

S3 Texture Compression

S3 Texture Compression or S3TC is a compression scheme for three or four color channel textures. This functionality is exposed by the ubiquitous extension `EXT_texture_compression_s3tc`. There are 3 forms of S3TC allowed by OpenGL.

S3 Texture Compression

EXT `EXT_texture_compression_s3tc` (http://www.opengl.org/registry/specs/EXT/texture_compression_s3tc.txt)

Contents

Compression Format

[DXT1 Format](#)

[DXT1 with 1-bit Alpha](#)

[DXT3 Format](#)

[DXT5 Format](#)

sRGB and S3TC

View textures and S3TC

See also

Compression Format

S3TC is a technique for compressing images for use as textures. Standard image compression techniques like JPEG and PNG can achieve greater compression ratios than S3TC. However, S3TC is designed to be implemented in high-performance hardware. JPEG and PNG decompress images all-at-once, while S3TC allows specific sections of the image to be decompressed independently.

S3TC is a block-based format. The image is broken up into 4x4 blocks. For non-power-of-two images that aren't a multiple of 4 in size, the other colors of the 4x4 block are taken to be black. Each 4x4 block is independent of any other, so it can be decompressed independently.

There are 3 forms of S3TC accepted by OpenGL. These forms are named after the old Direct3D names for these formats: DXT1, DXT3 and DXT5.

DXT1 Format

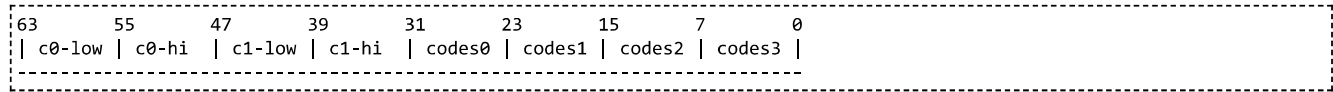
A DXT1-compressed image is an RGB image format. As such, the alpha of any color is assumed to be 1. Each 4x4 block takes up 64-bits of data, so compared to a 24-bit RGB format, it provides 6:1 compression. You can get a DXT1 image by using the `GL_COMPRESSED_RGB_S3TC_DXT1_EXT` as the internal format of the image.

Each 4x4 block stores color data as follows. There are 2 16-bit color values, `color0` followed by `color1`. Following this is a 32-bit unsigned integer containing values that describe how the two colors are combined to determine the color for a given pixel.

The 2 16-bit color values are stored in little-endian format, so the low byte of the 16-bit color comes first in each case. The color values are stored in RGB order (from high bit to low bit) in 5_6_5 bits.

The 32-bit unsigned integer is also stored in little-endian format. Every 2 bits of the integer represent a pixel; the 2 bits are a code that defines how to combine `color0` and `color1` to produce the color of that pixel. In order from highest bit to lowest bit (after the little-endian conversion), the pixels are stored in row-major order. Every 8 bits, 4 2-bit codes, is a single row of the image.

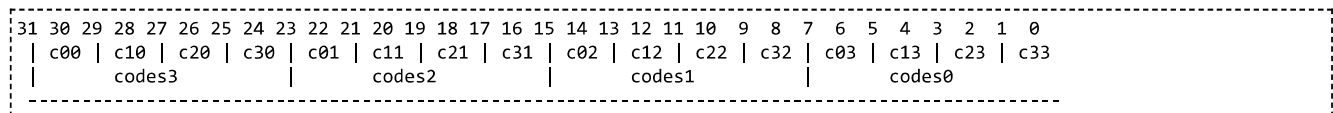
Here is a diagram of the setup:



c0-low is the low byte of the 16-bit color 0; similarly, *c0-hi* is the high byte of color 0. To reconstitute color 0, simply do this: $((\text{bytes}[1] \ll 8) + \text{bytes}[0])$, where `bytes` is the an array containing the above sequence of bytes. Color 1 would come from $((\text{bytes}[3] \ll 8) + \text{bytes}[2])$.

Similarly, the *codes* are the bytes that make up the 32-bit integer bitcodes. They have to be rebuilt in reverse order, so *codes3* is the left-most byte.

Once rebuilt into their proper order, you can get the individual 2-bit values like this. The The pixel values are in cXY form, where Y goes bottom to top as is standard for OpenGL:



The interpretation of the 2-bit values depends on how `color0` and `color1` compare to each other. If the integer value of `color0` is greater than `color1`, then the 2-bit values mean something different than if `color0` is less than or equal to `color1`. The meaning of the 2-bit values is as follows:

code	<code>color0 > color1</code>	<code>color0 <= color1</code>
0	<code>color0</code>	<code>color0</code>
1	<code>color1</code>	<code>color1</code>
2	$(2 * \text{color0} + \text{color1}) / 3$	$(\text{color0} + \text{color1}) / 2$
3	$(\text{color0} + 2 * \text{color1}) / 3$	Black

The arithmetic operations are done per-component, not on the integer value of the colors. And the value "Black" is simply R=G=B=0.

DXT1 with 1-bit Alpha

There is a form of DXT1 available that provides a simple on/off alpha value. This format therefore uses an RGBA base format. To get this format, use the `GL_COMPRESSED_RGBA_S3TC_DXT1_EXT` internal format.

The format of the data is identical to the above case, which is why this is still DXT1 compression. The interpretation differs slightly. You always get an alpha value of 1 *unless* the pixel uses the code for Black in the above table. In that case, you get an alpha value of 0.

Note that this means that the RGB colors will also be 0 on any pixel with a 0 alpha. This also means that bilinear filtering between neighboring texels will result in colors combined with black. If you are using premultiplied alpha blending, this is what you want. If you aren't, then it almost certainly is not what you want.

When using OpenGL to compress a texture, the GL implementation will assume any pixel with an alpha value < 0.5 should have an alpha of 0. This is another reason to manually compress images.

DXT3 Format

The DXT3 format is an RGBA format. Each 4x4 block takes up 128 bits of data. Thus, compared to a 32-bit RGBA texture, it offers 4:1 compression. You can get this with the `GL_COMPRESSED_RGBA_S3TC_DXT3_EXT` internal format.

Each block of 128 bits is broken into 2 64-bit chunks. The *second* chunk contains the color information, compressed almost as in the DXT1 case; the difference being that color0 is always assumed to be less than color1 in terms of determining how to use the codes to extract the color value. The first chunk contains the alpha information.

The alpha 64-bit chunk is stored as a little-endian 64-bit unsigned integer. The alpha values are stored as 4-bit-per-pixel alpha values. The alpha values are stored in row-major order, from the highest bit of the 64-bit unsigned integer.

DXT5 Format

The DXT5 format is an alternate RGBA format. As in the DXT3 case, each 4x4 block takes up 128 bits. So it provides the same 4:1 compression as in the DXT3 case. You can get this with the `GL_COMPRESSED_RGBA_S3TC_DXT5_EXT` format.

Just as for the DXT3 format, there are two 64-bit chunks of data per block: an RGB chunk compressed as for DXT1 (with the same caveat as for DXT3), and an alpha chunk. Again the *second* chunk is the color chunk; the first is the alpha.

Where DXT3 and DXT5 differ is how the alpha chunk is compressed. DXT5 compresses the alpha using a compression scheme similar to DXT1.

The alpha data is stored as 2 8-bit alpha values, α_0 and α_1 , followed by a 48-bit unsigned integer that describes how to combine these two reference alpha values to achieve the final alpha value. The 48-bit integer is also stored in little-endian order.

The 48-bit unsigned integer contains 3-bit codes that describe how to compute the final alpha value. These codes are stored in the identical order as the codes in DXT1; they simply are 3 bits in size rather than 2.

Just as in the DXT1 case, the codes have different meanings depending on how α_0 and α_1 compare to one another. Here is the table of codes and computations:

code	alpha0 > alpha1	alpha0 <= alpha1
0	alpha0	alpha0

1	alpha1	alpha1
2	$(6*\alpha_0 + 1*\alpha_1)/7$	$(4*\alpha_0 + 1*\alpha_1)/5$
3	$(5*\alpha_0 + 2*\alpha_1)/7$	$(3*\alpha_0 + 2*\alpha_1)/5$
4	$(4*\alpha_0 + 3*\alpha_1)/7$	$(2*\alpha_0 + 3*\alpha_1)/5$
5	$(3*\alpha_0 + 4*\alpha_1)/7$	$(1*\alpha_0 + 4*\alpha_1)/5$
6	$(2*\alpha_0 + 5*\alpha_1)/7$	0.0
7	$(1*\alpha_0 + 6*\alpha_1)/7$	1.0

sRGB and S3TC

Images compressed with S3 texture compression can also be in the sRGB colorspace. This extension does not qualify as a ubiquitous extension, but it is widely supported nevertheless.

As with any sRGB operation, the question arises as to whether the two key colors (which are in the sRGB colorspace) are converted to a linear colorspace before the decompression step or after. While converting before decompression is preferred, the specification allows implementations to do the colorspace conversion afterwards the decompression. Doing so means linearly interpolating the color values, which generally should not be done in a non-linear colorspace like sRGB.

In general, hardware that is incapable of GL 3.0 or above will do the colorspace conversion after decompression, while 3.0 or better hardware (aka: Direct3D 10-capable or better) will do the conversion before decompression.

sRGB S3 Texture Compression

EXT [EXT_texture_sRGB](http://www.opengl.org/registry/specs/EXT/texture_compression_sRGB.txt) (http://www.opengl.org/registry/specs/EXT/texture_compression_sRGB.txt)

View textures and S3TC

The `glTextureView` defines which image formats between source and destination textures are valid. However, because S3TC is not core, their behavior is not defined in the `EXT_texture_compression_s3tc` (http://www.opengl.org/registry/specs/EXT/texture_compression_s3tc.txt) extension. It is instead defined by the `ARB_texture_view` (http://www.opengl.org/registry/specs/ARB/texture_view.txt) extension, which extends the compatibility table with these entries:

S3TC texture view compatibility

Class	Internal formats
GL_S3TC_DXT1_RGB	GL_COMPRESSED_RGB_S3TC_DXT1_EXT, GL_COMPRESSED_SRGB_S3TC_DXT1_EXT
GL_S3TC_DXT1_RGBA	GL_COMPRESSED_RGBA_S3TC_DXT1_EXT, GL_COMPRESSED_SRGB_ALPHA_S3TC_DXT1_EXT
GL_S3TC_DXT3_RGBA	GL_COMPRESSED_RGBA_S3TC_DXT3_EXT, GL_COMPRESSED_SRGB_ALPHA_S3TC_DXT3_EXT
GL_S3TC_DXT5_RGBA	GL_COMPRESSED_RGBA_S3TC_DXT5_EXT, GL_COMPRESSED_SRGB_ALPHA_S3TC_DXT5_EXT

See also

- Khronos Data Format Specification section on S3TC formats (<https://www.khronos.org/registry/DataFormat/specs/1.1/dataformat.1.1.html#S3TC>)
-

Retrieved from "http://www.khronos.org/opengl/wiki/opengl/index.php?title=S3_Texture_Compression&oldid=13899"

This page was last edited on 1 July 2017, at 02:36.