DirectX 6 Texture Map Compression

by Dan McCabe and John Brothers

Texture maps add visual detail to a scene without increasing its geometric complexity. However, texture memory is a relatively scarce resource, forcing game developers continually to tweak their software and artwork to fit into the limited texture memory on the graphics accelerator. Even with the availability of Intel's Accelerated Graphics Port (AGP), which lets the graphics accelerator directly access texture maps stored in system memory, bus and memory usage — as well as bandwidth — are still very limited. A solution to these problems is texture map compression, which greatly reduces not only the amount of memory that a texture map occupies, but also the bandwidth required to fetch texture data.

S3 devised a compression scheme specifically for texture maps, called S3TC, which yields benefits readily visible to programmers while maintaining the quality of artists' creations (Figure 1). Microsoft recently licensed this technology and made it the basis of DirectX 6's texture map compression. As such, this compression format should have broad hardware and software support.

Taking Advantage of Reduced Memory and Bandwidth

The speed at which texture data can be accessed generally limits sustained 3D fill rates, particularly when high-quality filtering modes, such as trilinear, are used. With a given memory or bus bandwidth, much more texture data can be read with compressed textures. The effective bandwidth with texture compression is the actual data rate multiplied by the compression ratio. So, for AGP-2x, where the maximum theoretical bandwidth is 512MB/s, the effective theoretical bandwidth with six-to-one compression is 3.0GB/s. By boosting the sustained fill rate, game performance can be dramatically improved when texturing directly from system memory over AGP or when reading a texture out of frame buffer memory. Of course, the most obvious benefit to using compressed textures is that they require less storage space. This can be taken advantage of in a number of ways.

BETTER TEXTURE RESOLUTION. Normally, texture storage requirements strain the limits of available memory. With tex-

Dan McCabe has enjoyed working on computer graphics for the last 20 years. A significant portion of that time was spent at IBM Research, where he worked on 3D rendering as well as dynamics simulation and the collision detection problem. Dan is currently a 3D architect with S3 Inc., where he is defining key aspects of the next several generations of 3D hardware. Although S3 is headquartered in Santa Clara, Calif., Dan is based in S3's Bellevue, Wash., office.
John Brothers is VP of architecture and software development at S3 and has been instrumental in the design and development of S3's recently announced Savage3D accelerator for the past two years. He is now working on future high-end 3D graphics platforms in development at S3 and is leading a great software group in putting out high performance, bug-free drivers.

ture compression, you can use higher-resolution (larger) textures, as well as a greater variety of textures at any given time. Larger textures provide more surface detail on objects and can prevent the blurriness that's often a consequence of overstretching small texture maps. Larger texture maps also allow for much more detail than would be possible without compression. And you can increase the number of texture maps in use at any one time, enabling more varied scenes.

Because the rendering surface is usually stored in memory along with your textures, you can also increase the resolution of your rendering surface with the additional memory made available by compression. Needless to say, you might not be able to increase your resolution in all aspects simultaneously, but having more memory available gives you more flexibility and options for improving your title.

MIP-MAP USE. Increased amounts of texture memory also make it easier to take advantage of MIP-maps. While MIP-maps require 30 percent more storage to house the down-sampled MIP-levels, texture compression easily frees up this extra storage. Using correctly computed MIP-maps, you can efficiently eliminate aliasing artifacts that would otherwise appear when mapping multiple texels to one screen pixel. The correct way to compute MIP-levels is with a low-pass filter, normally a box filter. Computing MIP-levels with point-sampling to get "sharper images," as one graphics chip maker has recommended, completely eliminates the intended benefit of MIP-mapping, which is to do high-quality texture antialiasing. Beware of computing MIP-levels with point sampling.

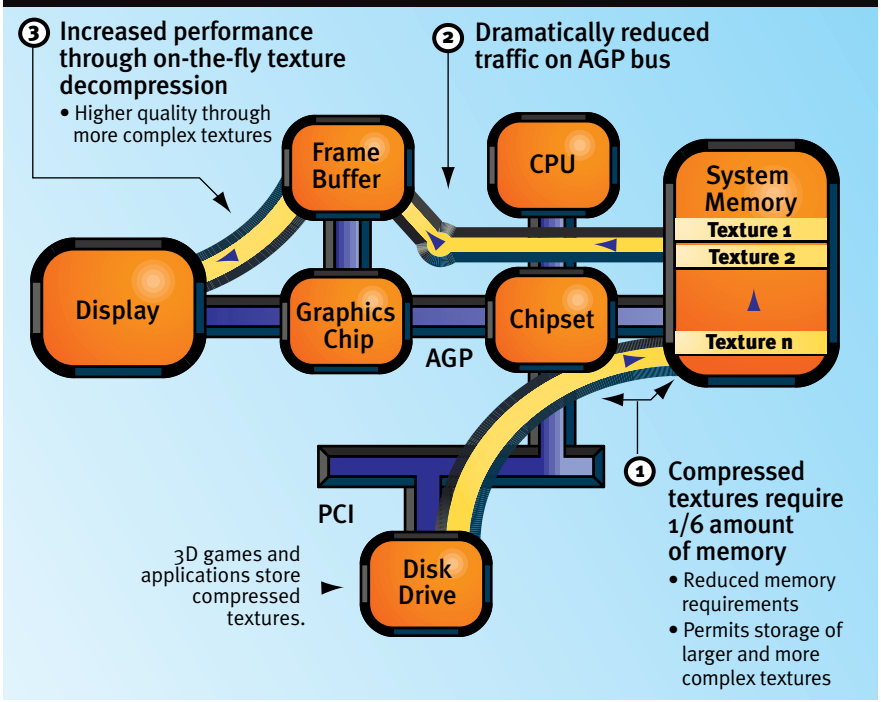
As discussed so far, texture compression can improve overall image quality without impacting performance. In fact, texture compression should actually boost performance significantly. While MIP-maps were mainly invented to eliminate texture aliasing artifacts, they also happen to increase the performance of your rendering hardware quite a bit. With MIP-mapping, texture fetches are very localized. For that reason, your renderer can use a much higher percentage of data read to generate subsequent pixels that will need texels from the same area of the texture. Also, because the texture fetches are localized, your application can read data in larger bursts with fewer page breaks in memory. Such an implementation increases the effective bandwidth over the AGP bus, system memory, or frame buffer — if the texture happens to be there. Without MIP-mapping, accesses to texture memory become random, wrecking havoc on bus and memory efficiency.

Data bandwidth, particularly texture read bandwidth, will often be the limiting factor in achieving high, sustained fill rates. Getting around these limiting factors and achieving high, sustained fill rates is what we're all really after, as high paper numbers don't do much to speed real applications.

TRIPLE-BUFFERED RENDERING. Texture compression also frees up memory that you can use for triple-buffering. If you're double buffering to synchronize buffer swaps with vertical retrace to avoid tearing, you're probably aware of the engine stalls that this method causes. Triple buffering can eliminate these engine stalls. Triple buffering can also boost frame rates, especially as frame rates increase and the cost of synchronizing buffer swaps with vertical retrace increases. While switching to triple buffering can result in a 30 percent frame rate increase, it does so at the expense of increased frame buffer requirements. But if you've compressed the texture data, you should already have this memory available.

IMPROVED SUSTAINED FILL RATE. The amount of memory that your graphics engine can transfer in a given unit of time is bounded by system design. One of the factors limiting sustained fill rates is the speed at which textures can be fetched from memory. Compressed texture maps consume less bandwidth (only 25 to 33 percent of the bandwidth required for uncompressed data) and therefore can be fetched faster. The bottom line is that compressing texture maps will result in faster rendering.

FIGURE 1. *S3 Texture compression (S3TC) is the DirectX 6 standard.*

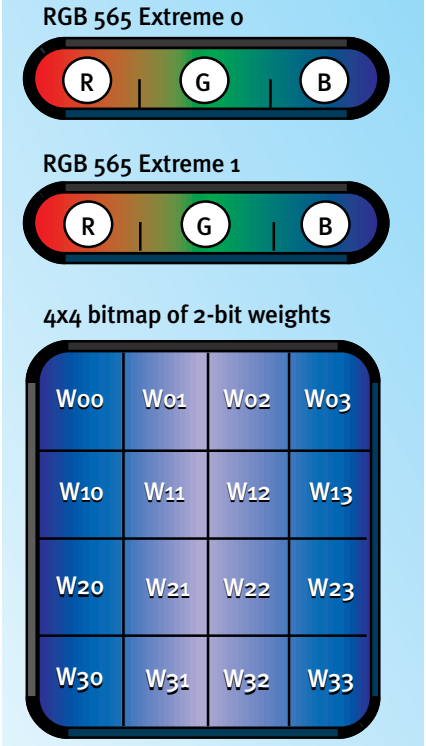


How It Works

S3TC compresses textures to a fixed size of 4 bits per texel for opaque textures or 8 bits per texel for textures requiring more than 1 bit of transparency. An image rendered with compressed textures is virtually indistinguishable from an image rendered with the original uncompressed texture maps. This image quality is the primary reason Microsoft adopted S3TC as the basis for DirectX's texture compression.

The compression scheme breaks a texture map into 4×4 blocks of 16 pixels. For texture maps with just 1 bit of transparency or none, each texel is represented by a 2-bit index in a 4×4 bitmap, for

FIGURE 2. *A 4×4 pixel block encoded as two 16-bit color extrema, and a bitmap of 2-bit interpolation weights.*



a total of 32 bits. In addition to the 4×4×2 bitmap, each block has two 16-bit colors in RGB565 format. From these two explicitly encoded colors, S3TC derives two additional colors, yielding a four-color lookup table. The system then uses these 2-bit indices to look up the texel color in this table. In total, the 16 texels are encoded using 64 bits, for an average of 4 bits per texel. The same scheme can also support 1 bit of transparency. When a 4×4 block includes transparent texels, the two encoded colors are swapped to indicate that the block has just three colors — one of the bitmap encodings (11) indicates a transparent texel. The third color is derived differently in this case. (Figure 2).

If your game makes uses of more sophisticated transparency, you can encode an additional 64 bits for transparency information in each 4×4 block. S3TC provides two mechanisms to encode complex transparency effects: it either explicitly encodes the 4 most significant bits of each pixel in the alpha channel in a 4-bit-per-pixel bitmap, or it employs a linear interpolation scheme similar to that used for color encoding. With the explicitly encoded variant, you can capture additional transparency information by dithering the alpha channel prior to truncating it to 4 bits per pixel. One great thing about this scheme is that the blocks are completely self-contained and no additional data needs to be fetched to decode the 16 texels in a block. No code book, for example, is required, as in vector quantization schemes. This advantage is important, because having to do two fetches to decode a texel is a serious performance problem. Additionally, you won't need to hassle with managing code books or palettes.

Simple Decoding Hardware

Decoding compressed texture blocks is a simple process and, therefore, very inexpensive to implement in hardware. This simplicity lends itself to very fast implementations and a straightforward approach to replicating decoding logic in order to decode multiple pixels in parallel.

S3TC's simple decoding scheme is able to achieve high-quality results by computing a linear approximation of the color space in a small block. Recall Taylor series mathematics, which states that any function can be adequately approximated over a small interval by the first two terms in the Taylor series: the constant term and the linear term. This is precisely what S3TC does in the color space of the block.

Using S3TC Texture Compression

Using S3TC texture compression in your application is very simple as it's directly supported by DirectX 6. Your artists create artwork using the same tools that they've used in the past. You perform a one-time compression as you create your distribution medium. Then, when your software run time loads the compressed texture map, a simple modification of your existing code suffices to load the compressed texture.

Although decompressing the encoded texture map is a simple matter, compressing it properly is a complex task that can be time consuming if you want the maximum possible quality. Therefore, you'll most likely be compressing your texture maps when you create them (or at least, before you place them on your distribution medium). To assist you with this conversion, S3 has made several compression tools available on *Game Developer* magazine's web site. Even if you don't use precompressed textures in your application, S3TC's fast encoder can still deliver 95 percent of the expected image quality.

If you use Adobe Photoshop to create or manipulate your artwork, you can use the S3TC Photoshop plug-in (S3TC.8BI) to extract compressed texture maps seamlessly from that graphics tool. This plug-in lets you open and save S3TC files as if they were native to Photoshop. This approach is the best one to take, especially if you're using a relatively uncommon image format for your textures.

On the other hand, if you're using an image editor other than Photoshop, you can create and view S3TC texture map files with our standalone utility, S3TC.EXE. This utility accepts a number of commonly used image formats, such as .BMP, .JPG, .TIF, and .TGA. It also allows you to create compressed texture map files with or without MIP-maps. With either the standalone utility or the Photoshop plug-in, creating and viewing a compressed texture map is straightforward and unobtrusive to your workflow.

If you don't want to compress at author time, you have several alternatives. You can compress during your game's installation, when the game is started, or when levels are loaded. These alternatives are possible, because DirectX 6 will include an API to compress your texture maps at run time. Bear in mind, however, that run-time compression isn't as fast, so we encourage you to compress off-line whenever possible. Lastly, because Microsoft will have an API to decompress textures very quickly in software, there's no danger of having problems with hardware that doesn't have built-in decompression support.

Choosing the Optimal Compression Level

Several variants of S3TC are supported within DirectX 6, depending on the level of transparency support that your application needs. Each of these formats has its own Four-Character Code (FOURCC) that you use to create the texture map surface.

Microsoft has defined five new FOURCCs. If your texture map is completely opaque, uses only 1 bit of alpha, or uses color-key transparency, you should be using FOURCC DXT1. This format is the most compact representation of the new compressed texture map formats. It lets you switch, at the block level, between fully opaque blocks and blocks with minimal transparency. Each block of 16 pixels is encoded in 8 bytes for an average of 4 bits per pixel.

If your texture maps have more complex transparency effects, you can use one of the DXT2, DXT3, DXT4, or DXT5 formats. All of these formats use an additional 8 bytes to encode transparency information, for a total of 16 bytes per block or an average of 8 bits per pixel. DXT2 and DXT3 explicitly encode alpha information by capturing the 4 most significant bits of the alpha channel. Dithering on the alpha channel can also increase the effective number of bits that are represented. You would use DXT2 when your transparency has premultiplied alpha (which Microsoft is advocating for the latest release of DirectX) and DXT3 for the more traditional nonpremultiplied alpha representation. DXT4 and DXT5 represent the transparency channel using a 3-bit linear-interpolation scheme similar to that which encodes color information. Again, use DXT4 for premultiplied alpha and DXT5 for nonpremultiplied alpha. DXT1 is by far the most useful format because it compresses down to 4 bits per texel and provides excellent color resolution and 1 alpha bit. The entire texture map must be classified by a single FOURCC code. Allocating a compressed texture map surface couldn't be simpler — do what you've been doing previously, but use the new FOURCC code instead.

Helpful for Internet-based Games

Texture-map compression is useful whenever memory size or bandwidth are an issue (all the time). One obvious application of this technology is transferring texture maps or images over the Internet. If you're creating VRML worlds for walkthroughs on the Internet, you'll want to use texture maps to provide visual detail to the objects. While the bandwidth of a local graphics system is already a concern, network bandwidth of Internet applications is an even more critical consideration. Using compressed textures in your VRML world will increase their user friendliness and increase the acceptance of VRML for your clients. And with Microsoft's Chrome project racing to completion, you can expect to see more mixing of 2D and 3D graphics on the same web page for a unified look across all graphics elements. Expect to benefit from texture compression in this environment as well.

Texture compression saves memory and bandwidth, and you'll find that taking advantage of S3TC within DirectX 6 is trivial. You can easily implement its simple decoder in hardware, so you'll likely be seeing that functionality on chips soon enough. Regardless of the hardware support in the short term, the support for fast decompression built into the DirectX 6 API makes this format a reliable solution. It will work on all platforms beginning with DirectX 6.

LISTING 1. *Handling S3TC compressed textures.*

Step 1: Load compressed texture from the file.

```
{
      DDSURFACEDESC ddsd;
      DWORD dwfilecode,dwBodySize;
      BYTE* body;
      FILE* Fp=fopen("test.s3t","rb");
      fread(&dwfilecode,sizeof(DWORD),Fp);              // Skip the filecode
      fread(&ddsd,sizeof(DDSURFACEDESC),Fp);            // Loaded the surface descriptor
      fread(&dwBodySize,sizeof(DWORD),Fp);              // Get the size of the texture
      body = (BYTE*)malloc(dwBodySize*sizeof(BYTE));    // allocate texture memory
      fread(body,dwBodySize,Fp);                        // Read the body
}
```

Step 2: Create the texture surface.

```
{
      LPDIRECTDRAWSURFACE     lpdds;
      // Assume that your DirectDraw interface is represented by lpDD
      lpDD->CreateSurface(&ddsd,&lpdds,NULL);           // Not exactly - verify

      // Fallback strategy
      // If CreateSurface fails due to lack of video memory
      // OR the DDSCAPS_NONLOCALVIDMEM flag to ddsd.ddsCaps.dwFlags
      // call CreateSurface again to create the surface in AGP memory
}
```

Step 3: Load the compressed data onto the texture surface.

```
{
      lpdds->Lock(NULL,&ddsd,NULL,NULL);
      memcpy(ddsd.lpSurface,body,dwBodySize);
      lpdds->Unlock();
}
```

Sample Code

Sample code to load and use compressed S3TC textures in DirectX is presented in Listing 1. With that code, you' re locked and loaded. Use this texture surface anywhere within your Direct3D game or application. Check the S3TC web site for more information about how to use S3TC in DirectX 6 and in the OpenGL S3TC extensions. ■

FOR FURTHER INFO

S3 Inc.
http://www.s3.com

DirectX 6
http://www.microsoft.com/directx/pavilion/default.asp

Intel's AGP
http://developer.intel.com/pc-supp/platform/agfxport/AGP_FAQ.HTM