# Graphic Elements: Designing a High-Performance, Highly General Graphics Subsystem

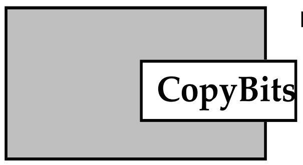
**Al Evans**

*Graphic Elements is a highly general graphics subsystem which offers performance on a par with dedicated "sprite" systems. A Graphic Element is an abstract entity that knows how and where to draw itself on a computer display, and may (or may not) know how to respond to all possible causes of change in its appearance: the passage of time, contact with another Graphic Element, and direct action by the user. This paper discusses the design goals set for Graphic Elements, the design decisions made on the basis of those goals, and the insights gained by the author in the process of implementing those decisions in the real world of current microcomputers.*

## 1 — Introduction

During the first fifteen years of the history of microcomputers, all systems for doing high-performance graphics were "compromised" in one way or another. On the Apple ][, odd and even pixels were of different colors. On the Apple ///, you could accomplish limited animation by dynamically changing character sets. On the early PCs, with their plethora of video cards and modes, the bus was so slow that it was often better to keep a copy of the screen in RAM, compare the new pixels to the old, and send out only the ones that had actually changed.

```
PROCEDURE CopyBits(srcBits, dstBits: BitMap; srcRect, dstRect: Rect;
                   mode: INTEGER; maskRgn: RgnHandle);
```



**CopyBits Made Graphics Programming FUN!!**

In 1983 and 1984, Apple's Lisa, and later the Macintosh, offered the first glimmer of hope. From the viewpoint of the application program, their video buffers looked like arrays of contiguous pixels, and they came with a fast built-in CopyBits procedure which was guaranteed to work the same on every machine. Today, these features seem trivial — but back then, they were victories worth celebrating.

And we did celebrate — with MacPaint and MacDraw, with many changes of font, size, and style, with PageMaker, Dark Castle, Déja Vu, and my own Cap'n Magneto.

But there were still severe limitations. Pixels could be either black or white. Although CopyBits made it easy to use offscreen bitmaps, there was practically no space to store an offscreen image — only 80K of heap space in the original Macintosh. Even writing directly to screen memory was slowed by the necessity of sharing access with the video-display hardware. The first generation of color Macintoshes did not improve the situation greatly — although we now had 256 colors, lack of memory and slow bus speeds limited us to the same kinds of approaches we had used on the older black-and-white machines.

Finally, in 1991, there was the Macintosh //si. It shipped at a reasonable price — less than I spent for my first Apple ][ — with 5 megabytes of RAM and an 80-meg drive. Although its video RAM was not fast, at least its speed wasn't limited by being out on a slow bus. To me, this was an indication that Apple was beginning to recognize the importance of fast video displays, and that the use of high-performance interactive graphics would soon become a day-to-day reality for large numbers of people.

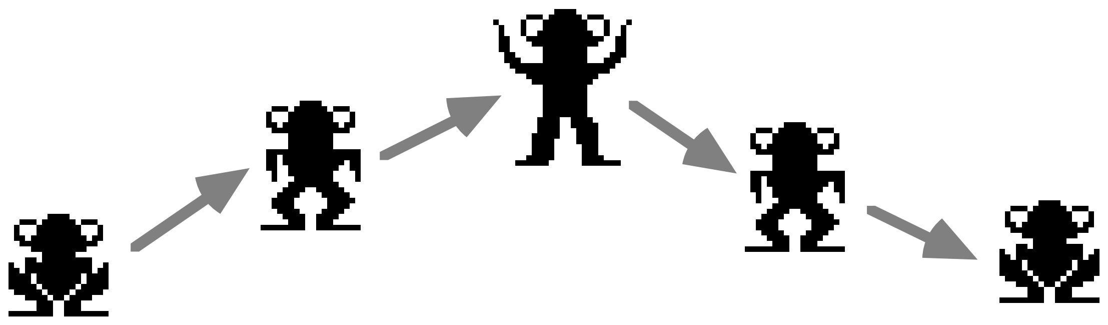
There are two basic kinds of high-performance interactive graphics engines. The first kind is like a camera and projector. The "camera" constructs each frame, and the "projector" transfers it to the monitor screen. All 3D modelling and "walkaround" engines, everything which is "Doom-like", is of this kind. In these engines, there is an implicit assumption that everything changes on every frame. Thus there is little to be gained by keeping track of exactly which pixels will change in the next frame, and efforts to enhance performance must concentrate on speeding up the modeler and renderer which construct the frames and the blitter which transfers them to the screen.

The second kind of graphics engine is more like a machine for cel animation. In cel animation, there is a background which normally remains stationary, and one or more layers of "cels" containing individual graphics which are overlaid on the background to construct each frame. Animation is obtained by changing the image on and/or the position of one of the cel layers. A computer engine performing this kind of animation benefits, of course, from fast routines for constructing its frames and transferring them to the screen. But in addition, it can and should take advantage of the fact that most pixels don't change from one frame to the next.

Graphic Elements began its life as a graphics engine of this second kind; in the world of microcomputers, normally called a "sprite engine". Through a process of repeated application to real problems and iterative redesign, it has evolved into a highly flexible tool for machine-human interaction, the best solution for a large class of problems in real-world computer graphics.

**2 — Design Objectives in Graphic Elements**

In this paper, I will pretend that the design of Graphic Elements sprang forth fully formed, as from the forehead of Zeus. Of course, this is far from the truth.



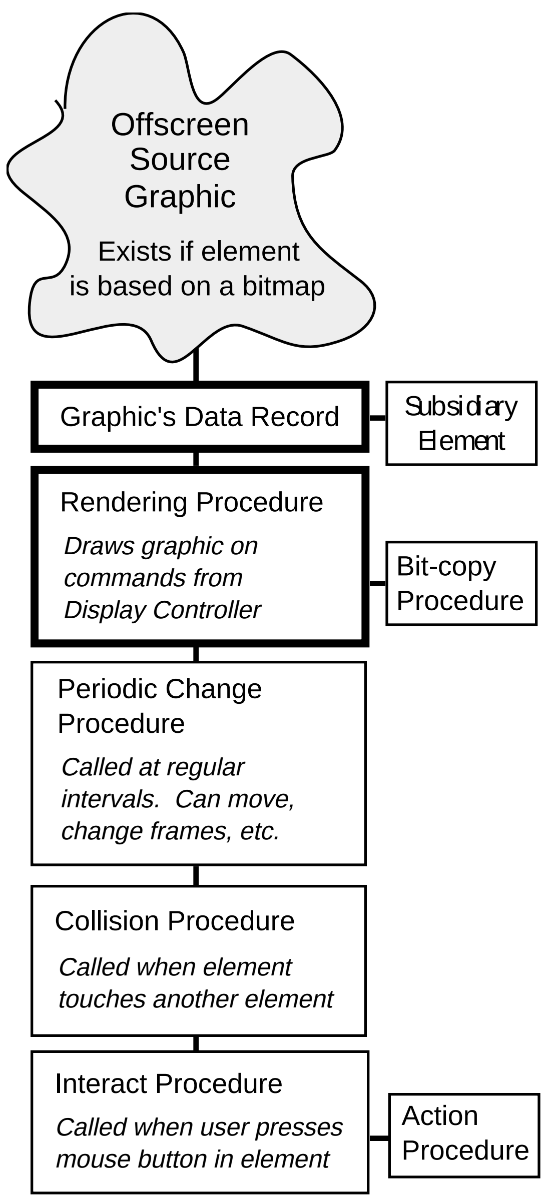
The truth is that I spent much of my limited spare time during 1992 testing animation techniques, using Ricardo Batista's "Color Sprite Manager" from one of the Apple Developer CDs as an initial test bed. My first modest objective was to get eight 32X32 color shapes moving and animating between a foreground and background on the //si. I was convinced that if I could attain this level of performance without "cheating" on compatibility and generality, machines capable of running a good general animation system would be widely available by the time I could actually finish designing and implementing such a system. By early 1993, I had accomplished this objective, and was ready to start working on "the real thing".

My spare time in 1993 was devoted to designing, testing, redesigning and retesting, and to the first use of my evolving system in actual application programming. As might be expected, this led to a complete redesign, from the top down and from the bottom up. This redesign led to the first version of Graphic Elements, and the API has remained very stable since late 1993, with only a few changes in parameters and types, and the addition of several calls and capabilities. GE is now approaching my ideal of a "perfect" graphics subsystem, a long-nurtured image of how an application program, a graphics subsystem, and a computer ought to interact in a perfect world.

In such a perfect graphics subsystem, the application program would tell an individual element of the screen display what to do, then step out of the way while the graphic executed its commands. For example, the application program might say "change frames and update your position every 30 ms; bounce according to this rule if you contact a wall; explode and disappear if you are touched by one of these." Then the application program would go off and do whatever else it needed to do, relying on the individual graphic to maintain the correct on-screen appearance.

Each individual graphic would know what it needed to know in order to play its part in the application: where it was at any given moment and how to draw itself, or any portion of itself that needed to be drawn. In addition, it would be capable of responding, by itself, to any or all of the possible causes for a change in its appearance: the passage of time, contact with another graphic, and direct action by the user.

In between the application program and the individual graphic would be a controller, sequencer, and event-distributor which would know nothing at all about the application program and very little about the individual graphics. This module would keep track of time and memory, would call upon the individual graphics to do


Offscreen Source Graphic
Exists if element is based on a bitmap
Graphic's Data Record
Subsidiary Element
Rendering Procedure
Draws graphic on commands from Display Controller
Bit-copy Procedure
Periodic Change Procedure
Called at regular intervals. Can move, change frames, etc.
Collision Procedure
Called when element touches another element
Interact Procedure
Called when user presses mouse button in element
Action Procedure

whatever needed to be done at a given moment, and would transfer completed frames to the video hardware. In addition, it would provide general services — access to individual graphics, movement routines, and visibility-control routines — to both the application and the individual graphics.

**3 — Design Decisions**

Based on the testing I had done, I began the implementation of Graphic Elements with a set of design decisions about its basic operating principles already in place. Although they are stated separately, all of these decisions are interdependent — a change in any one of them would likely lead to a change in the others.

Although it is a system of objects, I chose to write GE in C (and 68K assembly, where appropriate). The reasons for this decision were purely practical — C compilers were more highly-evolved than C++ compilers, and it is easy to link a C library to an application created with any compiler. The limited needs of the system for object subclassing were easy to handle explicitly, by the use of typedefs, custom creation routines, custom disposal routines, and function pointers.

The fact that Graphic Elements was intended to be a completely general system dictated several features of its design. For example, performance enhancements based on palette tricks or on requiring certain alignments or sizes of rectangles to be blitted could not be considered; neither could tricks like pixel-doubling blitters. Similarly, I considered it improper for the system to require that the user be in a certain graphics mode, or to place any restrictions on the size or location of the GrafPort it was drawing into. The system had to be able to deal with whatever graphic environment the application program handed it.

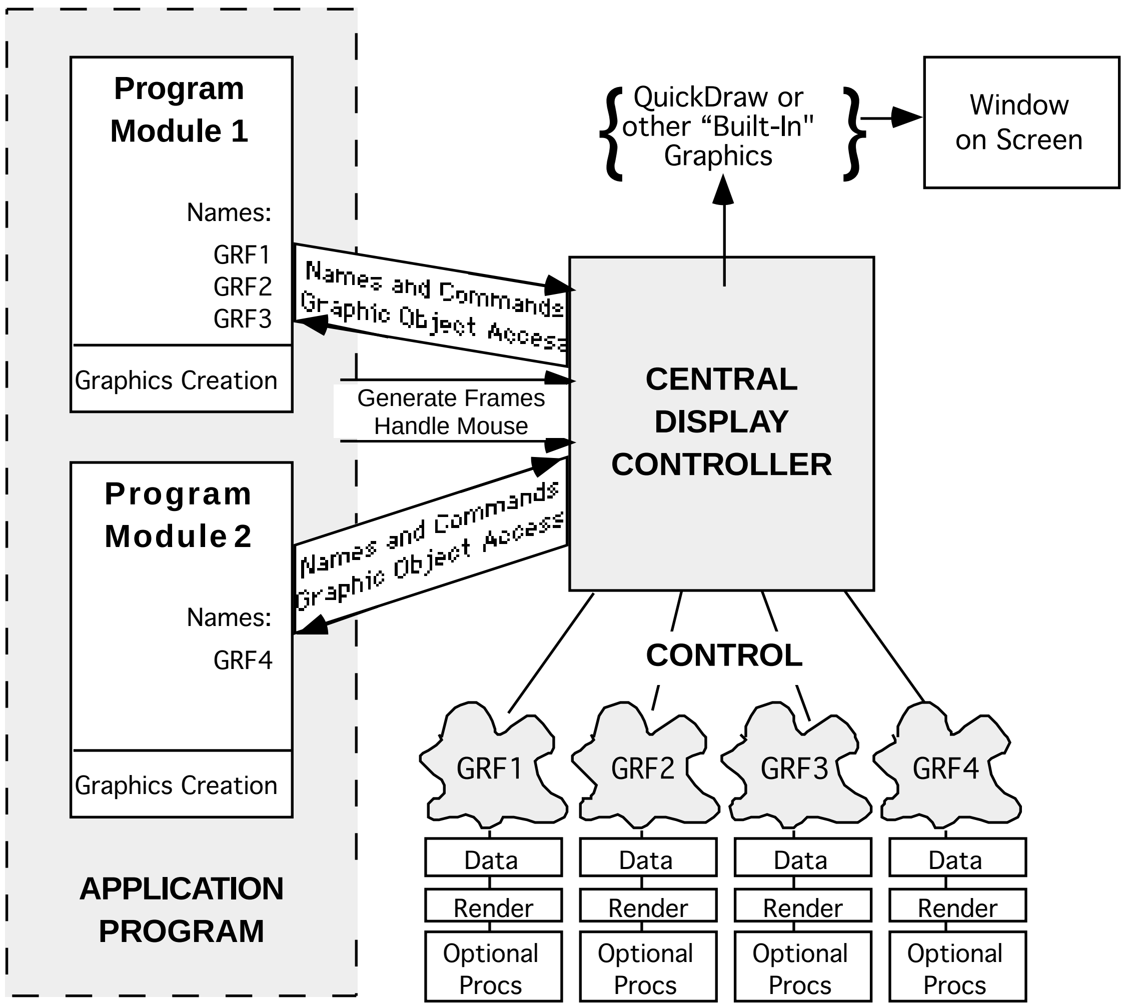
This intended generality, along with the extreme difficulty of debugging low-level, high-performance graphics routines from the level of the application, dictated the error-handling philosophy of the system. Stated simply, this philosophy is “when in doubt, do nothing.” If a routine has (or thinks it has) the data it needs to perform its function, it will do so. Otherwise, it will do nothing at all. This approach to error control works well in the context of a visual-display system[*], where most bugs are immediately obvious, and the ones that are not are extremely difficult to find.

*Offscreen*

In its initial incarnation, Graphic Elements would support only 8-bit graphics in its offscreen operations. This is a good compromise between versatility, on the one hand, and speed and memory consumption, on the other. In brief, the offscreen graphics data space required by an application using GE is usually between two and four times the on-screen area, perhaps more if animation is used extensively. For a 640 X 480 pixel screen, this is between 600K and 1.2 megs of offscreen memory. The use of 16- or 32-bit offscreen graphics would double or quadruple both the memory consumption and the time spent moving pixels from one place to another.

All graphics would be constructed offscreen, then blitted on-screen. This design decision simplifies other parts of the system, particularly the determination of

[*] This error-handling has been tested more than once, by the unintentional omission of some or all of the required graphics from application resources or data files. Except for the lack of graphics, such applications are completely functional.



which portions of individual graphics need to be drawn versus which parts of the final image must be transferred to the screen. It allows for the easy and "legal" use of custom blitters for the offscreen image. It eliminates the necessity of synchronizing to the vertical retrace of the monitor, which is tricky in terms of code and expensive in terms of time. And it insures that the "graphics memory" which is accessed the most is normal DRAM, and thus cachable.

This use of offscreen memory for all construction also makes it possible to rely on CopyBits() for all on-screen display. This brings automatic benefits which far outweigh any possible disadvantages — automatic support of all graphic modes, automatic support of multiple-monitor systems, and guaranteed compatibility with future Macintosh systems.

The "speed cost" of all this generality is difficult to quantify, and will vary from system to system. However, on the //si I was using when I first worked on Graphic Elements, I determined that if I did all my construction directly on the screen, the *maximum* possible time savings in the overall process was in the neighborhood of 17%. This number is likely to be much lower — perhaps even negative — on more modern machines.

**4— Some Solutions**

During the development of Graphic Elements, I spent weeks trying to find out what was actually happening, as compared to what I thought should happen based on the code I meant to write. The larger the system, and the more conceptually "asynchronous" it is, the more difficult and important it becomes to find out what it is really doing. I used both active and passive profiling, along with ad hoc techniques such as histograms showing the frequencies of times spent animating versus times spent doing other things in the main event loop. I also wrote small test applications to apply various extreme sets of conditions to the system, in order to see where it broke down.

All profilers tell you where your code is spending its time. "Active" profilers — such as the ones that come with Metrowerks' and Symantec's development environments — keep track of the number of times a routine is entered, and the amount of time between entry and exit.

"Passive" profilers — such as the one that comes with MPW — sample the program counter at regular intervals, and keep track of the number of times they have caught the processor executing in a given address range. After the test run, the output of the passive profiler is correlated with a link map and a ROM map for the computer used in testing, in order to assign routine names to these address ranges.

Active profilers can tell you precisely how many calls you make to a routine, and how long that routine takes to execute. However, code must be inserted, by the compiler, at the entry and exit of each routine being profiled. So they are useless for calls to ROM, to libraries, or to anything else for which you do not have source code. The sampling technique used by passive profilers overcomes this problem. If you have a link map of the library or a map of the ROM, they can give a very accurate picture of the relative amounts of time spent in various operations, even in ToolBox calls. But passive profiling provides no information on how many times a routine has been called. Each type of profiling provides data you can't get from the other.

When profilers don't provide enough detail, a simple Macintosh trick for precise timing of individual routine calls is to use the Time Manager. If you call PrimeTime() with a delay specified as a negative number, the Time Manager works in microseconds instead of milliseconds. Code like that at the bottom of the page can be used to get a precise indication of how long it takes to execute an individual routine.

It is important to be sure the machine is as idle as possible when testing your code's performance. File sharing should be off, no other applications should be running. There will always be some unknowable amount of "stolen" time included in your results, but it is reasonable to assume that this time is spread evenly through your code.

The many hours spent testing, rewriting, and retesting led me to several conclusions which had the quality of "revelations" — they seemed perfectly obvious, but only AFTER I reached them.

*Optimize Algorithms, Not Code*

Any high-performance graphics system is going to spend at least 90% of its total time moving bits from one place to another. The temptation is strong to spend the bulk of one's optimizing time working on low-level blitter code — and indeed, each machine cycle saved in the inner loop of a blitter can save thousands of cycles per frame.

Remember, though, that the absolute number of pixels which must be transferred increases as the square of the rectangle bounding those pixels. It is at least equally fruitful to concentrate on moving the minimum number of pixels possible. The "revelation" here was that there is only one time this number can be determined — immediately before the frame is generated.

In Graphic Elements, any change or movement merely adds to a list of "dirty" rectangles — rectangular areas of the image which must be redrawn because of the change. This list is kept sorted, and rectangles added to it are merged with rectangles aready on the list according to an

```
// 30 seconds in µsec for timer — never time out
#define thirtySeconds 30L * 1000 * 1000

TMTask      aTimeTask;

unsigned long msUsed;

// Init Timer
aTimeTask.tmAddr = nil;
aTimeTask.tmWakeUp = 0;
aTimeTask.tmReserved = 0;

...

// Start timer, call routine, stop timer
PrimeTime((QElemPtr) &aTimeTask, -thirtySeconds);
RoutineToBeTimed();
RmvTime((QElemPtr) &aTimeTask);

// Calculate number of milliseconds used
msUsed = (thirtySeconds + aTimeTask.tmCount) / 1000;
```

algorithm derived from the results of profiling. Immediately before frame-generation time, this list is compared against each object to determine what part of that object, if any, needs to be redrawn on this cycle. The list is reused to transfer only those portions of the "world" which have actually changed to the screen.

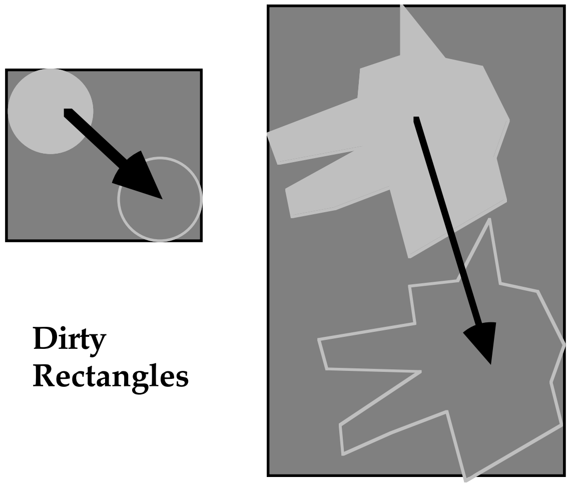
This way of doing things has a side-effect which has sometimes been useful: since no actual drawing is done when they are called, the basic routines which control visibility, movement, and frame-changing are all interrupt-safe. The Graphic Elements system supports this by actually keeping two lists of dirty rectangles, so that the application can continue making changes to one even while the other is used to generate a new frame.

*Don't Overlook the Small Things*

In looking over the output from my active profiling, I found several routines which used only fractions of a percent of the total time, but which were called tens or hundreds of times as often as the blitting code. These were essentially routines for doing rectangle arithmetic — unions, intersections, and offsetting.

The "revelation" here is that, even though such routines don't account for much time in the overall scheme of things, each cycle saved in one of them is worth tens or hundreds of cycles saved in, for example, the setup portion of a blitter.

For Graphic Elements, I wrote optimized assembly versions of all the rectangle-arithmetic routines for the



**Dirty Rectangles**

68K (and optimized C versions for the PowerMac). The savings would have been even greater had I been using the ToolBox versions of these operations, with their associated trap overhead.

*Memory Allocation Strategies*

It was apparent that several kinds of data entities in the system would need to be allocated and deallocated very rapidly and kept in lists, some of them ordered. As you can see from the paragraphs above, rectangles are a good example of such an entity. In all of my prior work with the Macintosh, I had left memory management up to the system. I knew when to lock handles and how to keep the heap from getting fragmented — but NewHandle() and NewPointer() had always sufficed for memory allocation in the software I worked on. Now I needed to allocate and deallocate hundreds or thousands of rectangles a second. I studied and tested various memory-management techniques, and decided that the best approach was to allocate fixed-sized records from a pre-allocated, pre-linked chunk.

This kind of memory management is perfect when you have relatively small numbers of same-sized memory objects that must be allocated and deallocated rapidly. It is also very simple; a sample implementation is given on the following pages.

The "revelation" here is that allocating a record in memory need only take two moves and a clear. If you need fast dynamic memory allocation, don't be afraid to do it yourself.

Graphic Elements makes extensive use of list-handling routines similar to those shown above, except that they are expanded to include the allocation and management of multiple "chunks" of list entries. These routines play roughly the same role as the use of overridden "new" and "delete" operators in an object-oriented language.

As I tested the system and began to use it in real applications, I ran into increasing numbers of situations where a graphic "grew" some piece of auxiliary data: a path record, a string, or a mask to be used in animation. The bookkeeping for these memory allocations began to get messy. I wanted the application program to be able to free all memory occupied by an object just by calling DisposeGrafElement() on it, or to free up a whole world of elements just by calling DisposeGEWorld(). But now, the application had to remember what spare parts it had attached to various Grephic Elements, and dispose of them as well.

I added fields to the element record to hold pointers to extra data for use in various parts of its operation, plus a field for a pointer to a "custom disposal" procedure. If it exists, this procedure will be called automatically during the disposal of the element. This mechanism plays the part of the "virtual destructor" in an object-oriented system.

The "revelation" here is that the best time to make provisions for the complete destruction of an object is at the time of its creation.

*The Paradigm-Glitch Test*

The most significant of these revelations, though, came

```
// Type definitions for managing lists of fixed-size records

typedef struct LMember *LMemberPtr;

typedef struct LMember {
   LMemberPtr     next;
   long           data;  // Dummy data field
} LMember;

typedef struct LHeader *LHeaderPtr;

typedef struct LHeader {
   LMemberPtr     listHead;
   LMemberPtr     free;
   short          entrySize;  // Actual size of data field
} LHeader;

// Code to implement list of fixed-size records

// Link entire list, starting at firstEntry
void ChainList(Ptr firstEntry, short entrySize, short nEntries)
{
   short    elemCount;
   Ptr      currElem, nextElem;

   currElem = firstEntry;
   for (elemCount = 1; elemCount < nEntries; elemCount++) {
      nextElem = currElem + entrySize;
      ((LMemberPtr) currElem)->next = (LMemberPtr) nextElem;
      currElem = nextElem;
   }
   ((LMemberPtr) currElem)->next = nil;
}

// Allocate memory for header and list
// Place all list members on “free” chain
LHeaderPtr InitList(short chunkSize, short entrySize)
{
   LHeaderPtr  hdr;

   hdr = (LHeaderPtr) NewPtrClear(chunkSize * entrySize + sizeof(LHeader));
   if (!hdr) return nil;
   hdr->entrySize = entrySize;
   hdr->listHead = nil;
   hdr->free = (LMemberPtr) (((Ptr) hdr) + sizeof(LHeader));
   ChainList((Ptr) hdr->free, entrySize, chunkSize);
   return hdr;
}
```

```
      // Return next free list entry
      LMemberPtr AllocateEntry(LHeaderPtr thisList)
      {
         LMemberPtr  freeEntry = thisList->free;

         if (freeEntry) {
            thisList->free = freeEntry->next;
            freeEntry->next = nil;
         }
         return (freeEntry);
      }

      // Delete entry "thisMember" and move it to free list
      void DeleteEntry(LHeaderPtr thisList, LMemberPtr thisMember)
      {
         LMemberPtr  aMember, prevMember = nil;

         if (thisList->listHead == thisMember)
            thisList->listHead = thisMember->next;
         else {
            aMember = thisList->listHead;
            while (aMember != thisMember) {
               prevMember = aMember;
               aMember = aMember->next;
            }
            prevMember->next = thisMember->next;
         }
         thisMember->next = thisList->free;
         thisList->free = thisMember;
      }
```

from one of those simple but perverse cases that seems to arise in any software system, no matter how well designed and implemented  Take, for example, the case of two small balls bouncing over a background. As I described above, during frame generation the rectangles made "dirty" by the movement of these balls will be compared to the rectangle of the background, to determine the minimum portion of the background which must be redrawn on this cycle.

But suppose one ball is in the upper left corner of the scene, and the other is in the lower right? Then, regardless of how small the balls themselves are, the "minimum portion" of the background to be redrawn will be the whole thing. I considered several possible solutions to this problem, ranging in quality from ungainly to downright ugly.

The "correct" answer was already there, latent in the system's design. First, for memory conservation, GE loads the actual bits of a graphic only once, regardless of how many elements use these bits. Second, in order to handle some common forms of animation, there is a facility for chaining groups of elements together so that they can be manipulated by the application program as a single entity. All I had to do was write a new kind of Graphic Element which would take height and width specifications for the rectangles of a grid, then create an appropriate number of elements from the same graphic and chain them together. In this way, the "invalidations" caused by the two balls would only affect each other when the balls were close together.

The revelation here was that if you design a software system well, there will be ways of using the system you never imagined when you were building it. Sufficient generality can be used to patch cracks in the paradigm.

The corollary is equally important: when you find a well-designed system, don't hesitate to imagine uses for it that the designers never intended. Two good examples in the Graphic Elements system are the "grabber" element and the QuickTime movie element*.

### *The Grabber Graphic Element*

One of the possible causes for a change in the appearance of a graphic on the display is the user's interaction with the graphic. The system supports this in a very

* Complete source code for both of these, as well as the rectangle-grid element described above, are included in the Graphic Elements release.

## API Overview — Managing GEWorlds

*A new GEWorld capable of holding Graphic Elements is created by the following display controller call:*

```
GEWorldPtr NewGEWorld(GrafPtr worldPort, Rect *worldRect,
                           Fixed scale, CTabHandle worldColors);
```

The `worldRect` and `scale` parameters determine the position and size of the GEWorld within `worldPort`. Briefly, if the world has a certain position and size in a GrafPort at `scaleOneToOne` (0x00010000), it will have the same relative position and size, in a GrafPort half the size of the original, at a scale of 0x00008000 (0.5 in fixed-point notation).

*After creating and populating a GEWorld, the application makes contact with the display controller at only a few points:*

```
void DoWorldUpdate(GEWorldPtr world, Boolean invalidate);
```

This is called every time through the main event loop (with `invalidate` false) and from the application's draw routine (with `invalidate` true) to keep the world's onscreen representation up to date.

```
Boolean MouseDownInSensor(GEWorldPtr world, Point gMousePt);
```

This is called from the "in content" part of the application's mouse click handler. It returns true if an element in the world has handled the click.

*When it is completely finished with a GEWorld, the application program calls:*

```
void DisposeGEWorld(GEWorldPtr world);
```

This destroys a world and everything in it, freeing all memory. Custom dispose procedures of individual elements are called as required.

general way: any element can be assigned a "sensor rectangle", a tracking procedure, and an action procedure. When the mouse button goes down, the application program calls MouseDownInSensor() from its event-handling code. The system looks through its list of registered active rectangles, and dispatches the event to the "topmost" element which handles mouse events in that location, if any. That element's tracking procedure then takes over for as long as the mouse button is pressed, and calls the element's action procedure as appropriate.

But GE itself makes no assumptions about what such a sensor-type element might do. To the system, it is just an element like any other. Because of this generality, it was possible — and even easy — to write a Graphic Element which has as its sole function the on-screen manipulation of other elements.

This "grabber" is, essentially, a sensor-type element which covers the entire "world" at the "topmost" level when it is active. When the user presses the mouse button and its tracking procedure is called, it searches for an element under the current mouse position. If it finds such an element, the grabber "captures" it and moves it to follow the mouse for as long as the button is held down. When the grabber has captured another element, its rendering procedure draws a rectangle around that element. Otherwise, it does nothing.

*QuickTime Movie as a "Sprite"*

Apple's QuickTime is an excellent example of a "well-designed system". Although its API is optimized to make it easy for the application programmer to play back digital-video movies, QuickTime and parts of QuickTime can be used for many purposes at a variety of different granularities.

When I heard that a then-future version of QuickTime would support "sprite-like" graphics in QuickTime movies, I accepted the challenge of adopting QuickTime movies into the Graphic Elements system. Because of the generality of the design paradigms of both systems, this proved to be almost trivial — a hundred lines of C code, including blanks and comments, with only two small hacks.

The first was due to the fact that QuickTime ignores the clip region of a GrafPort, so the visible region of the port was manipulated explicitly to get the fastest update times possible (calling SetMovieDisplayClipRegion() was much slower). This is necessary because, as a Graphic Element, the movie may well have to be drawn more often than once per frame.

The second hack was required in order to be certain that changes in the screen position of the movie were passed on to QuickTime. The current top-left point of the movie element's location is saved in an extra field in its record, and the element's autochange procedure compares its current location to this field on each iteration. If it has moved, the autochange procedure calls SetMovieBox() for the new location and updates the field.

The result is a fully functional Graphic Element that is also a fully functional QuickTime movie, with only the inevitable constraints due to the fact that it is being "displayed" in an 8-bit graphics environment and the fact that it must be copied one extra time to reach the screen.

**5— The Animation Cycle**

After all the elements used by an application program have been initialized, the system connects to the main line of the application's code at only three points. When the user clicks the mouse in a window containing a GEWorld, the application calls MouseDownInSensor() to allow active elements in that world to handle the mouse click. As described above, MouseDownInSensor() dispatches the mouse click to the topmost element having a sensor rectangle which contains the mouse point.

Everything else except incrementing the world's timer (which is done from an interrupt-level task) happens when the application program calls DoWorldUpdate(). The program calls this function to redraw the whole world in response to update events, and "as often as possible" — normally once each time through its event loop — to allow the GE system to handle time-based and graphics tasks.

DoWorldUpdate() begins by performing a "reality check" — unless the world has been invalidated, or the world is active and the last update was more than a millisecond ago, it just returns. This is done to avoid useless processing — the minimum time granularity in Graphic Elements is 4ms, and the minimum time interval that is actually "useful" in computer graphics is around 15ms.

If these tests are passed and the world is not being drawn due to an update event, each element which has an "AutoChange" procedure (and for which the interval between changes has passed) is called to perform its periodic action. Of course, these actions can result in collisions, which can result in other actions, and so on. But as mentioned above, the only actual effects of these actions at this point are changes to the elements' internal states and changes in a list of rectangles. So this AutoChange processing is relatively fast.

After processing changes due to the passage of time, DoWorldUpdate() checks the world's timer again, to see whether it is time to generate another frame onscreen. Each world can have its own "minimum time between frames" value.

If it is time to generate a new frame, DoWorldUpdate() swaps the world's two rectangle lists, so that possible changes generated by interrupt tasks can continue while it is drawing. Then it intersects each rectangle on the world's "redraw" list with the "screen" rectangle of each

## API Overview — Universal Services

*The display controller provides services to the application, to the individual elements, or to any other software entity with access to a GEWorld pointer. These are services that deal with elements in their global context, as parts of a world, and can be used with any type of element, regardless of how it is constructed or rendered, or of what it does.*

```
GrafElPtr FindElementByID(GEWorldPtr world, OSType objectID);
```

Although elements are generally managed by reference to their IDs, this function returns a pointer for direct access to an element's fields.

```
void ShowElement(GEWorldPtr world, OSType elementID, Boolean showIt);
```

Basic visibility control.

```
void MoveElement(GEWorldPtr world, OSType elementID, short dh, short dv);

void MoveElementTo(GEWorldPtr world, OSType elementID, short h, short v);

void PtrMoveElement(GEWorldPtr world, GrafElPtr element, short dh, short dv);

void PtrMoveElementTo(GEWorldPtr world, GrafElPtr element,
                                       short h, short v, Boolean doScale);
```

Basic movement. The "Ptr" versions are for cases, such as an element's AutoChange procedure, where a pointer to the element is already available. The `doScale` parameter in the last call determines whether the `h` and `v` parameters are scaled to the world (the normal case), or whether they are interpreted as absolute pixel positions (sometimes useful).

```
void SetElementPlane(GEWorldPtr world, OSType elementID, short newPlane);
```

Set element's front-to-back position.

**API Overview — Setting Up the Action**

*The display controller provides services used to set up an individual Graphic Element to respond to the possible causes for a change in its appearance:*

```
void SetAutoChange(GEWorldPtr world, OSType elementID,
        AutoChangeProc changeProc, Ptr changeData, short changeIntrvl);
```

Set up element for periodic action.

```
void SetCollision(GEWorldPtr world, OSType elementID,
                    CollisionProc collideProc, short collidePlane);
```

Set up element to react to contact with other elements.

```
void AddSensorToList(GEWorldPtr world, OSType id, Rect *sensorRect);

void RemoveSensorFromList(GEWorldPtr world, OSType id);

void SetSensorAction(GEWorldPtr world, OSType sensorID,
                                       SensorAction newAction);
```

Set up element to react to mouse clicks.

```
void SetCleanupProc(GEWorldPtr world, OSType elementID,
                                       CleanupProc elemCleanup);
```

Set a function which will be called when the element is destroyed.

element in the world. The result — or rather, the union of all the results — is stored in each element's "draw rectangle". At the end of this operation, the system knows exactly what portions of all elements must be drawn in order to refresh the display for each rectangle on the list.

DoWorldUpdate() then calls the rendering procedure of each of these elements, in order from back to front. The graphics environment has been set so that each element will draw itself into the world's offscreen "stage" GWorld.

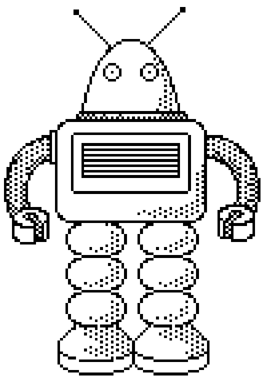
Finally, DoWorldUpdate() copies each rectangle from the list from offscreen to onscreen, resulting in an updated frame on the display.

**6— Conclusion**

Graphic Elements began as a system to cover 90% of all needs for high-performance computer graphics, without requiring any special graphics knowledge on the part of the application programmer. On one hand, this results in limitations — the system will probably be slower, or less memory-efficient, that a graphics system designed specifically for a given application. On the other, the generality of Graphic Elements leads to new possibilities.

For example, one of the most interesting elements created so far represents the musical score in a music education program. Each score page is constructed offscreen from a display list, shown onscreen, and the notes on it are highlighted as the music is played via MIDI or digital sound. The score is interactive, and allows the user to set a selection on the page or across multiple pages with the mouse. This kind of functionality is easy to obtain with Graphic Elements, and very difficult with a simple "sprite system".

This generality, in particular the clear distinction between the display controller and the individual graphics, also makes Graphic Elements easy to adapt to new environments. GE worlds are easy to "wrap" in screen savers or in the views or panes of any application framework. Further, GE's paradigm is meant to be easy to implement in any OS meeting certain minimal requirements. It has already been ported to Windows 95 and BeOS, and source code written using the standard elements provided with Graphic Elements will compile and execute the same way in all three systems.



## API Overview — A Typical Graphic Element

*The simplest Graphic Elementcan be created with a single call from the application. It can then be completely "forgotten" by the application, if it is something like a background or piece of scenery.*

*A more typical and interesting element type is the frame-sequence graphic (FSG), a standard animation based on two or more PICTs, considered as a series of frames. Here is the API for the standard frame-sequence graphic included in the system:*

```
GrafElPtr NewAnimatedGraphic(GEWorldPtr world, OSType id, short plane,
        short resNum, short mode, short xPos, short yPos, short nFrames);
```

Create a new frame-sequence graphic from a series of `nFrames` PICT resources starting at `resNum`.

```
typedef enum { singleframe=0,
               reciprocating,
               loop,
               oneshotvanish,
               oneshotstop,
               oneshotloop} AnimSequence;
```

The types of animation that can be applied automatically to the standard FSG. "One shot" animations are very common in actual applications, and the last three types cover their variations: disappear at end of animation, stop at end of animation, go from beginning to end to beginning and stop in original position.

```
void AnimateGraphic(GEWorldPtr world, OSType elementID,
                           short interval, AnimSequence sequence);
```

The application calls `AnimateGraphic()` to start one of the built-in animation sequences. The sequence then proceeds automatically.

```
void SetAnimDirection(GEWorldPtr world, OSType elementID, Boolean forward);
```

FSGs can be run from frame 1 to frame nFrames, or from frame nFrames down to frame 1.

```
void SetMirroring(GEWorldPtr world, OSType elementID,
                               Boolean mirrorH, Boolean mirrorV);
```

FSGs can be mirrored left-for-right and/or top-for-bottom.