TECHNIQUES FOR
REAL-TIME FRAME BUFFER ANIMATION

Computer Graphics Laboratory
New York Institute of Technology
Old Westbury, New York 11568
Paul S. Heckbert
USA

Abstract

Techniques have been developed which allow animation sequences to be played back in real time using a frame buffer. Since the pixel memory of the average frame buffer is not fast enough to be reloaded in real time, the pixels are left static, and other components of the frame buffer are used to create the illusion of motion. A survey of colormap and pan/zoom techniques is presented which, in addition to playback of pre-computed animation, allow real-time shading of 3-D surfaces and simulation of some video editing effects. The techniques are appropriate for low to medium-cost frame buffers.

Paul Heckbert is Software Manager at the New York Institute of Technology Computer Graphics Lab in Old Westbury, New York. His responsibilities include rendering algorithm R&D and graphics software maintenance. He received a B.S. in Applied Mathematics from M.I.T. in 1980. While there he became interested in computer graphics at the Architecture Machine Group.

Appeared in Proceedings of Computer FX 84, London, Oct. 1984, pp. 57-67. This version is missing all figures.

1. INTRODUCTION

Within the past ten years, frame buffers have become the standard technology for high quality image display in interactive computer graphics. They can display pictures consisting of shaded colored areas, while vector displays are usually limited to monochromatic lines. Unfortunately, it usually takes several seconds to rewrite the large amount of memory in a frame buffer, so real-time raster animation is severely limited. Dynamic vector displays, on the other hand, are refreshed from a relatively small memory, the *display list*, making that technology more conducive to animation. However, clever use of the colormap and pan/zoom controls of a frame buffer's video generator allow limited real-time animation.

After defining the average frame buffer, and how pictures can be packed into it, we describe both pan/zoom and colormap techniques for animation. In addition to playback of pre-computed animation, these techniques allow real-time shading of 3-D surfaces and simulation of some video editing effects.

1.1. The Standard Frame Buffer

See [Foley and van Dam, 1982] for a general introduction to computer graphics, [Baecker, 1979] for a history of the frame buffer, or [Blinn, 1979] for an excellent crash course in raster graphics.

We will assume a frame buffer with roughly the following specifications:

- 512x512 array of pixels, with 8 bits per pixel
- colormap with 8 bits in and 24 bits out (8 bits per component)
- panning control capable of single pixels shifts in x and y
- integer zoom factors from 1 to 16 (sometimes just powers of 2)
- 512x480 displayed window
- NTSC video standard 30Hz interlaced display (one field every 60th of a second)
- interface to host allowing colormap and pan/zoom control registers to be rewritten during vertical blanking in a glitch-free manner

The following techniques are not limited to frame buffers with these specifications; we state them simply as a point of departure.

The specifications above are intended to typify the *average*, medium cost frame buffer. Colormapped systems with up to 12 bits per pixel are available, and these are even more powerful than the average for real time animation. At greater depths, the size of the colormap gets out of hand, so typically the pixels are broken into three fields for the red, green, and blue, and each component is passed through its own colormap for color correction. If each component gets 8 bits, we have the common 24-bit frame buffer.

Due to VLSI developments and cheaper, faster memories, there are now special-purpose frame buffers and real-time graphics generators with capabilities far beyond the average. Although it is tempting to discuss the possibilities for real-time animation using these state-of-the-art devices, we will restrict ourselves to the medium cost, average frame buffer. This compromise position should make the techniques more applicable to the cheap frame buffers found in many personal computers. Hopefully some of these methods will remain useful decades hence, when current hardware will undoubtedly be obsolete.

2. PICTURE PACKING

Some pictures are easy to display using an average frame buffer, others not. Simple graphics such as text or pie charts do not strain a frame buffer's powers. Displaying pictures containing a variety of colors using an 8-bit frame buffer can be difficult, however, requiring complex algorithms [Heckbert, 1982].

It is only when there are "unused bits" somewhere in the frame buffer that we can squeeze more frames in. If the frames we wish to display are not simple (not low res. and/or few colors) at the outset then we must either shrink them or quantize them so that they will "use fewer bits". Each of these operations can be tricky when working with colormaps which aren't just linear ramps. To shrink a picture, it's usually best to filter areas rather than point sampling, in order to avoid aliasing (jaggies). This is easy in a 24-bit frame buffer, where a full gamut is available, but in an arbitrary colormap the interpolated colors are usually missing (since colormaps are not generally convex sets in color space) so a nearest neighbor mapping must be used. Quantizing a picture to fewer colors can be difficult too, but the resulting savings significant, especially when dithering is used [Heckbert, 1982]. Since the frame buffer has limited storage, we must sacrifice either spatial or color resolution in order to make room for temporal resolution.

In order to visualize the possibilities, the frame buffer should be thought of as a block of memory which can be mapped to the screen in various ways, depending on the colormap, pan, and zoom settings. With an average frame buffer, any rectangular window (from a list of legal sizes) in the frame buffer can be zoomed and panned to fill the screen and any pixel value mapped to any color. Assuming that we have F frames each with a spatial resolution of X by Y each using K colors, we can think of the frame buffer as a three dimensional array fb[F,Y,X] where each array element has $\log_2 K$ bits. If $F \cdot Y \cdot X \cdot \log_2 K \leq 512 \cdot 512 \cdot 8$ then there is a chance that everything will fit.

3. PANNING AND ZOOMING

Pan/zoom animation divides the screen spatially into a number of miniature frames, and plays them back by moving the picture relative to the screen, much as film is moved past a projector. The simplest layout for the frames is a rectangular array of M rows and N columns. They are usually sequenced in row major order (figs. 1, 2).

For playback, the frame buffer is panned so that each frame is centered in turn, with optional zooming to magnify each to fill the screen. This technique (and most of the colormap techniques to follow) work best when the animation sequence forms a cycle.

If zooming is not used, all of the miniature frames can be seen as the picture is panned, but unfortunately only in the center are they played in sequential order, everywhere else on the screen the order is erratic. We can see why by considering the picture just to the right of center. The frame to the right of i is $i+1$ except at the end of the line where it is $i + 1 - N$ (assuming wraparound). This backwards jump in time causes the erratic order. We would like to find an alternative to row major order which eliminates this.

Most frame buffers cause the picture to wrap around as it is panned, as if it were on the surface of a torus. If we rearrange the frames so that they're in a straight line (a spiral on the torus) then we can make every instance advance smoothly. A nice example is the arrangement for 12 frames shown in figure 3. Since 3 and 4 are relatively prime (have no factors in common), a diagonal path with wraparound passes through every square before returning to the first. Figure 4 shows the more general arrangement, which works even when M and N are not relatively prime. This skewed arrangement gives flawless unzoomed panning when M and N divide into the dimensions of the frame buffer and the number of frames equals $M \cdot N$.

4. COLORMAP ANIMATION

If the pixel values are set up properly, short animation sequences can be played back in real time simply by changing the colormap. This technique is unusual because there is no analog to it in other media. With standard reflective or transparent images on paper or film, there are only a few global controls over the image: brightness, contrast, hue (filters), and focus. With a frame buffer, however, we can turn an ordinary grayscale image into a fantastic rainbow of colors at the touch of a button. This effect is also possible using video colorizers, but not with the repeatability and programmability of a digital computer.

For colormap animation, the colormaps can either be pre-computed or calculated on the fly. In either case, for true real-time playback the colormap loading must be synchronized with vertical blanking. If not, objectionable video glitches will often result.

4.1. Colormap Cycling

Colormap cycling is a popular effect which shifts all the colors in a colormap by one position. When run continuously on an arbitrary image, it gives the illusion of motion because colors appear to flow across the screen [Booth and MacKay, 1982]. The motion is usually unpredictable unless the pixel values are laid down in a controlled manner. We will now explore several alternatives for controlled colormap animation. The appropriate technique for a given animation depends on the degree of overlap between frames, as we will see.

4.2. Non-Overlapping Frames

The simplest technique is a special case of colormap cycling called *alternate color animation* [Shoup, 1979]. Consider a cartoonish animation of F frames with only two colors in which a foreground object moves across the screen. If the foreground shapes do not overlap from one frame to another then each can be assigned its own pixel value, say values 1 through F, and pixel value 0 used for background (fig. 5). Now, to display frame i we simply put the foreground color in color i and the background color in all the others. In an 8-bit frame buffer, this allows sequences of 255 frames to be displayed. Moving objects of several colors are also possible, with a consequent decrease in the maximum number of frames.

This technique works perfectly when no two objects overlap, but one is rarely this lucky. There are several ways to eliminate small overlaps:

(1) A priority order can be assigned to the frames, so that, say, frame i overrides frame j if i>j (fig. 6). With this rule, later frames “take bites out of” earlier frames, an error which is sometimes acceptable, depending on the shape of the objects and the extent of overlap.

(2) Decide the priority on a pixel-by-pixel basis, so that all frames competing for the same pixel have an equal chance. The decision can be made using a random number generator.

(3) For slowly moving objects, overlaps are often limited to consecutive frames (in fig. 6, all frames but 4 and 6). For this case, a checkerboard patterning technique can be employed. A pixel at (x,y) in frame i is filled in only if it is inside the object and i+x+y is even (fig. 7). This ensures that consecutive frames will never share the same pixel, since i+x+y and (i+1)+x+y can’t both be even.

All of the tricks described above solve the overlap problem by eliminating overlap, consequently distorting the shapes. There are better ways to deal with overlap.

4.3. Fully Overlapping Frames

If the frames are 1-bit pictures, we can place each in a separate bit plane of the frame buffer [Negroponte, 1977]. This allows 8 arbitrary 1-bit frames to be packed into an average frame buffer. Frame i is displayed by loading a colormap which selects the foreground or background color depending on bit i of the pixel value. This technique is called *bit plane extraction*. The frame buffer is partitioned in depth, in contrast to the spatial partitioning used for pan/zoom animation.

By grouping planes, we can display four 2-bit frames or two 4-bit frames. Each has its applications. Because of the four-color map theorem, any map can be encoded into 2 bits per pixel, so that four maps will fit in an 8-bit frame buffer. Packing two 4-bit frames is often useful in order to alternate quickly between two pictures when making comparisons.

Generalizing further, we note that the number of colors in each frame need not be a power of two. As long as the product of the number of colors in each frame is less than the number of possible pixel values (256), the animation will fit.

4.4. Partially Overlapping Frames

When objects partially overlap we'd like to find an encoding technique which allows more than 8 frames to be packed into a frame buffer [Shoup, 1979]. In fact, there is an algorithm which includes alternate color animation and bit plane extraction as special cases. The central idea is to find the sequence of colors which each pixel steps through. For an animation of F frames with K colors per frame, the number of unique color sequences can range between K (if all frames are equal) and K^F (if all possible sequences occur). Pixels with equal sequences may as well have equal pixel values. Frames are encoded into the frame buffer one-by-one, creating new pixel values when new sequences are discovered. As the frame buffer is being filled, the program keeps track of the sequence corresponding to each pixel value, thereby building a colormap for each frame. These colormaps are loaded in rapid succession in order to play back the animation. The number of frames which will fit ranges between 8 and 255 depending on the degree of overlap. Figure 8 shows three frames which use 2 colors, 2 colors, and 3 colors, respectively, but which have only 7 color sequences, not the $2 \cdot 2 \cdot 3 = 12$ theoretically possible.

4.5. Orientation Coding

The previous techniques pack pre-computed animation sequences into a frame buffer and create motion by changing the colormap. It is also useful to store a single picture in a frame buffer and use the colormap to interpret it. The information stored will often be something more abstract than light intensity. A popular application of this idea performs shading of 3-D surfaces using colormap manipulations. At each pixel, the orientation of the vector normal (perpendicular) to the surface is found, and encoded into the pixel value [Blinn, 1979], [Sloan and Brown, 1979]. This is called *orientation coding*. It can be used on any type of surface: polygon, quadric, patch, etc.

The normal at each pixel can be encoded using two variables since a unit vector has two degrees of freedom (z can be calculated given x and y, since $x^2 + y^2 + z^2 = 1$). The normal vectors represented by the code are called the *orientation list*. The most common orientation code scales and quantizes the x and y components of the normal to lie between 0 and 15, and then tacks together these two 4-bit numbers into an 8-bit pixel value (like fig. 9). Unfortunately, this code results in artificial faceting (fig. 11a). Several alternative codes based on polar coordinates have been suggested, but they are marginal improvements [Bass, 1981], [Evans, 1984]. The greatest improvement results from perturbing the normals randomly before quantization (fig. 11b) [Evans, 1984]. This dithering makes the surface look rough, but eliminates the faceting.

If we wish to minimize the faceting effect without resorting to dithering, we should think of creation of the orientation list as a two-dimensional quantization or clustering problem. We can improve upon the uniform x-y quantization discussed above. To minimize artifacts, the quantization should be done adaptively, depending on the distribution of normal vector orientations in the scene [Heckbert, 1982]. Each normal can be thought of as a point on a hemisphere, or, projecting onto the x-y plane, as a point in a circle (fig. 10). The modest improvement in appearance due to adaptive quantization may not, however, be worth the effort except for low depth frame buffers.

Adaptive quantization is especially effective for polyhedra, since each face has a single normal. As long as there are fewer than 255 uniquely oriented faces, this will give results superior to uniform quantization.

Once the list of orientations (one per pixel value) and an orientation coded picture have been computed, the picture can be shaded by changing the colormap. Since the pixel value encodes surface orientation but not position, we can accurately model any shading formula which is a function of orientation only. This precludes correct shading for local light sources or perspective projections, since highlight locations are dependent on position within the scene when the light or eye is not infinitely distant. Shadows are also impossible.

To compute the colormap, the shading function [Foley and van Dam, 1982] is evaluated for each of the fewer than 256 orientations. Since the shading function is evaluated so few times per frame, it can be an expensive one (area light sources, directional lights, etc.). Depending on the number of lights, computing each colormap may take more than a frame time, but a reasonable computer will allow several updates per second at the minimum. When the colormap is animated to simulate illumination by several moving, colored lights, the effect is much more dramatic than can be conveyed on this black and white page.

The author has developed an interactive *light editor* which allows light sources to be positioned on an infinite radius sphere using a tablet, with about 4 colormap updates per second on a PDP 11/34. The user also has control over each light's color, diffuse/specular ratio, and highlight size. While using this program to set up lights for 3-D scenes, we've found the greatest limitation to be that all surfaces must have the same reflectance properties (color, glossiness, etc.). This problem could be solved if a

12-bit frame buffer were available, since the extra 4 bits would allow 16 surface types. Even with an 8-bit frame buffer, several bits could probably be used for surface type if the orientations were chosen adaptively, and dithering employed.

Orientation coding has been used very cleverly by Nelson Max for his film *Carla's Island* [Max, 1981]. He simulated the changing illumination of the ocean during sunset using colormap animation. Although he had only 8 bits per pixel, he was able to shade several surface types by exploiting the simple geometry of his scene when choosing an orientation code.

Evans has described a method which allows 7-color diffuse surfaces to be orientation coded using a 24-bit frame buffer [Evans, 1984]. His method is restricted to the seven surface colors: red, green, blue, magenta, yellow, cyan, and white. These colors can be represented by a 3-bit code: 100=red, 010=green, 111=white, etc. Evans computes an orientation coded picture as usual, reserving pixel value 0 for black background. This 8-bit picture is written to the three components of the frame buffer in a special way. Each of the three bits of surface color determines whether the orientation code or pixel value 0 is used for each byte of the pixel. For example, if C is the orientation code, then a white surface would get the 24-bit pixel value (R,G,B)=(C,C,C) while a red surface would get (C,0,0). When the colormap is animated in the usual way, (C,0,0) gets mapped to a red color because color 0 is black. Evans' method has several problems: highlights have the wrong color, and the selection of surface colors is poor. The latter restriction could be eliminated by dithering the surface color.

4.6. Colormap Mixing

Whenever colormap manipulation is being used, it is trivial to simulate video effects like fade, key, and cross-dissolve. Frames can be mixed in numerous ways, including transparency and priority ordering [Blinn, 1979]. Applications for colormap mixing include both animation and stills.

5. CONCLUSIONS

Colormap and pan/zoom animation, although limited, are powerful and economical techniques which allow real-time playback. Depending on the user's requirements, combinations of the techniques described above can be used to shift those limitations between the temporal, spatial, and chromatic dimensions.

6. ACKNOWLEDGEMENTS

Dan Franzblau and Walter Bender helped me milk the Ramtek 9300 at M.I.T. during the spring of '79. Rebecca Allen and Amber Denker at N.Y.I.T. gave me great encouragement.

7. REFERENCES

Baecker, Ronald, “Digital Video Display Systems and Dynamic Graphics”, *Computer Graphics (SIGGRAPH ’79 Proceedings),* vol. 13, no. 2, Aug. 1979, pp. 48-56.

Bass, Daniel H., “Using the Video Lookup Table for Reflectivity Calculations: Specific Techniques and Graphic Results”, *Computer Graphics and Image Processing,* vol. 17, no. 3, Nov. 1981, pp. 249-261.

Blinn, James F., “Raster Graphics”, *Tutorial: Computer Graphics,* Kellogg Booth, ed., IEEE catalog no. EHO 147-9, 1979, pp. 150-156.

Booth, Kellogg S., and Stephen A. MacKay, “Techniques for Frame Buffer Animation”, *Proceedings of Graphics Interface ’82,* May 1982, pp. 213-220.

Evans, Kenneth B., “Realtime Lighting Manipulation in Color via Lookup Tables”, *Proceedings of Graphics Interface ’84,* May 1984, pp. 173-177.

Foley, James D., and Andries van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass., 1982.

Heckbert, Paul S., “Color Image Quantization for Frame Buffer Display”, *Computer Graphics (SIGGRAPH ’82 Proceedings),* vol. 16, no. 3, July 1982, pp. 297-307.

Max, Nelson L., “Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset”, *Computer Graphics (SIGGRAPH ’81 Proceedings),* vol. 15, no. 3, Aug. 1981, pp. 317-324.

Negroponte, Nicholas, “Raster Scan Approaches to Computer Graphics”, *Computers and Graphics,* vol. 2, no. 3, 1977, pp. 179-193.

Shoup, Richard G., “Color Table Animation”, *Computer Graphics (SIGGRAPH ’79 Proceedings),* vol. 13, no. 2, Aug. 1979, pp. 8-13.

Sloan, Kenneth R., Jr., and Christopher M. Brown, “Color Map Techniques”, *Computer Graphics and Image Processing,* vol. 10, no. 4, Aug. 1979, pp. 297-317.

fbanim captions

fig. 1: row major 3x4 array

fig. 2: row major 4x8 array

fig. 3: diagonal 3x4 array

fig. 4: skewed 3x4 array

fig. 5: non-overlapping
(numbers are pixel values)

fig. 6: priority order

fig. 7: checkerboard patterning

fig. 8a: three frame animation and its composite

fig. 8b: colormaps for each of the three frames

fig. 9: 4x4 orientation code (4 bits)
(standard one is 16x16, using 8 bits)

fig. 10: adaptive orientation quantization
circles are centers of *dirichlet domains*

fig. 11a: undithered

fig. 11b: dithered

fig. 12: dithered,
bump mapped torus

(all three using 16x16 x-y orientation code)