History

Graphics were DIFFICULT the first 15 years

A// -- odd and even pixels different colors

A/// -- Animation by changing character sets

Lisa and Mac offered first easy graphics programming

Pixels in rectangular arrays

CopyBits() easy and fun

First break from graphics MODES

MacPaint->MacWrite; leads to desktop publishing

Still major limitations, speed and memory

80K free heap in original Mac

Early color Macs limited by memory, speed, and bus bandwidth

1991, Mac //si

Inexpensive

Plenty of memory

Video RAM not fast, but faster

High performance graphics would soon become day-to-day reality

Kinds of graphics engines

Camera/Projector

3D modelling, walkaround, Doom

Everything changes on each frame

Cell animation

Background normally remains stationary

One or more layers containing moving/changing graphics

Most pixels don't change from frame to frame

Graphic Elements is this kind

Normally called a “sprite engine”

Design Objectives

First objective: Get 8 32X32 sprites, foreground and background

No cheating on generality or compatibility

Used Ricardo Batista's Color Sprite Manager as initial testbed

Spare-time project, accomplished by early 1993

1993: Redesigning, retesting, first use in actual applications

Led to complete redesign, top-down and bottom-up

First version of Graphic Elements

Since then, the API has remained stable through wide variety of applications

My idea of a perfect graphics subsystem

Application program sets the graphic up to do what it needs to do, then ignores it.

Individual graphic type (class) knows what it needs to do to play its part in the application
Draw itself
Respond to any or all possible causes for change in appearance
Passage of Time
Contact with another graphic
Action by User
Controller - Coordinator - Event-distributor between application program and individual graphic
Knows nothing about application program
Knows very little about individual graphic types
Knows memory management
Tells individual graphics to do what needs to be done
Generates updated frames for display hardware
Provides general services to application and graphics
Access
Movement
Visibility

Design Decisions

All of these decisions are INTERDEPENDENT!

Chose to write in C (and 68K)

C compilers were better
C libraries were easily linked
Limited subclassing needs could be handled explicitly
But this will probably change in next version

Designed to be completely general, no special tricks for better performance

Special palette arrangements
Required alignments
Pixel-doubling
Graphics modes
Size or location of GrafPort
Had to accept whatever graphics environment the application set up

“When In Doubt, Do Nothing” error handling

If a routine thinks it has what it needs, it acts
Otherwise, it does nothing
Works well for visual-display system
Most bugs are immediately obvious
The rest are extremely difficult to find
Has been tested by omission of some or all graphics resources

8-Bit Offscreen Graphics

Compromise between versatility and speed/memory consumption
Offscreen space required usually 2-4X onscreen
Use of 16- or 32-bit graphics would double or quadruple this requirement and the time required for moving bits

All Graphics Constructed Offscreen, Blitted Onscreen

Simplifies determination of what needs to be updated
Allows for easy/legal use of custom blitters
Eliminates need to synch to vertical retrace
Ensures that graphic memory most often accessed is normal (cachable) DRAM

CopyBits() is Used for All Offscreen->Onscreen Copying

Automatic support of all graphics modes
Automatic support of multiple monitors
Guaranteed compatibility with future systems

Generality Has a Cost In Speed

Maximum possible savings going direct-to-screen was 17% on //si
Probably lower, or even negative, on newer machines

Meeting Design Parameters

What is Actually Happening?

Active Profiling
Tells how many times a routine is called, and how long it takes per call
Useless for ROM or library calls; must have source code
Passive Profiling
Tells how much time the processor spent in different locations in memory
Is correlated with link maps/ROM maps to determine how long it spent in different routines
Does not tell how many TIMES a routine has been called
Ad hoc Techniques
Histograms showing frequencies of times spent animating versus times spent doing other things
Test applications applying extreme conditions to see where the system breaks down
“Point” Timing using Time Manager
Testing leads to “Revelations” — Conclusions that seem obvious, once they are reached

Optimize Algorithms, Not Code

High-Performance Graphics Systems Spend At Least 90% of Their Time Moving Bits
The temptation is strong to spend the bulk of optimizing time working on low-level blitter code
But the absolute number of pixels to be transferred increases as the square of the bounding rectangle
Concentrate on moving the minimum number of pixels possible
This number can be determined ONLY immediately before a new frame is generated

GE Uses Dirty-Rectangle List

List of rectangles that need to be redrawn kept in scan order
Movement and changes just cause a new rectangle to be added list
List is compared against all objects before frame generation to determine what parts of what objects need redrawing
Rectangles on list are transferred to screen
Useful side-effect: object motion and change are interrupt-safe
GE keeps two lists, so that one can be used while screen updates continue

Don't Overlook the Small Things

Several Routines Used Small Amounts of Total Time, But Were Called Often
These routines can be overlooked in passive profiling
Each cycle saved is worth MANY cycles in a part of the code used less frequentlyt
GE: Optimized assembly/C for rectangle arithmetic routines
Don't forget trap overhead, where applicable!

Memory Allocation

Needed to allocate hundreds or thousands of rectangles per second
GE: allocate fixed-size records from pre-allocated, pre-linked chunk
Perfect for relatively small numbers of fixed-size objects that need to be allocated/deallocated rapidly
Allocation takes two moves and a clear

Creation and Destruction

Graphics grew extra bits of data
Application should be able to dispose of them without extra bookkeeping
Added pointer fields for extra data and function pointer for custom disposal procedure
The best time to make provisions for destruction is at time of creation
This is a good general practice, and helps a lot in writing clean code

Paradigm Glitches and Expansion

Example of Grid Element (demo)

Example of Grabber Element (demo)

Example of QT Movie (demo)

The GE Animation Cycle

After initialization, contact with application at three points

MouseDownInSensor() when user clicks mouse
DoWorldUpdate(true) for update events

DoWorldUpdate(false) “as often as possible”

On each call to DoWorldUpdate()

Is the world active and (invalid or more than 1 ms old)?

Give time to all elements that have requested it for periodic actions

Changes can lead to collisions, and thus other changes

But changes only affect rectangle lists and elements' internal states, so this processing is fast

Is it time to generate another frame onscreen?

Swap rectangle lists so that changes can continue while drawing

Intersect each rect on list with screen rect of every active element

Union of results becomes element's draw rect

After this operation, system knows exactly what parts of each element must be redrawn to refresh display

Call rendering procedure of each element with non-empty draw rect

Elements draw in order, from back to front

Graphics environment is set so that each element draws into offscreen "stage" world

Copy each rectangle on the list from offscreen to onscreen

Conclusion

Intended to cover 90% of all high-performance graphics needs without requiring specialized graphics-programming knowledge

Leads to limitations — probably slower or less memory-efficient than a special-purpose engine

But its generality leads to new possibilities

Graphics which would not be possible with the average “sprite system” (score page demo)

Ease of adaptation to different frameworks -- After Dark screensavers, MacApp, TCL, and PowerPlant views/panes

Ease of porting — Windows95, BeOS