Return

Mission: Compressible

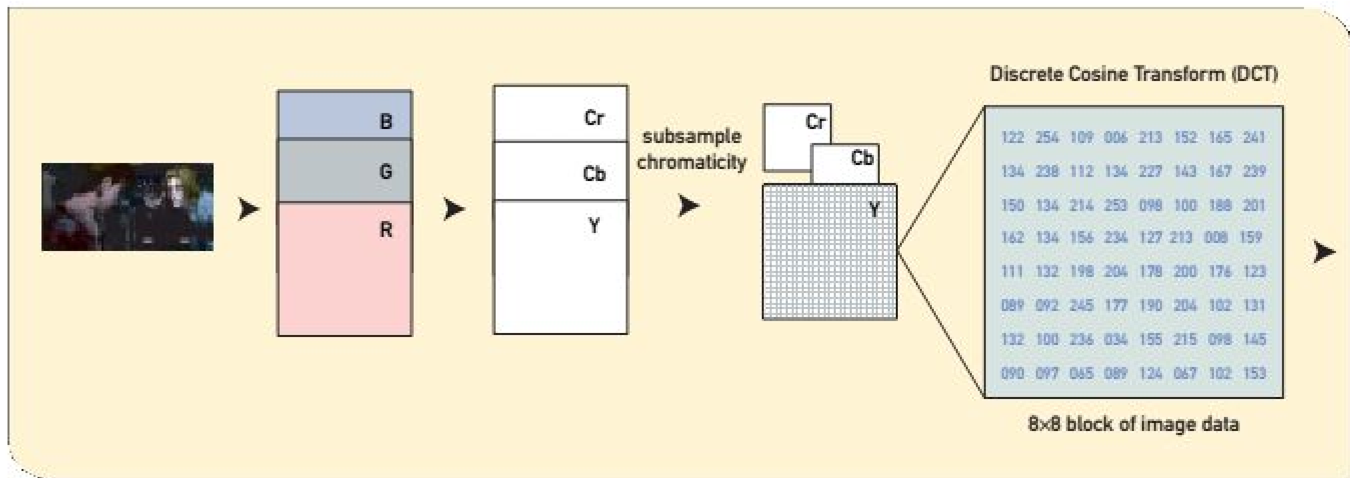
Achieving Full-Motion Video on the Nintendo 64

TODD MEYNINK | *When there's no surf to be found, Todd's busy pretending to be a software engineer at Angel Studios. Drop him a line at todd@angelstudios.com.*

RESIDENT EVIL 2 for the Nintendo 64 was the first game on a cartridge-based console system to deliver full-motion video. Angel Studios' team brought this two-CD game, comprising 1.2GB of data, to a single 64MB cartridge. A significant portion of this data was more than 15 minutes of cutscene video. Achieving this level of compression, meeting the stringent requirement of 30Hz playback, and delivering the best video quality possible was a considerable challenge.

To look at this challenge another way, let's put it into numerical perspective. The original rendered frames of the video sequences were 320×160 pixels at 24-bit color = 153,600 bytes/frame. On the Nintendo 64 RESIDENT EVIL 2's approximately 15 minutes of 30Hz video make a grand total of 15•60•30•153,600 = 4,147,200,000 bytes of uncompressed data. Our budget on the cartridge was 25,165,824 bytes, so I had to achieve a compression ratio of 165:1. Worse still, I had to share this modicum of cartridge real estate with the movie audio.

The Playstation version of RESIDENT EVIL 2 displays its video with the assistance of a proprietary MDEC chip but because the N64 has no dedicated decompression hardware, our challenge was compounded further. To better understand the magnitude of the implementation hurdles, consider that it is analogous to performing full-screen MPEG decompression at 30Hz, in software, on a CPU roughly equivalent in power to an Intel 486. Fortunately, the N64 has a programmable signal processor called an RSP that has the ability to run in parallel with the CPU.



TYPE	USED AS REFERENCE?	TYPE OF PREDICTION	COMMENTS
Intrapictures (I-frames)	Yes	N/A	Typically encoded with a JPEG-like algorithm. They start and end each GOP.
Predicted-pictures (P-frames)	Yes	P-frames support forward prediction from a previous I-frame.	The motion-compensated prediction errors are DCT coded.
Bidirectional (interpolated) pictures (B-frames)	No	A B-frame is a forward, backward or bidirectional picture created by, and relative to, other I and P frames.	

FIGURE 1 (top). A summary of the encoding process.
FIGURE 2 (bottom). GOP encoding methods.

A Brief JPEG Primer

In order to simplify the timing and synchronization issues, I chose an MPEG-1-style (henceforth referred to as MPEG) compression scheme for the video content only. (Audio was handled separately, which I'll discuss later in this article.)

As an introduction to the relatively complex issues of applying MPEG compression to the video sequences of the game, let me present a brief primer on JPEG compression.

First, the image is converted from RGB into YCbCr. This process converts the RGB information into luminance information (Y) and chromaticity (Cb and Cr):

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.2989 & 0.5866 & 0.1145 \\ 0.1687 & -0.3312 & 0.5 \\ 0.5 & -0.4183 & -0.0816 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Inverting the coefficient matrix and applying it to YCbCr finds the inverse transformation. This color model exploits properties of our visual system. Since the human eye is more sensitive to changes in luminance than color, I could devote more of the bandwidth to represent Y than Cb and Cr. In fact, I can halve the size of the image with no perceptible loss in image quality by storing only the nonweighted average of each 2×2-pixel block of chromaticity information. This way the Cb and Cr information is reduced to 25 percent of its original size. If each of the three components (Y, Cb, Cr) represented 1/3 of the original picture information, the subsampled version now adds up to 1/3 + 1/12 + 1/12 = 1/2 the original size.

Second, each component is broken up into blocks of 8×8 pixels. Each 8×8 block can be represented by 64-point values denoted by this set:

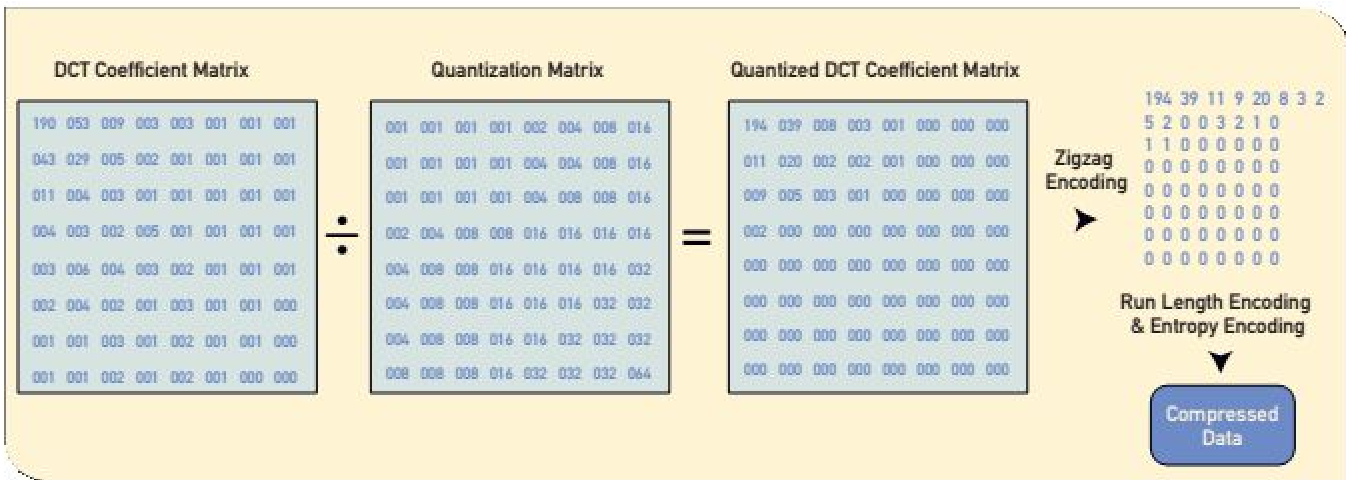
$$\{f(x,y) \in 0 \le x \le 7,\ 0 \le y \le 7\}$$

where x and y are the two spatial dimensions. The discrete cosine transform (DCT) transforms these values to the frequency domain as $c = g(F_u, F_v)$, where c is the coefficient and F_u and F_v are the respective spatial frequencies for each direction:

$$\frac{1}{4} \cdot C(u) \cdot C(v) \cdot \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cdot \cos\left(\frac{(2x+1) \cdot u \cdot \pi}{16}\right) \cdot \cos\left(\frac{(2y+1) \cdot v \cdot \pi}{16}\right)$$

$$\text{where } C(u) = C(v) = \frac{1}{\sqrt{2}} \text{ for } u,\ v = 0 \text{ and } 1 \text{ otherwise}$$

The output of this equation gives another set of 64 values known as the DCT coefficients, which is the value of a particular frequency — no longer the amplitude of the signal at the sampled position (x,y). The coefficient corresponding to vector (0,0) is the DC coefficient (the DCT coefficient for which the frequency is zero in both dimensions) and the rest are the AC coefficients (DCT coefficients for which the frequency is nonzero in one or both dimensions). Because sample values typically vary gradually from point to point across an image, the DCT processing compresses data by concentrating most of the signal in the lower values of the (u,v) space.



For a typical 8×8 sample block, many — if not all — of the (u,v) pairs have zero or near-zero coefficients and therefore need not be encoded. This fact is exploited with run-length encoding.

Next, the 64 outputted values from the DCT are quantized on a per-element basis with an 8×8 quantization matrix. The quantization compresses the data even further by representing DCT coefficients with precision no greater than is necessary to achieve the desired image quality. This tunable level of precision is what you modify when you move the JPEG compression slider up and down in Photoshop when you save an image.

In the third step (ignoring the detail that the DC components are difference-encoded), all of the quantified coefficients are ordered into a "zigzag" sequence. Since most of the information in a typical 8×8 block is stored in the top-left corner, this approach maximizes the effectiveness of the subsequent run-length encoding step. Then the data from all blocks is encoded with a Huffman or arithmetic scheme. Figure 1 summarizes this encoding process.

Both JPEG and MPEG are "lossy" compression schemes, meaning that the original image can never be reproduced exactly after being compressed. Information is lost during JPEG compression at several points: chromaticity subsampling, quantization, and floating-point inaccuracy during the DCT.

Motion Picture Compression

JPEG compression attempts to reduce the spatial redundancy in a single still image. In contrast to a single frame, video consists of a stream of images (frames) arriving at a constant rate (typically 30Hz). If you examine consecutive frames in a movie, you'll generally find that not much changes from one frame to the next. MPEG exploits this temporal redundancy across frames, as well as spatial redundancy within a frame.

To deal with temporal redundancy, MPEG divides the frames up into groups, each referred to as a "group of pictures," or GOP. The size of the GOP has a direct effect on the quality of the compressed images and the degree of compression. A GOP's size represents one of the many trade-offs inherent to this process. If the GOP is too small, not all the temporal redundancy will be eliminated. On the other hand, if it is too large, images at the start of the GOP will look substantially different from images toward the end (imagine a scene change partway through), which will adversely affect the quality of reconstructed images.

To improve compression, frames are often represented by composing chunks of nearby "reference" frames. The frames within a GOP are generally encoded via one of three methods, as shown in Figure 2.

Figure 3 shows the different frames and their roles and relationships. This example introduces an intracoded picture (I-frame) every eight frames. The sequence of intrapictures (I), predicted pictures (P), and bidirectional pictures (B) is IBBBPBBBI. When GOP size is varied, only the number of B-frames on either side of the P-frame ever changes. Note that this sequence represents the playback sequence and is not necessarily the order in which frames are stored. Storing frames 1,2,3,4,5,6,7,8 as 1,5,2,3,4,9,6,7,8 would make sense since the I- and P-frames are read first, facilitating construction of B-frames as soon as possible and reducing the number of frames that need to be kept around in order to decode the stream successfully.

Prediction and interpolation are employed in a technique

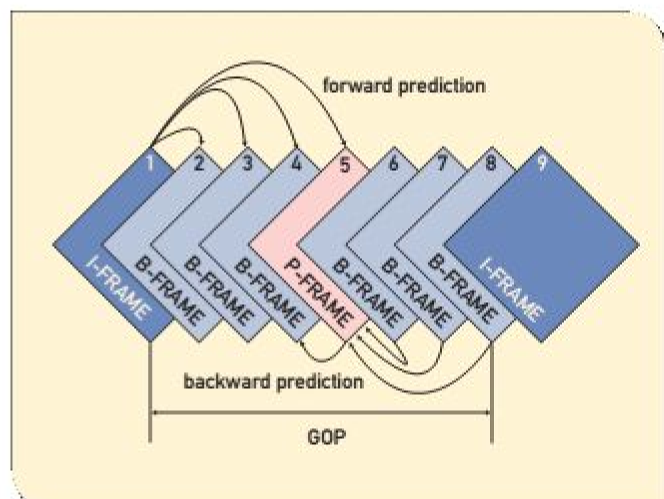


FIGURE 3. Relationships between the I-, B-, and P-frames.

called "motion compensation." Prediction assumes that the current picture can be modeled as a transformation of the picture at some previous time. Interpolated pictures work similarly with past and future references, combined with a correction term.

It makes no sense to use an entire image to model motion within a frame, so modeling motion must be done with smaller blocks. MPEG uses a macro-block consisting of 16×16 pixels (think of a macro-block as four of our 8×8 DCT blocks). This approach illustrates another trade-off between the quality of the image and the amount of information needed to represent the image. An estimate of per-pixel motion would look best, but would be way too big, while a quarter-image block would look pretty ordinary but take up very little space. In a bidirectionally coded picture, each 16×16 macro-block can be of type intra, forward predicted, backward predicted, or average. Note that the 16×16 block used for compensation need not lie on a 16×16-pixel boundary.

A cost function typically evaluates which macro block(s) from which image represents the current block in the current image. This cost function measures the mismatch between a block and each predictor candidate. Clearly an exhaustive search, in which all possible motion vectors are considered, would give the best result, but that would be extremely expensive computationally. Figure 4 shows the relative sizes of the different frame types.

JPEG Compression/ Decompression Steps

JPEG Compression

1. Preparation of data blocks (RGB→YCbCr)
2. Source encoding
   - Discrete cosine transform (DCT)
   - Quantization
3. Entropy encoding
   - Run-length encoding
   - Huffman or arithmetic coding

JPEG Decompression

1. Entropy decoding
   - Huffman or arithmetic coding
   - Run-length coding
2. Source decoding
   - Dequantization
   - Inverse discrete cosine transform (IDCT)
3. Preparation for display (YCbCr→RGB)

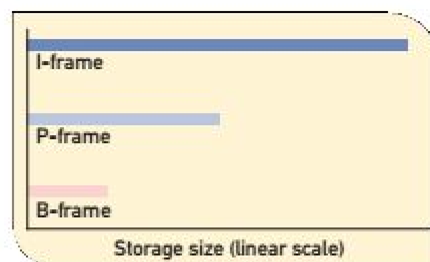


FIGURE 4. Relative sizes between I-, B-, and P-frames.

Implementation

My first step to implement the full-motion video for the N64 version of RESIDENT EVIL 2 was to develop a PC-based compression/decompression platform that could be debugged and tuned easily. This let me experiment with different GOP sizes, bit rates, and other variables. It quickly became apparent that this optimization challenge would be a war between image size, image quality, and decoding complexity.

There were severe memory constraints. As you probably know, without an expansion pack, the N64 has only 4MB of RAM. This memory is divided up among program code, heap, stack, textures, frame buffers, the Z-buffer, and so on. For a large game, it's likely that there will be room for only two frame buffers at any reasonable resolution and color depth. Keep in mind also that you need space to hold the necessary reference frames (I-frames and P-frames) in memory to compute the predicted frames. This requirement came down to three frames (I,P,I) of YCbCr data at 24-bit color. Obviously the resolution of the video dictates exactly how much RAM this requires.

I tested many different parameter settings, the most fundamental of which was bit rate. A higher bit rate naturally led to higher quality. Unfortunately, simply raising the bit rate to a point where acceptable quality was exhibited across the board required too much storage space. In our case, a quick calculation gives us our target mean compressed frame size: 25,165,824 bytes / 27,000 frames = 932 bytes per frame.

Higher resolution improved the image quality up to a point, but it quickly fell off after that. The reason for the falloff is that only a limited number of bits are available to describe all the pixels in a frame. While a high-resolution movie may look good when little in the scene is changing, rapid motion or a scene change may not be adequately described at the same bit rate. This artifact becomes extremely noticeable when the boundaries of motion blocks are discontinuous, which gives the movie a "blocky" look. Additionally, increasing the resolution means more macro-blocks, more inverse DCTs, more motion compensation, more color space conversion, and generally more decoding time. It quickly became apparent to us that displaying movies encoded at the source resolution would not be possible at 30Hz.

Since movies would be displayed at a lower resolution than the source, we needed a mechanism for scaling the decoded frames back up to full-screen resolution. We tried pixel doubling, but the results were unsatisfactory even on an NTSC screen (which hides a lot of the larger "pixel" definition). Next I tried using the N64's `rectcopy` routine (part of the N64's software library) with bilinear interpolation. This approach gave better results and remained in place until I tried a custom microcode routine — which in turn gave way to a reduced screen resolution, which the RDP scaled up automatically for free. Reduced resolution also gave the added bonus of reducing memory requirements for the frame buffers.

I tried decoding the movies to both 16-bit RGB and 32-bit RGBA frame buffers. The 32-bit image gave superior results, especially across gradations of color, though at the time the performance hit didn't justify the extra memory and processing requirements. The target color depth had several implications. Foremost were the increased memory requirements of the frame buffers. At the time I was evaluating this approach, running at source resolution and color depth would not have been pos-

Full-motion video stills from Resident Evil 2 for N64.

sible given the memory constraints. A secondary implication was the increase in computing time required to process the larger frames, further hampered by the N64's less-than-stellar memory performance. At this point, I had movies running at low resolution at 30Hz and roughly within the size requirements, but the image quality left a lot to be desired and this problem needed to be addressed. I began to think about optimization.

Rewriting the Algorithm in Microcode

My decompression algorithm was written in C, and its computation time was spread over a large portion of the source. I was not going to reap large benefits from optimizing code with MIPS Assembly without a Herculean effort and far more time than I had. While I had never dealt with the N64's signal processor (the RSP) before, I knew that its vector nature and potential to run in parallel held the keys to improved performance. (For a walk-through of the process of calling microcode programs, DMA-ing data in and out of micro-memory, passing arguments, and more, refer to Mark DeLoura's article, "Putting Curved Surfaces to Work on the Nintendo 64," November 1999.)

After getting a simple "add 2 to this number" function to work, I began to port portions of the C-based decoding code over to microcode. This task was by no means simple. The only avenue for debugging the microcode was to crash the RSP at various places and read the data cache to verify that things up to that particular point were working correctly. This process was very laborious.

A direct result of the difficulties of developing microcode was that I would only have time to rewrite a finite number of routines. A fixed-point rewrite of the inverse discrete cosine transform seemed like an obvious choice. After several painstaking days of coding and verifying this routine, it was ready for prime time. Unfortunately, the rewrite actually caused the routine to perform more slowly. My investigation revealed that a cache issue was causing this problem. As each block of pixel data is read and prepped for decode, it becomes resident in the CPU's data cache. For the RSP to process it, the data must be DMA'd from main memory to the RSP's DMEM. After processing, it must then be DMA'd back into main memory. The CPU's cache doesn't know that this potentially asynchronous process has modified the data, so those cache lines must be "invalidated" and reread to ensure that the CPU is operating on up-to-date data. The bottom line was that all this extra memory thrashing was swamping any benefit gained by the efficiency of the RSP's SIMD instructions.

My next stop was the motion compensation code. Unfortunately, the amount of code required to handle all the different kinds of motion compensation was prohibitive. The RSP's lack of a `shift` instruction didn't promise a clean implementation. Clearly, the code which finally brought the decompressed image to the screen (without further CPU intervention) stood to gain the most benefit.

Rewriting the color-space conversion (CSC) routines to take advantage of the RSP's vector architecture proved to be quite successful. The RSP was uniquely suited to this sort of task. Once the RSP had performed the conversion, the RGB data could be DMA'd from DMEM directly to the frame buffer, avoiding the earlier caching problems. This bought a noticeable performance increase and provided a corresponding increase in image quality, but I was still a long way from the quality of the original FMV.

The Epiphany

At this point, my implementation was getting closer to my goal, but problems remained. First, the image quality was still not as good as I had hoped. Second, the data files required to support this inadequate quality were already substantially over their size budget. And finally, the decoding still took too long and I couldn't see an easy way to improve it — especially since I was trying to also reduce the bit rate.

Then the idea struck me: what if I skipped every other frame and interpolated at run time? I knew if I could get this approach to work, it would simultaneously

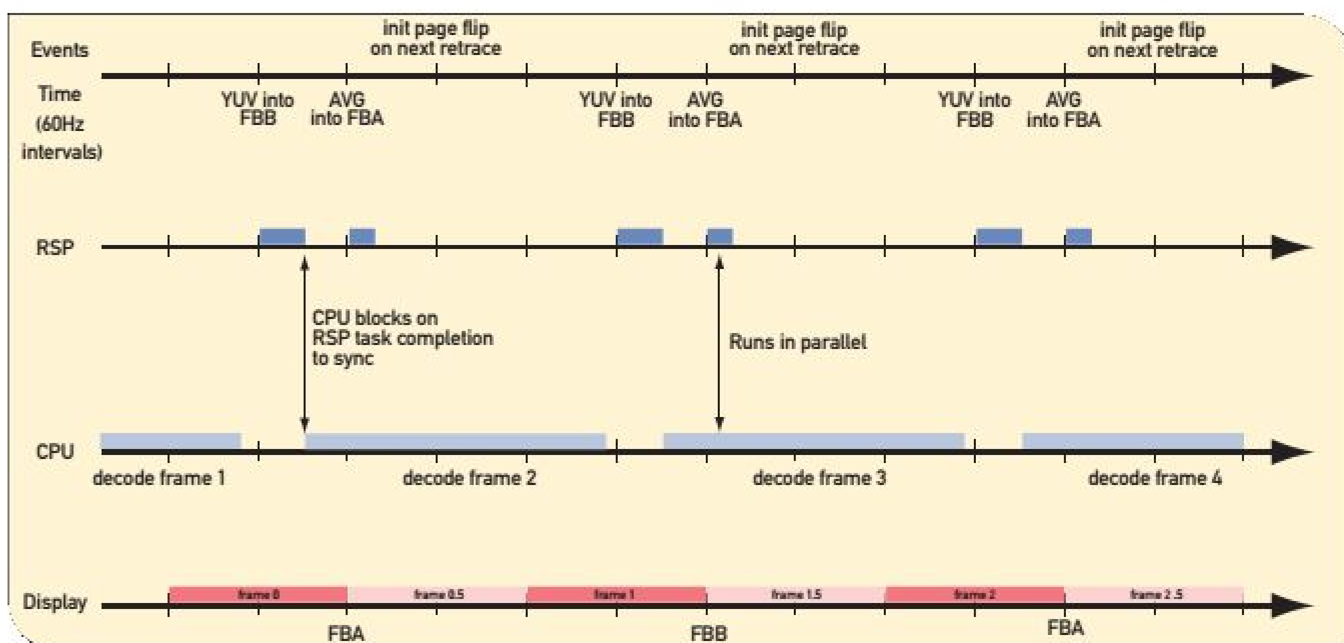


FIGURE 5. A processing time line.

halve the bit rate and double the decoding time. I was banking on the hope that it would be difficult to differentiate data decoded at 30Hz from data decoded at 15Hz with interpolation.

At first I considered using triple buffering to decode two frames, and then interpolating between the two to generate the intermediate frame. But memory restrictions quickly ruled out this approach and any of its variants.

I eventually found the solution. In it, the RSP average routine effectively swaps in a new frame without a page flip by beating the retrace gun down to the bottom of the screen thanks to some fast microcode. From a conceptual standpoint, this tricky timing allowed me to achieve triple buffering with only two buffers. (See Figure 5, a processing time line, and Figure 6, a UML state machine that runs the CPU thread in parallel with the RSP.)

With this approach, each frame had almost 1/15th of a second to decode. Skipping every other frame halved the memory footprint. This made the inclusion of all the clips possible, and also allowed us to improve the quality with the space left over. And we still had extra decoding time to burn, which we put to good use by increasing the movie resolution to further improve image quality.

It wouldn't have been possible to implement this solution without a scheduler. The scheduler used was part of a sophisticated operating system written by fellow team members Chris Fodor and Jamie Briant. In addition to supporting multiprocessing and multi-threading, it provided detailed information about and management of the N64's hardware. This was pivotal to taking full advantage of the machine. Once I fleshed out the algorithm, implementation with the OS's scheduler was straightforward.

Continuous Improvement

Shortly after we implemented this system, we created a demo for E3 1999. It was very gratifying to walk past Capcom's booth and hear people arguing over whether they were playing the game on an N64 or a Playstation. Unfortunately, the video quality on the N64 was still noticeably below that of the original Playstation game.

One of the reasons for this was that smooth color gradients were not reproducing well. I experimented with a cheap form of dithering as a postprocessing step. (Credit goes to Alex Ehrath, my fellow RE2-N64 programmer, for this idea.) As YCbCr data was converted to 16-bit RGB, I kept track of the lower-order bits that were being masked off, added these lower-order bits to the following pixel before it was masked off, and so on. The red, green, and blue channels were processed independently. While this technique provided a noticeable improvement when the frames were considered in isolation, differences from frame to frame made it look as if there were some sort of static interference when they were played as a movie. The modulation of the interpolated frames only amplified this problem.

The bad reproduction of gradients was especially noticeable in dark areas. To compensate for this, I experimented with gamma correction as a preprocessing step prior to encoding. My goal was to even out the perceived difference in intensity between dark colors and lighter colors. Unfortunately, this approach just gave the movies a washed-out look.

Next, I tackled the age-old challenge of trying to make the image on the NTSC display resemble those shown on an RGB monitor. We drew ten vertical bars across the screen, moving from black to gray to white as an intensity reference image. On an NTSC screen, the middle bar looked more red than gray, even on expensive reference monitors. After several iterations, we moved to Photoshop and applied a combination of color boosting, contrast/brightness adjust-

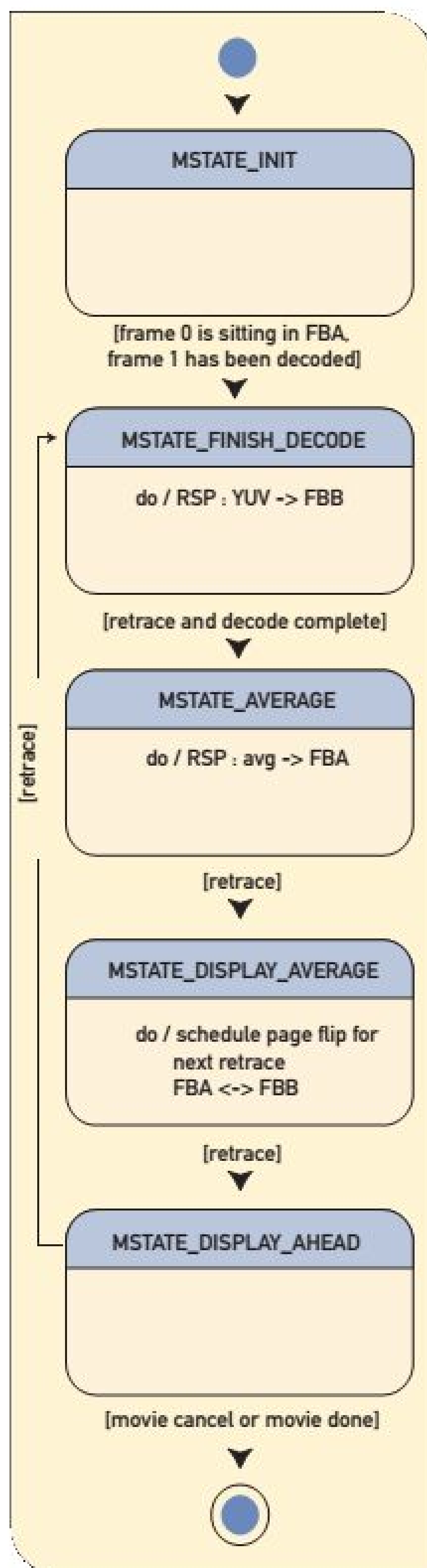


FIGURE 6. A UML state machine that runs the CPU thread in parallel with the RSP.

ments and level altering images prior to encoding until we felt we had a combination that improved the final image quality substantially.

Finally, in another attempt to improve color gradient reproduction, I retried a previously rejected technique. In earlier tests, the full 24-bit color output had looked marginally better, but extra computation and memory requirements had ruled it out. Now that the color space conversion had been moved to microcode, and a multi-processing approach had bought us much longer decoding times, I could get 24-bit color with little extra cost. A single day's coding brought startling results, and when combined with the improved color from the Photoshop preprocessing, the true-color output improved the display quality dramatically. Colors were reproduced even more vibrantly and patchy blotches became smoothly transitioning gradients. At last, I had achieved what I was after. To download an archive containing the final CSC microcode, go to the *Game Developer* web site at www.gdmag.com.

Scripting and Synchronization with Audio

Fortunately, both Leon and Claire's (the two main player-characters in RE2) games shared many sections of video, which I factored out into shared "video clips." This substantial task resulted in hundreds of clips ranging in length from a minute to a second. Movie playback was then achieved by replaying a sequence of clips. The ability to "hold" on a particular frame while the frame counter ticked by provided some additional compression. These sequences of movie clips and holds were played back through scripts that bestowed a substantial amount of flexibility.

Audio compression and playback was handled separately from the video. Audio clips were triggered on particular frames.

Dividing movies into clips gave us the ability to vary the bit rate according to content. Fast action meant larger changes from frame to frame, which led to more compression artifacts requiring higher bit rates to compensate. Conversely, relatively calm scenes could be encoded at a much lower bit rate.

Changes in scenes at low bit rates were problematic when they occurred between I-frames. Until the next I-frame swung by, the sudden change caused the remainder of the GOP to display with highly noticeable compression artifacts. Quality could be preserved across changes in scene at low bit rates by making new clips with cuts on the scene change boundaries.

An Industry First

If we were to do another similar N64 project, we would definitely implement the same technology and tricks I've described here to any video sequences used. However, many of these techniques can be applied on any platform where file size is a major concern. For instance, factoring out all common "film" sequences and replaying individual clips back to back via a script to re-create the original can afford a large space savings. Ensuring the clips are built on scene-change boundaries allows you to lower the bit rate and still maintain quality. Also, compensating for loss of color saturation and levels due to compression prior to encoding can yield a result closer to the original.

Bringing full-motion video to the N64 is challenging both in terms of achieving the necessary compression to support video on a cartridge system and the software required to play the compressed data back in real time. Relentlessly trying and retrying everything brought us a great result and an industry first: high-quality video on a cartridge-based console.

Discuss this article in Gamasutra's Connection!
www.gamasutra.com/discuss/gdmag

FOR MORE INFORMATION

BOOKS

Raghavan, S. V. and S. K. Tripathi. Networked Multimedia Systems. *Upper Saddle River, N.J.: Prentice Hall, 1998.*

Foley, J. D., and others. Computer Graphics: Principles and Practice, *2nd ed. Reading, Mass.: Addison-Wesley, 1996.*

WEB RESOURCES

www.mpeg.org