IBM PC ANIMATION – CRUDE BUT EFFECTIVE

William T. Verts
COINS Department
University of Massachusetts
Amherst, MA 01003

ABSTRACT

Owners of IBM PC's (or equivalent) equipped with the primitive Color Graphics Adapter (CGA) have trouble in creating convincing animated effects. The CGA lacks hardware that allows double buffering. While double buffering is possible on the more advanced adapters, most graphics packages restricted to the CGA are forced to redraw each image directly on the display screen. This is extremely distracting when an animated effect is desired. Experiments with the motion of a vertex through a Delaunay Triangulation show that it is extremely difficult to determine which point is moving when the mesh must be redrawn on the screen after each change in position. This paper presents a mechanism for achieving relatively smooth animation on systems equipped only with the CGA. The technique simulates double buffering by treating an off-screen area of memory as the display area for graphics commands. To then "instantly" update the screen the entire off-screen memory area is copied into the area of memory reserved by the display adapter. Tests using a slow PC show that a sixteen kilobyte image frame can be copied to the screen memory in under one twelfth of a second, sufficient for the illusion of smooth motion.

INTRODUCTION TO THE CGA

Why Use the CGA?

The Color Graphics Adapter was the first and most primitive graphics card produced for the IBM Personal Computer (PC). Although more advanced adapters have been produced and have become popular in later years, many machines are still equipped only with the CGA. Similarly, there is a large software base requiring that an adapter have hardware compatibility with the CGA. Upgrading to a newer adapter (and its corresponding monitor) may be an expense people are not willing to pay. Owners of the IBM-PC Portable (the "luggable", not the lap-top) may also have difficulty in finding a replacement adapter that provides the composite video signal required by the internal monitor.

What Can the CGA Do?

The CGA supports two major text modes: 80 columns by 25 lines and 40 columns by 25 lines. Each character occupies one byte, and all characters are printable. Character definitions conform to standard ASCII for characters in the range 32..126 (the printable ASCII characters), with IBM-PC specific characters for the remainder of the 256 patterns. Associated with each character is an *attribute byte* that describes how the character is to be shown on the screen: characters may be one of 16 colors, on one of 8 background colors, and may blink.

The CGA also supports two graphics modes made up of a rectangular grid of *pixels* (picture elements, or spots of color on the screen), where each pixel may be one of a limited set of colors. The two modes are 320x200 pixels with four colors available per pixel, and 640x200 pixels with two colors per pixel. In addition, there is a little known "unofficial" mode that allows 160x100 pixels with sixteen colors per pixel, but this mode is accessible only by directly programming the video driver chip in the CGA. Note that 320x200 mode, for example, means that visible on the screen are 200 horizontal *raster lines* with 320 pixels per line.

Along with the special plug for the color monitor, the CGA card has an coaxial cable plug which provides a *composite video* signal of the image on the screen. Users could RF-modulate this video signal and use a standard NTSC television rather than being forced to purchase a special purpose monitor (the 25x40 text mode is present to compensate for the limited video bandwidth of most TV's). It is very easy to record the images produced by the CGA on a standard Video Cassette Recorder, as long as the VCR is capable of accepting composite video without the RF component. Composite video is not produced by the higher definition adapters such as the Enhanced Graphics Adapter (EGA).

CGA DISPLAY MODES

The *video memory* for the CGA is mapped onto the address space of the PC as a 16K block starting at absolute address `$B800:$0000` (16K is shorthand for 16 kilobytes, where a kilobyte is 1024, or 2^{10} bytes). Text and graphics are displayed by storing bytes into that 16K block, which the CGA continuously scans to form the video signals.

Text Modes

A CGA text screen occupies 4K bytes in 80 column mode and 2K bytes in 40 column mode. Half of those bytes contain the characters displayed and half contain the attribute information for each character. The 16K memory reserved for the CGA is partitioned into multiple pages (four 80 column and eight 40 column pages), any of which can be the *active page* (the page being written to) and any can be the *visible page* (shown on the screen). This technique provides a simple mechanism for "instantaneously" changing the screen by writing text into an invisible page, then instructing the CGA to make that page visible (as with the SCREEN instruction in Microsoft's Basic interpreter (IBM; 1982)). Short sequences of text-based animation (four or eight frames depending on the text mode) are created by writing images into each page and then in a loop making each page visible for a short time.

Graphics Modes

In graphics mode the entire 16K block of video memory is used by a single screen, which is always visible. The CGA lacks the memory and hardware needed to support more than one graphics page. There is no native hardware support for animation as there is in the EGA and the more advanced graphics adapters.

For the two native graphics modes the 16K video memory is partitioned into two 8K fields. The first 8K field contains the even numbered raster lines [0, 2, ..., 198] and the second 8K field contains the odd numbered raster lines [1, 3, ..., 199]. Each raster (one line of pixels) occupies 80 bytes, and the raster lines are consecutive within a field (there are no free bytes between rasters). There are 192 unused bytes at the end of each field.

In 640x200 graphics mode each byte represents eight pixels, one bit per pixel. Each pixel has two values: 0 corresponds to black (always the background color), and 1 corresponds to a single *foreground* color selected by the user. This foreground color is one of the sixteen native CGA colors.

In 320x200 graphics mode each byte represents four pixels, two bits per pixel. Each pixel has four values, where 0 corresponds to a user selected *background* color (again, one of the sixteen native CGA colors). Colors 1 through 3 are fixed depending on which of four *palettes* (numbered 0 through 3) has been selected. For example, in palette 0 the four colors are background, green, red, and brown. Although four colors can appear on the screen simultaneously, the user can not arbitrarily select which four are to be displayed.

Selecting the graphics mode and attributes of the screen (palette, background color, and foreground color, as required) are accomplished through calls to the operating system

and by directly setting the CGA hardware registers to the appropriate values (Hogan; 1988) (Crayne, Girard; 1985). Most high level languages for the PC such as Turbo Pascal (Borland International; 1987, 1988) supply procedures to simplify controlling the screen.

Snow

The CGA comes into conflict with the processor of the PC when they simultaneously access the same byte in video memory. This conflict shows up on the screen as flashes of brightly colored "snow" in text modes and in the 160x100 graphics mode (which the PC still "thinks" is a text mode). Snow can be eliminated under software control by waiting to store bytes into video memory until the horizontal or vertical video retrace interval. This technique significantly slows screen operations, but because it is a software technique it can be switched on and off as desired. Some new implementations of the CGA do not suffer from snow.

CGA ANIMATION

Borland International has included with recent versions of Turbo Pascal, Turbo C, Turbo BASIC and Turbo Prolog routines that provide device independent graphics support for a large number of graphics adapters (Borland International; 1987, 1988). Software support for animation is present if multiple screens are available in the hardware. There is of course no such support for the CGA.

Although DOS (the operating system) contains functions for manipulating pixels in the video memory, a graphics package can be written to modify "virtual pixels" in any 16K block of memory according to the storage rules listed earlier. A new image prepared in an off-screen *frame* becomes visible when it is copied to the video memory of the CGA.

Time and Space

"Active" animation frames reside in main memory data-structures, so copying one frame to another is strictly a memory-to-memory transfer. Tests on a 4.77Mhz PC/XT show that one frame can be copied to another in about one-twelfth of a second. Machines with a higher clock rate can copy frames even faster. In addition, the PC/AT can make most advantage of its 16-bit bus to increase transfer speed if two conditions are met: frames must be word aligned in memory (the base address is an even number), and the frame-copy code must transfer 16-bit words instead of 8-bit bytes.

Each frame occupies 16K bytes of memory. Main memory is restricted on the PC to 640K bytes, of which portions must be reserved for DOS, device drivers, main memory RAM-disks, any TSR programs (Terminate and Stay Resident, "pop-up" programs), and the animation program itself. Forty frames would completely fill the 640K bytes of main memory; the practical limit is much lower. A 640K system without TSR's or a RAM-disk can generally support a program requiring 22 frames, which is exactly what a 360K byte, 5 1/4 inch diskette can hold.

An image can be loaded into memory from a 360K floppy disk in about one second, and from a hard disk in about one-fifth of a second, depending on the quality of the disk drives.

Systems equipped with EMS (Expanded Memory) can allocate a very large chunk of EMS memory as a RAM-disk and keep the bulk of unused images stored there "off line". A copy from RAM-disk is orders of magnitude faster than pulling the images off of magnetic media.

PASCAL CODE EXAMPLES

Basic Techniques

In (Adams; 1988) a few off-screen CGA frames are located at fixed places in high memory. This technique limits the number of frames available to those defined at compile-time and permits conflicts to occur between image frames and code that may accidentally extend into the reserved areas. By dynamically allocating memory for frames from the Pascal heap instead, the number of off-screen frames available can be determined at run-time. All code examples that follow are written in Borland's Turbo Pascal and are compatible with version 3.0 or later.

Two type definitions are critical to the construction of animation frames: an array type of the correct size (16K) and a pointer type that points to objects of that array type. The Pascal code fragment below shows those type definitions and two variables of the pointer type.

```
Type     Screen          = Array [0..16383] Of Byte ;
         Screen_Pointer  = ^Screen ;

Var      A, B            : Screen_Pointer ;
```

Screens are allocated for variables `A` and `B` of type `Screen_Pointer` from the heap with standard Pascal procedure `New`. The expression `New(A)` allocates a 16K block from the heap and assigns the address of that block to `A` (in Turbo Pascal a function can be installed to return `Nil` if an allocation attempt fails, rather than causing the program to abort). Alternatively, the expression `A := Ptr($B800:$0000)` directly assigns to variable `A` the address of the CGA video memory. Copying screen `B` into screen `A` is accomplished by the simple assignment statement `A^ := B^`.

The Frame Handler

Although a single off-screen memory block is sufficient to perform all feats of graphics animation, many animation tasks are simplified by having two or more off-screen frames. The definition below is of an array of pointers to animation frames, and an index variable that indicates which of those frames is *active* (being issued graphics commands).

```
Var      Active_Table     : Array [0..40] Of
                                    Screen_Pointer ;
         Current_Active   : -1..40 ;
```

The `Active_Table` array can be made as large as necessary, although the limit of 40 insures that there are enough entries to completely fill the 640K system memory with frames. All entries in `Active_Table` are initialized to `Nil`. When a off-screen frame is *opened*, memory is allocated for it from the heap and its address is assigned to the proper entry in `Active_Table`. When a frame is *closed*, its memory is returned to the heap with Pascal procedure `Dispose`, and the corresponding entry in `Active_Table` is set back to `Nil`.

By definition, `Active_Table[0]` always refers to the visible screen; its value is either `$B800:$0000` (if opened and in graphics mode) or `Nil` (if closed and in text mode).

`Current_Active` is the index into `Active_Table` of the active frame. The active frame must be open (i.e., when `Current_Active` is greater than or equal to zero `Active_Table[Current_Active]` will not be `Nil`). `Current_Active` is initialized to –1, and otherwise equals –1 when no screen is active (several may be opened, but none will receive graphics).

Copying Frames

Any open screen may be copied into the currently active screen by the expression: `Active_Table[Current_Active]^ := Active_Table[Target]^`. Animation is accomplished by copying a screen into the visible screen (`Active_Table[0]`).

Clearing Frames

The active frame can be cleared with the expression: `FillChar(Active_Table[Current_Active]^, SizeOf(Screen), #0)` This expression uses Turbo Pascal routine `FillChar` to flood the 16K block with zeroes (`#0` is the character with ordinal value zero). Frames may be initialized to other values by changing the flood character.

Loading and Storing Frames

It is very useful to save images to and load images from disk. A code fragment for storing the image in the active frame is shown below; loading images from disk requires similar code. `File_Name` must contain a valid DOS file name.

```
Type     Frame_File       = File Of Screen ;

Var      File_Name        : String ;
         Outfile          : Frame_File ;

Assign   (Outfile, File_Name) ;
Rewrite  (Outfile) ;
Write    (Outfile, Active_Table[Current_Active]^) ;
Close    (Outfile) ;
```

Painting Pixels

The most important action that can be performed on a graphics screen is to set a particular pixel to a particular color. For the CGA, with each raster line occupying a contiguous group of bytes, this task is broken into three phases: determining the location of the correct raster line, locating the correct byte within that raster, and setting the correct pixel within that byte.

Assume that variables `X` and `Y` contain the coordinates of a visible point in the active frame, and that the pixel will be set to the value in `Color`. A code fragment to set the pixel to the desired color is given below. Variables `X`, `Y`, `Offset` And `Index` are of type `Integer`, variables `Color`, `Mask`, `Shift`, `Pixel` and `Image` are of type `Byte`.

The code is identical for both 320x200 mode and 640x200 mode except for the four lines marked with the `(*!*)` comment. Those lines each contain pairs of adjacent numbers where the rightmost of each pair is commented out with curly braces. The code is set up for 320x200 mode. To change to 640x200 mode, delete the curly braces and place them around the leftmost number of each pair.

```
(* Compute the offset of the correct raster *)
Offset  := (Y Div 2) * 80 ;
If Odd(Y) Then Offset := Offset + 8192 ;
(* Compute the offset of the correct byte *)
Offset  := Offset + (X Div 4 { 8 }) ;          (*!*)
(* Determine pixel position within byte *)
Index   := (X Mod 4 { 8 }) ;                   (*!*)
(* Invert Index and scale by bits per pixel *)
Shift   := (3 { 7 } - Index) * 2 { 1 } ;       (*!*)
(* Determine mask to modify correct pixel *)
Mask    := ($03 { $01 } SHL Shift) ;           (*!*)
(* Build colored pixel to go in new place *)
Pixel   := (Color SHL Shift) AND Mask ;
(* Get image byte from active screen *)
Image   := Active_Table[Current_Active]^[Offset] ;
(* Modify the pixel *)
Image   := Pixel OR (Image AND NOT Mask) ;
(* Replace image byte back into active screen *)
Active_Table[Current_Active]^[Offset] := Image ;
```

This code fragment is very inefficient. Execution speed can be improved by replacing the computations of `Offset` and `Mask` with table look-ups of precomputed values. Expressions containing `Div` and `Mod` by powers of two can be replaced with shifts. Many of the assignment statements can also be condensed into a few long statements (variable `Image` is not even necessary).

Drawing Lines

The Bresenham algorithm is the classic algorithm for painting lines between any two points. The algorithm works by stepping one pixel at a time from one endpoint of the line to the other, painting a spot of color at each pixel. This algorithm can be easily written in assembly language because it uses integers rather than real numbers to determine the deviation of the painted line from the true slope. A FORTRAN implementation of this algorithm appears in (Bowyer, Woodwark; 1983).

Using the Bresenham algorithm is inefficient when drawing horizontal lines on the CGA. A horizontal line routine can take advantage of the knowledge that each byte in an image frame contains several pixels from the same raster line, and that a horizontal line is contained in a contiguous group of bytes. The first and last bytes of the group (containing the left and right endpoints of the line) must be masked off so only pixels that are part of the line are changed. Bytes in-between the first and last bytes are completely part of the line, and those bytes all receive the same value: a byte where all pixels have the desired color.

USAGE AND EXAMPLES

This section presents several applications of the frame animation technique. All of the examples listed here have been implemented on a standard 4.77Mhz IBM PC/XT equipped with a CGA and an 8087 numeric coprocessor. Each example demonstrates one or more places where standard CGA graphics is inferior to the frame animation technique.

Real-Time #1: Removing Plotting Distractions

An experiment was conducted to observe what happens to a Delaunay Triangulation as one of the vertices moves through the plane. When the mesh is repainted directly on the screen between iterations, it is extremely difficult to visually isolate the moving point from those that are not moving. Repainting gives an illusion of motion to lines that have not changed positions between iterations. When the graphics are painted into an off-screen frame, then "instantly" copied to the visible screen, the motion of the

point becomes quite apparent. On a 4.77Mhz machine, painting the graphics is slower than generating the Delaunay Triangulation for up to 20 points, even using an $O(N^4)$ mesh generation algorithm.

Real-Time #2: Illusion of Motion

Three dimensional, near real-time rendering of simple molecular models can be performed on a PC. Atoms are represented by spheres, which have the characteristic that after scaling, rotation, and perspective transforms are applied a sphere still looks like a filled circle. The molecular model is rotated, then atoms in the model are sorted according to distance from the viewer. Filled circles are painted into the off-screen frame from the atom that is furthest away to the atom that is closest. The radii of the circles depend on the relative sizes of the corresponding atoms and on how far they are from the viewpoint. Atoms that are small or are far away tend to be obscured by large or nearby atoms.

When the image is completely rendered it is copied to the screen. The time between updates is dependent on the complexity of the molecular model and the sizes of the circles that must be drawn, but small molecules can be processed in just a few seconds. The short pause between complete frames is far less distracting than the "popcorn effect" of having the circles plotted directly on the visible screen for each iteration.

Real-Time #3: Time Constraints

A graphic simulation of an analog clock (one with hands) requires that the screen be updated once every second. For this task there need to be two off-screen frames. One frame contains the background image of the clock face, which does not change, and the other is a work screen needed to generate each new clock image. The background image is copied to the work frame, the hands are added in their new orientation, and the result is then copied to the visible screen. Despite the fact that the process requires two 16K frame copies, there is ample time to generate each new image within the one-second time limit.

Menus and Rubber-Banding

A mouse driven graphics paint program can take advantage of the animation facilities in several ways. The work image and the menu image are kept in separate frames so that the menu does not interfere with the drawing being generated. When the menu image is needed the work image is copied off to a temporary frame and the menu image is copied to the visible screen. When the menu action is complete, the work image is copied back to the screen.

Keeping an "Undo" screen is trivial. By saving the work image before each new operation is performed, the undo frame always contains the last valid image and is always available in case a mistake is made.

The animation facility is fast enough to support *rubber-banding*. When drawing lines, boxes or other objects it is useful to display the new object moving or changing in size according to the position of the mouse until the size and position of the object are satisfactory. This can be accomplished by saving the work image, then as the mouse moves the saved image is copied to a temporary area where the figure is added in its new orientation, and the result is then copied to the visible screen. When the correct position has been determined, the figure is added permanently to the image.

Preprocessed Frame Movies

When several images of a scene have been rendered where each is slightly different from its predecessor, the images can be copied into the visible screen area fast enough to create the illusion of smooth motion. A sequence of 20 images of the Earth, where in

each successive image the Earth has been spun by 18 degrees, will create the illusion of a smoothly spinning planet.

The motion will be a lot coarser if there are more animation images than will fit into main memory. For example, a 64 image sequence occupies one megabyte of memory and each frame must be loaded successively off of the hard disk. Loading from the hard disk is a lot slower than moving an image around in main memory.

Mandatory Off-Screen Graphics

If the PC has no CGA compatible graphics adapter, graphics images can still be produced in off-screen frames and saved for display on another machine.

CGA EXTENSIONS

A graphics package is minimally complete if it contains procedures for setting any pixel to any allowable color and for examining any pixel to see what color it contains.

Clearing the screen, drawing lines between any two points (clipping those lines to the screen if necessary) and copying chunks of an image from one place to another can be completely described in terms of setting and examining individual pixels, but these tasks can often be optimized by carefully considering how image memory is organized.

An animation graphics package must also be able to allocate and de-allocate frames, move images from frame to frame, load images from disk, store images onto disk, and make any of the frames the active frame for graphics commands.

In addition to the techniques presented here, there are several ways to extend the capabilities and usefulness of the graphics package.

Graphics Modes

Other than the three graphics modes that can be displayed on the CGA monitor (640x200 two color, 320x200 four color, and 160x100 sixteen color), graphics modes can be supported that cannot be shown on the CGA monitor. It is a simple intuitive step to realize that off-screen graphics techniques can be used for formats the CGA does not naturally support, so long as no attempt is made to directly display images in those formats. An "image processing" mode, for example, is 128x128 pixels with 256 colors per pixel (each pixel occupies one full byte). This format contains the largest square that will fit into a 16K byte CGA frame ($16384 = 128^2$). All graphics primitives can be executed on frames in this mode, except that frames may not be copied to the video memory. The other modes become very useful for previewing, as long as one is willing to accept restrictions on the number of colors available or the size of the visible area.

Circles and Ellipses

A routine that draws a circle is useful. An ellipse routine is also useful to have, but is a somewhat more complex algorithm than the circle. In addition to the Bresenham line routine, (Bowyer, Woodwark; 1983) also contains a simple circle drawing routine that uses only integers. Integer routines can be efficiently coded in machine language for high speed. An integer ellipse routine appears in (Van Aken; 1984). These routines paint only the outline of the circle or ellipse, but both can be modified to paint solid figures. The ellipse can be used in place of the circle routine by selecting identical values for the major and minor axes, and it can also be used to paint *visually correct* circles on screens that do not have a 1:1 aspect ratio.

Color Mixtures and Patterns

Startling color mixture effects can be achieved with little extra overhead by painting figures with patterns instead of with solid colors. Pseudo-colors can be created by mixing two or more colors in a checkerboard. This is particularly useful on the CGA because of the limited number of available colors.

It is very simple to define patterns as screen invariant so that the color of every pixel can be determined uniquely from its pattern, regardless of what objects are to be painted on the screen. Drawing lines or figures then "uncovers" the pattern in the shape of the drawn object. This technique results in strange visual effects: animation on a screen invariant background pattern more resembles object shaped windows moving over the pattern than patterned objects in motion. It is more complex to write an efficient pattern handler that is object relative than one that is screen relative.

CONCLUSIONS

The CGA (what an acquaintance refers to as the Brain Damaged Adapter), is indeed a primitive graphics standard. Owners of IBM PC's equipped with more advanced adapters, and those who have machines specifically designed for animation, may be amused at the thought of doing animation on the CGA. There are much better ways of performing animation than with the CGA, but when it is the only choice it can be teased into producing spectacular effects.

BIBLIOGRAPHY

Adams, L., 1988. High-Performance Graphics in C — Animation and Simulation, Windcrest Books (a division of TAB Books), Blue Ridge Summit, PA

Borland International, 1987. Turbo Pascal 4.0 Owner's Handbook, Borland International, Scotts Valley, CA

Borland International, 1988. Turbo Pascal 5.0 User's Guide, Borland International, Scotts Valley, CA

Borland International, 1988. Turbo Pascal 5.0 Reference Manual, Borland International, Scotts Valley, CA

Bowyer, A., Woodwark, J., 1983. A Programmer's Geometry, Butterworths, London

Crayne, C. A., Girard, D., 1985. The Serious Assembler, Baen Enterprises, New York, NY

Hogan, T., 1988. The Programmer's PC Sourcebook, Microsoft Press, Redmond, WA

IBM, 1982. BASIC, Reference Manual

Van Aken, J. R., 1984. An Efficient Ellipse-Drawing Algorithm: IEEE Computer Graphics and Applications, Volume 4 #9 (September 1984), pp. 24-35

TRADEMARKS