Multiple graphics can be superimposed, displayed side-by-side in a graphics array, or displayed in rapid succession as an animation. These techniques can all start the same way—you produce an array $\lambda$ of Graphics objects representing complete figures. You’ll probably complete one or several of them with many trial-and-error steps, and adjust options so that their appearance is compatible with the desired effect. Then you’ll build the entire $\lambda$ with option DisplayFunction -> Identity to avoid excessive output. The array entries are the objects returned by Show, perhaps via a Plot function or the twoDG function draw. The desired superposition, array, or animation effect is produced by appropriately processing $\lambda$.

**Superposition**

The most common multiple-graphics technique is superposition: displaying multiple Graphics objects, each atop the previous so that the resulting figure is the union of all their elements. There’s no need for this with figures produced by the same Plot function. Each of Plot, ParametricPlot, PolarPlot, and ImplicitPlot will graph a list of functions; to use ListPlot or PolarListPlot to graph several functions, simply graph the union of the corresponding lists. Similarly, there’s no need to superimpose figures that you build explicitly from graphics primitives. Just render the union...
of their display lists. Superposition is required, though, when you must use different methods to produce different portions of the desired figure.

For example, consider figure 1, which you might use to illustrate three aspects of the motion of a point \( P \) according to equations \( x = a \cos t, \ y = b \cos t, \) with \( a > b, \) where \( t \) represents time. First, \( P \) follows an elliptical path that you can draw with \texttt{ParametricPlot}. Second, a \texttt{ListPlot} showing points \( P \) at equally spaced time intervals demonstrates that \( P \) moves fastest near the ends of the shorter axis and slowest near those of the longer. (Consider the dot spacing.) Finally, two pairs of line segments, each drawn with graphics primitives from one focus of the ellipse to a point \( P \) to the other focus, can emphasize that all such polygons have the same length. The following code produced the figure.

\begin{verbatim}
  a = 1;  b = 0.7;
  F = {Sqrt[a^2 - b^2], 0};
  P[t_] := {a Cos[t], b Sin[t]};
  n = 50;  dt = 2\[Pi]/n;
  ellipse = ParametricPlot[P[t], {t,0,2\[Pi]},
    PlotStyle -> AbsoluteThickness[0.5],
    DisplayFunction -> Identity];
  PP = Table[P[k*dt], {k,0,n}];
  dots = ListPlot[PP, PlotStyle -> AbsolutePointSize[3],
    DisplayFunction -> Identity];
  L = {AbsolutePointSize[5], Point[F], Point[-F],
    AbsoluteThickness[1],
    AbsoluteDashing[{8, 4}], Line[{-F, PP[[20]], F}],
    AbsoluteDashing[{2, 2}], Line[{-F, PP[[40]], F}];
  \lambda = {ellipse, dots, Graphics[L]};
  Show[\lambda, Axes -> False, AspectRatio -> Automatic,
    DisplayFunction -> $DisplayFunction]
\end{verbatim}

\footnote{The \textit{foci} of this ellipse have coordinates \( \pm \sqrt{a^2 - b^2}, 0. \)}
Several trials were required without the `DisplayFunction → Identity` option, showing the ellipse, dots, and display list `L` separately (see figure 2) then together, to find appropriate thickness, point size, and dashing directives, and to select indices 20 and 40. The `AspectRatio` and final `DisplayFunction` options overrode the default `GoldenRatio` and explicit `Identity` values used in the `ParametricPlot` and `ListPlot` lines. `Show` automatically adjusted the `PlotRange` option to accommodate all the points.

You can ask many questions about how `Show` might reconcile conflicting option values in the entries of list `\(\lambda\)`. For example, the author wondered, would a background specification in one entry be regarded as opaque, and thus obliterate the part of the figure corresponding to preceding entries? Experimentation found the answer to be `no`. As for some other options discussed in section 11.6, `Show` evidently accepts the first entry’s `Background` option and doesn’t change it once it’s set. Determining what happens with other such conflicts will require similar experimentation.

**Side-by-side with `GraphicsArray`**

The simplest multiple-graphics effect is side-by-side display. There’s no need for that when you use the author’s technique of exporting *Mathematica* figures for import into a word processor—you can display them however you want. But for an interactive presentation done completely with *Mathematica* you may want to override its standard method of displaying an array of `Graphics` objects as a vertical sequence of figures.

For this purpose, *Mathematica* provides function `GraphicsArray`. For example, the code `Show[GraphicsArray[\(\lambda\)], ImageSize → 936]` produced figure 2. The `ImageSize` option, 936 printer’s points = 6½ inches, the width of this page, overrode the default, four inches. Without it, the tick labels would be too crowded to read. Notice that in this situation, `Show` doesn’t adjust the `PlotRange` option for consistency. (You wouldn’t want it to—the graphics might be unrelated.) Moreover, most options to `Show` won’t affect the individual figures. For example, you can’t remove the axes that way. The options discussion in section 11.6 noted that frames are handled differently to avoid collisions, and you can adjust the spacing between graphics.

To compare the entries of a long list \(\lambda = \{\lambda_1, \ldots, \lambda_n\}\) of `Graphics` objects, you may want to display them in a two-dimensional array, with \(m\) figures in each row, where \(m\) divides \(n\). You can accomplish this with the built-in *Mathematica* list manipulation
function \textit{Partition}. For example, with \( m = 2 \) and \( n = 2k \), \( \mu = \text{Partition}[\lambda, 3] \) has value \( \{\{\lambda_1, \lambda_2\}, \{\lambda_3, \lambda_4\}, \ldots, \{\lambda_{n-1}, \lambda_n\}\} \), and \( \text{Show}[\mu] \) displays \( k \) rows of 2 figures.\(^2\)

![Figure 11.10.2 GraphicsArray version of figure 1](image)

\textbf{Animation}

Generate a list of related figures with the same options—perhaps plots of lists of more and more points, or a sequence of graphs of a function that depends on a changing parameter, all using the same axes and plot ranges. Watch the display: It’s like an example of a recreational book common more than a century ago, with such figures in the same position at the right edges of successive pages. Stare at that position and riffle the pages. You see a primitive movie.

As the industrial age developed, that device was supplanted by movies. The figures were printed on frames of a film strip, and projected in rapid succession on a screen. As the electronic age develops, you can use the same technique with \textit{Mathematica}. You \textit{must} use it, even though it’s so primitive. You \textit{must} create a list of figures analogous to the frames of the film, then instruct the front end to display them in rapid succession.

There’s no facility for the double-buffering technique common in computer graphics, that takes advantage of the fact that your computer can probably generate a new frame in the same time that it takes to retrieve and display a previously computed

\(^2\) If \( n = 2k + 1 \), \( \mu \) will still be a list of \( k \) lists of two entries each; \textit{Partition} ignores the odd leftover entry.
one. You must generate all the frames first, then instruct the front end to display them in rapid succession. Animating a list of a few hundred frames requires an enormous amount of memory.

The following discussion explains how to animate with Mathematica, with an example, but it can’t display the results effectively. For that you must execute the code that you’ll find on the CD that accompanies this book, and work through the animation exercises at the end of this section.

The example is based on figure 1. That figure doesn’t effectively convey the idea of the motion. You have to deduce or be told, for instance, that the point moves counterclockwise. To demonstrate the motion, animate the figure. Consider the following stripped-down code, which produces a list of fifty figures, consisting of the single point $P$ in fifty successive positions.

\begin{verbatim}
a = 1;  b = 0.7;
P[t_] := {a Cos[t], b Sin[t]};
n = 50;  dt = 2\pi/n;
Do[L = {AbsolutePointSize[3], Point[P[k*dt]]};
   Show[Graphics[L], PlotRange->{{-1.2,1.2},{-0.9,0.9}}],
   {k, 0, n-1}]
\end{verbatim}

If you watch the output as it’s generated, you see fifty figures scrolling by, each of which shows one point, $P$, at a different position. When output stops, you’re looking at the last figure, in which $P = <1, 0>$. Double-click the second innermost cell bracket at right. This “collapses” the vertical array of fifty figures so that only the first shows. The bracket now has an arrow at the bottom. Double-click anywhere within the cell (not the bracket). The Mathematica front end now displays the fifty figures in rapid succession, cyclically. You can control it with the buttons at the bottom of the screen—as your mouse passes over them, explanatory screen tips appear. Stop the animation by clicking the cell.

Can you program Mathematica to make its front end start the animation immediately, so that you don’t have to fumble in public trying to double-click a cell bracket that

---

3 In many mathematical animations, as in many computer games, relatively few pixels change from one frame to the next. The double-buffering technique uses two frames, either of which it can display. While displaying one, it modifies parts of the other to get the next frame. When modification is complete, it switches the frame being displayed, and carries out the necessary modifications to the other one.
you can hardly see? Yes, by including the following code after the Do loop just displayed, in the same cell:

```mathematica
SelectionMove[SelectedNotebook[], All, GeneratedCell];
FrontEndTokenExecute["SelectionCloseAllGroups"];
SelectionAnimate[SelectedNotebook[], AnimationDisplayTime -> 0.1]
```

The first line selects all cells generated by evaluating the current cell, just as double-clicking that bracket does. **Caution:** You must ensure that the current cell generates no other output! The second line collapses the selected group of cells, and the third displays them in rapid succession. The option is needed to slow down the animation so that you can see it.4 You can vary its value to control the speed.

The *twoDG* package contains the definition *animate* := {...}, where ... stands for the three code lines just displayed. If you use the package, you can animate a list of figures simply by placing the code line *animate* in the same cell immediately after the code that generates the list.5

This display isn’t very informative. To enhance it by including the polygon that joins the foci to the moving point, replace the *L* assignment with

```mathematica
L = {AbsolutePointSize[3], AbsoluteThickness[0.5],
    Point[F], Point[P[k*dt]], Point[-F],
    Line[{-F, P[k*dt], F}]}
```

This animation, with twirling focal radii, is somewhat more memorable. The dashing effect was eliminated because it would be distracting. The figures still don’t show the full path. You have to imagine previous positions of *P* as it goes around. And further enhancement might distract you and inhibit that image memory. You can help prevent that by showing all the points from the beginning up to the current position. In the *L* assignment just displayed, replace Point[P[k*dt]] by

```mathematica
Table[Point[P[j*dt]], {j, 0, k}]`
```

---

4 This is required even though 0.1 is the default value. That’s probably a *Mathematica* bug.

5 Front-end programming is really beyond the scope of this book. The documentation is so poor that it’s really an experimental activity. For example, the author couldn’t code *animate* as a function with an argument to specify the *AnimationDisplayTime* option. That’s possibly because the relationship isn’t clear between *SelectionMove* and the location of the code being evaluated. Use the *Mathematica* help browser and see Glynn and Gray (1999, part 14) for more information. (Gray invented front-end programming.)
You might want to pause to reflect about the completed figure, then repeat for a better look. It’s hard to stop the animation at a particular frame by clicking. Here’s a makeshift method. Once you’ve seen the previous animation, stop it anywhere. Double-click the second innermost cell bracket—the one with the arrow at the bottom. That expands the “stack” of figures. Scroll to the last one, for \( k = 50 \), where you want to pause. Click its innermost bracket, then right click for its properties. Click Option Inspector, then the + beside Graphics Options, then the + beside Animation. Click on the value of the AnimationDisplayTime option, and change it from the default 0.1 second to 5 seconds. Click on Apply then exit from this dialog box. Recollapse the stack of figures, then double-click to automate. You’ll see the pause, as desired.

Can you program the front end to take care of this detail, too? Yes, insert the following code lines in the same cell immediately after the code that generates the list of figures to be animated:

```mathematica
SelectionMove[SelectedNotebook[], After, GeneratedCell];
SelectionMove[SelectedNotebook[], Previous, Cell];
SetOptions[NotebookSelection[SelectedNotebook[]],
  AnimationDisplayTime -> 5];
SelectionMove[SelectedNotebook[], All, CellGroup];
FrontEndTokenExecute["SelectionCloseAllGroups"];
FrontEndTokenExecute["SelectionAnimate"]
```

The first line is equivalent to placing the front-end insertion point immediately after all the output cells. The next two select the previous cell—the last frame of the animation—and change one of its options. The fourth selects the encompassing cell group, and the next two close and animate it.

Although somewhat richer in detail than the code in the `animate` definition in `twoDG`, these lines seem logically no more complex. Nevertheless, the author was unable to assemble them into a feature that you could use by including a single line in your animation program.  

---

6 See the previous footnote.
Exercises

1. Fix the code for figures 1 and 2 so that the latter displays no axes, uses aspect ratio 1, and the same `PlotRange` value throughout.

2. Use superposition to draw a figure from an earlier chapter of the book.

3. Display a neat graphics array corresponding to the implicit plots in figure 11.4.10, with `dv = 0.02`, for `n = -3` to `3`. Make a corresponding animation with smaller `dv` and larger `n`.

4. Animate a sequence of epicycles such as figure 11.4.7 for a range of `b` values between 0 and 2. Why aren’t negative `b` values interesting?

5. In the last version of the example under the heading Animation, replace the graphics element `Table[Point[P[j*dt]],{j,0,k}]` with `Point[P[k*dt]]` and the portion of the elliptic curve corresponding to `t` values from 0 to `k*dt`.

6. Animate the process of drawing the graph of the example in figure 11.4.7. Each frame should show point `P` and the portion of the epicycloid between the starting point `<x, y> = <0, 1>`.

7. Use animation to verify visually that the angle bisectors theorem holds for a wide variety of cases, by constructing and animating a list of figures like figure 11.8.2, with `A` varying smoothly along a semicircle with center at the origin that passes through the position shown in the figure. **Suggestion:** Use the code in section 11.9.7.

---

7 Demonstrations of this sort are wonderfully easy with Geometer’s sketchpad software (Jackiw 1999). You select points such as `A, B, e` in figure 11.8.2 by pointing and clicking, and construct the bisectors as in section 11.8 using interactive tools modeled on drawing programs. Then you can drag `A` around with your mouse. All constructed elements dependent on `A` move with it. You observe that wherever you drop `A`, the bisectors remain concurrent. You can’t do that with Mathematica, but exercises 7–8 come as close as possible.