Why Graphic Elements?

A few years ago, during one of those lulls between projects which every so often overtake an independent software developer, I started work on a “sprite system” for the Macintosh. I had spent the past several years mostly working on business-oriented, database-intensive software. I wanted to see for myself how much graphics performance I could squeeze out of the then-new generation of Macs, as compared to the 68000-based, black-and-white only models for which I had targeted most of my previous software.

After two or three iterations of the development cycle, including many hours of profiling, I had a sprite system with which I was fairly satisfied. It would put multiple animated pictures on the screen, in between a foreground and a background. Performance was satisfactory, and compatibility was excellent. Then I started using it in “real” application programming. And the number of ugly hacks began to grow.

Here I needed a line of text in a sprite “world”, but it should be real text, not just a bitmap. No problem — I'd just change the sprite data structure to contain a function pointer to a draw routine, and add an accessor or two to the library so that the draw routine could find all the data it needed.

Here I needed a graphic which was almost, but not quite, like a sprite over a background. Actually, it was like 50 slightly different sprites, any one of which might be “on” or “off” or “somewhere in between” at any given moment. Another hacked draw routine, and a modification to the sprite loader routine to assure that the PICT shared by these sprites as a source graphic would not be loaded 50 times.

Here I needed an interactive set of “cassette-deck” buttons. No problem — just put the “out” positions of the buttons into the background, make sprites of the “in” positions, and write a routine to handle mouse clicks inside the screen rectangle which bounds the button graphics.

The exceptions just kept coming. Each of them was easy enough to handle, using something that was not quite exactly a sprite. But the source code kept getting messier and messier, and clarity of function was becoming a thing of the past.

It was then I realized that what I had was a paradigm problem. By thinking of the onscreen graphics as sprites, I was automatically limiting my concept of these graphics. Sprites were things put on the screen by an application program, and jerked around like puppets according to keypresses and mouseclicks under the

command of this program. It was time to reconsider the essential nature of onscreen computer graphics.

Every computer graphic, from a line of text to an element of a complex animation, is part of a communication between the application program and the user of that program. Every computer graphic has a life cycle — at some point during the operation of the application program, it is created; at some time after its creation, it is deleted.

At any moment during its life cycle, a graphic may be visible or invisible. When it is visible, it has a position on the screen in three dimensions (horizontal, vertical, and "in front of" or "behind" other graphics) where it must be able to be drawn at any time. While a computer graphic is shown on the screen, it can move or change in appearance due to four possible causes.

A graphic can always be altered by a direct action of the application program. But in the vast majority of instances, changes in the position or appearance of an onscreen graphic are caused by the passage of time (animation), by interaction with another onscreen graphic (collision), or by the actions of the user (selection and manipulation with the mouse). All of these causes are, at least conceptually, somewhat separate from the action of the application program.

The more I thought about it, the more it became apparent that I could formalize and abstract all of these causes of change, and define an onscreen graphical object in such a way that any graphic, regardless of whence it came or how it was rendered, could have the inherent ability to respond directly to all of these possible causes of change.

These considerations led to the birth of Graphic Elements. At a minimum, a Graphic Element is a data structure that knows how to draw itself. In itself, this requirement permits all rendering to be controlled by a separate software entity, the display controller, which is responsible for keeping track of what needs to be drawn at any moment and for calling the rendering procedures of the individual Graphic Elements as needed.

Any individual Graphic Element may also have the ability to deal with all of the possible causes of change: an "autochange procedure" which is called periodically to react to the passage of time, a "collision procedure" to react to interaction with other onscreen graphics, a "tracking procedure" and an "action procedure" to interact directly with the user. The application program, of course, retains the ability to retrieve any Graphic Element and act upon it explicitly.

Two great advantages of this architecture immediately became apparent. First, it isolates the application program from the details of the graphics code. For most

purposes — show/hide, move, etc. — the main line of the application code can treat any graphic just like any other graphic. This leads to a “natural” isolation of graphics-related programming into separate source

modules, interfaced to the application-level program only through their initialization functions, the four-character IDs of the graphics they describe, and whatever accessor functions are needed for reading and changing the states of these graphics.

The second great advantage of the Graphic Elements architecture is that it isolates both application-level and graphics-level code from the implementation details of any particular operating system. In itself, the Graphic Elements system makes only two assumptions about the graphics capabilities of a system: that it can set up an area of memory into which individual Graphic Elements can draw just as though they were drawing to the screen, and that it can then copy relevant portions of this memory to the screen. This isolates all system-specific code to the rendering procedure of the individual Graphic Element, and opens the door to high-performance graphics code that is platform-independent at the source level.

The addition of a library of standard Graphic Elements led to a third advantage: the ability to handle many common graphics and animation tasks in a few lines of application code. For example, the white puff of smoke from the barrel of a gun can be loaded, moved into place, and set up for the correct type of animation in only six lines of code. From then on, it will automatically "do the right thing" whenever it is shown. A standard button, switch, or slider can be set up and connected to its action procedure in only two lines of code. From then on, its action procedure will be called automatically as the user manipulates it with the mouse.

Since its first implementation, the Graphic Elements system has been used to write thousands of lines of application code. Much of this code has used Graphic Elements as a sprite system, and it still fills this role very well, often with a code savings of 30-50% (based on lines of code) compared to an ordinary sprite system.

But the most interesting Graphic Elements are those which go beyond the capabilities of most sprite systems. For example, one Graphic Element acts as a realtime highlight for the notes in a musical score, blinking on and off behind the notes as they are played. Another provides a versatile and automatic onscreen representation for the crossword-puzzle data structure in a spelling activity. A third provides the ability to pick up and manipulate other Graphic Elements.

In a sense, Graphic Elements is an inversion of the usual relationship between a sprite system and the other graphics services provided by a computer's operating system. In the usual application program environment, the sprite system is just one of many forms of graphics indifferently supported by the operating system. In

Graphic Elements, all forms of graphics are — or can be — “adopted” by the sprite system, actually

becoming sprite-like entities. Like all worthwhile paradigm shifts, this inversion has given results which go beyond those which were expected, both quantitatively and qualitatively.

Quantitatively, Graphic Elements yields an immediate and substantial savings in the number of lines of application code required to implement many common forms of animated computer graphics. Qualitatively, Graphic Elements makes easy many kinds of onscreen graphic effects which were difficult to achieve using earlier programming techniques.