11.9 Finishing touches with \textit{twoDG}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Concepts} \\
Streamlining output with function \texttt{draw} \hrulefill \\
Drawing line segments and polygons \hrulefill \\
Clipping lines and rays \hrulefill \\
Arcs and circles \hrulefill \\
Angle marks \hrulefill \\
Ticks \hrulefill \\
Example: figure 11.8.1 \hrulefill \\
Arrowheads \hrulefill \\
Exercises \hrulefill \\
\hline
\end{tabular}
\end{center}

This section continues the presentation of the \textit{twoDG} package begun in sections 11.2, 11.7, and 11.8. It’s organized around construction of figure 11.8.1. The package was designed to make it as easy as possible to draw figures of that type. The goal is realized by

- specialized functions that construct graphics elements for angle marks, ticks, and arrowheads,
- a convenient interface between the package’s algebraic features and \textit{Mathematica} graphics primitives, and
- a function that simplifies the last stages of the process by assembling display and option lists into a \texttt{Graphics} object and calling \texttt{Show}.

These topics are discussed next, in reverse order.

\textbf{Function \texttt{draw}}

To draw a geometric figure such as those in section 11.8, you assemble a display list of graphics primitives and directives corresponding to all the details of the figure. From that list you build a \texttt{Graphics} object \( \gamma \), which you pass to function \texttt{Show} to render. Graphics options whose default values you must change can be built into \( \gamma \) or supplied
as arguments to Show. You must set some options, such as AspectRatio → Automatic, virtually every time. Those should become standard arguments to Show. Others may be particular to the task at hand, are probably being set experimentally, and should be included in γ with the other data for the figure you’re constructing.

This programming is tedious. The display list is often quite long, and you need many experimental trials to get everything right. If the process isn’t streamlined, you won’t do it.

Function draw eliminates a step or two at the end of the process. Over many trials, it can save appreciable effort. There are two versions:

\[
\text{draw}[\text{L}_\_] := \text{Show}[\text{Graphics}[\text{L}], \text{AspectRatio} \to \text{Automatic}];
\]

\[
\text{draw}[\text{L}_\_, \text{M}_\_] := \text{Show}[\text{Graphics}[\text{L}],
\quad \text{Append}[\{\text{M}\}, \text{AspectRatio} \to \text{Automatic}]]
\]

The first builds the Graphics object, sets the aspect ratio and calls Show. It’s often convenient to invoke it with the //draw syntax at the end of the line of code that builds L. Its second version allows you to specify another option or list of options, M. You could even use M to override AspectRatio → Automatic with another AspectRatio setting, because M is placed first in the resulting list and Show ignores later occurrences of the same option. Draw returns the Show value, which is Graphics[L] with its option list updated.

**Drawing segments and polygons**

The Point ADT serves two purposes in twoDG. It’s used with many operations for computing coordinates of points in a figure, and as a graphics primitive for drawing them. How do you draw lines? You don’t—they’re infinite in extent. You draw line segments, which are determined by their endpoints. Common notation for the segment between points P and Q is PQ or \( \overline{PQ} \). Neither is possible with Mathematica. Instead, the package uses the multiplication operator \(*\):

\[
\text{Point}[\{\text{x}_0, \text{y}_0\}] \ast \text{Point}[\{\text{x}_1, \text{y}_1\}] ^:=
\quad \text{Line}[\{\{\text{x}_0, \text{y}_0\}, \{\text{x}_1, \text{y}_1\}\}]
\]
This operator may be elided. If \( P \) and \( Q \) are \texttt{Point} objects then both \( P*Q \) and \( P \ Q \) represent the segment between \( P \) and \( Q \). In the latter situation, the space is essential.

As an example, consider the following code, which produced figure 1, an incomplete version of figure 11.8.1:

\[
\begin{align*}
P &= \texttt{Point}[0,1.2]; \quad \texttt{Oh} = \texttt{Point}[-0.8,0]; \quad Q = \texttt{Point}[0.7,0]; \\
OP &= P \vee \texttt{Oh}; \quad OQ = \texttt{Oh} \vee Q; \quad h = \texttt{bisector1}[OP,OQ]; \\
X &= h \cap \texttt{vertical}[0.35]; \\
Y &= OP \cap \texttt{pointPerpendicular}[X,OP]; \\
Z &= OQ \cap \texttt{pointPerpendicular}[X,OQ]; \\
L &= \{\texttt{AbsolutePointSize}[5], \texttt{AbsoluteThickness}[1], \texttt{Oh}, X, Y, Z, \texttt{Oh} P, \texttt{Oh} Q, \texttt{Oh} X, X Y, X Z\}; \\
draw[L]
\end{align*}
\]

The first line of this code gives specific numerical coordinates for points \( P, O, Q \). They were adjusted experimentally to look best. The second code line defines the angle’s legs and its bisector. The assignment \( OP = \texttt{Oh} \vee P \) was used initially, but yielded the wrong figure. As reported in section 11.8, \texttt{bisector1} can represent the bisector of either pair of angles determined by the legs. Evidently the bisector in the desired figure is on the positive side of one of \( \texttt{Oh} \vee P \), \( \texttt{Oh} \vee Q \) and the negative side of the other. The signs of all three coefficients of one of these equations must be changed. Reversing the order of \( \texttt{Oh} \) and \( P \) in the join operation does that.

The third line of code establishes a point \( X \) on bisector \( h \) by intersecting it with a vertical line whose horizontal coordinate was chosen by experiment. The next two lines locate \( Y \) and \( Z \) as intersections of legs with their perpendiculars through \( X \). Now the display list is constructed. First, point size and line thickness, determined by experiment. Then the points and line segments emphasized in the argument that the figure will illustrate. Function \texttt{draw} renders the figure.

---

\footnote{\( \texttt{Oh} \) is used because \( \texttt{O} \) is a reserved symbol in \textit{Mathematica}.}
This example was somewhat tedious. It would be far worse without use of twoDG features.

Representing a polygon $UVWX$ in a display list using this programming style requires a repetitive sequence such as $U\ V\ ,\ V\ W\ ,\ W\ X$. The polygon function helps reduce the tedium (particularly for longer sequences). It replaces a sequence of Point objects by the Line primitive determined by the corresponding sequence of pairs of coordinates:

$$\text{polygon} := \text{Line}\left[\text{Table}\left[\{\text{c1}[L[[k]]],\ \text{c2}[L[[k]]]\},\ {k,1,\text{Length}[L]}\right]\right]$$

This definition lets you replace that example sequence by $\text{polygon}\{U,V,W,X\}$. There’s an example under the next heading. The twoDG package also includes an analogous function filledPolygon that converts a list of Point objects into a Polygon primitive.

**Clipping**

Occasionally you want to draw the graph $g$ of a lineq object. That must be done by drawing a segment with a Line primitive, which requires the coordinates of the ends of the segment. To compute those, you need the coordinates of the borders of the entire figure. Then you must determine the which border segments $g$ intersects, and compute the intersection coordinates. This process is called clipping. The twoDG package provides functions for this purpose and for clipping a ray (half-line).

Function lineClipped has two forms. The first performs a clipping task slightly more general than what was just described: $\text{lineClipped}\{\text{LL,UR},g\}$ returns a Line object, the segment corresponding to the intersection of

- the line represented by lineq object $g$, with
- the rectangle with horizontal and vertical edges determined by Point objects LL and UR (lower left and upper right).
The second form, \texttt{lineClipped[G,g]}, computes the segment determined by line \texttt{g} and the borders of the entire figure represented by \texttt{Graphics} object \texttt{G}.ootnote{You can use the second form to append the clipped segment of a \texttt{lineq} object \texttt{g} to a display list \texttt{L} that already contains points that determine the borders. Check the list with \texttt{draw}, then execute \texttt{AppendTo[L,lineClipped[%,g]]} //\texttt{draw}. \textit{Caution:} \texttt{draw} calls \texttt{Show} to render \texttt{L}, and \texttt{Show} may change the \texttt{PlotRange} option “automatically”, which would affect subsequent clipping.} It uses the \texttt{AbsoluteOptions} function described in section 11.6 to find appropriate points \texttt{LL} and \texttt{UR}, then calls the first form of the \texttt{lineClipped} function.

Function \texttt{rayClipped} has two analogous forms, \texttt{rayClipped[\{LL,UR\},P,Q]} and \texttt{rayClipped[G,P,Q]}. It computes the “visible” segment of the ray that starts at \texttt{P} and passes through \texttt{Q}.

Clipping is extremely tedious, and the method used isn’t particularly important in general mathematics. For that reason, the codes for the \texttt{twoDG} clipping functions aren’t discussed here. You can find them on the CD enclosed with the book. Clipping is such a frequent operation, however, that its efficiency is extremely important for high-speed interactive graphics applications. For that reason, clipping algorithms are studied in detail in computer graphics books.ootnote{For example, Foley and van Dam 1982.} The algorithm used in \texttt{twoDG} hasn’t been optimized, because efficiency isn’t important for its target applications.

For a demonstration of line clipping, consider the following code, which produced figure 2.

\begin{verbatim}
X = Point[1,0]; Y = Point[0,1]; U = X + Y; V = U + X; W = 2X; \lambda = \{origin, U\}; Table[lineClipped[\lambda, lineq[1,3,k]],\{k,-8,2,0.1\}] \cup Table[lineClipped[\lambda, lineq[3,-1,k]],\{k,-4,2,0.1\}] \cup {polygon[{U,V,W,X}]} //draw
\end{verbatim}

\textbf{Figure 11.9.2} Line clipping
### Arcs and circles

*Mathematica*’s built-in graphics primitive `Circle` doesn’t complain if you specify a negative radius. Two `twoDG` features protect against this sort of error and make it easier to draw the most common circle:

```mathematica
Unprotect[Circle];
Circle[Point[x_, y_], r_] := 
  If[r < 0, noCircle, Circle[{x, y}, r]];
unitCircle = Circle[origin, 1]
```

The undefined symbol `noCircle` is used as an error signal. The next function represents the circular arc with radius `r` and center `Z` (a `Point` object) from direction `α` counterclockwise to direction `β`, where `0 ≤ α, β < 2π`.

```mathematica
arc[Z_, r_, α_, β_] := If[r < 0, noCircle,
  Circle[{c1[Z], c2[Z]}, r, {α, If[α ≤ β, β, β + 2π]}]]
```

Further `twoDG` circle features are explored in exercises 7 and 8.

### Angle marks

Geometric figures such as figure 11.8.1 often include marks to indicate that certain angles are congruent. The `twoDG` function `angleMark` constructs a graphics element to mark angle `POQ` with `n` circular arcs from ray `OP` counterclockwise to ray `OQ`, starting at radius `r` and stepping outward at intervals of width `dr`:

```mathematica
angleMark[P_, O_, Q_, r_, dr_, n_] := Module[{α, β},
  α = direction[O, P];  β = direction[O, Q];
  Table[ arc[0, r+k*dr, α, β], {k, 0, n-1} ]]
```

When `n = 1`, the most common situation, argument `dr` is irrelevant. A second version of the function simplifies that case:

```mathematica
angleMark[P_, O_, Q_, r_] := angleMark[P, O, Q, r, 0, 1]
```

---

4 *Usually* circular, not elliptic, since the `twoDG` function `draw` chooses aspect ratio 1.
It's customary to indicate a right angle with a special angle mark that uses the vertex, legs, and two additional segments to build a small interior square. The \textit{twoDG} function \texttt{rightAngle} constructs a graphics element to mark right angle \( \text{POQ} \) with such a square, with edge length \( s \):

\begin{verbatim}
rightAngle[p_,o_,q_,s_] := Module[{t,u,T,U,V},
t = s/distance[o,p];  u = s/distance[o,q];
T = t(p - o);   U = u(q - o);
polygon[{o+t, o+t+u, o+u}]]
\end{verbatim}

\textbf{Ticks}

Diagrams such as figure 11.8.1 often include marks to indicate that certain segments are congruent. The \textit{twoDG} function \texttt{tick} constructs a graphics element to mark segment \( \text{PQ} \) with \( n \) ticks of length \( u \), centered with an interval of width \( t \):

\begin{verbatim}
tick[p_,q_,u_,t_,n_] := Module[{m,d,T,U,dT,dU},
m = midpoint[p,q]; d  = distance[p,q];
T = q-p;   dT = (t/(2d))T;
U = Point[-c2[T],c1[T]]; dU = (u/(2d))U;
Table[ (m + k dT + dU)(m + k dT - dU), {k,-(n-1),n-1,2} ]]
\end{verbatim}

This code computes the vector \( T \) from \( P \) to \( Q \), its length \( d \), and a perpendicular vector \( U \) with the same length. Vectors \( dT \) and \( dU \), parallel to \( T \) and \( U \) respectively, correspond to half an interval between ticks and to half a tick. Finally, this function builds a list of \texttt{Line} objects corresponding to the ticks. When \( n = 1 \), the most common situation, argument \( t \) is irrelevant. A second version of the function simplifies that case:

\begin{verbatim}
tick[p_,q_,u_] := tick[p,q,u,0,1]
\end{verbatim}

A good example of this kind of programming is the following code, which produced the angle marks and ticks in figure 11.8.1:

\begin{verbatim}
t = 0.05;  u = 0.03;
AppendTo[L,
  {angleMark[Q,Oh,X,0.1], angleMark[X,Oh,P,0.12],
   rightAngle[Oh,Y,X,t],   rightAngle[X,Z,Oh,t],
   tick[X,Y,t], tick[Oh,Y,t,u,2],
   tick[X,Z,t], tick[Oh,Z,t,u,2]}]
\end{verbatim}
Insert it just before `draw` in the code displayed earlier for figure 1.

**Arrowheads**

So many mathematical diagrams include arrows that *Mathematica* provides a standard add-on package, `Graphics`Arrow`, with facilities for creating a great variety of such pointers. So great, in fact, that it’s hard to figure out how to draw simple arrows. That package is beyond the scope of this book, but *twoDG* includes a simple function that you may use directly, or tailor easily to fit your needs. Function `Arrowhead` constructs a graphics element representing a right angle with vertex \( V \) (a `Point` object) and legs of length \( d \), pointing in direction \( \theta \):

\[
\text{arrowhead}[V_, \theta_, d_] := \text{Module}\{\{P,Q\},
  P = V + d*\text{cis}[\theta + 5\pi];
  Q = V + d*\text{cis}[\theta + 3\pi/4];
  \{V,P,V,Q\}\}
\]

For an example, see exercise 4.

**Exercises**

1. Use *twoDG* features to draw figure 11.8.2. Take care to interrupt the bisector of angle \( C \) at \( I \). Draw a second figure that eliminates some of the clutter inside the triangle, but includes its *incircle*, which is tangent to all three edges. *Suggestions*:
   i. Erase a region by drawing with the background color.
   ii. According to the proof of the angle bisectors theorem in section 11.8, the intersection of the bisectors is equidistant from the edge lines.

2. Use *twoDG* features to draw a figure to illustrate the perpendicular bisectors theorem. (See exercise 4 in section 11.8.) Mark all right angles and all pairs of congruent segments. In your figure, include the *circumcircle*, which passes through all three vertices.
3. In figure 3, some vertices of a square have been joined to edge midpoints to form a kite. Use twoDG features to draw it. Be sure that the filled region doesn’t obscure the line segments.
4. Use *twoDG* features to draw figure 4, an arrow commonly needed in oversized advanced algebra diagrams to represent an inclusion map. *Suggestion:* This solution used two circular arcs, two line segments, and an arrowhead.

5. Use *twoDG* function `rayClipped` to draw figure 5.

6. Use *twoDG* features to draw figure 6 precisely. Mark all congruent segments with ticks and all pairs of congruent angles with angle marks. Mark any right angle. Give an argument that no further ticks or angle marks are appropriate. Measure the angles of triangle $AOE$ in degrees.

7. The *twoDG* package overloads the intersection operator to represent intersections of a circle with a line, and of two circles. Values $L$ of formulas

   $$\text{Circle}[Z,r] \cap \text{lineq}[a,b_,c_] \quad \text{Circle}[Y,q] \cap \text{Circle}[Z,r]$$

   can be an empty list, a list of one `Point` object, or a list of two. You can use `Length[L]` to determine how many points constitute the intersection. If you attempt to intersect a circle with itself, the operator returns the undefined symbol `badIntersection`. Without consulting the *twoDG* package, write code for these overloaded operators, and test it for a set of valid and invalid inputs. Then compare your results with the code in the package, and explain the differences. *Suggestion:* Subtracting the equations of two nonconcentric circles yields a linear equation satisfied by any intersection points; compute the intersection of its graph with one of the circles.

8. Use *twoDG* features to draw the unit circle $U$, the point $P<3,2>$, the two tangents from $P$ to $U$, and auxiliary lines and/or circles you use in constructing the tangents. Indicate any right angles used, with marks situated so that no other parts of the figure pass through the little squares. *Suggestion:* Since the tangents are perpendicular to the radii at the points of tangency, and right angles are inscribed in semicircles, the intersections lie on the circle through $P$ and the origin.

9. Angle and tick marks are specified in *twoDG* using the same scale as the figure being drawn, rather than the fractions of total figure width or printer’s-point measurements utilized by *Mathematica* graphics options. Is that appropriate?