

`.lt` Same as `.eq`, except less-than.

`.le` Same as `.eq`, except less-than or equal.

`.gt` Same as `.eq`, except greater-than.

`.ge` Same as `.eq`, except greater-than or equal.

`.round` Rounds the top number on the stack to the nearest integer. Round basically adds 0.5 and then does the `.floor` operation, described above.

`.ceiling` Replaces the top number on the stack with the smallest integer greater than it. The ceiling is always an integer and it is always greater than or equal to the number. It is only equal to the number if the number itself is an integer.

`.floor` Replaces the top number on the stack with the largest integer smaller than it. It is always less than or equal to the number, and is equal to it only if the number is an integer.

`.truncate` Truncates the top number on the stack to the integer closest to zero. Truncate is like `.floor` (see above) for positive numbers and like `.ceiling` (see above) for negative numbers.

The above doesn't seem like much if you've never worked with reverse Polish notation, but you can do just about any calculation you like using it. For example, here's a little **Geometer** program that will draw a graph of the function $\sin x + \cos 2x$:

```
.geometry "version 0.2";
v1 = .free(-1.00204, -0.398773, "A");
v2 = .free(0.130879, -0.366053, "B");
xval = .f.vv(v1, v2);
yval = .f.rpn(xval, .sin, xval, 2.000000, .mul,
    .cos, .add);
v3 = .v.ff(xval, yval, .smear);
```

It isn't very interesting, because the values mostly lie outside the drawing window, and need to be better scaled to make a reasonable diagram, but it does behave as advertised. The x coordinate is simply taken to be the distance between the points `v1` and `v2`. The y coordinate is calculated using the `rpn` expression. Follow along to see what happens. First the `xval` is placed on the stack and its sine is taken. Another copy of it is put on the top of the stack (above $\sin x$), then a 2 is added to the stack, the top two elements are multiplied, yielding $2x$ on top, whose cosine is taken, and that result is added to $\sin x$ below, yielding $\sin x + \cos 2x$. Those values are then used as the coordinates of `v3`, which has a `.smear` color, so as you drag around `v1` or `v2`, the point `v3` will leave a smeared track along the curve $y = \sin x + \cos 2x$.

After each `.f.rpn` expression is evaluated, the stack is cleared, so you can't pass information on to other commands, although this may occur in the future, so it's a good idea to leave only the final result you're interested in on the stack when you're done, for compatibility with future versions of **Geometer**.

5.7 Transformation

It's sometimes convenient, especially in scripts, to be able to move your objects around on the screen. Most important motions are rigid, but from time to time it's useful to have non-rigid motions as well. **Geometer** provides a simple set of transformations that can be applied to points yielding other points. Let's begin with the simplest set:

```
[P] = .v.vtranslate([P], [F], [F]);
[P] = .v.vscale([P], [F], [F]);
[P] = .v.vrotate([P], [F]);
```

These operations yield a new point that's transformed relative to the old one. The `.v.vtranslate` command, for example, makes a new point that's translated by the given amounts in the x and y directions from the old one. In the following example:

```
v1 = .free(0, 0);
v2 = .v.vtranslate(v1, .3, .5);
```

the point `v1` will be drawn at the center of the drawing area with coordinates $(0, 0)$ and `v2` will be translated from `v1` by .3 units in the x direction and .5 units in the y direction. If you then grab `v1` with your mouse and drag it around, `v2` will remain .3 units to the right and .5 units above `v1`.

The rotation and scaling commands, `.v.vrotate` and `.v.vscale`, do their rotation and scaling relative to the origin at the center of the drawing area. So in the following example:

```
.degreemode;
v1 = .free(.2, .2);
v2 = .v.vscale(v1, 2.0, 1.5);
v3 = .v.vrotate(v1, 45);
```

the x and y coordinates of `v1` are multiplied by 2.0 and 1.5, respectively, to yield the point `v2` at $(0.4, 0.3)$. Similarly, the rotation by $45°$ about the origin will put point `v3` at $(.28284, 0)$. The rotation is counter-clockwise about the origin, and since `v1` lies on the $45°$ line, it will be rotated to the $90°$ line, namely, the y axis.

In case you've never messed with geometric transformations, the order does make a difference – a rotation followed by a translation is almost never the same as the translation followed by the rotation. Probably the best place to learn about this would be in a textbook on computer graphics.

But just a couple of hints may be enough to get you through. If you'd like to rotate about a point that's not at the origin, you can translate it to the origin, then rotate, and then translate back. So, for example, to rotate the point `v1` about the point $(.2, .3)$ by 48 degrees, the following chunk of code will do the trick:

```
.degreemode;
v2 = .v.vtranslate(v1, -.2, -.3, .in);
v3 = .v.vrotate(v2, 48, .in);
v4 = .v.vtranslate(v3, .2, .3);
```

The intermediate points `v2` and `v3` are painted the `.in` (= invisible) color so they won't appear in your diagram.

The trick above works great if you only have a point or two that you'd like to rotate about an unusual origin. If you'd like to use the same operation on a whole set of points, it would be a pain to make up all the intermediate points for each of the original points.

Geometer has a primitive object called a transformation (listed as `[X]` in the syntax charts, and as "`x`" within the command words). Once you build a transformation (like a rotation of 48 degrees about the point $(.2, .3)$, you can then apply it to a large series of points.

Here's the code to do exactly that, and then to transform all the vertices of the triangle `v1`, `v2`, `v3` to a new triangle `v4`, `v5`, `v6`:

```
v1 = .free(0, 0);
v2 = .free(.5, 0);
v3 = .free(0, .5);
id = .x.identity();
x1 = .x.translate(id, -.2, -.3);
x2 = .x.rotate(x1, 48);
x3 = .x.translate(x2, .2, .3);
v4 = .v.vx(v1, x3);
v5 = .v.vx(v2, x3);
v6 = .v.vx(v3, x3);
```

The code above builds a transformation starting from the identity, which represents no transformation at all, by applying one transformation after the other to it. The

entire transformation that represents a rotation about the point $(.2, .3)$ is saved in the transformation called `x3`. Then `x3` is applied to each of the vertices of the original triangle in turn to make the new, rotated, triangle.
Here's a list of all the commands that deal with transformations:

```
[X] = .x.identity();
[X] = .x.translate([X], [F], [F]);
[X] = .x.scale([X], [F], [F]);
[X] = .x.rotate([X], [F});
[P] = .v.vx([P], [X]);
[X] = .x.xxf([X], [X], [F]);
[X] = .x.f9([F], [F], [F], [F], [F], [F], [F], [F], [F]);
```

The use of the top 5 is obvious, but the final two require some explanation. `.x.xxf` is a way to choose between two transforms, depending on the value of the number. If the number is zero, the first transform is used; otherwise the second. This can be useful in a script when you want to change from transform to transform in the middle.

The final one, `.x.f9` allows you to specify a completely general affine transformation using 9 numbers. The way transformations are handled internally is using 3 by 3 matrices, which are multiplied together to generate more complex transformations. For example, the rotation matrix is as follows, assuming that the angle of rotation is x:

$$\begin{pmatrix} \cos x & \sin x & 0 \\ -\sin x & \cos x & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Although the points are only two dimensional, a third entry, equal to 1.0 and called the "w" coordinate) is added to them before the matrix multiplication. At the end, the new x and y coordinates are divided by the new w coordinate to get the two dimensional point. Usually this division is unnecessary – the w coordinate after transformation will be equal to 1.0.

But as an example, let's look at rotation by 30 degrees of the point $(2, 1)$ The matrix multiplication operation looks like this:

$$(2 \quad 1 \quad 1)\begin{pmatrix} \cos 30^\circ & sin30^\circ & 0 \\ -\sin 30^\circ & \cos 30^\circ & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Since $\cos 30^\circ = .86602$ and $\sin 30^\circ = .5$, the operation is:

$$(2 \quad 1 \quad 1)\begin{pmatrix} .866025 & .500000 & 0 \\ -.500000 & .86602 & 0 \\ 0 & 0 & 1 \end{pmatrix} = (1.23205 \quad 1.86602 \quad 1).$$

If we divide through by the resulting w coordinate of 1.0 (and it isn't too hard to divide by 1.0), we get the point $(1.23205, 1.86602)$ which is the result of rotating the point $(2, 1)$ by 30 degrees in a counter-clockwise direction.

For completeness, here are the matrices equivalent to translation by t_x in the x direction and t_y in the y direction:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{pmatrix},$$

and of a scale by s_x in the x direction and s_y in the y direction:

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

5.8 Macros

Imagine that you want to make a whole bunch of triangles connecting various sets of three points. Without macros, the only way to proceed would be to put down a long series of three points followed by the three lines that connect them. Each triangle would thus require 3 lines of code. With macros, you can reduce this to 4 lines per triangle, and with more complicated drawings, the savings can be far more.
Here's how to do the triangle example using a triangle macro:

```
.geometry "version 0.2";
.macro triangle(.vertex v1, .vertex v2, .vertex v3)
{
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
}
v1 = .free(-0.764826, 0.554192, "1");
v2 = .free(-0.515337, 0.766871, "2");
v3 = .free(-0.384458, 0.300613, "3");
v4 = .free(0.265849, 0.476483, "4");
v5 = .free(0.703476, 0.595092, "5");
v6 = .free(0.658487, -0.0429448, "6");
triangle(v1, v2, v3);
triangle(v4, v5, v6);
```

The macro `triangle` takes three parameters, all of which are points, and from them it constructs three lines. Each time it is called, it generates a new set of three lines from the given points.

At present, the macro facility is rather simple-minded – parameters can be any of the usual types: `.vertex`, `.line`, `.circle`, `.flt`, `.conic`, et cetera.

If a name within the macro is either a parameter name or is a name of an item produced within the macro, that local version is used. If it is not a parameter name or locally generated item, **Geometer** assumes that it is a global value.

For example, the following macro will draw a line from the point that's a parameter "`v`" to the fixed point v0:

```
v0 = .pinned(0, 0);
.macro linetozero(.vertex v)
{
    l1 = .l.vv(v0, v);
}
```

Macros can return a single value. For example, here is the syntax for a macro that takes two points and returns a circle having the two points as a diameter:

```
.macro .circle diamcircle(.vertex v1, .vertex v2)
{
    vmid = .v.vvmid(v1, v2, .in);
    .return c = .c.vv(vmid, v1);
}
v1 = .free(-0.2975, -0.12);
v2 = .free(0.1975, -0.1825);
v3 = .free(0.045, 0.3225);
c1 = diamcircle(v1, v2, .red);
c2 = diamcircle(v1, v3, .green);
```

Note that the point "`vmid`" is colored invisible so that it doesn't show up every time you make a new circle. Note also that the macro calls that produce `c1` and `c2` can have properties. In this case, the two different expansions of the macro will draw a red and a green circle. If you want, you can then use `c1` or `c2` in future constructions.

There is a known bug in the **Geometer** macro package that does not allow you to pass `.flt` parameters that are constant. For example, the following code will not work to generate a point with coordinates $(.3, .3)$:

```
.macro .vertex v(.flt f)
{
    .return vv = .v.ff(f, f);
}
v1 = v(.3);
```

You can get around the bug with this minor modification to the code, since as long as the `.flt` parameter is not constant, the macro code works fine:

```
.macro .vertex v(.flt f)
{
    .return vv = .v.ff(f, f);
}
f3 = .f.rpn(.3);
v1 = v(f3);
```

5.9 Scripts

To put a diagram on auto-pilot so it will pass through a fixed set of positions, use the `.script` command. It takes the following form:

```
[F] = .script([FLOAT], [FLOAT], [FLOAT]);
```

Each [FLOAT] represents a literal floating point number (in other words, you can't put in variable [F] types). In the command they are interpreted as the minimum value for flt, its maximum value, and the step size. If a line like this appears in your file, it means that when you press the *Run Script* button, the number on the left side of the "=" sign will start at minimum and will take steps of the size indicated by the other parameter until it reaches the maximum value. Thus, `f = .script(0, 1, .01);` will take 100 steps to proceed through the script. You can then use the value of `f` as you would any other number. In radian mode, if you want an angle to run from 0 to 2π in steps of .02 radians, do this: `f = .script(0, 6.2831853, .02);`

As an example, here's a simple script that drives a point in a circle of radius 1 about the origin in 100 steps:

```
.geometry "version 0.2";
f = .script(0, 360, 3.6);
vx = .f.rpn(f, .cos);
vy = .f.rpn(f, .sin);
v = .v.ff(vx, vy);
```

`f` is the angle in degrees and runs from 0 to 360 (to go completely around the circle). We then take the cosine and sine of the resulting number to generate the x and y coordinates of the point `v`. Run the script, and `v` moves smoothly around in a circle.

Normally, the *Run Script* button in **Geometer** is deactivated, but as soon as a `.script` command is read from the file, the button is activated. Only one script command is allowed per **Geometer** file.

5.9.1 Recording Animations

It is possible to save an EPS file for each stage of a running script. If you use the *Print Script* command in the *File* pull-down menu before running the script. This activates the saving of Encapsulated PostScript files for one execution of the script. The files are written in the same directory as the **Geometer** file and are given names related to that file. For example, if the file name is `Frog.T` and there are 20 steps in the script, PostScript files named `Frog000.eps`, `Frog001.eps`, ..., `Frog019.eps` will be generated.

It is up to you to figure out what to do with those files, but various programs can be used to combine them into animated GIFs that could be used on web pages, for example. For a detailed example, see section 6.15 in the teacher's tutorial chapter.

That's all there is to the `.script` command, but it can be a very powerful tool.

5.10 More On Colors

There's a little more to the color commands that was not described earlier. For example, there is a color `.black` that can be used. Since the background is black, this command is sort of like making it invisible, but such items can be selected (although you won't

be able to tell except for any side effects you may cause on moving a selected, invisible point).

In addition to all the non-colors like `.smear`, `.blink`, `.blink1`, `.blink2` and `.invisible` (or `.in`, for short), **Geometer** deals with 32 colors whose names are `.c0`, `.c1`, ..., `.c31`. The first 8 of these have aliases: `.c0 = .black`, `.c1 = .red`, `.c2 = .green`, `.c3 = .yellow`, `.c4 = .blue`, `.c5 = .magenta`, `.c6 = .cyan`, and `.c7 = .white`.

The colors from `.c8` through `.c15` are non-black, and `.c16` on are simply white. However, you can redefine any of these colors (including the first 8, although you'd probably be nuts to do so) as in the following example:

```
.c10 = (.4, .5, .8);
```

The three floating-point values must lie between 0.0 and 1.0 and they represent the amount of red, green, and blue in the color to be displayed. The example above will define color `.c10` to use 40% of full red, 50% of full green, and 80% of full blue.

You can use the alternate color names exactly as you do the standard names like `.red`, `.blue`, et cetera.

5.11 The Display Attribute

If `.display` appears in the property list of an angle or of a line segment, then the length of the segment or measure of the angle (appropriately displayed in degrees or radians, depending on the mode) is displayed with the name of the segment or angle in the upper left corner of the window and is continuously updated.

5.12 Reference

5.12.1 Primitive Types

Here is a list of all the allowed properties:

Line types
- `.segment` – joins the two endpoints
- `.ray` – from first point forever through second
- `.longline` – infinite line through the two points

Line Widths
- `.width=2.0` – (default is 1.0)

Line styles
- `.solidline` – default
- `.dashline`
- `.dotline`

Line marks
- `.line0slash` – default

.line1slash
.line2slash
.line3slash

Polygon types
.outlinepoly – default
.solidpoly
.hashpoly
.hashpoly1
.hashpoly2

Point types
.circpoint – default
.diamond
.plus
.cross
.soliddiamond
.square
.dot
.nomark

Angle types
.ring1 = 1 ring – default
.ring2 = 2 rings
.ring3 = 3 rings
.slash1 = 1 ring with slash
.slash2 = 2 rings with slash
.slash3 = 3 rings with slash
.dslash1 = 1 ring with double slash
.dslash2 = 2 rings with double slash
.dslash3 = 3 rings with double slash
.right = right angle mark
.noangle = no angle mark

Colors
.white = .c7 = white – default
.black = .c0 = black
.red = .c1 = red
.green = .c2 = green
.yellow = .c3 = yellow
.blue = .c4 = blue
.magenta = .c5 = magenta
.cyan = .c6 = cyan
.invisible = .in = not normally drawn
.smear = color smears while point is dragged
.c8, .c9, ..., .c31 = additional colors
.blink = first blinking color
.blink1 = second blinking color
.blink2 = third blinking color

Layers

`.l0`, `.l1`, ..., `.l31` (note: lower-case 'L's)
`.l0on`, ..., `.l31on` – all layers from this number on
`.tol0`, ..., `.tol31` – all layers up to the number
Example: `.l7on` = `.l7`, `.l8`, ..., `.l31`

Display Value
`.display`

Stack commands
`.add`, `.sub`, `.mul`, `.div`, `.mod`, `.neg`
`.sin`, `.cos`, `.tan`, `.atan2`, `.abs`, `.exp`,
`.log`, `.rand`, `.dup`, `.clear`, `.pop`, `.roll`
`.floor`, `.ceiling`, `.round`, `.truncate`, `.copy`, `.exch`,
`.eq`, `.ne`, `.lt`, `.le`, `.gt`, `.ge`

Names
anything enclosed in "double quotes".

5.12.2 Command List

Here are forms of the rest of the recognized commands. Note that the set may increase in the future as new primitives are added. There is no possibility of conflict with user-defined names, since all the new primitives will be defined with names beginning with a period.

In the listing below, only the required fields are specified; any number of additional properties can be specified in any order. The types of the parameters are important and are checked. For example, ".l.vv", which makes a line from two points, will barf if its parameters are not variables representing points.

In what follows, `[P]` stands for any point identifier, `[C]`, for any circle identifier, and so on. The list of possibilities includes: `[P]` = vertex, `[L]` = line, `[C]` = circle, `[CON]` = conic, `[BEZ]` = Bézier curve, `[F]` = flt, `[X]` = transformation, `[A]` = angle, `[ARC]` = arc, `[POLY]` = polygon. `[RPN]` = a valid sequence of rpn commands.

Items like `[INT]`, `[FLOAT]`, and `[HEX]` indicate a fixed, floating point or hex number. `[INT2]` represents an integer that is 1 or 2; `[INT4]` can take only the values 1, 2, 3, 4. Hex numbers must have the form 0xhh...h, where h is a hex digit: 0, 1, ..., 9, a, b, ..., f.

Finally, `[LAYER]` and `[COLOR]` stand for a valid layer or color specification.

`[P] = .free([FLOAT], [FLOAT]);` Make a completely unconstrained point whose initial coordinates are given by the two floating-point numbers. The coordinate system goes roughly from -1.0 to 1.0 in each direction, but obviously if the window is non-square, this isn't right. With a non-square window, the smaller dimension goes from -1.0 to 1.0; the larger dimension is scaled appropriately. The point $(0, 0)$ is always exactly in the center of the drawing area.

`[P] = .pinned([FLOAT], [FLOAT]);` Exactly the same as .free, except that the new point cannot be dragged with the mouse.

`[P] = .vonl([L], [FLOAT], [FLOAT]);` Make a point constrained to be on the line, so it has one degree of freedom. The floating point values are the point's initial coordinates (which need not be exactly on the line – the point will instantly be projected to the line for the first display).

`[P] = .vonc([C], [FLOAT], [FLOAT]);` Make a point which is constrained to be on the circle, so it has one degree of freedom. The floating point values are the point's initial coordinates (which need not be on the circle – the point will instantly be projected to the circle for the first display).

`[P] = .v.ff([F], [F]);` The values of the `[F]`s are evaluated, and used as the point's x and y coordinates.

`[P] = .v.vvmid([P], [P]);` Make a point constrained to be the midpoint of the other two points.

`[P] = .v.ll([L], [L]);` Make a point constrained to be at the intersection of the two given lines, or at infinity in the appropriate direction if the lines are parallel.

`[P] = .v.lc([L], [C], [INT2]);` The point is at the intersection of the line and the circle. Since there are two possible intersections, they are distinguished by the value of the integer, which must be either 1 or 2.

`[P] = .v.cc([C], [C], [INT2]);` The point is at the intersection of the two circles. Since there are two possible intersections, they are distinguished by the value of the integer, which must be 1 or 2.

`[P] = .v.vvf([P], [P], [F]);` A new point is constructed between the first and second point parameters in the ratio given by `[F]`. If $\widehat{[F]}$ is zero, the new point will be on top of the first point. If the ratio is 1, it will be on top of the second. If the ratio is $2/3$, it will be $2/3$ of the distance between the first and second given points. Ratio values outside the range $[0, 1]$ also make sense and are interpreted in the obvious way—a ratio of -1 would put the point on the opposite side from the second point of the first, and with a distance equal to the distance between the given points.

`[P] = .v.avv([A], [P], [P]);` A new point is made that makes an angle with the other two points equal to the given angle. The angle is measured in a counterclockwise direction.

`[P] = .v.ccenter([C]);` The new point is at the center of the given circle.

`[P] = .v.lvmirror([L], [P]);` The mirror image of the point through the line is generated.

`[P] = .vonconic([CON], [FLOAT], [FLOAT]);` The point is forced to remain on the given conic section. The two floats are the current coordinates.

`[P] = .v.vcinv([P], [C]);` Inverts the given point through the circle.

`[P] = .v.lconic([L], [CON], [INT2]);` The point is at the intersection of the line and the conic section. There are up to 2 possibilities, so `[INT2]` can be 1 or 2.

`[P] = .v.vvvbisect([P], [P], [P]);` The new point is on the angle bisector of the angle formed by the other three points. It is on the inside of that angle.

`[P] = .v.vx([P], [X]);` The new point is the old one with the given transformation applied.

`[P] = .v.vtranslate([P], [F], [F]);` Construct the new point by translating the old one by the first flt in the x-direction and by the second in the y-direction.

`[P] = .v.vscale([P], [F], [F]);` The new point is the old point scaled by the first flt in the x-direction and by the second in the y-direction.

`[P] = .v.vrotate([P], [F]);` Construct the new point by rotating the old one by flt counter-clockwise about the origin. The angle represented by flt is dependent on the angle mode: degrees or radians.

`[P] = .v.lcvother([L], [C], [P]);` The new point is at the intersection of the line and the circle and is guaranteed to be different from the given point.

`[P] = .v.ccvother([C], [C], [P]);` The new point is at the intersection of the two circles and is guaranteed to be different from the given point.

`[P] = .v.vvvharmonic([P], [P], [P]);` A new point is the harmonic conjugate of the first three. If the first three points are called A, B, and C, the result, X, satisfies $\mathcal{H}(AC, BX)$.

`[L] = .l.vv([P], [P]);` The new line passes through the two given points.

`[L] = .l.vlperp([P], [L]);` A new line is constructed, passing through the given point and perpendicular to the given line.

`[L] = .l.vlpar([P], [L]);` A new line that passes through the given point and is parallel to the given line is constructed.

`[L] = .l.vc([P], [C], [INT2]);` A new line which passes through the point and is tangent to the given circle is constructed. In general, there are two possibilities, so `[INT2]` is 1 or 2.

`[L] = .l.ccext([C], [C], [INT]);` The new line is an exterior tangent to the two given circles. `[INT2]` is 1 or 2, since there are two possibilities.

`[L] = .l.ccint([C], [C], [INT2]);` The new line is an interior tangent to the two given circles. `[INT2]` is 1 or 2, since there are two possibilities.

`[L] = .l.conicv([CON], [P], [INT2]);` Constructs a line tangent to the conic that passes through the point. There are 2 possibilities, so `[INT2]` must be 1 or 2.

`[L] = .l.vvperp([P], [P]);` This line is the perpendicular bisector of the segment connecting the two points.

`[C] = .c.vv([P], [P]);` The new circle is constrained to have its center at the first point, and to pass through the second.

`[C] = .c.vvv([P], [P], [P]);` A new circle that passes through all three points is constructed.

`[C] = .c.lll([L], [L], [L], [INT4]);` A circle that is tangent to all three lines is constructed. There are, in general, 4 possibilities, so `[INT4]` is 1, 2, 3, or 4. The usual circle—the one that sits inside the triangle determined by the three lines—has `[INT4]` equal to 1.

`[C] = .c.vf([P], [F]);` The circle has a center at the point, and a radius equal to the value of flt.

`[C] = .c.ccinv([C], [C]);` The new circle is the inverse of the first circle through the second circle.

`[C] = .c.lcinv([L], [C]);` The new circle is the inverse of the line through the circle.

`[C] = .c.vcrad([P], [C]);` Constructs a circle centered at the point, and having the same radius as the given circle. This emulates a compass for straight-edge and compass constructions.

`[BEZ] = .bez.vvvv([P], [P], [P], [P]);` A new cubic Bézier curve is constructed having the four points as control points.

`[F] = .f.rpn([RPN]);` An `[RPN]` list is a set of comma-separated tokens that are chosen from among flt, angle, float, and stack commands. The stack commands are listed above. The sequence is evaluated as if it were a PostScript program, and the top number on the stack is the final value of flt. If there's an error, flt is set to something huge and no error is reported.

`[F] = .f.vv([P], [P]);` flt is set to the distance between the two points.

`[F] = .f.area([POLY]);` flt is set to the area of the polygon.

`[F] = .f.vvvratio([P], [P], [P]);` Calculates the ratios of the distances between the first and second and second and third points.

`[F] = .script([FLOAT], [FLOAT], [FLOAT]);` Only one of these commands can appear in a file. The three floats are interpreted as the minimum value, maximum value, and step size. If this command appears, when the *Run Script* button is pressed, the value of flt is set to the minimum value, and is incremented by the step size until it is larger than the maximum value. Then the script stops.

`[F] = .f.vxcoord([P]);` The x coordinate of point is assigned to flt.

`[F] = .f.vycoord([P]);` The y coordinate of point is assigned to flt.

`[A] = .a.vvv([P], [P], [P]);` The new angle has a vertex at the second of the three points, and has as edges rays from the center to the other two points.

`[A] = .a.f([F]);` This makes an angle of size flt. The `[F]` can be interpreted as either degrees or radians, depending on **Geometer**'s mode.

`[ARC] = .arc.vvv([P], [P], [P]);` This is similar to the angle command, but the arc drawn is centered on the second point and has a radius such that it begins at the first point. It ends on the line from the second to the third point, and goes in a counter-clockwise direction.

`[P] = .polygon([INT], [P], ..., [P]);` The value of `[INT]` is the number of points to follow – it must be between 3 and 10, inclusive.

`[CON] = .conic.vvvvv([P], [P], [P], [P], [P]);` Five points determine a conic section passing through all of them.

`[CON] = .conic.lllll([L], [L], [L], [L], [L]);` Construct a conic section tangent to all five lines.

`[X] = .x.identity();` Makes the identity transformation.

`[X] = .x.rotate([X], [F]);` Add a rotation with xform to make a new one flt is the rotation in degrees or radians, depending on the mode of geometer.

`[X] = .x.scale([X], [F], [F]);` Add a scale to the given xform. The two flts represent x- and y- scaling.

`[X] = .x.translate([X], [F], [F]);` Add a translation to the given xform. The two flts represent x- and y- translation.

`[X] = .x.xxf([X], [X], [F]);` If the value of flt = 0, the first xform is used; otherwise the other. This is a way to get a conditional transformation.

`[X] = .x.f9([X], [F], ..., [F]);` There are nine flt values that are used as entries into a transformation matrix which is then multiplied by the given xform to make a new one. The new transformation matrix is filled as follows, assuming the flt values are numbered 0 to 8 as they appear above:

$$\begin{pmatrix} \text{flt0} & \text{flt1} & \text{flt2} \\ \text{flt3} & \text{flt4} & \text{flt5} \\ \text{flt6} & \text{flt7} & \text{flt8} \end{pmatrix}.$$

`.layercondition([F], [F], [F], [HEX]);` If the value of the second flt lies between the values of the first and third, set the layers to the value of the hex number. This is generally used in scripts. Here's an example that turns on layer 1 for the first third of the script, layer 2 for the second third, and layer 3 for the final third:

```
f = .script(0, 1, .02); // steps of .02
.layercondition(0.0, f, .3333333, 0x1);
.layercondition(.3333333, f, .6666666, 0x2);
.layercondition(.6666666, f, 1.0, 0x4);
```

Generally the first and third flt values are constants, but that's not required.

`.text("Any text you want.");` Makes a line of text to be displayed at the bottom of the screen. The order in which text lines appear dictates the order in which they are displayed. In the current implementation, at most 8 lines are visible. Use the text in combinations with layers for more information.

There are some special characters you can insert in the text. See the section on text for more information on how to add characters like some of these: $\angle$, Δ, $\cong$, and $\overline{XYZ}$.

`// comment text ...` Any line beginning with "//" is ignored, but is saved and printed in any output files generated.

`[COLOR] = ([FLOAT], [FLOAT], [FLOAT]);` `[COLOR]` can be replaced by any of the valid color identifiers, usually `.c8`, `.c9`, ..., `.c31`, but it can be the default colors as well. The command above sets the red, green, and blue components for the given color identifier. For example to make `.c18` have red, green, and blue components of .1, .2, and .3, respectively, include the following command:

```
.c18 = (.1, .2, .3);
```

These definitions can be placed anywhere in the file, but they are only evaluated once as the file is parsed. When the file is saved, they will be written out at the top of the file. `.c8` through `.c15` have some interesting initial colors; `.c16` through `.c31` are initially white.

It's a good idea to make the three values between 0.0 and 1.0.

`[LAYER];` `[LAYER]` stands for any valid layer identifier, for example `.l0`, `.l1`, ..., `.l31`, or `.l0on`, `.l1on`, ... , `.l31on`; These are default layers for the file and are treated exactly as are the default colors.

`.degreemode;` or `.radianmode;` Sets the mode for printing and reading angles. **Geometer** is in degreemode by default;

`.macro` These are covered in Section 5.8.

5.12.3 The Text Editor

Geometer's text editor is fairly brain-dead, but it will only be used to edit the tiny **Geometer** files. It is a pretty standard cut-and-paste editor, but it lacks some nice features like using a double click to select a word, or shift-select to extend a selection.

There are a few speed keys (most of which are based on Emacs) for those of us who would prefer to avoid using the mouse, if possible. Here's a list of those speed keys:

Ctrl-a	Cursor to beginning of line
Ctrl-b	Cursor left one character
Ctrl-c	Copy selection

Ctrl-d	Delete character ahead of cursor
Ctrl-e	Cursor to end of line
Ctrl-f	Find command
Ctrl-k	Kill to end of line
Ctrl-n	Cursor down a line (next line)
Ctrl-o	Open up a line at the cursor
Ctrl-p	Cursor up a line (previous line)
Ctrl-Q	Quit (and don't save changes)
Ctrl-s	Save file
Ctrl-u	Kill to beginning of line
Ctrl-v	Paste command
Ctrl-w	Cut command (for Emacs users)
Ctrl-x	Cut command
Ctrl-y	Paste command (for Emacs users)

The arrow keys work in the obvious way, **Home** and **End** move to the beginning and end of the file, and the **Backspace**, **Tab**, and **Delete** keys work in the obvious way.

5.12.4 Secret Commands

Geometer has a couple of "secret" commands that are only available via the keyboard. Here's what they do:

Ctrl-l (This is the lower-case "L" key.) This command toggles the line width to be wider than normal. If you so screen-saves to snag an image, the thicker lines can be much easier to see.

→ **(right arrow)** Moves everything to the right a tiny bit. This actually changes the coordinates of the points.

← **(left arrow)** Moves everything a bit left.

↑ **(up arrow)** Moves everything a bit up.

↓ **(down arrow)** Moves everything a bit down.

Page Up Increases the size of the figure (zooms in) by a factor of 1.01. The zooming is relative to the center of the drawing area.

Page Down Zooms out by a factor of 1/1.01.

Ctrl-Page Up Rotates all geometry about the center of the drawing area by an angle of 1 degree clockwise.

Ctrl-Page Down Rotates all geometry about the center of the drawing area by an angle of 1 degree counter-clockwise.

5.12.5 Startup Options

What follows is true, and you can modify **Geometer**'s startup options as described below, but the easier way to do it is now to use the *Set Options* entry in the *Edit* pull-down menu. It will fire up a little user interface to set all the options below.

When **Geometer** starts, it tries to read a file called `.geometerrc` in your home directory (the directory in the environment variable HOME). If the HOME environment variable is not set, **Geometer** will look in the current directory. Currently, there are a few default values which can be set in that file – the command to invoke the editor, and the size of the initial **Geometer** window.

Here is a sample version of .geometerrc that sets all:

```
size 800 700
displayfont TIMES
printcmd gsview32 /p %s
printbw
fontsize 18
fontstyle BOLD
indexmode
nonames
browser netscape.exe
texteditsize 12
notips
nomsginfo
```

This will start with a window for the geometry that's 800 pixels wide and 700 high in the second case. (Note that the width and height don't include the menu on the left (about 200 pixels) nor the pulldown on top (about 30 pixels). If you're using **Geometer** to make illustrations in Encapsulated PostScript, it's a good idea to have width and height equal, since the PostScript file only prints what's between -1 and 1 in both dimensions, and a square window will show exactly that. **Geometer** will attempt to run in color index mode (rather than RGB, or full color mode)—use this if your computer has a fairly low-powered graphics card (for example one that can only display 256 colors). If you're not sure what kind of graphics you have, try it and if it doesn't work, take the line out of your startup options file. The colors will be a little better in RGB mode. The `nonames` tells **Geometer** not to add names to points. This can be toggled on and off through the menu entry *Point Names* or by typing **Ctrl-t**.

By default, the help system will use the internet explorer browser, but if you use something else, add a line like the last one to your file.

`texteditsize 12` sets the size, in pixels, of the font used in the **Geometer** text editor. By default, it is 14 pixels high. This example sets it to 12 pixels high.

Similarly, the print command will stuff the print file names into the

Geometer can draw text on the screen in a few fonts, including:

```
HELVETICA (default)
TIMES
COURIER
```

These are available in:

```
PLAIN (default)
BOLD
ITALIC
BOLDITALIC
```

If your printer can only print black and white, include the line `printbw` to get better printing.

The `notips` line tells **Geometer** that you do not want to see the "Tip of the day" when you start the program.

Finally, `nomsginfo` will cause **Geometer** not to display the help information as the cursor moves over the buttons in the control area.

Chapter 6

Teacher's Tutorial

This tutorial is for teachers (or bright students) wishing to use **Geometer** to construct fancy diagrams of their own. Before reading this, you should already know generally how to use **Geometer**—how to construct points, lines, et cetera, and how to manipulate them using the graphical user interface (GUI).

A huge number of examples are, of course, available on the CD that comes with the book. The difference is that there the programs come without explanation.

6.1 A Simple Construction: The Circumcircle

Let's warm up with a simple construction—we'll construct the circumcircle of an arbitrary triangle.

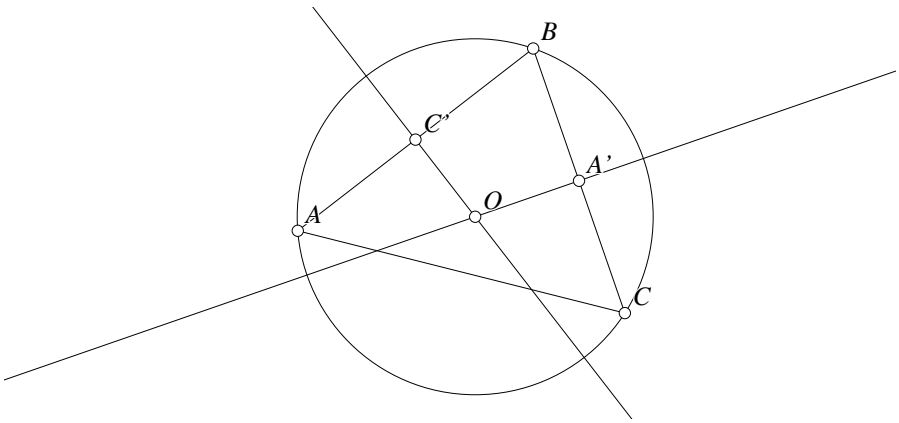

Figure 6.1: Construction of a Circumcircle
`Teachers/Circumcircle.T` [P]

The first thing to do is to get a (fairly) firm idea in your head about how you want the finished **Geometer** diagram to behave. In this example, I've chosen to do it as a five step process (see Figure 6.1):

1. Show the triangle alone with some text to describe the problem to be solved.

2. Construct the perpendicular bisectors of two of the sides first by finding the midpoints of the sides, and then by constructing lines perpendicular to those sides at their midpoints (two steps).

3. Identify the circumcenter as the point of intersection of those two perpendicular bisectors.

4. Construct the circumcircle using the circumcenter as center and any of the vertices of the triangle as a point on the circle.

If you're not doing it already, run **Geometer** *on your computer and follow along as we construct this example.*

Here is how I would do it in detail.

1. In the first few steps, we'll construct as much of the diagram as possible using the standard GUI tools. Beginning with an empty file, make three points (that will automatically be labeled A, B, and C). Connect those point to make the original triangle. This is what will be displayed (together with some text) when the diagram is opened.

2. Next, use the command *PP=>P Mid* to find the midpoints of the segments AB and BC. **Geometer** will label these new points "D" and "E" which isn't bad, but I prefer names like C' and A' for the midpoints opposite vertices A and C, respectively. To change the names, click on the point, hold down the `Ctrl`-key, and type `n`. A dialog box will appear with the name, and you can edit them to the new names C' and A'. Press the *OK* button to complete the edits.

3. To find the perpendicular bisectors of AB and BC, use the *PL=>L Perp* command. The first thing you notice is that these lines probably don't look right—they are ray-like. That's because you are in the mode of making line segments, and you would like to make lines instead. (We make lines that are infinite in both directions for the perpendicular bisectors since they may meet outside the triangle—for medians or angle bisectors we would probably handle the situation differently.) Anyway, click on each of those lines to select them and then select *Line* under the *Line Type* button in the GUI.

4. Next find the circumcenter using the command *LL=>P* button and clicking on the two altitudes. If you're following along exactly, **Geometer** probably labeled it F, and you may want to change its name to O as you did in step 2 by selecting it and typing `Ctrl-n`.

5. Finally, using the *Ctr Edg=>C* command, make a circle centered at O and passing through point A. This completes the construction, so you may want to move points A, B, and C to check your work, and to save the file (perhaps as `circumcircle.T`).

6. By far the easiest way to proceed is with the text editor, so when you're happy with your diagram, under the pull-down menu *Edit* issue the *Edit Geometry* command. An editor window will appear with text that looks roughly like this (the coordinates will obviously be different since you clicked in different places on your screen than I did when you were placing points A, B, and C):

```
.geometry "version 0.31";
v1 = .free(-0.407186, 0.0209581, "A");
v2 = .free(0.239521, 0.598802, "B");
v3 = .free(0.502994, -0.161677, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
v4 = .v.vvmid(v1, v2, "C'");
v5 = .v.vvmid(v2, v3, "A'");
l4 = .l.vlperp(v4, l1, .longline);
l5 = .l.vlperp(v5, l2, .longline);
v6 = .v.ll(l4, l5, "O");
c1 = .c.vv(v6, v1);
```

   Most of what appears above is pretty obvious: `v1`, `v2`, and `v3` are the points labeled A, B, and C, `l1` is the line connecting `v1` and `v2` (or, in other words, the line AB), and similarly for lines `l2` and `l3`.

   Look at the rest of the entries to make sure you understand how they correspond to the items in your drawing. They appear in exactly the same order you issued the GUI commands, and if you have any doubt, the reference section in the advanced tutorial section in Section 5.12.2 explains the syntax and semantics precisely.

7. OK, let's begin with a single change. (When you get good at this, you'll make a whole bunch of changes at once, but for now, let's just make one minor edit to see how it works.) Be very careful to follow instructions *exactly* so that you don't get an error. We'll talk about how to deal with errors later (see step 11).

   Let's make the point `v4` so that it changes colors as you step through the construction. We would like to have it invisible at first, to appear in a blinking color next, and then to change to blue for the rest of the construction. To do this, modify the line:

```
v4 = .v.vvmid(v1, v2, "C'");
```

   to become:

```
v4 = .v.vvmid(v1, v2, "C'", [.in, .blink, .blue]);
```

The information between the square brackets is the color information. The `.in` says that on layer 0, the color is invisible. Then the `.blink` says to paint this point in a blinking color on layer 1, and the final `.blue` says that from layer 2 on, the object should always be painted in blue. Remember the comma after the text: `"C'"`.

After you make this edit, save the file using the *Save* command under the *File* pull-down menu in the text editor window. If you did make a typing error, type the *OK* button on the alert menu and you will be put back in the text editor with the line containing the error highlighted. If you made no error, you should be back to the GUI form of **Geometer**, but the point C' will be invisible.

8. Test your change. Do this by clicking on the *Start* button under *Layer Control* to show only layer 0, and then click on the *Next* button just below it to go to layer 1. If you've done everything right, the point C' will appear and will be blinking. Press *Next* again, and C' should change to blue. Repeated presses of *Next* should have no effect—it will remain blue for the rest of the layers.

   Notice that all the rest of your diagram is visible in all steps. That's because by default, all items are created to appear on all layers.

9. Next, let's make the diagram so that when it is loaded, only layer 0 will be shown so the user will not have to press the *Start* button to get going. Issue the *Edit Geometry* command again (under the *Edit* pulldown menu, or simply by typing `Ctrl-e`), and add a final line to the file that looks like this:

   ```
   .l0;
   ```

   (That's "ell-zero", not "ell-oh"). It tells **Geometer** to begin displaying this diagram displaying only layer zero.

   Save your changes (use *Save* under the *Edit* pulldown, or simply by typing `Ctrl-s`), and note that the diagram appears with only the "0" layer lit under *Layer Control*. If you like, you can press the *Next* button a few times to see that it still has the desired behavior for the appearance of point C'.

10. Bring up the text editor again with *Edit Geometry*, and before you do anything else, take a close look at what you've got:

    ```
    .geometry "version 0.31";
    .l0;
    v1 = .free(-0.407186, 0.0209581, "A");
    v2 = .free(0.239521, 0.598802, "B");
    v3 = .free(0.502994, -0.161677, "C");
    l1 = .l.vv(v1, v2);
    l2 = .l.vv(v2, v3);
    l3 = .l.vv(v3, v1);
    v4 = .v.vvmid(v1, v2, [.in, .blink, .blue], "C'");
    v5 = .v.vvmid(v2, v3, "A'");
    l4 = .l.vlperp(v4, l1, .longline);
    l5 = .l.vlperp(v5, l2, .longline);
    v6 = .v.ll(l4, l5, "O");
    c1 = .c.vv(v6, v1);
    ```

Notice that **Geometer** has changed what you put in! The `.l0;` command has been moved to the top of the file, and the layer information you typed in for point C' has been moved to appear before the `"C'"`.

The reason is that slight reorderings like this make no difference, and **Geometer** doesn't remember exactly what you typed in—it just remembers the effect you want, so when it prints its version, that version may be slightly different from what you typed. The layer command moved to a different line, but the vast majority of other reorderings are within a single line.

11. OK, now let's see how to deal with typing errors. Intentionally make an error by typing an `X` at the end of the line that begins with `l3` to make:

```
l3 = .l.vv(v3, v1);X
```

and save the file with `Ctrl-S`. **Geometer** will barf and will display an alert notifier with the obscure message, "Line 9: Expected '='". For now ignore the message and press the *OK* button on the notifier and you will be returned to the text editor with line 9 highlighted. Usually all that's required is a quick glance to see what's wrong and to fix it, in this case by deleting the `X` you just typed. Go ahead and delete the `X` and save the file again, and you should be back to where you were.

Now edit the file again, but add an `X` to the ends of two different lines and try to save the file. You'll get an error message, but only about the first error, and only that line will be highlighted. Fix that error, try to save again, and you'll be pointed at the second bad line, et cetera. **Geometer** basically gives up as soon as it hits the first error.

Note that sometimes the error is really on the previous line—**Geometer** may not notice there's anything wrong until it tries to interpret the next line.

With more experience, you can use the information in the error message as well. The one above, "Line 9: Expected '='", occurred because beginning with the `X`, **Geometer** was expecting something like

```
X = .free(0, 0);
```

Geometer does not require the commands to be one per line; there can be multiple commands on a single line, or one command can stretch across many input lines.

12. Now that we're comfortable with errors let's continue to fix up our diagram. Bring up the editor and make all of the changes shown below. The first listing is the current state of your file; below it is the target version. The last 5 lines all require modification. Don't necessarily make all the changes at once; edit a line or two, save the file to see if there are errors, fix those errors if necessary, and then edit the file again to continue with the changes.

```
.geometry "version 0.31";
.l0;
```

```
v1 = .free(-0.407186, 0.0209581, "A");
v2 = .free(0.239521, 0.598802, "B");
v3 = .free(0.502994, -0.161677, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
v4 = .v.vvmid(v1, v2, [.in, .blink, .blue], "C'");
v5 = .v.vvmid(v2, v3, "A'");
l4 = .l.vlperp(v4, l1, .longline);
l5 = .l.vlperp(v5, l2, .longline);
v6 = .v.ll(l4, l5, "O");
c1 = .c.vv(v6, v1);

.geometry "version 0.31";
.l0;
v1 = .free(-0.407186, 0.0209581, "A");
v2 = .free(0.239521, 0.598802, "B");
v3 = .free(0.502994, -0.161677, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
v4 = .v.vvmid(v1, v2, [.in, .blink, .blue], "C'");
v5 = .v.vvmid(v2, v3, [.in, .blink, .blue], "A'");
l4 = .l.vlperp(v4, l1, [2 .in, .blink, .blue], .longline);
l5 = .l.vlperp(v5, l2, [2 .in, .blink, .blue], .longline);
v6 = .v.ll(l4, l5, [3 .in, .blink, .blue], "O");
c1 = .c.vv(v6, v1, [4 .in, .blink, .white]);
```

After you have completed the edits above, test your diagram by pressing the *Next* button to see that it behaves correctly—on layer 0, only the triangle appears. On layer 1, the two midpoints, A' and C' appear blinking. On layer 2, the perpendicular bisectors appear blinking, and the midpoints change to blue. On layer 3, point O appears, blinking, and the perpendicular bisectors are blue. On layer 4, the required circle is blinking—all the other construction items are blue. On layers 5 and beyond, the circle is white.

The only new thing that we've done is to add some multipliers in the color/layer specifications. In the specification for point O, for example:

```
v6 = .v.ll(l4, l5, [3 .in, .blink, .blue], "O");
```

the `3` in front of the `.in` signifies that three invisible layers appear. This multiplier could appear anywhere in the specification, and you don't need to use it on input. For example, suppose you wanted some item to be red for the first three steps, then invisible for 2, and finally to appear white for the rest of the layers, you could type this color/layer specification that will work fine:

```
[.red, .red, .red, .in, .in, .white]
```

But after you save it, **Geometer** will print it as:

```
[3 .red, 2 .in, .white]
```

13. Now it's time to add some text. Edit your geometry and add the lines to the end of the file so that the final result looks something like this (again, you may wish to make these edits a few at a time and test them by saving the file and using the *Next* button in the control panel area of **Geometer**):

```
.geometry "version 0.31";
.l0;
v1 = .free(-0.437126, 0.0748503, "A");
v2 = .free(0.239521, 0.598802, "B");
v3 = .free(0.502994, -0.161677, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
v4 = .v.vvmid(v1, v2, [.in, .blink, .blue], "C'");
v5 = .v.vvmid(v2, v3, [.in, .blink, .blue], "A'");
l4 = .l.vlperp(v4, l1, [2 .in, .blink, .blue], .longline);
l5 = .l.vlperp(v5, l2, [2 .in, .blink, .blue], .longline);
v6 = .v.ll(l4, l5, [3 .in, .blink, .blue], "O");
c1 = .c.vv(v6, v1, [.white, 3 .in, .blink, .white]);
.text("Given \triangleABC, construct its circumcircle.", .l0);
.text("Move points A, B, and C to see
what 'circumcircle' means.", .yellow, .l0);
.text("Find the midpoints C' and A' of
segments AB and BC, respectively.", .l1);
.text("Construct a line through C' perpendicular
to AB and a line through A' perpendicular
to BC.", .l2);
.text("Let O be the intersection of those
perpendicular bisectors.", .l3);
.text("The circle centered at O that passes
through A is the required circumcircle.", .l4);
.text("Press 'Next' to continue ...", .red, .tol3);
```

A lot of new ideas are used above—we'll look at them command by command. In the first line you typed:

```
.text("Given \triangleABC, construct its circumcircle.", .l0);
```

the text that appears between the double quotes will appear in the **Geometer** diagram. **Geometer** understands certain combinations of letters, like "`\triangle`", above, to represent a special symbol, in this case, the triangle symbol: $\triangle$. The line will appear in **Geometer** as:

> Given $\triangle$ABC, construct its circumcircle.

The line will appear in the default color (white), and the final token on the line, "`.l0`", tells **Geometer** to display it only on layer 0. In this case, "`.l0`" is shorthand for: "`[.white, .in]`".

The next two lines:

```
.text("Move points A, B, and C to see
what 'circumcircle' means.", .yellow, .l0);
```

Display the two lines of text between the double quotes as two lines in **Geometer**. The lines will also appear in layer 0 only, but this set will be in yellow. The text will look like this on the screen (in yellow, of course):

> Move points A, B, and C to see
> what 'circumcircle' means.

Notice that the line breaks on the screen appear wherever newlines were typed in the `.text` command.

Not much new appears in the next four commands, except that the text is presented on different layers: `.l1` for layer 1, et cetera.

Finally, the last line:

```
.text("Press 'Next' to continue ...", .red, .tol3);
```

is drawn in red, and the layer control "`.tol3`" means that it appears from layer 0 to layer 3—in other words, on four layers. It could have been done with: `[4 .red, .in]`.

Geometer *always* interprets a file in order, so if there are multiple lines of text that appear on the visible layer, they will be drawn as they appear in the file. In the example above, there are three `.text` commands that display something on layer 0, so the layer 0 text will look like this on the **Geometer** screen:

> Given $\triangle$ABC, construct its circumcircle.
> Move points A, B, and C to see
> what 'circumcircle' means.
> Press 'Next' to continue ...

The first line will be drawn in white, the next two in yellow, and the final line in red.

There, you've finished with the first example! One final thing worth noting is that although the demonstration "officially" ends on the fifth layer (layer 4), if you step on to layer 5, you get a non-blinking version of everything that is suitable for printing. In addition, the printing version has no text (although it could, if you wanted to, but usually the text in your document will be sufficient). Figure 6.1 is what you get if you "print" the figure that appears on layer 5.

6.2 A Simple Proof: Equal Sides $\Longrightarrow$ Equal Angles

In this section, we'll construct a **Geometer** diagram to prove that if two sides of a triangle are equal, then the angles opposite those sides are also equal.

The first problem we face is how to construct the diagram so that the student will be able to manipulate it in such a way that two of the lines will remain the same length. We are trying to construct something like Figure XXX, where segments AB and AC are always equal. There are various approaches, but for this example, I've chosen to make points B and C completely free, but to constrain point A to lie on the perpendicular bisector of the segment BC. See Figure 6.2

Here are the steps I followed; as before, I highly recommend that you follow along and generate the same diagram in your own version of **Geometer**.

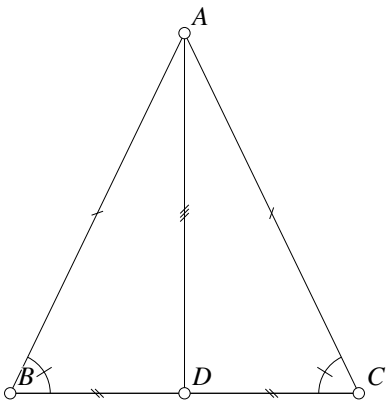

Figure 6.2: Equal Sides Have Equal Angles Opposite Them
`Teachers/Sidesangles.T` [P]

1. We know that as you create points, **Geometer** names the first one "A", the second one "B", et cetera. Being too lazy to change the names of points, I created three free points, A, B, and C, and then deleted point A. To delete a point, select it by clicking on it, and then type `Ctrl-d`, or choose the *Delete* command from the *Edit* pull-down menu.

   In fact, the fastest approach is this: Hold the `Ctrl` button down while you click on *Free P* so that you can create a bunch of points, click once to make the point A, and immediately type `Ctrl-d` to delete it, then click twice more to make points labeled B and C. Finally, click *Cancel Repeat Mode* to get out of the repeat creation mode.

2. Next, construct the segment BC (using *PP=>L*) that will be one edge of the triangle, and find the midpoint of it using *PP=>P Mid*. This midpoint will be labeled D, which is what we want. Now, using *PL=>L Perp* make the perpendicular bisector of the segment BC passing through D*. Finally we are in a position to add point A using *P on L* and clicking on the perpendicular bisector you just constructed. **Geometer** will label this new point "E", so type `Ctrl-n` to edit its name to "A". Finally, click on the perpendicular bisector to select it, and change its color to "invisible" using the *Color* button in the control area of **Geometer**. Complete the triangle by constructing segments AB and AC.

   Test your diagram (and save it to disk if you like) by moving points A, B, and C. B and C should move completely freely, but point A should be constrained to lie on the perpendicular bisector of BC (and hence the lengths of AB and AC will remain equal, as we desired).

*__Geometer__ provides another command, *PP=>L Perp Bis*, that constructs the perpendicular bisector of a segment directly without requiring the midpoint. This command (available only in the pull-down menu) could be used, but there's no real advantage, since we are going to need the point D later anyway.

To complete the proof, we are going to need line segment AD, so draw that now as well.

3. Now we have all the basic lines that we'll need, but we need to fix up the layer colors, add text, and add a few more interesting touches.

   Let's begin by marking the lines and angles that are supposed to be congruent so that the student can see more clearly what's going on. Select line segment AB and use the cascaded pull-down *Style* menu to set that line to *1 Slash*. Do the same thing for segment AC. Both will now be drawn with a single slash through them.

   We also want to show that the base angles, $\angle ACB$ and $\angle ABC$ are equal, so first use the *Angle Type* button in the control panel to set the default angle type to *1 Slash*. Now use the *PPP=>A* command in the control panel to make two angles above. Click on the points in the order they appear in the angle description; for example, to draw $\angle ACB$, click on point A, then C, and finally, B. In certain cases, the angles you get may be "inside-out"—in other words, the reflex angle is drawn. If that happens, there's a *Flip Angle* command under the *Edit* pull-down menu that's also available as the keyboard `Ctrl-a` command.

4. Now we have most of the geometry we need, so lets fix up the layer colors using the text editor. When you issue the first *Edit Geometry* command, this is roughly what you should have in the text editor:

```
.geometry "version 0.31";
v2 = .free(-0.254936, -0.168969, "B");
v3 = .free(0.429834, -0.171934, "C");
l1 = .l.vv(v2, v3);
v1 = .v.vvmid(v2, v3, "D");
l2 = .l.vlperp(v1, l1, .in);
v4 = .vonl(l2, 0.090513, 0.537185, "A");
l3 = .l.vv(v4, v2, .line1slash);
l4 = .l.vv(v4, v3, .line1slash);
l5 = .l.vv(v4, v1);
ang1 = .a.vvv(v4, v3, v2, .slash1);
ang2 = .a.vvv(v3, v2, v4, .slash1);
```

   Convert it to what follows by adding the layer color information to four of the lines, and by adding the `.text` entries:

```
.geometry "version 0.31";
.l0;
v2 = .free(-0.254936, -0.168969, "B");
v3 = .free(0.429834, -0.171934, "C");
l1 = .l.vv(v2, v3);
v1 = .v.vvmid(v2, v3, [.in, .blink, .blue], "D");
l2 = .l.vlperp(v1, l1, .in);
v4 = .vonl(l2, 0.090513, 0.537185, "A");
l3 = .l.vv(v4, v2, .line1slash);
l4 = .l.vv(v4, v3, .line1slash);
l5 = .l.vv(v4, v1, [2 .in, .blink, .blue]);
ang1 = .a.vvv(v4, v3, v2, [.white, 3 .in, .blink, .white], .slash1);
ang2 = .a.vvv(v3, v2, v4, [.white, 3 .in, .blink, .white], .slash1);
```

```
.text("Show that if AB \congruent AC in \triangleABC, then
\angleACB = \angleABC.", .l0);
.text("Move points A, B, and C.", .yellow, .l0);
.text("Let D be the midpoint of segment BC, so
BD \congruent CD.", .l1);
.text("Construct line AD which is congruent
to itself.", .l2);
.text("\triangleACD \congruent \triangleABD by SSS, since
AB \congruent AC, AD \congruent AD, and BD \congruent CD.", .l3);
.text("Since \triangleACD \congruent \triangleABD, we
have \angleACB \congruent \angleABC, which we
wanted to show.", .l4);
.text("Press 'Next' to continue ...", .red, .tol3);
```

You now have a proof that is acceptable, but which can be vastly improved.

For example, it would be nice to have all the congruent segments marked as are AB and AC, and it would be nice to somehow highlight the two triangles that we showed to be congruent: $\triangle ACD$ and $\triangle ABD$.

The easiest way to do this is using the GUI of **Geometer**. We want the segments BD and CD to be congruent, so just use the *PP=>L* command to add those lines, but before you do that, issue the *2 Slash* command under the *Styles→Lines* cascading pull-down menu. The layer colors will be wrong, but we can fix those later in the text editor.

Using *Next*, we can also get to a point where we can see the segment AD, and put three slashes on it with the *3 Slash* command under the same cascading menu.

To fix the layer colors for the segments BD and CD, use the editor to convert:

```
l6 = .l.vv(v2, v1, .l1, .line2slash);
l7 = .l.vv(v3, v1, .l1, .line2slash);
```

to

```
l6 = .l.vv(v2, v1, [.in, .white], .line2slash);
l7 = .l.vv(v3, v1, [.in, .white], .line2slash);
```

Save these changes, and test the file again—it's a little better, but we can improve it just a little by having all three sets of congruent sides blinking in three different blinking colors (`.blink`, `.blink1`, and `.blink2`) on layer 3, where we state that the triangles are congruent by SSS. Those modifications, plus a couple of others to make everything appear in white on the layer past the end yield a file that looks something like this:

```
.geometry "version 0.31";
.l0;
v2 = .free(-0.254936, -0.168969, "B");
v3 = .free(0.429834, -0.171934, "C");
l1 = .l.vv(v2, v3);
v1 = .v.vvmid(v2, v3, [.in, .blink, 2 .blue, .white], "D");
l2 = .l.vlperp(v1, l1, .in);
v4 = .vonl(l2, 0.090513, 0.537185, "A");
l3 = .l.vv(v4, v2, [3 .white, .blink, .white], .line1slash);
l4 = .l.vv(v4, v3, [3 .white, .blink, .white], .line1slash);
```

```
l5 = .l.vv(v4, v1, [2 .in, .blink, .blink2, .white], .line3slash);
ang1 = .a.vvv(v4, v3, v2, [.white, 3 .in, .blink, .white], .slash1);
ang2 = .a.vvv(v3, v2, v4, [.white, 3 .in, .blink, .white], .slash1);
.text("Show that if AB \congruent AC in \triangleABC, then
\angleACB = \angleABC.", .l0);
.text("Move points A, B, and C.", .yellow, .l0);
.text("Let D be the midpoint of segment BC, so
BD \congruent CD.", .l1);
.text("Construct line AD which is congruent
to itself.", .l2);
.text("\triangleACD \congruent \triangleABD by SSS, since
AB \congruent AC, AD \congruent AD, and BD \congruent CD.", .l3);
.text("Since \triangleACD \congruent \triangleABD, we
have \angleACB \congruent \angleABC, which we
wanted to show.", .l4);
.text("Press 'Next' to continue ...", .red, .tol3);
l6 = .l.vv(v2, v1, [.in, 2 .white, .blink1, .white], .line2slash);
l7 = .l.vv(v3, v1, [.in, 2 .white, .blink1, .white], .line2slash);
```

6.3 A Trapezoid has Perpendicular Diagonals

Next, we'll construct a proof that the diagonals of a trapezoid are perpendicular. (A trapezoid is a convex quadrilateral with four equal sides.)

The first problem is to draw a figure that the student can manipulate in such a way that four points always form a trapezoid. The solution I selected has two completely free points, A and B, and the third point C lies on a circle centered at B and passing through A. This will guarantee that $AB \cong BC$. To do this construction, put down two free points A and B, draw the circle centered at B and passing through A, and then use the *P on C* to make the point C lying on that circle. We don't want the circle cluttering up the diagram, so click on the circle to select it, and paint it the invisible color.

The locations of A, B, and C completely determine D. Since we know that the theorem is true, we can take advantage of it, and construct D. The easiest way I could think to do it is to draw the line AC (which we will need later as part of the proof), and then to reflect the point B across AC to get the fourth point D of the trapezoid. To do this, you need the command *LP=>P Mirror* that can be found in the cascading pulldown menu *Primitives→Point*.

Connect the four points AB, BC, CD, and DA with segments to form the trapezoid. Now kick yourself, select the segments individually, and convert each to a segment with a single slash through it with the cascading pull-down menu *Styles→Line→1 Slash*. (Kick yourself because if you had selected this style before drawing the four lines, they would all automatically have gotten the slash.)

Finally, draw in the two diagonals (but this time, remember to turn off the slash before doing so with the menu entry *Styles→Line→No Mark*).

Test the diagram by moving points A, B, and C to make sure that D moves appropriately, save the file, and now you're ready to start building in the proof.

Use the editor to put in the first few lines that explain the theorem to prove, and remember to add the line `.l0;` to put **Geometer** into the proof mode when the file is loaded.

In short, add these lines. (Note the use of the `\congruent` and `\perp` commands that draw the congruence symbol (≅), and the perpendicular symbol (⊥).

```
.l0;
.text("Prove that the diagonals of a trapezoid
are perpendicular.  In other words, if
AB \congruent BC \congruent CD \congruent DA then AC \perp BD.
Move points A, B, and C.", .l0);
```

Next we show that various internal angles are equal (for example, $\angle ACD \cong \angle CAD$) since they lie opposite equal sides. To do this, make the angles appear at the appropriate steps of the proof, and at the same time, it would be nice if the sides opposite them also blinked, perhaps in a different color.

We continue to step through the proof, adding a line of text at a time, and then modifying the layer colors of the various lines and angles so that things blink and stop blinking at the correct times, and eventually, we get to the following fairly good proof:

```
.geometry "version 0.31";
.l0;
v1 = .free(-0.299401, 0.508982, "A");
v2 = .free(-0.38024, -0.11976, "B");
c1 = .c.vv(v2, v1, .in);
v3 = .vonc(c1, 0.22768, 0.0599201, "C");
l1 = .l.vv(v1, v3, [3 .white, .blink2, .white]);
v6 = .v.lvmirror(l1, v2, "D");
l2 = .l.vv(v1, v2, [2 .white, .blink1, .blink, 3 .white,
    .blink2, .white], .line1slash);
l3 = .l.vv(v2, v3, [2 .white, 2 .blink1, .white, .blink,
    .white, .blink2, .white], .line1slash);
l4 = .l.vv(v3, v6, [.white, .blink1, .white, .blink1,
    .white], .line1slash);
l5 = .l.vv(v6, v1, [.white, .blink1, .white, .blink,
    .white, .blink, .white], .line1slash);
l6 = .l.vv(v2, v6);
v7 = .v.ll(l1, l6, .l7on, "O");
.text("Prove that the diagonals of a trapezoid
are perpendicular.  In other words, if
AB \congruent BC \congruent CD \congruent DA then AC \perp BD.
Move points A, B, and C.", .l0);
.text("In \triangleACD, AD \congruent CD, so
\angle CAD \congruent \angle ACD.", .l1);
ang1 = .a.vvv(v7, v1, v6, [.in, .blink, 2 .white,
    .blink, .white], .slash1);
ang2 = .a.vvv(v6, v3, v1, [.in, .blink, 2 .white, .blink,
    .white], .slash1);
.text("Similarly, n \triangleACB, AB \congruent CB, so
\angle CAB \congruent \angle BCA.", .l2);
ang3 = .a.vvv(v2, v1, v7, [2 .in, .blink, .white, .in], .slash2);
ang4 = .a.vvv(v1, v3, v2, [2 .in, .blink, .white, .in], .slash2);
.text("But AD \congruent AB, CD \congruent CB, and AC
is congruent to itself, so \triangleBAC \congruent \triangleDAC.", .l3);
.text("Since \triangleBAC \congruent \triangleDAC, we have
\angleDAC \congruent \angleDCA \congruent \angleBAC \congruent \angleBCA.",
    .l4);
ang5 = .a.vvv(v1, v3, v2, [4 .in, .blink, 2 .white, .blink,
    .white], .slash1);
```

```
ang6 = .a.vvv(v2, v1, v3, [4 .in, .blink, 2 .white, .blink,
    .white], .slash1);
.text("Since \angleBCA \congruent \angleDAC, lines AD and BC are
parallel, so the transversal BD makes \angleADB \congruent CBD.",
.l5); ang7 = .a.vvv(v1, v6, v2, [5 .in, 2 .blink1, .white],
.ring2); ang8 = .a.vvv(v3, v2, v6, [5 .in, 3 .blink1, .white],
.ring2); .text("But since AB \congruent AD in \triangle ABD we
have \angle DBA \congruent ADB \congruent \angleCBD.", .l6); ang9
= .a.vvv(v6, v2, v1, [6 .in, 2 .blink1, .white], .ring2);
.text("Let O be the intersection of AC and DB.  Then since AB
\congruent BC, \angleOBA \congruent \angleOBC, and \angleBAO
\congruent \angleBCO, \triangleAOB \congruent \triangleCOB.",
    .l7);
.text("Since \triangleAOB \congruent \triangleCOB,
\angleBOA \congruent \angleBOC, but since those
are supplementary angles,
\angleBOA = \angle BOC = 90\degrees.", .l8);
ang10 = .a.vvv(v1, v7, v2, [8 .in, .blink, .white], .right);
ang11 = .a.vvv(v2, v7, v3, [8 .in, .blink, .white], .right);
.text("Press 'Next' to continue ...", .red, .tol7);
```

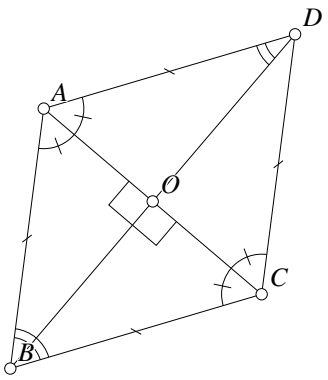

Figure 6.3: Perpendicular Diagonals of a Trapezoid
`Teachers/Trapezoid.T` [P]

Try loading a copy of this proof from the file `Trapezoid.T` and step through it as you read the following comments about what goes on for each layer setting.

The code above was not modified to its current condition in one pass. I stepped to each stage of the proof, figured out what colors I wanted to change, and then went in with the editor and changed the layer colors on those items for that step only. Then I tested the entire diagram again. It took about an hour to get it to its current state, starting from scratch.

Layer 0: This is a standard starting configuration; the statement of the theorem appears, the diagram shows the initial conditions (in this case, equal-length lines), there are instructions about how to manipulate the figure to test the theorem, and there is an indication that there is more to follow "*Press 'Next' to continue ...*".

Layer 1: Two angles are proven equal since they lie opposite equal sides. The equal sides are highlighted in one blinking color and the angles that must also be equal appear in the diagram for the first time in a different blinking color.

Layer 2: Next, we repeat the same argument for a different pair of sides and angles. The same highlighting technique is used, but since all we know at this point is that the second pair of angles are equal to each other (and not necessarily to the first pair of angles), we display the new angles with a double arc.

Layer 3: The corresponding sides of the triangles that are proven congruent are displayed in three different blinking colors. Each corresponding pair is shown in a different color.

Layer 4: Now we know that the four angles mentioned above are all equal because of the congruence of the two triangles in the previous step, so they can be displayed with the same angle markings. To do this, the angles $\angle BAC$ and $\angle BCA$ are actually included in the diagram twice—once with a single arc, and once with a double arc. The double arc version is displayed for the first few steps of the proof while the single arc version is invisible. Once we know they are equal, the double arc version is invisible and the single arc version is shown.

Layer 5: The lines known to be parallel are shown in a blinking color, and the corresponding equal angles cut by a transversal are shown in a different blinking color.

Layer 6: Another angle is shown to be equal to the equal angles made by the transversal of the parallel lines, so all three equal angles are shown as blinking.

Layer 7: The angle, side, and angle used to prove the triangles are congruent by ASA are shown in three different blinking colors.

Layer 8: This is the end of the official proof, and the angle between the diagonals of the trapezoid is shown as a right angle, which is what we are trying to prove. Notice that the "*Press 'Next' to continue ...*" has disappeared.

Layer 9: There's nothing interesting here for the student, but we can use layer 9 to print a PostScript file for inclusion in documentation or whatever. The result on layer 9 is displayed in Figure 6.3.

6.4 Intersection of Three Circles

If points A', B', and C' are selected on the sides BC, CA, and AB of $\triangle ABC$, then the three circles passing through $AB'C'$, through $BA'C'$, and through $CB'A'$ all meet at a point.

This is an easy construction to make using **Geometer**, and the code for the diagram `ThreeCircles.T` is shown below:

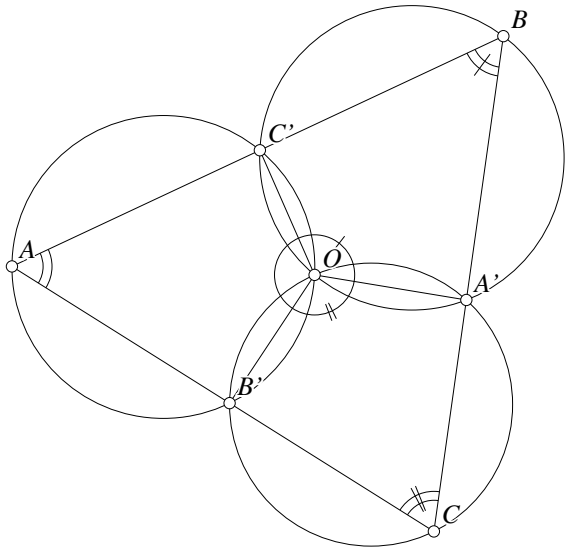

Figure 6.4: Intersection of Three Circles
Teachers/ThreeCircles.T [P]

```
.geometry "version 0.31";
.l0;
v1 = .free(-0.571856, 0.0598802, "A");
v2 = .free(0.571856, 0.595808, "B");
v3 = .free(0.41018, -0.556886, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
v4 = .vonl(l1, 0.005177, 0.33027, "C'");
v5 = .vonl(l2, 0.48574, -0.0181719, "A'");
v6 = .vonl(l3, -0.0652644, -0.258284, "B'");
c1 = .c.vvv(v6, v5, v3);
c2 = .c.vvv(v5, v4, v2);
c3 = .c.vvv(v1, v4, v6, [.white, 3 .in, .magenta, .white]);
v7 = .v.cc(c1, c2, 2, [.in, .blink, .white], "O");
.text("Theorem: Given any \triangleABC, select
points A', B', and C' on BC, CA, and AB,
respectively.  The circles passing through
CB'A', BA'C', and AC'B' all meet at a point.
Move points A, B, C, A', B', and C'.", .l0);
.text("Let O be the intersection different from
A' of circles BA'C' and CB'A'.  Construct segments
OA', OB' and OC'.", .l1);
l4 = .l.vv(v7, v5, [.in, .blink, .white]);
l5 = .l.vv(v7, v6, [.in, .blink, .white]);
l6 = .l.vv(v7, v4, [.in, .blink, .white]);
.text("Since CA'OB' and BA'OC' are concyclic,
we have:", .l2);
```

```
.text("(1) \angle A'OC' + \angle C'BA' = 180\degrees
(2) \angle B'OA' + \angle A'CB' = 180\degrees", [2 .in, .white, .yellow, .in]);
ang1 = .a.vvv(v5, v7, v4, [2 .in, .blink, .blink1, .white], .slash1);
ang2 = .a.vvv(v4, v2, v5, [2 .in, 2 .blink, .white], .slash2);
ang3 = .a.vvv(v6, v7, v5, [2 .in, 2 .blink1, .white], .dslash1);
ang4 = .a.vvv(v5, v3, v6, [2 .in, .blink1, .blink, .white], .dslash2);
ang5 = .a.vvv(v4, v7, v6, [3 .in, .blink1, .white]);
ang6 = .a.vvv(v3, v1, v2, [3 .in, .blink, .white], .ring2);
.text("(3) \angleC'BA' + \angleA'CB' + \angleB'AC' = 180\degrees
(4) \angleA'OC' + \angleB'OA' + \angle C'OB' = 360\degrees
Calculate (3)+(4)-(1)-(2) giving", .l3);
.text("Since", .l4);
.text("\angleC'OB' + \angle B'AC' = 180\degrees", .l3, .l4);
.text("We know that AC'OB' are concyclic.", .l4);
.text("Press 'Next' to continue ...", .red, .tol3);
```

Step through the proof in **Geometer** as you read the discussion about each layer below, and refer to the text above to see how the various effects are achieved.

Layer 0: As usual, the initial layer simply shows the theorem, explains how to manipulate the figure, and indicates with the "*Press 'Next' to continue ...*" that there is more to come.

Layer 1: Newly constructed lines and points appear in a blinking color.

Layer 2: Pairs of supplementary angles are shown in different blinking colors. Watch what happens to equations (1) and (2) on the next layer, and look at the code to see how this was done.

Layer 3: Sets of angles that add to 180° (since they are the angles of a triangle) and angles that add to 360° are shown in different blinking colors.

Layer 4: Again, check to see how the information from the top equation was carried over from the previous layer.

Layer 5: Finally, the proof is long over, but this page can be used to make a PostScript diagram like the one that appears in Figure 6.4.

6.5 A Binary Counter

This example provides a nice example of how the various layer colors work. The goal is to illustrate a binary counter where the digit 0 is represented by the color red and the digit 1 by the color green. As you step through the "proof", one layer at a time is shown, and the colors of four different squares will display the binary value for that number as a combination of red and green squares.

You can make the squares (or whatever other shapes you may like) using **Geometer**'s GUI, or, as I did in this example, you can simply type in the exact coordinates that you like so that the objects are uniform and uniformly spaced. Here is the code for the binary counter:

```
 .geometry "version
0.31"; .l0; v1a = .pinned(0.5, -0.1, .in); v1b = .pinned(0.5, 0.1,
.in); v1c = .pinned(0.7, 0.1, .in); v1d = .pinned(0.7, -0.1, .in);
v2a = .pinned(0.2, -0.1, .in); v2b = .pinned(0.2, 0.1, .in); v2c =
.pinned(0.4, 0.1, .in); v2d = .pinned(0.4, -0.1, .in); v4a =
.pinned(-0.1, -0.1, .in); v4b = .pinned(-0.1, 0.1, .in); v4c =
.pinned(0.1, 0.1, .in); v4d = .pinned(0.1, -0.1, .in); v8a =
.pinned(-0.4, -0.1, .in); v8b = .pinned(-0.4, 0.1, .in); v8c =
.pinned(-0.2, 0.1, .in); v8d = .pinned(-0.2, -0.1, .in); p1 =
.polygon(4, v1a, v1b, v1c, v1d, [.red, .green, .red, .green,
          .red, .green, .red, .green, .red, .green, .red, .green,
          .red, .green, .red, .green], .solidpoly);
p2 = .polygon(4, v2a, v2b, v2c, v2d, [2 .red, 2 .green, 2 .red,
          2 .green, 2 .red, 2 .green, 2 .red, .green], .solidpoly);
p4 = .polygon(4, v4a, v4b, v4c, v4d, [4 .red, 4 .green, 4 .red,
          .green], .solidpoly);
p8 = .polygon(4, v8a, v8b, v8c, v8d, [8 .red, .green], .solidpoly);
```

The four polygons, `p1`, `p2`, `p4`, and `p8`, represent the four binary digits. The one's digit alternates red and green on every step; the two's digit alternates every two steps, et cetera.

6.6 An Improved Binary Counter

The code below improves on the counter above. Instead of using red and green to represent the digits, we actually draw out a zero and a one. The zero is made with an ellipse, and the one with a line segment. To draw the zero, I played around with the five points until I had a shape I liked; then I pinned the points by converting the `.free` to `.pinned` in the editor. To get exact copies of my ellipses, I simply translated the original points to the right to make additional copies.

Then I used the same alternation of colors in the layer color commands, but made sure that when the zero was showing, the one was not, and vice-versa.

Here's the code:

```
.geometry "version 0.31";
.l0;
va = .pinned(0, 0.01, .in);
vb = .pinned(0.05, 0, .in);
vc = .pinned(0.026, 0.2, .in);
vd = .pinned(0.025, -0.2, .in);
ve = .pinned(0.0479042, -0.0898204, .in);
wa = .pinned(0.025, 0.2, .in);
wb = .pinned(0.025, -0.2, .in);
va1 = .v.vtranslate(va, 0.100000, 0.000000, .in);
vb1 = .v.vtranslate(vb, 0.100000, 0.000000, .in);
vc1 = .v.vtranslate(vc, 0.100000, 0.000000, .in);
vd1 = .v.vtranslate(vd, 0.100000, 0.000000, .in);
ve1 = .v.vtranslate(ve, 0.100000, 0.000000, .in);
wa1 = .v.vtranslate(wa, 0.100000, 0.000000, .in);
wb1 = .v.vtranslate(wb, 0.100000, 0.000000, .in);
va2 = .v.vtranslate(va, 0.200000, 0.000000, .in);
```

```
vb2 = .v.vtranslate(vb, 0.200000, 0.000000, .in);
vc2 = .v.vtranslate(vc, 0.200000, 0.000000, .in);
vd2 = .v.vtranslate(vd, 0.200000, 0.000000, .in);
ve2 = .v.vtranslate(ve, 0.200000, 0.000000, .in);
wa2 = .v.vtranslate(wa, 0.200000, 0.000000, .in);
wb2 = .v.vtranslate(wb, 0.200000, 0.000000, .in);
va3 = .v.vtranslate(va, 0.300000, 0.000000, .in);
vb3 = .v.vtranslate(vb, 0.300000, 0.000000, .in);
vc3 = .v.vtranslate(vc, 0.300000, 0.000000, .in);
vd3 = .v.vtranslate(vd, 0.300000, 0.000000, .in);
ve3 = .v.vtranslate(ve, 0.300000, 0.000000, .in);
wa3 = .v.vtranslate(wa, 0.300000, 0.000000, .in);
wb3 = .v.vtranslate(wb, 0.300000, 0.000000, .in);
zero1 = .conic.vvvvv(va3, vb3, vc3, vd3, ve3, [.white, .in,
      .white, .in, .white, .in, .white, .in, .white, .in,
      .white, .in, .white, .in, .white, .in]);
zero2 = .conic.vvvvv(va2, vb2, vc2, vd2, ve2, [2 .white,
       2 .in, 2 .white, 2 .in, 2 .white, 2 .in, 2 .white, .in]);
zero4 = .conic.vvvvv(va1, vb1, vc1, vd1, ve1, [4 .white,
       4 .in, 4 .white, .in]);
zero8 = .conic.vvvvv(va, vb, vc, vd, ve, [8 .white, .in]);
one1 = .l.vv(wa3, wb3, [.in, .white, .in, .white, .in, .white,
       .in, .white, .in, .white, .in, .white, .in, .white,
       .in, .white]);
one2 = .l.vv(wa2, wb2, [2 .in, 2 .white, 2 .in, 2 .white,
       2 .in, 2 .white, 2 .in, .white]);
one4 = .l.vv(wa1, wb1, [4 .in, 4 .white, 4 .in, .white]);
one8 = .l.vv(wa, wb, [8 .in, .white]);
```

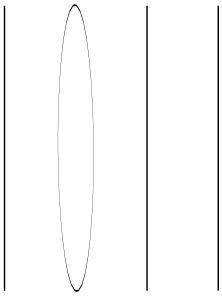

Figure 6.5: Layer 11 of the Binary Display Program
`Teachers/Binary1.T` [P]

Figure 6.5 shows the result that appears on layer number 11.

6.7 ♦Plotting Curves

You can (mis)use **Geometer** as a general graphing package, although it can be a bit clumsy. A lot of the clumsiness comes from the fact that **Geometer** insists on a coordinate system with $(0, 0)$ in the center of the drawing area and that runs from -1.0

to 1.0 in the shorter screen direction. In what follows, we'll assume that the **Geometer** drawing area is square, so its coordinate system will run from -1.0 to 1.0 in both directions.

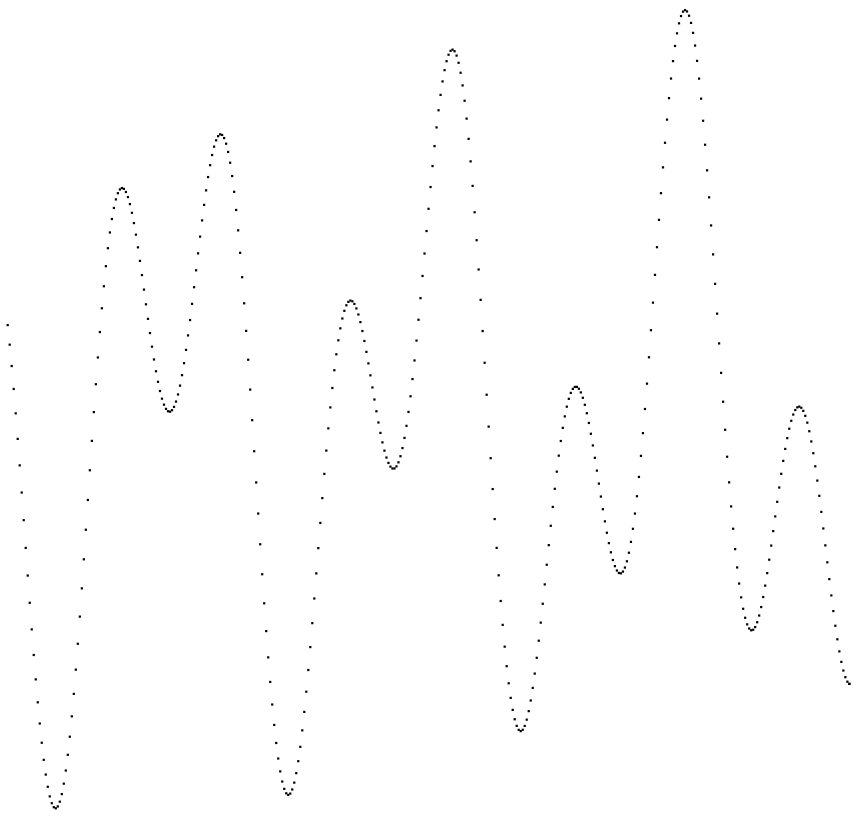

Figure 6.6: Plot of $f(x) = \cos 5x + \sin 11x$.
`Teachers/Plot.T` [S]

For this example, we'll simply plot the function:

$$f(x) = \cos 5x + \sin 11x.$$

Just eyeballing the function above, we can see that it must range in size between -2.0 and 2.0, so we'll probably want to multiply the x and y values by a number slightly smaller than 0.5 to make it fit. Thus, the input (x) values will also range from -2.0 to 2.0. In fact, if we make this input range a bit larger, the curve will run off both ends, and will thus be guaranteed to fill as much of the screen as possible. (By the way, the angles are measured in radians, or we wont see much!)
Once we've made those decisions, the code is pretty simple:

```
.geometry "version 0.31";
.radianmode;
x = .script(-2.100000, 2.100000, 0.010000);
y = .f.rpn(x, 5.000000, .mul, .cos, x,
        11.000000, .mul, .sin, .add, 0.450000,
        .mul);
x1 = .f.rpn(x, 0.450000, .mul);
v = .v.ff(x1, y, .smear, .dot);
```

The first line `.radianmode;` puts **Geometer** in radianmode; it measures angles in degrees by default.

The second line tells **Geometer** that the diagram is to be a script—that the *Run Script* button should be active, and when pressed, **Geometer** will repeatedly evaluate the entire program using values of `x` that run between -2.1 and 2.1 in steps of 0.01. If the spacing of the dots is wrong, you can make them more or less dense by changing value of 0.01 appropriately.

The third line does the calculation of the `y`-value: it takes `x`, multiplies it by 5 and takes the cosine, then takes another copy of `x` and multiplies it by 11, takes the sine, and then adds together the values. Finally, the result is multiplied by 0.45 (a number slightly less than 0.5) to keep the plot in range. `x1` is a similarly scaled value of `x`.

Finally, a point `v` is plotted with coordinates `x1` and `y`. It is plotted in the smearing color so that all the dots will appear on the screen. It is drawn as a single dot (using `.dot` as point type) so as not to clutter the screen too much.

The final result can be seen in Figure 6.6.

6.8 ♦Plotting Parametric Curves

It is also easy to plot parametric curves in the same way, where both the x- and y-coordinates are functions of a parameter t. Just let the `.script` command generate values of t, calculate x and y values using `.f.rpn`, and plot away.

Functions in polar coordinates are slightly more interesting. In this case, the function relates the angle θ with the radius, ρ. ρ or θ can usually be the parameter, and the other variable is calculated in terms of it, but then the resulting values must be converted to x- and y-coordinates so **Geometer** can deal with them. The conversion is simple, however:

$$\begin{aligned} x &= \rho\cos\theta \\ y &= \rho\sin\theta. \end{aligned}$$

As an example, let's plot the function

$$\rho = 0.5 + (\cos 10\theta)/3.$$

The result can be seen in Figure 6.7.
Here's the code to do it:

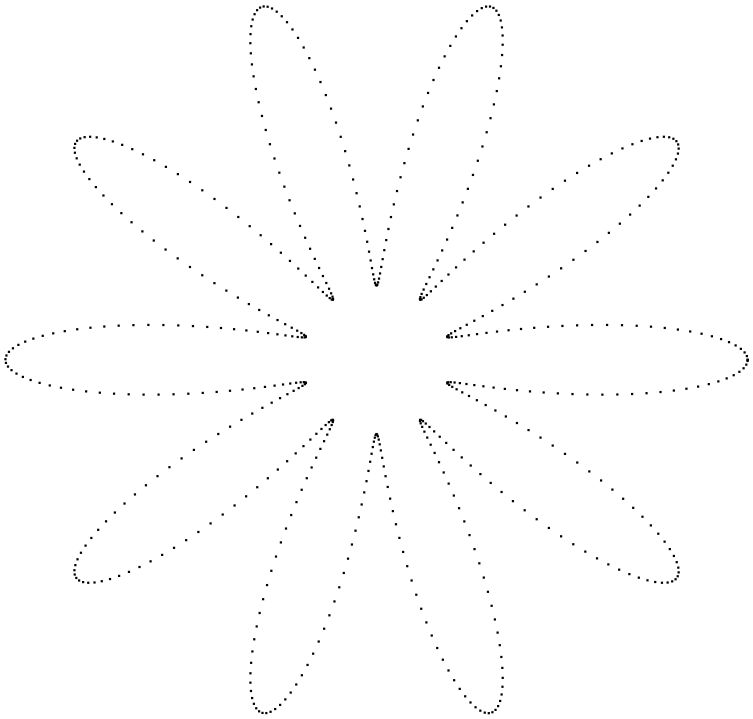

Figure 6.7: Plot of $\rho = 0.5 + (\cos 10\theta)/3$.
`Teachers/Plot1.T` [S]

```
.geometry "version 0.31";
.radianmode;
theta = .script(0.000000, 6.283100, 0.010000);
rho = .f.rpn(0.500000, theta, 10.000000, .mul, .cos,
        3.000000, .div, .add);
x = .f.rpn(theta, .cos, rho, .mul);
y = .f.rpn(theta, .sin, rho, .mul);
v = .v.ff(x, y, .smear, .dot);
```

6.9 Ellipse Macro

An ellipse is commonly defined in terms of two foci F_1 and F_2 and a length, l. The ellipse is the set of all points P such that $F_1P + F_2P = l$. Equivalently, it can be

defined in terms of the foci F_1, F_2, and a point P, and the point X is on the ellipse if $F_1P + F_2P = F_1X + F_2X$. This second definition is better for **Geometer** diagrams since **Geometer** allows you to manipulate points freely, and it's a little messier to manipulate a length. The conic sections that can be defined in **Geometer**, however, only include those that pass through 5 points, or those that are tangent to 5 lines.

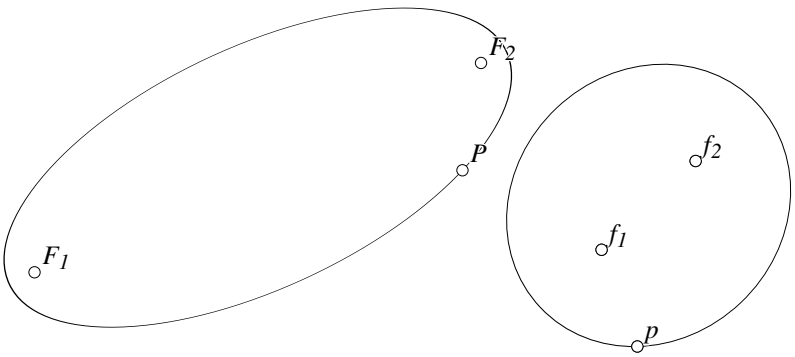

Figure 6.8: An Ellipse Macro
`Teachers/Ellipse.T` [P]

In this section we'll construct a macro that takes the two foci and a point on the boundary as input and draws the ellipse. In Figure 6.8 is a demonstration of two calls to the macro, one with focus points at F_1, F_2, and passing through point P, and the other with foci f_1, f_2, and passing through p.

Here is the **Geometer** code to generate Figure 6.8:

```
.geometry "version 0.32";
v1 = .free(-0.0299401, 0.479042, "F\sub{1}");
v2 = .free(0.389222, 0.317365, "F\sub{2}");
v3 = .free(0.203593, 0.0508982, "P");
v4 = .free(-0.233533, -0.505988, "f\sub{1}");
v5 = .free(0.673653, -0.41018, "f\sub{2}");
v6 = .free(0.718563, -0.473054, "p");
.macro conic(.vertex f1, .vertex f2, .vertex p)
{
    d1 = .f.vv(f1, p);
    d2 = .f.vv(f2, p);
    sum = .f.rpn(d1, d2, .add);
    r1 = .f.rpn(sum, 0.550000, .mul);
    r2 = .f.rpn(sum, 0.450000, .mul);
    c1 = .c.vf(f1, r1, .in);
    c2 = .c.vf(f2, r1, .in);
    c3 = .c.vf(f1, r2, .in);
    c4 = .c.vf(f2, r2, .in);
    v4 = .v.cc(c2, c3, 1, .in, "D");
    v5 = .v.cc(c1, c4, 2, .in, "E");
```

```
    v6 = .v.cc(c1, c4, 1, .in, "F");
    v7 = .v.cc(c2, c3, 2, .in, "G");
    con1 = .conic.vvvvv(p, v7, v6, v5, v4);
}
conic(v1, v2, v3);
conic(v4, v5, v6);
```

The code is fairly straight-forward—it takes the sample points, calculates the length (called `sum`), and then draws a pair of circles around the points of lengths .45 and .55 of the total length. The intersections of these circles will be points on the ellipse as well as the original point. The ellipse is the conic that passes through the original point and through the four circle intersections.

6.10 ♦♦Angle Subdivision

This example isn't particularly useful but it does make use of a bunch of new diagram construction techniques, especially the use of the arithmetic operations available for floating point numbers as well as some other tricks.

The diagram will consist of a fixed line marked from 2 to 11 at the top of the screen, and an angle $\angle ABC$ below whose size can be modified. The user can drag a point along the line at the top of the screen and it's position will represent an integer n from 2 to 11. The angle below is divided into two n equal parts. In other words, if $n = 2$, the angle is bisected; if $n = 3$, it is trisected, and so on. Figure 6.9 shows the diagram when the slider is in the region corresponding to $n = 7$. The angle $\angle ABC$ is divided into seven equal parts.

The code to do this was basically all typed in by hand. For the discussion below, it is broken into various chunks, but in the **Geometer** file it all appears together.

```
.geometry "version 0.31";
v1 = .v.ff(-1.000000, 0.800000, .in);
v2 = .v.ff(1.000000, 0.800000, .in);
l1 = .l.vv(v1, v2, .longline);
vp2 = .pinned(-0.9, 0.8, .nomark, "2");
vp3 = .pinned(-0.7, 0.8, .nomark, "3");
vp4 = .pinned(-0.5, 0.8, .nomark, "4");
vp5 = .pinned(-0.3, 0.8, .nomark, "5");
vp6 = .pinned(-0.1, 0.8, .nomark, "6");
vp7 = .pinned(0.1, 0.8, .nomark, "7");
vp8 = .pinned(0.3, 0.8, .nomark, "8");
vp9 = .pinned(0.5, 0.8, .nomark, "9");
vp10 = .pinned(0.7, 0.8, .nomark, "10");
vp11 = .pinned(0.9, 0.8, .nomark, "11");
vm1 = .pinned(-0.8, 0.8, .plus);
vm2 = .pinned(-0.6, 0.8, .plus);
vm3 = .pinned(-0.4, 0.8, .plus);
vm4 = .pinned(-0.2, 0.8, .plus);
vm5 = .pinned(0, 0.8, .plus);
vm6 = .pinned(0.2, 0.8, .plus);
vm7 = .pinned(0.4, 0.8, .plus);
vm8 = .pinned(0.6, 0.8, .plus);
```

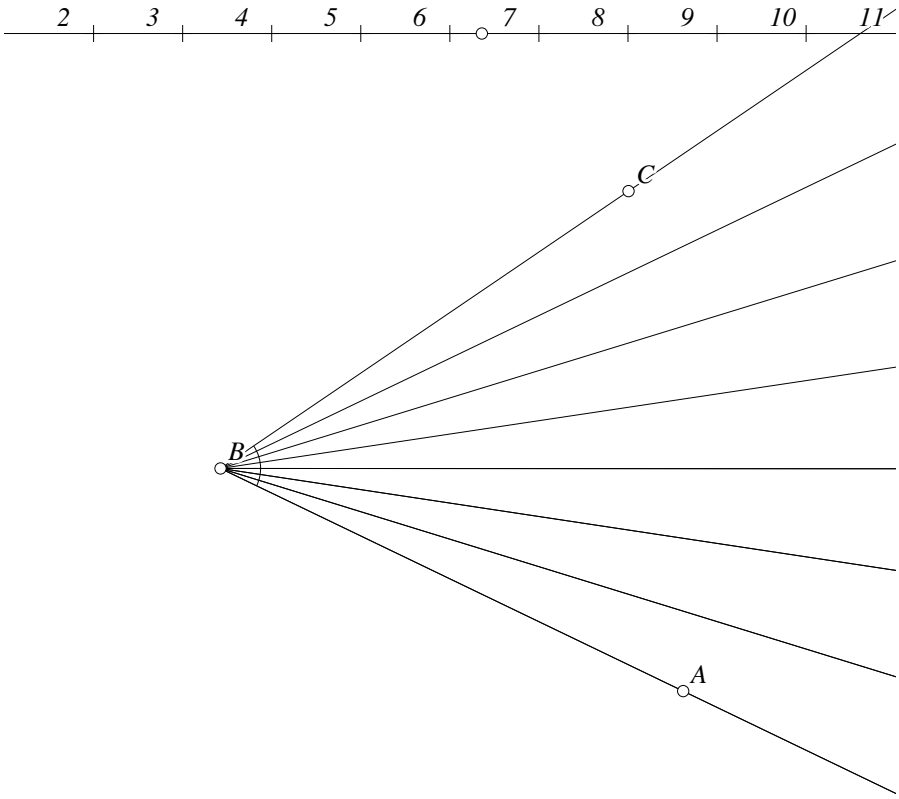

Figure 6.9: Subdivision of an Angle
`Teachers/Artificial.T` [P]

```
vm9 = .pinned(0.8, 0.8, .plus);
tab = .vonl(l1, -0.140719, 0.8, .cyan);
```

All this first chunk of code does is to draw the rule at the top of the viewing area. The coordinates for a square viewing area run from -1.0 to 1.0 in both directions, so the rule runs all the way across the drawing area with a y-coordinate of 0.8. The `vm1`, `vm2`, ... points are marked with crosses so that they divide the rule into 11 roughly equally-sized pieces and the `vp2`, `vp3`, ... points show no mark, but they appear between the other points and label the regions on the rule. Finally, the point called `tab` is stuck on the rule and can slide back and forth on it.

Note that all the points are pinned. This is so you can't inadvertently move them, but so that they will show up in the drawing.

```
value = .f.vxcoord(tab);
```

```
count = .f.rpn(value, 1.000000, .add, 0.200000, .div,
        2.000000, .add, .truncate);
```

The two lines above determine the value of n (which is called `count` in the code here). The first line takes the x-coordinate of the point `tab` that can slide along the line and stores it in a floating point number called `value`.

The next line converts `value` to a number between 2 and 11. We know that originally `value` is between -1.0 and 1.0 so if we add 1.0 to it and divide that result by 0.2 we will obtain a number between 0 and 10 (and it will, in fact be less than 10—something like 9.999 is the maximum value it can have).

Add 2 to that and truncate to the nearest integer and we've got a number between (and including) 2 and 11.

The `.f.rpn` line above does exactly the calculation described above. It puts `value` on the stack, then it puts 1.0 on the stack and adds the two, leaving the single result `value`$+1$ on the stack.

The next two entries, `0.200000` and `.div`, divide the number on the stack by 0.2. The next two entries add 2 to the result, and the final `.truncate` command rounds down to the nearest integer, so the result stored in `count` will be an integer between 2 and 11.

```
v3 = .free(0.55988, -0.571856, "A");
v4 = .free(-0.51497, -0.176647, "B");
v5 = .free(0.526946, 0.314371, "C");
l2 = .l.vv(v4, v3, .ray12);
l3 = .l.vv(v4, v5, .ray12);
ang = .a.vvv(v3, v4, v5);
```

The lines above were the only ones in this example that were entered using the GUI of **Geometer**. They draw the angle that is to be subdivided in the middle of the screen.

```
m2 = .f.rpn(ang, count, .div);
```

This line takes the measure of angle `ang`, divides it by `count`, and stores the result in the variable `m2`. The main angle has to be broken into a bunch of angles all having equal measures `m2`.

The problem, of course, is that depending on the size of `count` a different number of those angles need to be drawn. **Geometer** is not very good at conditional code, so what can be done?

```
x1 = .f.rpn(m2, 1.000000, count, .mod, .mul);
x2 = .f.rpn(m2, 2.000000, count, .mod, .mul);
x3 = .f.rpn(m2, 3.000000, count, .mod, .mul);
x4 = .f.rpn(m2, 4.000000, count, .mod, .mul);
x5 = .f.rpn(m2, 5.000000, count, .mod, .mul);
x6 = .f.rpn(m2, 6.000000, count, .mod, .mul);
x7 = .f.rpn(m2, 7.000000, count, .mod, .mul);
x8 = .f.rpn(m2, 8.000000, count, .mod, .mul);
x9 = .f.rpn(m2, 9.000000, count, .mod, .mul);
x10 = .f.rpn(m2, 10.000000, count, .mod, .mul);
```

OK, here's the dirty trick—we're going to draw 11 dividing lines, no matter what the angle is. It's just that for small values of `count`, lots of them will be drawn on top of each other.

Basically, to find the angle size, we take `m2` and multiply it by ten values: 1 (mod `count`), 2 (mod `count`), ..., 10 (mod `count`). Consider the situation where `count`$= 4$ to see what's going on:

$$\begin{aligned} 0 &= 4(\mathrm{mod}4) = 8(\mathrm{mod}4) \\ 1 &= 1(\mathrm{mod}4) = 5(\mathrm{mod}4) = 9(\mathrm{mod}4) \\ 2 &= 2(\mathrm{mod}4) = 6(\mathrm{mod}4) = 10(\mathrm{mod}4) \\ 3 &= 3(\mathrm{mod}4) = 7(\mathrm{mod}4), \end{aligned}$$

so ten lines are drawn, but two of them are drawn 3 times and two of them are drawn twice.

But now we have the angles, so all that remains is to draw them. The following straightforward code does the trick. It could probably be made shorter with a macro, but it was pretty simple just to get one set of three lines working correctly, and then to make nine more copies which were modified in the obvious way:

```
ang1 = .a.f(x1);
vsplit1 = .v.avv(ang1, v3, v4, .in);
lsplit = .l.vv(v4, vsplit1, .ray12);
ang2 = .a.f(x2);
vsplit2 = .v.avv(ang2, v3, v4, .in);
lsplit2 = .l.vv(v4, vsplit2, .ray12);
ang3 = .a.f(x3);
vsplit3 = .v.avv(ang3, v3, v4, .in);
lsplit3 = .l.vv(v4, vsplit3, .ray12);
ang4 = .a.f(x4);
vsplit4 = .v.avv(ang4, v3, v4, .in);
lsplit4 = .l.vv(v4, vsplit4, .ray12);
ang5 = .a.f(x5);
vsplit5 = .v.avv(ang5, v3, v4, .in);
lsplit5 = .l.vv(v4, vsplit5, .ray12);
ang6 = .a.f(x6);
vsplit6 = .v.avv(ang6, v3, v4, .in);
lsplit6 = .l.vv(v4, vsplit6, .ray12);
ang7 = .a.f(x7);
vsplit7 = .v.avv(ang7, v3, v4, .in);
lsplit7 = .l.vv(v4, vsplit7, .ray12);
ang8 = .a.f(x8);
vsplit8 = .v.avv(ang8, v3, v4, .in);
lsplit8 = .l.vv(v4, vsplit8, .ray12);
ang9 = .a.f(x9);
vsplit9 = .v.avv(ang9, v3, v4, .in);
lsplit9 = .l.vv(v4, vsplit9, .ray12);
ang10 = .a.f(x10);
vsplit10 = .v.avv(ang10, v3, v4, .in);
lsplit10 = .l.vv(v4, vsplit10, .ray12);
```

The 30 lines above can be replaced by the following 16 lines if you're willing to use a macro. Ten more of the earlier lines can also be moved into the macro if you wish—the lines where `x1`, ..., `x10` were defined.

```
.macro newangle(.flt f)
{
    ang1 = .a.f(f);
    vsplit1 = .v.avv(ang1, v3, v4, .in);
    lsplit1 = .l.vv(v4, vsplit1, .ray12);
}
newangle(x1);
newangle(x2);
newangle(x3);
newangle(x4);
newangle(x5);
newangle(x6);
newangle(x7);
newangle(x8);
newangle(x9);
newangle(x10);
```

6.11 Morley's Theorem

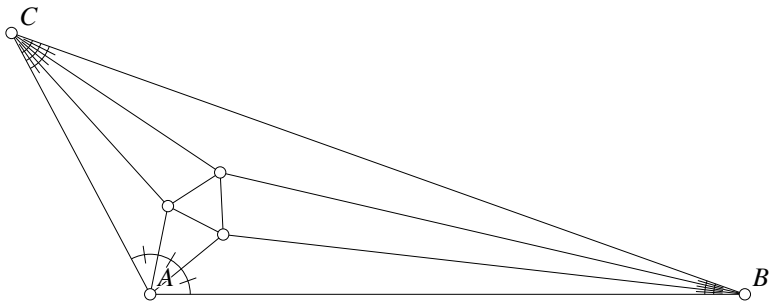

Figure 6.10: Morley's Theorem
`Teachers/Morley.D` [D]

Morley's Theorem (see Figure 6.10) states that if you examine the intersections of the angle trisectors of any triangle, they will meet in pairs to form an equilateral triangle.

Here's the actual **Geometer** code to draw the basic diagram for Morley's Theorem. The key parts are the three sets of five lines beginning with the `.a.vvv` commands. This gets the angle from the three points. That angle is then divided by 3 (well, multiplied by $1/3$, and the resulting number is converted back to an angle. That new angle is then used to construct two more points on the trisectors. There is some funny stuff to make the picture pretty—a lot of the lines are invisible because they were used in the construction, but in the nice illustration only parts of them are shown. Much of the diagram below can be constructed by pointing and clicking, but you will need to type in the three sections of code that generate the sets of trisectors.

```
.geometry "version 0.2";
v1 = .free(-0.874251, -0.760479, "A");
v2 = .free(0.0658683, 0.766467, "C");
```

```
v3 = .free(0.811377, -0.601796, "B");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
a = .a.vvv(v1, v2, v3, .noangle);
ll2 = .f.rpn(a, 0.333333, .mul);
a3 = .a.f(ll2);
vv1 = .v.avv(a3, v1, v2, .in);
ww1 = .v.avv(a3, vv1, v2, .in);
l4 = .l.vv(v2, ww1, .in, .ray12);
l5 = .l.vv(v2, vv1, .in, .ray12);
b = .a.vvv(v2, v3, v1, .noangle);
ll4 = .f.rpn(b, 0.333333, .mul);
b3 = .a.f(ll4);
vv2 = .v.avv(b3, v2, v3, .in);
ww2 = .v.avv(b3, vv2, v3, .in);
l6 = .l.vv(v3, ww2, .in, .ray12);
l7 = .l.vv(v3, vv2, .in, .ray12);
c = .a.vvv(v3, v1, v2, .noangle);
ll6 = .f.rpn(c, 0.333333, .mul);
c3 = .a.f(ll6);
vv3 = .v.avv(c3, v3, v1, .in);
ww3 = .v.avv(c3, vv3, v1, .in);
l8 = .l.vv(v1, ww3, .in, .ray12);
l9 = .l.vv(v1, vv3, .in, .ray12);
v4 = .v.ll(l8, l5, "Y");
v5 = .v.ll(l6, l9, "Z");
v6 = .v.ll(l7, l4, "X");
l10 = .l.vv(v1, v5, .red);
l11 = .l.vv(v1, v4, .red);
l12 = .l.vv(v4, v2, .red);
l13 = .l.vv(v2, v6, .red);
l14 = .l.vv(v6, v3, .red);
l15 = .l.vv(v3, v5, .red);
l16 = .l.vv(v5, v6, .yellow);
l17 = .l.vv(v6, v4, .yellow);
l18 = .l.vv(v4, v5, .yellow);
```

6.12 Drawing the Steiner Porism

How can Figure 6.11 be constructed using a computer geometry program?

Obviously, a set of equally-spaced circles that exactly fill the ring between two concentric circles was constructed, and the result was inverted to obtain a Steiner Porism with a lopsided pair of enclosing rings. The inverted circles will exactly fill the space between the lopsided rings.

It is not hard to work out the relative sizes of a pair of concentric circles that will allow for some fixed number n of circles to fit between them. In Figure 6.12 we see what we need to begin. In this case, there are nine circles, but let's just call that number n. The central angle between any pair of circle centers is $360°/n$, so vertex angle $\angle AOB = 360°$ of the isosceles triangle in the figure.

If that figure, $\triangle OMA$ is a right triangle where M is the point of tangency of two

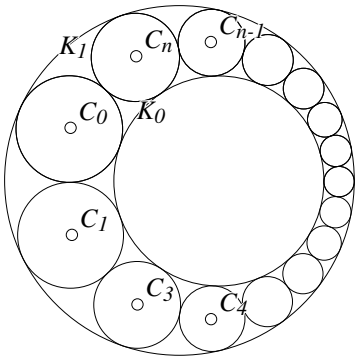

Figure 6.11: The Steiner Porism
`Teachers/Steiner.T` [S]

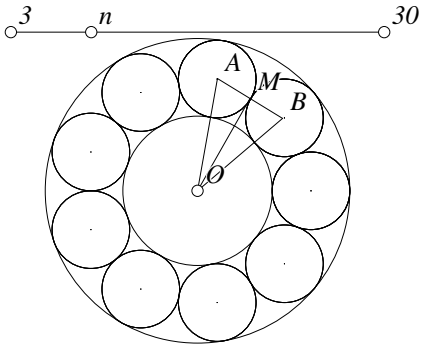

Figure 6.12: A Ball-Bearing Race
`Teachers/Steiner1.T` [M]

adjacent circles. Suppose the radius of the inner circle is R_1 and the radius of the small surrounding circles is R_2 (which will make the radius of the larger circle $R_1 + 2R_2$).

Then

$$\sin\left(\frac{180^\circ}{n}\right) = \frac{R_2}{(R_1 + R_2)},$$

and we can solve for R_2:

$$R_2 = \frac{R_1 \sin(\frac{180^\circ}{n})}{(1 - \sin(\frac{180^\circ}{n}))}.$$

The **Geometer** code below draws the figure (and a lot more besides). It includes a sort of slider at the top to change the number of surrounding circles to be anything from 3 to 30. The point labeled n can slide between the pinned `v1` and `v2`. The ratio is then converted using the `.f.rpn` commands to a number `n` between 3 and 30. Then `r2` is

calculated using the formula we obtained in the previous paragraph. (`r1` is arbitrarily set to be $.4$.)

Next, a macro is defined to draw circle number `i`. It finds the sine and cosine of the angle $180°(i/n)$ and multiplies them by the offset from the center, $R_1 + R_2$. These are the coordinates for the center of the i^{th} circle, and a circle of radius `r2` is drawn around that center.

Then 30 circles are drawn. The fact that fewer than 30 are needed doesn't matter; if `i` is too big, the circle corresponding to `i` will be drawn exactly on top of a previous one.

Note that since all the calculations are done in absolute coordinates, the circles will all be centered at the origin of the drawing, in the exact middle of the drawing area. This could be done differently, if desired. The final few lines of code draw the circles that inscribe and circumscribe that ring of circles. These, of course, also need to be calculated using the `.f.rpn` commands.

Finally, the listing is condensed from what **Geometer** would really put into its file—it would put each of the macro calls to `circ` on a separate line. By condensing them, you're saved looking at a page of almost identical commands.

This code, of course, only draws the circles between a pair of concentric circles. Additional code is needed to draw the Steiner Porism, since each of these circles should be inverted (which can be done most conveniently inside the macro). Then all the circles but the inverted versions should be painted the invisible color so all that appears in the **Geometer** diagram is the porism.

```
.geometry "version 0.2";
// Listing condensed by hand --
// all the calls to the circ macro were on different lines.
r1 = .f.rpn(0.400000);
v1 = .pinned(-1, 0.9, "3");
v2 = .pinned(1, 0.9, "30");
l1 = .l.vv(v1, v2);
v3 = .vonl(l1, -0.569395, 0.9, "n");
rat = .f.vvvratio(v1, v3, v2);
n = .f.rpn(rat, 27.500000, .mul, 3.500000, .add,
        .truncate);
a = .f.rpn(180.000000, n, .div);
r2 = .f.rpn(a, .sin, .dup, 1.000000, .exch,
        .sub, .exch, r1, .mul, .exch,
        .div);
.macro circ(.flt i)
{
    x = .f.rpn(a, 2.000000, .mul, i, .mul,
        .cos, r1, r2, .add, .mul);
    y = .f.rpn(a, 2.000000, .mul, i, .mul,
        .sin, r1, r2, .add, .mul);
    v = .v.ff(x, y, .dot);
    c = .c.vf(v, r2);
}
circ(0.000000); circ(1.000000); circ(2.000000);
circ(3.000000); circ(4.000000); circ(5.000000);
circ(6.000000); circ(7.000000); circ(8.000000);
circ(9.000000); circ(10.000000); circ(11.000000);
```

```
circ(12.000000); circ(13.000000); circ(14.000000);
circ(15.000000); circ(16.000000); circ(17.000000);
circ(18.000000); circ(19.000000); circ(20.000000);
circ(21.000000); circ(22.000000); circ(23.000000);
circ(24.000000); circ(25.000000); circ(26.000000);
circ(27.000000); circ(28.000000); circ(29.000000);
orig = .pinned(0, 0);
cin = .c.vf(orig, r1);
rout = .f.rpn(r1, r2, 2.000000, .mul, .add);
cout = .c.vf(orig, rout);
```

6.13 Apollonius' Problem

Apollonius' problem is to find three circles tangent to a given circle. This is done with a series of inversions, and is a bit tricky to illustrate with a **Geometer** diagram.

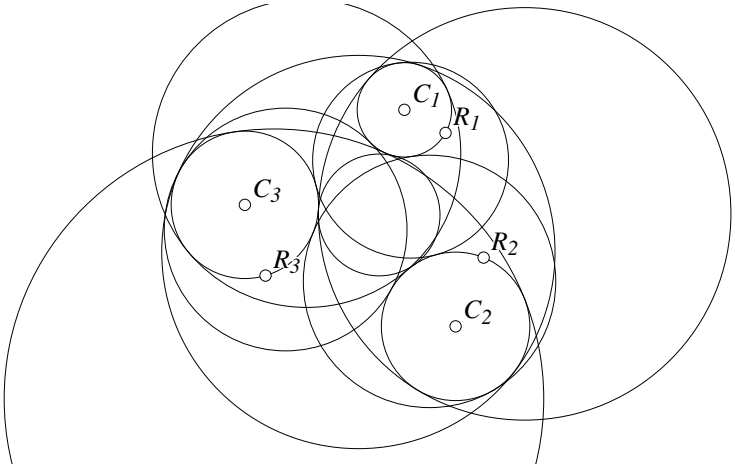

Figure 6.13: The Problem of Apollonius
`Teachers/Apollonius.T` [M]

There are up to eight possible solutions, and in Figure 6.13 an example is shown where all eight mutually tangent circles are shown.

The key idea is this: If we choose the circle of smallest radius and shrink it to a point, and at the same time either add or subtract the radius of the smallest circle to or from the radii of the larger circles, then solving Apollonius' problem for a point and two circles will yield a circle whose radius can be increased or decreased by the radius of the smallest circle to yield a solution[†].

Figure 6.14 demonstrates the general idea. The original circles for which the problem is to be solved are centered at C_1, C_2, and C_3, and they have radii R_1, R_2, and R_3,

[†]There are direct solutions as well that involve moving the line or lines parallel to themselves by a distance equal to the diameter of the smallest circle, just as we expand and shrink the diameters of the larger circles in the three-circle solution.

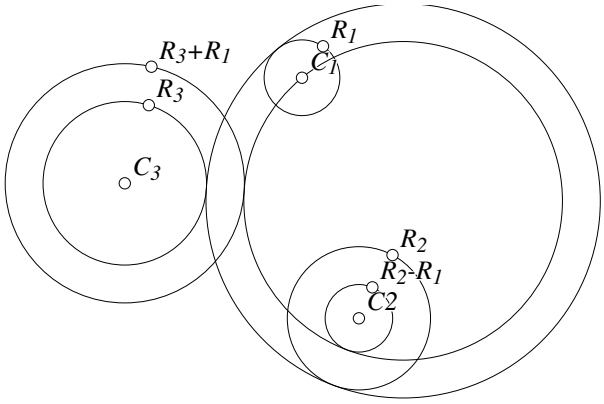

Figure 6.14: Addition and Subtraction of Radii
`Teachers/Apollonius1.T` [M]

respectively. Assume that R_1 is the smallest of the three radii. One solution can be obtained by drawing a circle centered at C_2 of radius $R_2 - R_1$ and by drawing a circle about C_3 of radius $R_3 + R_1$. Using techniques we learned earlier in the chapter, we can find a circle that is tangent to those new circles with the modified radii and passing through C_1 as shown in the figure. If that circle's radius is then increased by R_1, we have one of the eight possible solutions to the problem of finding circles mutually tangent to the three given circles.

Beware: This is trickier than it seems. Remember that there are up to four solutions to the "two circles and a point" problem, possibly having tangencies on both sides of both circles. Only one of these four circles can have its radius increased and still be tangent to the original circles. In fact, if you play with the **Geometer** diagram that generates Figure 6.13, you'll find that it works only for a limited range of values around the initial configuration—increase or decrease the radii too much and you'll find that the solutions jump to the other sides of circles, and on expansion or contraction of those circles, they no longer solve the problem.

It took a lot of work to produce this **Geometer** diagram—the editor was used repeatedly. Clearly, it could have been done with standard construction techniques, but the solution would have been hundreds of lines long. What follows is the complete code for that diagram, but broken into chunks with some commentary following each chunk.

It is pretty clear from the names which lines were done using the mouse and which were drawn with the editor. The mouse interface always makes up the same sorts of names, like `v1`, `v2`, et cetera, for points, and `c1`, `c2`, ..., for circles. Names like `r2sub` were typed in the editor. Also, there are a huge number of items drawn in the "invisible" color (`.in`). Typically, they were drawn in a color-coded way (internal tangents one color, external in another, for example), and when they were no longer needed for the construction, they were "erased" by turning them to an invisible color.

```
.geometry "version 0.2";
```

```
v1 = .free(0.169341, 0.542551, "C\sub{1}");
v2 = .free(0.348837, 0.44186, "R\sub{1}");
v3 = .free(0.392748, -0.401592, "C\sub{2}");
v4 = .free(0.51497, -0.101796, "R\sub{2}");
v5 = .free(-0.526167, 0.12827, "C\sub{3}");
v6 = .free(-0.436047, -0.180233, "R\sub{3}");
c1 = .c.vv(v5, v6);
c2 = .c.vv(v1, v2);
c3 = .c.vv(v3, v4);
r1 = .f.vv(v1, v2);
r2 = .f.vv(v3, v4);
r3 = .f.vv(v5, v6);
r2sub = .f.rpn(r2, r1, .sub);
r3sub = .f.rpn(r3, r1, .sub);
r2add = .f.rpn(r1, r2, .add);
r3add = .f.rpn(r3, r1, .add);
c3sub = .c.vf(v5, r3sub, .in);
c2sub = .c.vf(v3, r2sub, .in);
c3add = .c.vf(v5, r3add, .in);
c2add = .c.vf(v3, r2add, .in);
```

The code above draws the three initial circles, and it assumes that the radius `r1` is the smallest of the three radii. `r1` is added and subtracted from each of the other two radii and four circles are constructed centered at the same places as the other two circles, but with radii increased or decreased the appropriate amount.

```
v7 = .free(0.365897, 1.00522, .in, "7");
c4 = .c.vv(v1, v7, .in);
c5 = .c.ccinv(c2add, c4, .in);
c6 = .c.ccinv(c3add, c4, .in);
l2 = .l.ccext(c6, c5, 2, .in, .longline);
l3 = .l.ccext(c6, c5, 1, .in, .longline);
c7 = .c.ccinv(c3sub, c4, .in);
c8 = .c.ccinv(c2sub, c4, .in);
l21 = .l.ccext(c7, c8, 2, .in, .longline);
l31 = .l.ccext(c7, c8, 1, .in, .longline);
l1 = .l.ccint(c7, c5, 2, .in);
l4 = .l.ccint(c7, c5, 1, .in);
l5 = .l.ccint(c8, c6, 1, .in);
l6 = .l.ccint(c6, c8, 2, .in);
```

An arbitrary circle of inversion `c4` is drawn, and all four of the circles with modified radii are inverted in it. The common external and internal tangents to various pairs of those four circles are also drawn (after inversion, the center of the smallest circle went to infinity, so inverted solution to Apollonius' problem for two circles and a point will be these four lines tangent to the inverses of the circles with modified radii).

```
c10 = .c.lcinv(l2, c4, .in);
c11 = .c.lcinv(l3, c4, .in);
c14 = .c.lcinv(l21, c4, .in);
c15 = .c.lcinv(l31, c4, .in);
c9 = .c.lcinv(l1, c4, .in);
c12 = .c.lcinv(l6, c4, .in);
c13 = .c.lcinv(l4, c4, .in);
c16 = .c.lcinv(l5, c4, .in);
```

The tangent lines are inverted to find solution circles going through the center of C_1 and tangent to the circles with modified radii.

```
v8 = .v.ccenter(c9, .in, "I");
v9 = .v.ccenter(c12, .in, "J");
v10 = .v.ccenter(c13, .in, "K");
v11 = .v.ccenter(c16, .in, "L");
v12 = .vonc(c9, -0.474808, -0.263661, .in, "M");
v13 = .vonc(c12, 0.0527364, -0.175382, .in, "N");
v14 = .vonc(c13, -0.103302, -0.0870917, .in, "O");
v15 = .vonc(c16, 0.0408249, -0.602817, .in, "P");
rad1 = .f.vv(v8, v12);
rad11 = .f.rpn(rad1, r1, .sub);
cf1 = .c.vf(v8, rad11);
rad2 = .f.vv(v9, v13);
rad22 = .f.rpn(rad2, r1, .add);
cf2 = .c.vf(v9, rad22);
rad3 = .f.vv(v10, v14);
rad33 = .f.rpn(rad3, r1, .add);
cf3 = .c.vf(v10, rad33);
rad4 = .f.vv(v11, v15);
rad44 = .f.rpn(rad4, r1, .sub);
cf4 = .c.vf(v11, rad44);
```

The centers and points on the radii of four of the circles are found. From these, the radii can be determined, and `r1` can be added or subtracted as appropriate, and the new final circles can be found.

```
v16 = .v.ccenter(c11, .in, "Q");
v17 = .v.ccenter(c15, .in, "R");
v18 = .v.ccenter(c14, .in, "S");
v19 = .v.ccenter(c10, .in, "T");
v20 = .vonc(c11, -0.00771487, 0.608311, .in, "U");
v21 = .vonc(c15, -0.347598, 0.492195, .in, "V");
v22 = .vonc(c14, -0.314278, 0.371898, .in, "W");
v23 = .vonc(c10, 0.261844, 0.534324, .in, "X");
rad5 = .f.vv(v16, v20);
rad55 = .f.rpn(rad5, r1, .sub);
cf5 = .c.vf(v16, rad55);
rad6 = .f.vv(v17, v21);
rad66 = .f.rpn(rad6, r1, .add);
cf6 = .c.vf(v17, rad66);
rad7 = .f.vv(v18, v22);
rad77 = .f.rpn(rad7, r1, .sub);
cf7 = .c.vf(v18, rad77);
rad8 = .f.vv(v19, v23);
rad88 = .f.rpn(rad8, r1, .add);
cf8 = .c.vf(v19, rad88);
```

The operation above is repeated on the final set of four circles.

6.14 ♦♦Apollonius' Point

Apollonius' Point of a triangle is defined to be the common intersection of the three lines connecting the points of a triangle with the points of tangency of the three excircles with their circumscribing circle. See Figure 6.15. Construct a **Geometer** diagram that shows this point for an arbitrary triangle $\triangle ABC$.

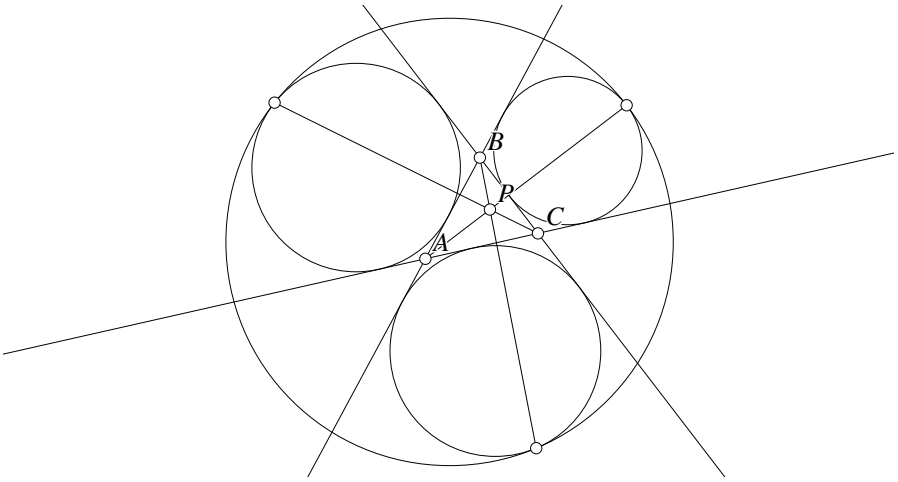

Figure 6.15: The Apollonius Point
`Teachers/ApolloniusPt.T` [M]

If we use the solution to Apollonius' problem in the previous section to obtain the circumscribing circle, we are faced with the problem that the construction there required the knowledge of which of the circles was the smallest. Not only that, but if there are 8 possible circles tangent to the three excircles, which one is the one that circumscribes them?

Feuerbach's Theorem comes to our aid, however. Feuerbach's Theorem states that the nine-point circle of a triangle is tangent to the three excircles of the triangle (and to the incircle as well, but that doesn't matter to us here).

If we can find an inversion that takes the excircles into themselves, the nine-point circle will be inverted to be the required tangent circle. Circles are inverted to themselves by any circle that is orthogonal to them. The circle orthogonal to all three excircles has its center at the radical center of the three circles, and we can find its radius by drawing a tangent to any of the circles from the radical center and using that point of tangency as a point on the diameter of the required circle.

So how do we find the radical center? It is the intersection of any pair or radical axes of pairs of the circles. The radical axis is easy to construct if the two circles intersect, but in this case, we know that none of the pairs do. So the usual trick is to find a circle that passes through both circles, to find the radical center of those three (the two non-intersecting circles and the one that intersects them both), and that point will

lie on the radical axis. Then choose another circle that intersects both and find the radical center for those three. That will be another point on the radical axis of the two non-intersecting circles. With two points on the radical axis, we can construct it.

There are a couple of strategies for finding a circle that intersects pairs of circles. Perhaps the easiest would be to find a circle that passes through their centers and through any other point, but then we'd need two pairs of such circles. This will work fine, but the solution here is to find two circles that intersect all three of the excircles so they can be used to find both of the radical axes. One circle that's guaranteed to work is the one that passes through the three circle centers. Another is obtained by taking the nine-point circle and making it a tiny bit larger so that it intersects all three. Remember that it is tangent to the three, so making it a tiny bit larger will cause it to intersect the three. In the construction here, it is 10% larger.

Once we have the circle orthogonal to all three circles, invert the nine-point circle through it, and it is the required outer tangent circle.

But now we need to find those points of tangency, and using the circle-circle intersection method is risky—due to numerical round-off, the circles might miss by a millionth of an inch and there will be no intersection. The easiest thing to do is to connect the centers of the circles with lines (which will pass through both circles perpendicularly), and find the intersections of those lines with the circles.

Here is the code that does the construction interleaved with a discussion of how it works. The vast majority was constructed with the **Geometer** GUI, but obviously a text editor was used from time to time.

It may be easier to follow this with the **Geometer** diagram displayed on the screen. If the meaning of any of the invisible points doesn't make sense, open the diagram with the text editor, change the invisible point to some obvious color, and redisplay.

```
.geometry "version 0.40";
v1 = .free(-0.158683, 0.0718563, "A");
v2 = .free(0.00299401, 0.374251, "B");
v3 = .free(0.149701, 0.122754, "C");
l1 = .l.vv(v1, v2, .longline);
l2 = .l.vv(v2, v3, .longline);
l3 = .l.vv(v3, v1, .longline);
c2 = .c.lll(l1, l2, l3, 2);
c3 = .c.lll(l2, l1, l3, 2);
c4 = .c.lll(l1, l3, l2, 2);
v27 = .v.ccenter(c3, .in);
v28 = .v.ccenter(c2, .in);
v29 = .v.ccenter(c4, .in);
```

The code above makes the triangle, draws the three excircles, and finds their centers. The centers are needed to make one of the circles that passes through all three of the excircles.

```
v4 = .v.vvmid(v1, v2, .in);
v5 = .v.vvmid(v2, v3, .in);
v6 = .v.vvmid(v3, v1, .in);
```

```
c5 = .c.vvv(v4, v5, v6, .in);
v9 = .v.ccenter(c5, .in);
r = .f.vv(v9, v6);
r1 = .f.rpn(r, 1.100000, .mul);
c6 = .c.vf(v9, r1, .in);
c7 = .c.vvv(v27, v28, v29, .in);
```

Circle `c5` is the nine-point circle (we know that the nine-point circle passes through the three midpoints of the sides). Then `r` is the radius of the nine-point circle, and we multiply it by 1.1 to get a new radius for a slightly larger circle centered at the same point (`v9`, the center of the nine-point circle), but guaranteed to intersect all three. `c6` is that larger circle, and `c7` is the circle passing through the centers of all three excircles.

```
v7 = .v.cc(c7, c2, 2, .in);
v8 = .v.cc(c7, c2, 1, .in);
v10 = .v.cc(c3, c7, 2, .in);
v11 = .v.cc(c3, c7, 1, .in);
v12 = .v.cc(c2, c6, 1, .in);
v13 = .v.cc(c2, c6, 2, .in);
v14 = .v.cc(c3, c6, 2, .in);
v15 = .v.cc(c3, c6, 1, .in);
```

Here are the eight intersections of the circles passing through all three excircles with the excircles themselves.

```
l4 = .l.vv(v11, v10, .in, .longline);
l5 = .l.vv(v7, v8, .in, .longline);
v16 = .v.ll(l4, l5, .in);
l6 = .l.vv(v14, v15, .in, .longline);
l7 = .l.vv(v13, v12, .in, .longline);
v17 = .v.ll(l6, l7, .in);
l8 = .l.vv(v16, v17, .in, .longline);
v18 = .v.cc(c4, c7, 2, .in);
v19 = .v.cc(c4, c7, 1, .in);
v20 = .v.cc(c4, c6, 1, .in);
v21 = .v.cc(c4, c6, 2, .in);
l9 = .l.vv(v21, v20, .in, .longline);
v22 = .v.ll(l9, l7, .in);
l10 = .l.vv(v18, v19, .in, .longline);
v23 = .v.ll(l5, l10, .in);
l11 = .l.vv(v22, v23, .in, .longline);
v24 = .v.ll(l11, l8, .in);
```

This is the construction of the radical center. Two pairs of radical axes are made for each of two pairs of excircles, their intersections are found to get two points on the radical axis of each pair of excircles, and then the radical axes are intersected at `v24` which is the radical center of the three excircles.

```
l12 = .l.vc(v24, c2, 2, .in, .longline);
v25 = .v.lc(l12, c2, 2, .in);
c1 = .c.vv(v24, v25, .in);
c8 = .c.ccinv(c5, c1);
```

`v25` is the point of tangency of a line from the radical center to one of the excircles. `c1` is the circle of inversion, and `c8` is the inverted nine-point circle that is the exterior tangent of the three excircles.

```
v26 = .v.ccenter(c8, .in);
l13 = .l.vv(v26, v29, .in, .longline);
l14 = .l.vv(v26, v28, .in, .longline);
l15 = .l.vv(v26, v27, .in, .longline);
v30 = .v.lc(l15, c8, 2);
v31 = .v.lc(l14, c8, 2);
v32 = .v.lc(l13, c8, 2);
```

Finally, the centers of the excircles and the center of the surrounding circle are connected with lines that are intersected with the various circles to get the exterior points.

```
l16 = .l.vv(v2, v32);
l17 = .l.vv(v1, v31);
l18 = .l.vv(v3, v30);
v33 = .v.ll(l16, l17, "P");
.text("Apollonius Point:  If the points of tangency
of the three excircles and their circumscribed
circle are connected to the opposite vertices of
a triangle, those lines are concurrent at
Apollonius' Point.", .l0);
```

Those exterior points are connected to the vertices of the triangle, and the point of Apollonius is found. There's also a short chunk of text to describe the construction.

6.15 Making Animated GIFs from a Script

This example uses Adobe Photoshop and Adobe ImageReady in combination with **Geometer** to produce an animated GIF that illustrates Miquel's Theorem. This particular example was made with the Photoshop CS and ImageReady CS versions of those programs. Other programs exist that let you patch files together to make animated GIFs, so this is only one way to go.

After messing around for a while, the following script seems to illustrate the theorem nicely:

```
.geometry "version 0.62";
t = .script(0.000000, 360.000000, 10.000000);
v0 = .free(-0.658683, -0.110778, .in);
x0 = .f.vxcoord(v0);
y0 = .f.vycoord(v0);
x1 = .f.rpn(t, .sin, 0.200000, .mul, x0, .add);
y1 = .f.rpn(t, .cos, 0.400000, .mul, y0, .add);
v1 = .v.ff(x1, y1);
v2 = .free(0.763473, 0.682635);
v3 = .free(0.350299, -0.724551);
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
```

```
l3 = .l.vv(v3, v1);
v7 = .free(-0.999, 0.999, .dot);
v8 = .free(0.999, 0.999, .dot);
v9 = .free(0.999, -0.999, .dot);
v10 = .free(-0.999, -0.999, .dot);
v44 = .free(0.00299401, -0.0149701, .in);
x44 = .f.vxcoord(v44);
y44 = .f.vycoord(v44);
x4 = .f.rpn(t, .sin, 0.300000, .mul, x44, .add);
y4 = .f.rpn(t, .cos, 0.050000, .mul, y44, .add);
v4 = .v.ff(x4, y4, .in);
l4 = .l.vlperp(v4, l2, .in);
l5 = .l.vlperp(v4, l1, .in);
l6 = .l.vlperp(v4, l3, .in);
v5 = .v.ll(l1, l5);
v6 = .v.ll(l2, l4);
v11 = .v.ll(l3, l6);
c1 = .c.vvv(v5, v6, v2);
c2 = .c.vvv(v6, v11, v3);
c3 = .c.vvv(v11, v5, v1);
```

We would like the script to illustrate the theorem with a series of triangles that change in a cyclical manner so that the animation can run and run with no apparent discontinuities.

Miquel's theorem states that for an arbitrary triangle, if three points are selected on the edges and if circles are drawn through each vertex and the selected points on the adjacent lines, then those circles will be concurrent.

The **Geometer** diagram above moves the vertex `v1` in an ellipse in 36 steps. The `t` is a sort of angle going from 0 to 360 degrees.

In the first attempt at this example, the points were simply constrained to lie on the edges of the triangle, but since they need to be re-projected to the lines after each movement, there was no guarantee that they would end where they started, and they usually did not. To fix this problem, it seemed like a good idea to select a point and drop perpendiculars to the three sides to select the points, since then they would return to the starting positions. Unfortunately, this point happens to be the point of concurrency of the circles, and the animation was far less interesting with a fixed point of intersection. Thus the code was modified to make the projection point `v4` itself move in an ellipse and this produced a suitable diagram.

The four vertices at the corners were useful for alignment in Adobe Photoshop.

Finally, the diagram above was not tested in the form above; the step size for `t` was made much smaller so that the movement was slow enough to assure that for a given arrangement of the starting points that none of the interesting parts of the figure left the diagram. Finally, it was saved as a file called `miquel.T`. When the script above is executed (after pressing the *Script Print* command), 36 files named `miquel000.eps` through `miquel035.eps` were created.

Figure 6.16 shows four of the 36 files created, evenly spaced though the animation.

Next, a new Photoshop file with length and width 668 pixels at 72 pixels per inch was created. This is the same size as the PostScript files generated by **Geometer**. A Photo-

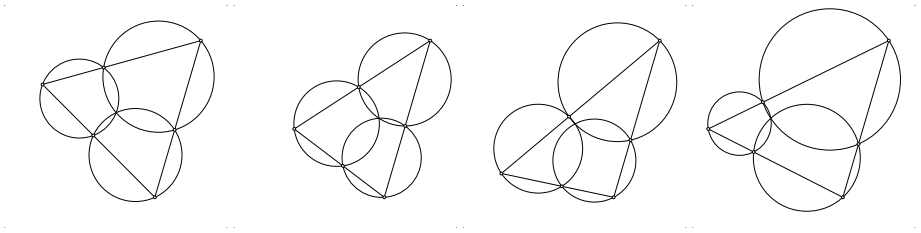

Figure 6.16: EPS Files Generated by a Script
`Teachers/miquel.T` [M]

shop action was created that opens a file, selects all, does a cut, goes to the other open file, does a paste, then closes the original file. Using the Photoshop Batch command, this action was applied to all 36 files generated by **Geometer** to create a Photoshop file with 37 layers: the original plus the 36 pasted images from the **Geometer** output.

The original layer was deleted and the file was saved as `miquel.psd` which is immediately opened in Adobe ImageReady. In the animation palette in ImageReady, the "Make frames from Layers" command was issued, and the resulting animation was saved as an animated GIF file called `miquel.gif`, which is included on the CD. Also included is a sample web page, `miquel.html` that illustrates how to include an animated GIF in html.

Chapter 7

Sleazy Hacks

The last time I wrote a book, the topic was computer graphics programming using the OpenGL graphics library. There are a lot of tricks that can produce amazing results using OpenGL (but are not obvious or straight-forward), and to give the readers some idea of what they are, I wanted to include a chapter entitled "Sleazy Hacks" that explained the tricks of the graphics wizard.

Unfortunately, the book was written under the auspices of Silicon Graphics, and a chapter named "Sleazy Hacks" didn't resonate with the corporate big-wigs, so the chapter title was changed to "Now That You Know". I have no idea what that means.

There aren't any corporate big-wigs associated with Geometer, so this chapter, since it serves the same purpose as the corresponding chapter in the OpenGL book, has the correct title.

Everything that Geometer can do is discussed in the tutorial chapters, but there are lots of non-obvious ways that the commands can be used to produce interesting, high-quality results.

Don't try to read this chapter until you've used Geometer a bit and have a feeling for what it can do, and what its deficiencies seem to be.

This chapter will provide some useful information, but probably your best source is to look at all the demonstrations that come with this book. Whenever you see an effect that you'd like to duplicate, all you have to do is to issue the *Edit Geometry* command, and all the tricks will be revealed. Just stare at the Geometer code until you understand what it does.

The information below is difficult to organize into a logical order, and for the most part, the topics are completely independent, so skipping around in this chapter is probably just as effective as reading it from beginning to end. On the other hand, the trick *you* need may be almost anywhere, so it wouldn't hurt to skim through the whole chapter to see what's there, and you can return for details later when you "need" the knowledge.

7.1 Drawing Tricks

7.1.1 Invisibility

You'll be surprised how often you need to use a few steps to get from where you are to where you want to be, and you don't want people viewing your efforts to see the intermediate steps. Remember that you can put all sorts of junk into your drawing to get exactly what you want, and then change the color of all the unneeded things to "invisible" afterwards. (Of course, your viewers can always click on the *Show Invis* command to see what you did, but at least you can do the same thing to see how I implemented the examples in this book.)

In fact, while you're building a proof or construction, you may find that it is very useful to work with the invisible items displayed—quite often the items that you've consigned to be invisible will be very useful for future constructions.

Remember also that the color layer commands can allow you to use invisibility selectively. For a complex construction, you may want to have some auxiliary line appear at step 7 (where it's really important), then to be visible for 2 steps, then to disappear for the rest because it isn't important. If the object in question happens to be the line segment connecting `v1` and `v2`, here's how to do it:

```
line = .l.vv{v1, v2, [7 .in, .blink, 2 .white, .in]);
```

The line will be invisible for 7 steps, then when it first appears it will blink, highlighting it's importance. For two more steps it will be visible in white, and then it will disappear for the remainder of the construction. (Remember that "step 7" is really the eighth step. Step 0 is the first, step 1 is the second, and so on.)

Make points that aren't free invisible if possible. Then the user won't try to manipulate them. Or you can pin them and draw them in the `.black` color.

7.1.2 Finding Things in the Editor

If you need to change a feature of a geometric object with the editor and your drawing is complicated, temporarily change to color of the object in question using the graphical user interface to some color that doesn't otherwise appear in your diagram. Then it'll be easy to find in the editor. Later, you can change the color back to the correct one.

7.2 Geometer Draws the Wrong Thing

Often you'll want to display a line segment, but the construction you use gives something different from what you want – the same line, but with different endpoints. With this constructed line, you can often pick the points you want on it, connect those to make the segment you want, and then make the original segment invisible.

For example, if you were trying to draw a square, and as part of your construction you used a line parallel to a given line, the line parallel doesn't have reasonable endpoints. But you can find where that line hits the other edges using the *LL=>P* command, connect them with another segment, and then paint the original long parallel line invisible.

A similar problem can occur with circles—if you draw the entire circle, the drawing gets too cluttered. You need to use circles if you need to find their intersections with lines or other circles, but the circles that are used for such things can be made invisible later. If you need to show just a segment of a circle, draw an arc on top of the invisible circle with a common center, but with endpoints that are appropriate to your diagram. Remember the command `.v.ccenter` that will find the center of a given circle if you don't happen to know the center. That center point can also be made invisible, if necessary.

There are other ways to make "invisible" points—paint them black, or draw them as `.nopoint`, for example, but those points can be selected and manipulated by the user, perhaps unintentionally. If you are preparing a drawing solely for use in publication, this can be the way to go. The points that are black or of type `.nopoint` are invisible in the drawing, but you can grab and move them with a mouse as you're adjusting the diagram before making the PostScript file that you need for something else.

7.2.1 The Wrong Segment

Suppose you're drawing a **Geometer** diagram to illustrate some theorem that states that three points, A, B, and C, lie on a line. Obviously you can draw a line through the three, but if the diagram is cluttered, you'd really prefer just to draw a segment connecting the outer two points, and the middle one will automatically be included. But many theorems, although they do guarantee that the three points will lie on a line, do not guarantee the order they will appear in.

The way around this is to draw three segments: AB, BC, and CA. Two of them will always be unnecessary, but depending on the order of the points, you can't know which two, and the extra lines will never clutter up the drawing—there is always a line there anyway.

The same idea could clearly be extended to more than three lines, but the number of small segments you need goes up.

7.2.2 The Wrong Tangent

Geometer has a few commands that have multiple possibilities, and all of them have to do with tangent lines. For example, the command that draws a tangent line from a point to a circle is not well-defined—if the point is outside the circle, there are two possibilities for this line.

What occurs internally is that **Geometer** solves the analytical equations for the line and circle simultaneously, and looks for the solution that gives a tangent line. To do this involves solving a quadratic equation, and if there are solutions, there are generally two

of them. (If the point is inside the circle, there are zero; if it is exactly on the circle, there is one, and if it is outside the circle, there are two.)

From your high-school algebra class, you may remember that to solve the quadratic equation:

$$ax^2 + bx + c = 0$$

you simply plug the numbers a, b, and c into the following equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

The "$\pm$" means that you can put in either a "$+$" or a "$-$" and each choice gives a different answer. In the case of a point-circle tangent, one choice gives one line and the other choice gives the other. Neither is a "preferred" answer.

What **Geometer** does is records which solution to use—the one with the $+$ or the one with the $-$, and depending on where you click on the circle, it chooses the tangent that hits the circle closest to where you clicked on it. The problem, of course, is that when you start moving around free points, the solution you want may suddenly switch to the other one, and the tangent line in your diagram will jump to the other one.

There is no general solution to this problem, but it seems to be the case that extremely often it is possible to include in your diagram both of the tangent lines. Then if the desired solution suddenly flips to the opposite one, the two tangent lines in your figure will flip to each other's positions, and the viewer will notice nothing.

Sometimes you can solve the problem easily with one of the commands `.v.lcvother` or `.v.ccvother`. These commands generate a point at the intersection of a line and circle or of a circle and a circle that are guaranteed to be different from the given point. If you need both intersections, there is no problem—click at both intersection points and you'll get the two different solutions. As you manipulate the figure they may flip, but they will still both be visible.

A problem can arise if the intersection was derived by other means, and thus there's no way to know which solution **Geometer** will use. As an example, there's a theorem that states that if a circle intersects the three sides of a triangle in two points each, and if the lines connecting the vertices of the triangle to one set of the points are coincident, then the lines connecting the vertices to the other set will also be coincident. One way to draw this would be to draw the three lines from the vertices through a common point, and then to find their intersections with the edges of the triangle. Draw a circle through those, and then find the other intersections of the circle with the sides. Unfortunately, the points are defined as intersections of pairs of lines, so there is no way to know which of the two solutions to the circle-line intersection problem is the "other" point. `.v.lcvother` or `.v.ccvother` will solve the problem nicely.

As a concrete example, suppose you are trying to draw a diagram illustrating Poncelet's Theorem (see Figure 7.1). Poncelet's Theorem has to do with a sequence of lines starting on an outer circle that are tangent to an inner one, and finally intersecting the outer circle at the next point. Sometimes, after some number of bounces like that (four bounces in the figure), the line comes back to where it started.

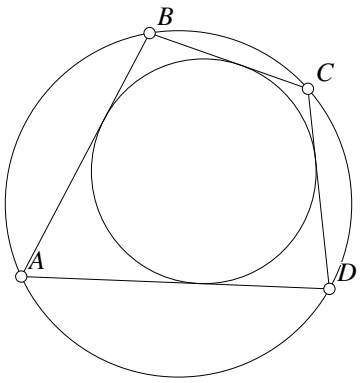

Figure 7.1: Poncelet's Theorem
`Sleaze/Poncelet.T` [M]

The "obvious" way to do this is to put point A on the circle, then draw a line tangent to the circle, extend that line to B, and do the same thing to make points C and D, finally coming back to point A. But each time you construct the tangent, there's a chance that **Geometer** will get the wrong one when you start moving point A around the circle.

A good way to draw this figure is to draw *both* tangents from A, from which you can find points B and D. From B and D, also draw both tangents, and so on. The final resulting **Geometer** code is shown below, and if you play with the diagram, you'll see that there is no visible flipping. The lines are crazily flipping on and off, but each time one flips to the other side, it's partner flips over to replace it. Note that the numbers 1, and 2 in the `.l.vc` commands indicates which solution **Geometer** is supposed to use, and note that for every point-circle pair, both a 1 and a 2 are used.

```
.geometry "version 0.2";
v1 = .free(0.0479042, -0.0778443, .in);
v2 = .free(-0.458084, 0.51497, .in);
v3 = .free(0.161677, 0.0688623, .in);
v4 = .free(0.0538922, 0.562874, .in);
c1 = .c.vv(v1, v2);
c2 = .c.vv(v3, v4);
v10 = .vonc(c1, -0.659588, -0.404813, .magenta, "A");
l9 = .l.vc(v10, c2, 2, .magenta);
l10 = .l.vc(v10, c2, 1, .magenta);
v11 = .v.lc(l9, c1, 2, .magenta, "B");
l11 = .l.vc(v11, c2, 1, .magenta);
l12 = .l.vc(v11, c2, 2, .magenta);
v12 = .v.lc(l12, c1, 2, .magenta, "C");
l13 = .l.vc(v12, c2, 1, .magenta);
l14 = .l.vc(v12, c2, 2, .magenta);
v13 = .v.lc(l14, c1, 2, .magenta, "D");
l15 = .l.vc(v13, c2, 1, .magenta);
l16 = .l.vc(v13, c2, 2, .magenta);
```

You can get the wrong tangent line in many circumstances, and in most of them a trick

like the one above can be used to eliminate a display problem. Of course if you're just trying to figure out how to solve a problem and not trying to produce a beautiful diagram to show a class to print in a book, you probably don't care if the thing blinks—it'll show you what you want.

Other commands that have multiple solutions include the internal and external tangents between two circles (two solutions each), the common tangent to three lines (four solutions), the intersection of a line and a circle (two solutions), the tangents from a point to a conic or the intersections of lines and conics.

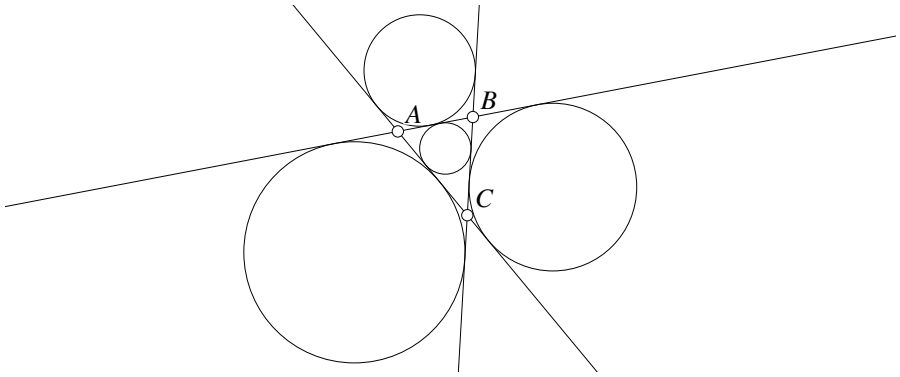

Figure 7.2: Four Tangent Circles
`Sleaze/Fourcircles.T` [M]

The only one that causes significant problems is the circle tangent to three lines where there are three possible solutions. See Figure 7.2. In the figure, one is called the incircle and the other three the excircles of $\triangle ABC$.

This time there are four solutions, and consequently **Geometer** puts a number between 1 and 4 in the `.c.lll` command. However, when you specify the command, if you click on the lines in a different order, solution 1 for one ordering of the lines may be the same as solution 3 for another ordering. **Geometer** blindly uses the lines in the specification in the order they appear. For that reason, the following code segment works perfectly, but to click it in with the mouse, you must be careful to click on the lines in the same order each time. In this case, the order was line AB, then line BC, and finally line CA. By clicking inside and outside $\triangle ABC$ as appropriate, the correct circles were selected. Note that in the code, the lines always appear in the same order: `l1` followed by `l2`, and finally `l3`.

```
.geometry "version 0.2";
v1 = .free(-0.288409, 0.313132, "A");
v2 = .free(0.122356, 0.391789, "B");
v3 = .free(0.0917666, -0.145702, "C");
l1 = .l.vv(v1, v2, .longline);
l2 = .l.vv(v2, v3, .longline);
l3 = .l.vv(v3, v1, .longline);
c1 = .c.lll(l1, l2, l3, 1);
c2 = .c.lll(l1, l2, l3, 3);
```

```
c3 = .c.lll(l1, l2, l3, 2);
c4 = .c.lll(l1, l2, l3, 4);
```

7.3 Geometer Deficiencies and Apparent Deficiencies

It is impossible to complete a program such as **Geometer**. Every time a new feature is added, it becomes obvious that three additional features are also needed. It is a fairly robust and useful program because it was used to draw all the illustrations in this book and it was also used as an experimental tool for the author to discover his own proofs of almost every theorem presented here.

Although no one would say **Geometer** is complete, it is also quite a bit more powerful than it may seem at first. In this section we'll talk about real deficiencies but also about features that seem to be missing, but are in fact present.

7.3.1 Using Angles

It is too bad that angles are treated slightly differently from simple floating point numbers (`flts`) but there are reasons for that. If you give an angle a name, you'd like to see that name drawn inside the angle opening, and if you give an `flt` a name, you'd generally just like to see the value printed on the display. Another minor problem is that angles can be expressed in **Geometer** either measured in degrees or radians. A `flt` is just a `flt`.

Luckily, it is easy to convert among the two forms. If you've got an angle that you need for a calculation, you can just use it as if it were a `flt`, and there's a simple command to turn `flts` back into angles. Here's an example. Suppose you're drawing a diagram with a trisected angle, and let `v1`, `v2`, and `v3` be the three points of the angle that's to be trisected. Here's some code that will do the job:

```
ang1 = .a.vvv(v1, v2, v3);
a3 = .f.rpn(ang1, 3.000000, .div);
ang3 = .a.f(a3);
v4 = .v.avv(ang3, v1, v2);
l3 = .l.vv(v4, v2);
```

The `.a.vvv` makes the `ang1` variable contain the angle determined by the three points, which we can then use directly in the `.f.rpn` command to do a calculation on it (in this case, divide it by 3 to trisect it). But the result of the `.f.rpn` command is a `flt`, so we use the `.a.f` command to turn it back to an angle. That angle (`ang3`) can then be used to construct other points and lines.

Another nasty problem is that angles are always directional. If you have an angle in a variable and you'd like to use it to construct another side, the measurement is always taken counter-clockwise relative to the central point. To make the problem concrete, suppose you'd like to have a diagram with three free points that determine an angle, and then you'd like to use two copies of that angle to be the base angles of an isosceles triangle. So as you move your three vertices, the base angles change appropriately.

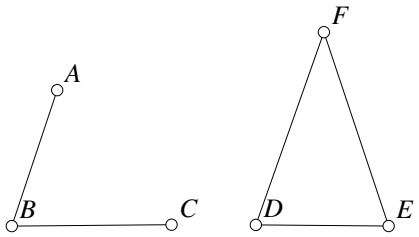

Figure 7.3: Copying an Angle
`Sleaze/Anglecopy.T` [M]

Figure 7.3 shows what we want. We want to be able to move points A, B, and C, and then use the resulting angle $\angle ABC$ twice as the equal angles in an isosceles $\triangle DEF$, where DE is the base. Here is the complete code to draw that diagram, even though it includes a bit of extra junk:

```
.geometry "version 0.2";
v1 = .free(-1.28394, 0.230712, "A");
v2 = .free(-1.65179, -0.871428, "B");
v3 = .free(-0.356492, -0.860066, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
ang1 = .a.vvv(v1, v2, v3, .noangle);
ang2 = .a.vvv(v3, v2, v1, .noangle);
v4 = .free(0.32695, -0.861679, "D");
v5 = .free(1.40373, -0.870804, "E");
l3 = .l.vv(v4, v5);
vx = .v.avv(ang1, v4, v5, .in);
vy = .v.avv(ang2, v5, v4, .in);
l4 = .l.vv(vy, v4, .in);
l5 = .l.vv(vx, v5, .in);
v6 = .v.ll(l4, l5, "F");
l6 = .l.vv(v6, v4);
l7 = .l.vv(v6, v5);
```

Only two lines were typed using the text editor—the ones that begin with `vx = .v.avv` and `vy = .v.avv`. All the rest was created with the standard point and click interface.

If you try to use `ang1` for both base angles, what will happen is that they will both take off from the endpoints in the same direction—counter-clockwise from the endpoint. For that reason, two versions of the same angle were made—`ang1` and `ang2` which measure the same angle, but with opposite orientations. If one of them is $50°$, the other will be 310^circ. But going $50°$ counter-clockwise is the same as going $310°$ clockwise so this will do exactly what you want.

7.4 Making Proofs or Constructions

If you want a demonstration where different information appears each time you press the “next” button, draw your initial figure with all layers enabled. Then disable layer

0, and draw the next set of stuff, then disable layer 1 and draw the next set, and so on.

Use the `.l1on`, `.l2on`, ... commands to make something continue to appear to the end of a proof. It's not too hard to go in and edit the file by hand to do this, or if you hold down the **Ctrl** key when you click on a layer button, it flips the state of all the following layers.

Use `.blink` or `.blink1` on steps of the proof that show a couple of lengths or angles are equal. Or you can use two different blinking colors for more complicated items like similar or congruent triangles.

7.5 Making Scripts

Somebody should write something for this someday. For now your best bet is to look at all the files on the CD that use scripts and see how they work. You can examine the text version of the files, and then load the files with **Geometer** to run them. Use a search tool to look through all of the `.T` and `.D` files on the CD for those that contain the text `.script`

7.6 Making Drawings for Publication

When you issue the command to make an "EPS" (Encapsulated PostScript) file, Geometer takes into account the current size and shape of the display area. It assumes that the window's x-dimension fills the page, and that the y-direction is relative to it. When the EPS file is produced, it will be 400 points wide—a little more than 5 inches.

If you are going to make an EPS file that includes a smeared primitive, the picture will probably look better if the smeared points or lines are more or less evenly spaced. For that reason, instead of dragging the control point by hand, write a script that automatically drags the control point in a uniform manner.

Chapter 8

Coordinate Systems

8.1 Barycentric Coordinates

Just for fun, here's a **Geometer** program that finds a point, given its barycentric coordinates.

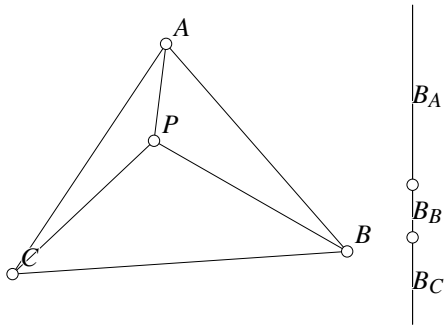

Figure 8.1: **Geometer** Construction Given Barycentric Coordinates
`Coordinates/Barymacro.T` [M]

Figure 8.1 shows the **Geometer** diagram for one set of barycentric coordinates. Here is the code that generates Figure 8.1:

```
.geometry "version 0.2";
v1 = .free(0.921847, 0.921847, .invisible, .circpoint);
v2 = .free(0.932504, -0.900533, .invisible, .circpoint);
l1 = .l.vv(v1, v2);
v3 = .vonl(l1, 0.927456, -0.0373089, .circpoint);
v4 = .vonl(l1, 0.929119, -0.321678, .circpoint);
l2 = .l.vv(v1, v3, "B\subscript{A}");
l3 = .l.vv(v3, v4, "B\subscript{B}");
```

```
l4 = .l.vv(v4, v2, "B\subscript{C}");
v5 = .free(-0.477798, 0.708703, .circpoint, "A");
v6 = .free(0.513322, 0.225577, .circpoint, "B");
v7 = .free(-0.765542, -0.669627, .circpoint, "C");
l5 = .l.vv(v5, v6);
l6 = .l.vv(v6, v7);
l7 = .l.vv(v7, v5);
Lac = .f.vv(v5, v7);
Lab = .f.vv(v5, v6);
Ba = .f.vv(v1, v3);
Bb = .f.vv(v3, v4);
Bc = .f.vv(v4, v2);
Ra = .f.rpn(Lac, Bc, Ba, Bc, .add,
        .div, .mul);
ca = .c.vf(v5, Ra, .invisible);
v8 = .v.lc(l7, ca, 1, .invisible);
Rb = .f.rpn(Lab, Ba, Ba, Bb, .add,
        .div, .mul);
cb = .c.vf(v6, Rb, .invisible);
v9 = .v.lc(l5, cb, 1, .invisible);
l8 = .l.vv(v9, v7, .invisible);
l9 = .l.vv(v6, v8, .invisible);
v10 = .v.ll(l8, l9, .circpoint, "P");
l10 = .l.vv(v5, v10);
l11 = .l.vv(v10, v7);
l12 = .l.vv(v10, v6);
```

The user can change either the shape of triangle ABC, or can change the relative sizes of B_A, B_B, and B_C by sliding the little markers on the vertical line on the right edge of the diagram. The ratio of the lengths on the line on the right should be the same as the ratio of the areas of the sub-triangles opposite vertices A, B, and C.

8.2 Trilinear Coordinates

A useful macro in **Geometer**, would be one that would generate a point given its trilinear coordinates. This can be a little tricky.

The most obvious approach is shown in Figure 8.2. Ignoring P_A for a moment, every point P whose distance from lines AC and AB is in the ratio $P_B : P_C$ lies on the line AP in the figure. Thus the required point must lie somewhere on this line. A similar line can be constructed through either B or C, and the intersection of those two lines will be at the point with the given trilinear coordinates.

Thus the line AP can be constructed if we can find a single additional point on it that is different from A. For the particular pair of lengths P_B and P_C construct the lines parallel to AC and AB at distances P_B and P_C, and those parallel lines will meet at a point on the required line.

The details of such a **Geometer** construction are shown in Figure 8.3. Two arbitrary points N and M are placed on the lines AB and AC, respectively. Perpendiculars to the lines are constructed at those two points. Then circles of radii P_B and P_C are constructed centered at M and N. The intersections of those circles with the perpendicular

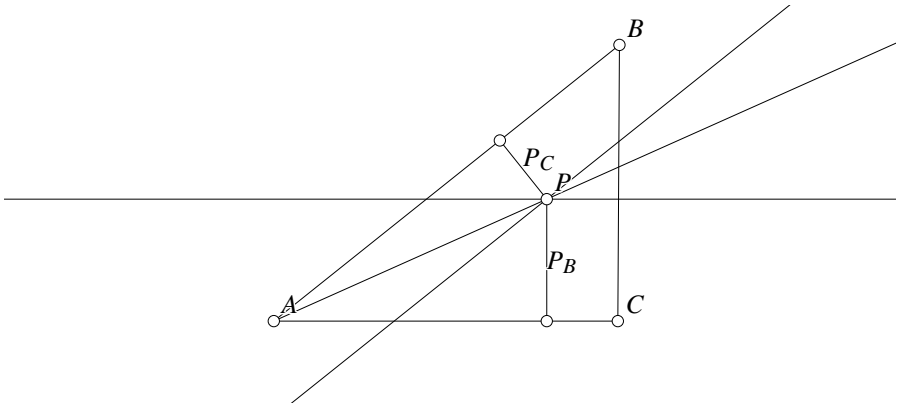

Figure 8.2: Finding P, given its trilinear coordinates
`Coordinates/Construct.T` [M]

lines are at points S and T which are at distances P_B and P_C from lines AC and AB, respectively. If parallels to AB and AC are constructed through S and T, they must meet in a point P such that the line AP includes all points having the correct $P_B : P_C$ ratio. If a similar line is constructed from either point B or C, the point satisfying all the ratios of the given trilinear coordinates will be at the intersection of those two lines.

The **Geometer** code that generates the point P for the arbitrary choice of $P_B = 0.2$ and $P_C = 0.3$ is shown below. Larger values of P_B and P_C, (such as 2 and 3) would work fine, but since **Geometer**'s coordinate system goes from roughly -1.0 to 1.0, the circles would be so large that you probably wouldn't even see them in the diagram.

```
.geometry "version 0.2";
v1 = .free(-0.886179, -0.589431, .circpoint, "A");
v2 = .free(0.853658, 0.800813, .circpoint, "B");
v3 = .free(0.845528, -0.589431, .circpoint, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
Pb = .f.rpn(0.200000);
Pc = .f.rpn(0.300000);
Vac = .vonl(l3, 0.247967, -0.589431);
Vab = .vonl(l1, 0.00952314, 0.126294);
Cb = .c.vf(Vac, Pb);
Cc = .c.vf(Vab, Pc);
Lb = .l.vlperp(Vac, l3);
Lc = .l.vlperp(Vab, l1);
v4 = .v.lc(Lb, Cb, 2);
v5 = .v.lc(Lc, Cc, 2);
l5 = .l.vlpar(v4, l3, .longline);
l6 = .l.vlpar(v5, l1, .longline);
v6 = .v.ll(l5, l6, .circpoint, "P");
```

The problem with this construction is that there are *two* intersections of a line with a circle, and this construction requires that the parallel lines be on the "inside" of the triangle. But the code above may flip from one to the other as the vertices A, B, and

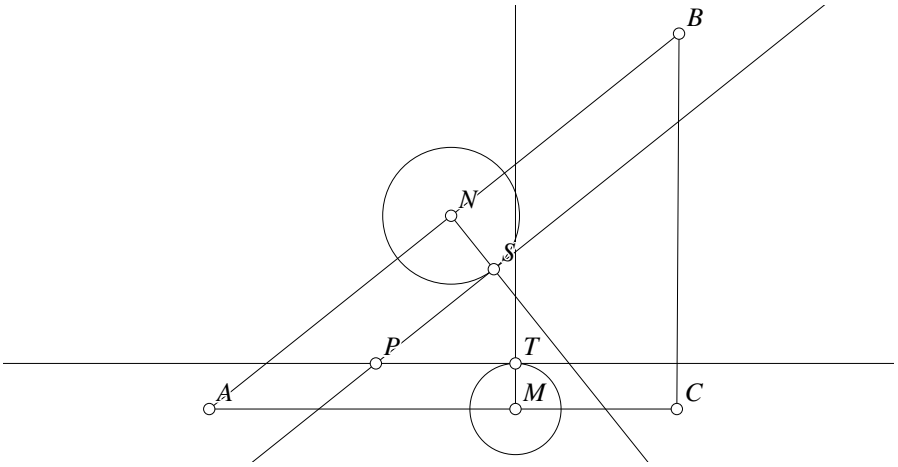

Figure 8.3: First Step of construction
`Coordinates/Macro1.D` [D]

C are moved. If you make an exact copy of this example in **Geometer**, and then turn the triangle inside out by moving the point C across the line AB, the parallel lines will stay on the same sides of AB and AC, and P will move outside the triangle.

Figure 8.4 shows an alternative approach. The angle A will be divided into two angles, α and β by the line through P. Since P_C and P_B are perpendicular to the sides of the triangles, $\sin\alpha = P_B/h$ and $\sin\beta = P_C/h$. Since the same side h is included in both triangles, we can solve for h in both of these equations and set them equal, giving $P_B \sin\beta = P_C \sin\alpha$.

Since $A = \alpha + \beta$, $\beta = A - \alpha$, so we have:

$$P_B \sin(A - \alpha) = P_C \sin\alpha.$$

Continuing by expanding $\sin(A - \alpha)$ and doing some algebra, we get:

$$\begin{aligned} P_B \sin A \cos\alpha - P_B \cos A \sin\alpha &= P_C \sin\alpha \\ P_B \sin A \cos\alpha &= (P_C + P_A \cos A)\sin\alpha \\ \frac{P_B \sin A}{P_C + P_B \cos A} &= \tan\alpha \\ \arctan\Big(\frac{P_B \sin A}{P_C + P_B \cos A}\Big) &= \alpha \end{aligned}$$

Once we have α we can construct the line through A making an angle of α relative to the line AC, and do a similar thing from another vertex.

Using the construction above, here is the complete **Geometer** code to generate a point P with given trilinear coordinates $P_A : P_B : P_C$. It actually does a bit more than that, and calculates all three lines through the vertices, and if you change `l1`, `l2`, and `l3` to be visible, you'll see that they all do meet in a point, as required.

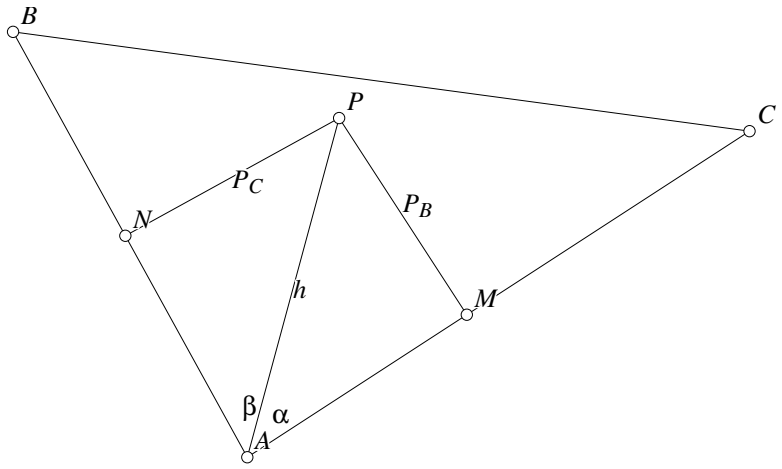

Figure 8.4: First Step of construction
`Coordinates/AltConstruct.D` [D]

```
.macro .vertex trilinear(.vertex v1, .vertex v2, .vertex v3,
    .flt f1, .flt f2, .flt f3)
{
    ang1 = .a.vvv(v3, v1, v2, .invisible);
    ang2 = .a.vvv(v1, v2, v3, .invisible);
    ang3 = .a.vvv(v2, v3, v1, .invisible);
    a1 = .f.rpn(f2, ang1, .sin, .mul, f3,
        f2, ang1, .cos, .mul, .add,
        .atan2, .invisible);
    a2 = .f.rpn(f3, ang2, .sin, .mul, f1,
        f3, ang2, .cos, .mul, .add,
        .atan2, .invisible);
    a3 = .f.rpn(f1, ang3, .sin, .mul, f2,
        f1, ang3, .cos, .mul, .add,
        .atan2, .invisible);
    A1 = .a.f(a1);
    A2 = .a.f(a2);
    A3 = .a.f(a3);
    va1 = .v.avv(A1, v3, v1, .invisible);
    va2 = .v.avv(A2, v1, v2, .invisible);
    va3 = .v.avv(A3, v2, v3, .invisible);
    l1 = .l.vv(v1, va1, .invisible);
    l2 = .l.vv(v2, va2, .invisible);
    l3 = .l.vv(v3, va3, .invisible);
    .return v4 = .v.ll(l1, l2, .invisible);
}
```

8.2.1 Malfatti's Problem

Using this macro to generate points given their trilinear coordinates it is possible to do a lot of interesting constructions in **Geometer** even if you don't know quite how to do

them using standard techniques.

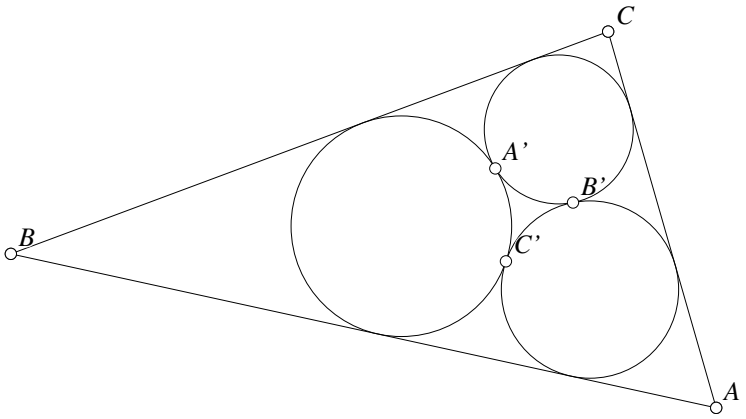

Figure 8.5: Malfatti's Problem
`Coordinates/Malfatti.T` [M]

For example, Malfatti's problem requires the construction of three circles within a triangle that are tangent to each other, and so that one pair are tangent to the sides of the triangle as in Figure 8.5.

This section would probably deserve a triple (or even quadruple!) black-diamond designation, but we're going to cheat and use the observation that locations and sizes of the three circles can be determined relative to the so-called "Ajima-Malfatti" points of the triangle.

The first Ajima-Malfatti point has the following trilinear coordinates:

$$\sec^4(A/4) : \sec^4(B/4) : \sec^4(C/4).$$

The second Ajima-Malfatti point has these trilinears:

$$\frac{1}{t_2}+\frac{1}{t_3}-\frac{1}{t_1} : \frac{1}{t_3}+\frac{1}{t_1}-\frac{1}{t_2} : \frac{1}{t_1}+\frac{1}{t_2}-\frac{1}{t_3},$$

where

$$\begin{aligned} t_1 &= 1+2[sec(A/4)cos(B/4)cos(C/4)]^2 \\ t_2 &= 1+2[sec(B/4)cos(C/4)cos(A/4)]^2 \\ t_3 &= 1+2[sec(C/4)cos(A/4)cos(B/4)]^2. \end{aligned}$$

The construction is accomplished because the points of common tangency of the circles labeled A', B', and C' in Figure 8.5 satisfy the following conditions:

1. The lines AA', BB', and CC' are concurrent at the first Ajima-Malfatti point.

2. If A'', B'', and C'' are the centers of the excircles opposite A, B, and C, respectively, then the lines $A'A''$, $B'B''$, and $C'C''$ are concurrent at the second Ajima-Malfatti point.

The rest of the details of the construction can be obtained by looking at the source code for the **Geometer** diagram for Figure 8.5. The definition of the `trilinear` macro has been removed from the listing below to save space:

```
v1 = .free(1.52239, -0.934328, "A");
v2 = .free(-1.86866, -0.19403, "B");
v3 = .free(1.00299, 0.874627, "C");
l1 = .l.vv(v1, v2);
l2 = .l.vv(v2, v3);
l3 = .l.vv(v3, v1);
ang1 = .a.vvv(v3, v1, v2, .in);
ang2 = .a.vvv(v1, v2, v3, .in);
ang3 = .a.vvv(v2, v3, v1, .in);
fa = .f.rpn(1.000000, ang1, 4.000000, .div, .cos,
        .div, .dup, .dup, .dup, .mul,
        .mul, .mul, 0.200000, .mul);
fb = .f.rpn(1.000000, ang2, 4.000000, .div, .cos,
        .div, .dup, .dup, .dup, .mul,
        .mul, .mul, 0.200000, .mul);
fc = .f.rpn(1.000000, ang3, 4.000000, .div, .cos,
        .div, .dup, .dup, .dup, .mul,
        .mul, .mul, 0.200000, .mul);
mf1 = trilinear(v1, v2, v3, fa, fb, fc, .in, "mf1");
c1 = .c.lll(l3, l2, l1, 2, .in);
c2 = .c.lll(l1, l3, l2, 2, .in);
c3 = .c.lll(l3, l1, l2, 2, .in);
v4 = .v.ccenter(c3, .in);
v5 = .v.ccenter(c1, .in);
v6 = .v.ccenter(c2, .in);
t1 = .f.rpn(1.000000, ang1, 4.000000, .div, .cos,
        .div, ang2, 4.000000, .div, .cos,
        .mul, ang3, 4.000000, .div, .cos,
        .mul, .dup, .mul, 2.000000, .mul,
        1.000000, .add);
t2 = .f.rpn(1.000000, ang2, 4.000000, .div, .cos,
        .div, ang1, 4.000000, .div, .cos,
        .mul, ang3, 4.000000, .div, .cos,
        .mul, .dup, .mul, 2.000000, .mul,
        1.000000, .add);
t3 = .f.rpn(1.000000, ang3, 4.000000, .div, .cos,
        .div, ang2, 4.000000, .div, .cos,
        .mul, ang1, 4.000000, .div, .cos,
        .mul, .dup, .mul, 2.000000, .mul,
        1.000000, .add);
T1 = .f.rpn(1.000000, t2, .div, 1.000000, t3,
        .div, .add, 1.000000, t1, .div,
        .sub);
T2 = .f.rpn(1.000000, t3, .div, 1.000000, t1,
        .div, .add, 1.000000, t2, .div,
        .sub);
T3 = .f.rpn(1.000000, t1, .div, 1.000000, t2,
        .div, .add, 1.000000, t3, .div,
        .sub);
```

```
mf2 = trilinear(v1, v2, v3, T1, T2, T3, .in, "mf2");
l4 = .l.vv(v1, mf1, .in);
l5 = .l.vv(v2, mf1, .in);
l6 = .l.vv(v3, mf1, .in);
lla = .l.vv(v4, mf2, .in);
llb = .l.vv(v5, mf2, .in);
llc = .l.vv(v6, mf2, .in);
v7 = .v.ll(llc, l5, .green, "B'");
v8 = .v.ll(llb, l4, .green, "C'");
v9 = .v.ll(lla, l6, .green, "A'");
v10 = .v.vvvbisect(v3, v1, v2, .in);
v11 = .v.vvvbisect(v1, v2, v3, .in);
v12 = .v.vvvbisect(v2, v3, v1, .in);
l7 = .l.vv(v12, v3, .in);
l8 = .l.vv(v2, v11, .in);
l9 = .l.vv(v10, v1, .in);
l10 = .l.vvperp(v8, v7, .in);
l11 = .l.vvperp(v9, v8, .in);
l12 = .l.vvperp(v9, v7, .in);
v13 = .v.ll(l12, l9, .in);
c4 = .c.vv(v13, v7, .green);
v14 = .v.ll(l10, l7, .in);
c5 = .c.vv(v14, v8, .green);
v15 = .v.ll(l8, l11, .in);
c6 = .c.vv(v15, v8, .green);
```

Chapter 9

Quickstart Guide

This section contains the bare minimum of information to install and use **Geometer** to view prepared files. The reference manual contains a tutorial chapter if you wish to learn to draw your own simple diagrams.

Geometer only works on Windows 95/98/NT machines.

Load the CD into your drive and run the install script. By default, **Geometer** will install into the directory `C:/Geometer` but you can put it elsewhere. All the interesting files are in that installation directory which contains a file called `README` that describes the arrangement of the installed files.

All **Geometer** diagrams are files whose name ends in `.T` or `.D` (the `.T` files are generally more interesting—they are theorems, and the `.D` files are simply drawings used in documents). You can view any of them by double-clicking on the diagram's icon.

Geometer diagrams consist of a points, lines, circles, and so on. The only thing you can manipulate in the drawing area with a mouse are some of the points. (If it's necessary to have a movable line in a diagram, that line is usually a line going through two points, and you can move one or both of the points to move the line. Similarly for all other non-point objects.)

To move a point, use the mouse to move the cursor over a point. Press down on the left mouse button, and move the mouse with that button held down. If that point is movable, the diagram will shift appropriately.

Many diagrams present step-by-step proofs or construction. In those diagrams, you can click on the *Next* button to advance to the next step of the proof/construction. You can begin again by pressing *Start* or you can go back one step by pressing *Prev* (for "previous").

Even if the diagram is a proof or construction, you can still manipulate the points as you step through. For example, suppose you get to a step in the proof that says, "Therefore, points A, B, C, and D all lie on a circle." If you don't believe it, you can at this point modify the drawing by moving points to see that in fact the four points do continue to

lie on a circle. Then once you've convinced yourself, you can continue with the proof or construction.

Finally, some diagrams have scripts. If the *Run Script* button in the control panel is not grayed-out, you can press it to run a prepackaged script. You can stop the running by clicking in the drawing window, or by letting the script run to completion.

In most cases for the prepared diagrams discussed above, each screen will contain instructions for how to proceed, like "Press 'Next' to continue ...", or "Press the 'Run Script' button." Diagrams that are just used for drawings (generally having a `.D` suffix) or that were obtained from other sources may or may not have good instructions, so in those cases, you're on your own.

Index