A Method for Load-Time Conversion of DXTC Assets to ATC

Ray Ratelis, John Bergman

Guild Software, Inc.

Abstract

Most games developed for PC and Console tend to store their textures in the game-industry standard DXTC (S3TC) format. In this paper we present a methodology for load-time conversion of DXTC assets into the ATC format, suitable for loading on Qualcomm Snapdragon hardware, featuring the Adreno series of mobile GPUs. Due to the relative similarities between the texture formats, this process does not incur significant CPU overhead. All types of DXTC textures may be successfully converted on load-time, including "DXT5N" alpha+green compressed normal maps. Some slight artifacts may also be introduced during conversion, due to the green channel differences, which we describe. For most game usage, we consider the artifacts to be acceptable. This conversion method is currently in production use by Vendetta Online.

1. Introduction

Game developers planning to bring existing PC and console titles to Android face a number of different challenges, including the proliferation of proprietary texture compression standards. DXTC, or DirectX Texture Compression (aka S3TC) is the game industry standard on Windows, Mac, Linux, Xbox and Playstation, etc. Artists may spend hundreds of hours tweaking their DXTC textures to get the best balance of efficiency and visual fidelity. Programmers may spend considerable time optimizing the usage of memory and rendering efficiency with specialized uses of RGBA channels, and testing different fallback cases for shaders that use the assets.

All of this can conspire to make the process of asset conversion a tedious one, filled with time spent checking and re-checking fallback cases for quality assurance, increasing the cost of development. In addition, for those titles that periodically patch their assets to update or improve the game, multiple asset packs must be maintained with separate on-going patches. For an MMORPG in particular, this can be especially burdensome.

It would be vastly preferable if DXTC could be broadly supported across all platforms, Android included. At the time of this writing, a mixture of concern over patent status and licensing costs appears to have prevented this from taking place in the mobile electronics industry. We can only hope that this might change in the future. For the moment, we offer the following solution for the specific case of *Qualcomm Snapdragon* chips, using the ATC compression format.

This conversion method was developed using only the public documentation of the ATC format made available by Chainfire on the XDA-Developers forum. No reverse engineering or NDA documentation was utilized in creating this technique. See the related public data here:

http://forum.xda-developers.com/showthread.php?t=437777

2. Format Information

The DXTC and ATC formats are very similar, which lends itself to this conversion. In particular:

- The first color is 555 for ATC, vs 565 for DXTC.
- The Most Significant Bit (MSB) of the first color for ATC defines the decoding method.
- The 2 bit value for each pixel is different, but syntactically the same.
- The Alpha part is identical between the two formats.

ATC		**DXTC**	
(Alpha data)		*(Alpha data)*	
XRRRRRGG	GGGBBBBB	RRRRRGGG	GGGBBBBB
RRRRRGGG	GGGBBBBB	RRRRRGGG	GGGBBBBB
xxyyxxyy	xxyyxxyy	xxyyxxyy	xxyyxxyy
xxyyxxyy	xxyyxxyy	xxyyxxyy	xxyyxxyy

ATC	**DXTC**
Where: • Bytes 0-1 are color0 (555) • Bytes 1-2 are color1 (565) • X is the decoding method • xx and yy are the 2 bit values for each of the 16 pixels (4x4 block).	***Where*** • Bytes 0-1 are color0 (565) • Bytes 1-2 are color1 (565) • xx and yy are the 2 bit values for each of the 16 pixels (4x4 block)
If x is 0 then the 2 bit value is the interpolator between color0 and color1. • 00 is rgb=color0 • 01 is rgb=color0 + (1/3)*(color1-color0) • 10 is rgb=color0 + (2/3)*(color1-color0) • 11 is rgb=color1	***The 2 bit value is an out-of-order interpolator between color0 and color1, if color0 is greater than color1:*** • 00 is rgb=color0 • 01 is rgb=color1 • 10 is rgb=color0 + (1/3)*(color1-color0) • 11 is rgb=color0 + (2/3)*(color1-color0)
If x is 1 then the 2 bit value is defined as: • 00 is rgb=0,0,0 • 01 is rgb=color0 - color1/4 • 10 is rgb=color0 • 11 is rgb=color1	***If color0 is less than color1:*** • 00 is rgb=color0 • 01 is rgb=color1 • 10 is rgb=(color0 + color1)/2 • 11 is rgb=0,0,0

3. Conversion Process

For DXTC color0>color1, the ATC decode method 0 is the most similar:

- The color0 value can be switched from 565 to 0555 (with a small loss of precision in the green component, tinting the color either towards green or away from green by one bit).
- The color1 value is identical.
- The 2 bit color values can have their bits swapped from DXTC to ATC like this:
    - 00 -> 00
    - 01 -> 11
    - 10 -> 01
    - 11 -> 10
- For DXT3/ATC_RGBA_EXPLICIT_ALPHA and DXT5/ATC_RGBA_INTERPOLATED_ALPHA, the alpha part is the same so no conversion needs to be done.

This doesn't generate perfect results, some banding can be seen in smooth grey gradients, and an assumption is made in that color0 is greater than color1, but artifacts are minimal in practice.

4. Example Source Code

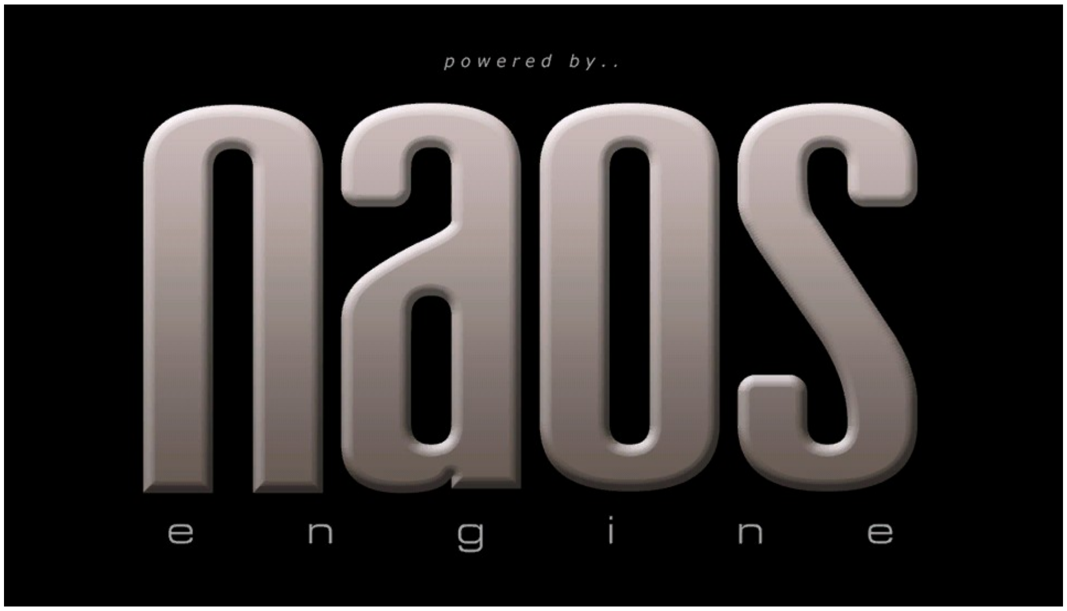
```
unsigned convertrgbbits(unsigned bits)
{
        /*

        dxtc where color_0 > color_1:
        00 ? color_0
        01 ? color_1
        10 ? 2/3 color_0 + 1/3 color_1
        11 ? 1/3 color_0 + 2/3 color_1

        dxtc where color_0 <= color_1:
        00 ? color_0
        01 ? color_1
        10 ? 1/2 color_0 + 1/2 color_1
        11 ? black (0,0,0)

        atc (method 0):
        00 ? color_0
        01 ? 2/3 color_0 + 1/3 color_1
        10 ? 1/3 color_0 + 2/3 color_1
        11 ? color_1

        atc (method 1):
        00 ? black (0,0,0)
        01 ? color_0 - 1/4 color_1
        10 ? color_0
        11 ? color_1

        */

        unsigned lut[4] = {0,3,1,2};
        unsigned a = lut[bits & 0x03];
        unsigned b = lut[(bits>>2) & 0x03];
        unsigned c = lut[(bits>>4) & 0x03];
        unsigned d = lut[(bits>>6) & 0x03];
        return a | (b<<2) | (c<<4) | (d<<6);
}

#pragma pack(1)
struct alphablock
{
        byte alpha0;
        byte alpha1;

        byte a[6];
};
struct DXT3alphablock
{
        byte b_a;
        byte d_c;
        byte f_e;
        byte h_g;
        byte j_i;
        byte l_k;
        byte n_m;
        byte p_o;
};
struct colorblock
{
        word color0;
        word color1;
        byte d_c_b_a;
        byte h_g_f_e;
        byte l_k_j_i;
        byte p_o_n_m;
};
struct DXT3block
{
        DXT3alphablock alpha;
        colorblock color;
};
struct DXT5block
{
        alphablock alpha;
        colorblock color;
};
#pragma pack()

// dxtc to atc conversion routines
// using info from http://forum.xda-developers.com/showthread.php?t=437777
void convertblock(colorblock *block)
{
        // change color values so color_0 is 555 and color_1 is 565
        // change bits so new 00 = 00, 01 = 11, 10 = 10, 11 = 01
        // MSB of color_0 is method (0 or 1) describing how to interpolate.

        // The only problem with this is that green is represented with less bits
for color 0 on ATITC so things tend to band a little and have a green/anti-green
tint.
```

```
        // I was thinking of doing some dithering but dithering would need to be
done on a 4x4 block so it would look like a checkerboard.


        unsigned color0b = (block->color0&0x01f);
        unsigned color0g = (block->color0&0x07C0)>>1;
        unsigned color0r = (block->color0&0xF800)>>1;

        block->color0 = color0r | color0g | color0b;

        block->d_c_b_a = convertrgbbits(block->d_c_b_a);
        block->h_g_f_e = convertrgbbits(block->h_g_f_e);
        block->l_k_j_i = convertrgbbits(block->l_k_j_i);
        block->p_o_n_m = convertrgbbits(block->p_o_n_m);

}


void convert_texture(int sourcefFormat, int destinationFormat, void *data)
{
        if(sourceFormat == DXT1 && destinationFormat == GL_ATC_RGB_AMD) {
                // change color values so color_0 is 555 and color_1 is 565
                // change bits so new 00 = 00, 01 = 11, 10 = 10, 11 = 01
                int s = max(1, w / 4) * max(1, h / 4);
                colorblock *blocks = (colorblock*)data;
                for(int i=0;i<s;i++) {
                        convertblock(blocks+i);
                }
        }
        else if(sourceFormat == DXT3 && destinationFormat ==
GL_ATC_RGBA_EXPLICIT_ALPHA_AMD) {
                // change color values so color_0 is 555 and color_1 is 565
                // change bits so new 00 = 00, 01 = 11, 10 = 10, 11 = 01
                int s = max(1, w / 4) * max(1, h / 4);
                DXT3block *blocks = (DXT3block*)data;
                for(int i=0;i<s;i++) {
                        convertblock(&(blocks[i].color));
                }
        }
        else if(sourceFormat == DXT5 && destinationFormat ==
GL_ATC_RGBA_INTERPOLATED_ALPHA_AMD) {
                // change color values so color_0 is 555 and color_1 is 565
                // change bits so new 00 = 00, 01 = 11, 10 = 10, 11 = 01
                int s = max(1, w / 4) * max(1, h / 4);
                DXT5block *blocks = (DXT5block*)data;
                for(int i=0;i<s;i++) {
                        convertblock(&(blocks[i].color));
                }
        }
}
```

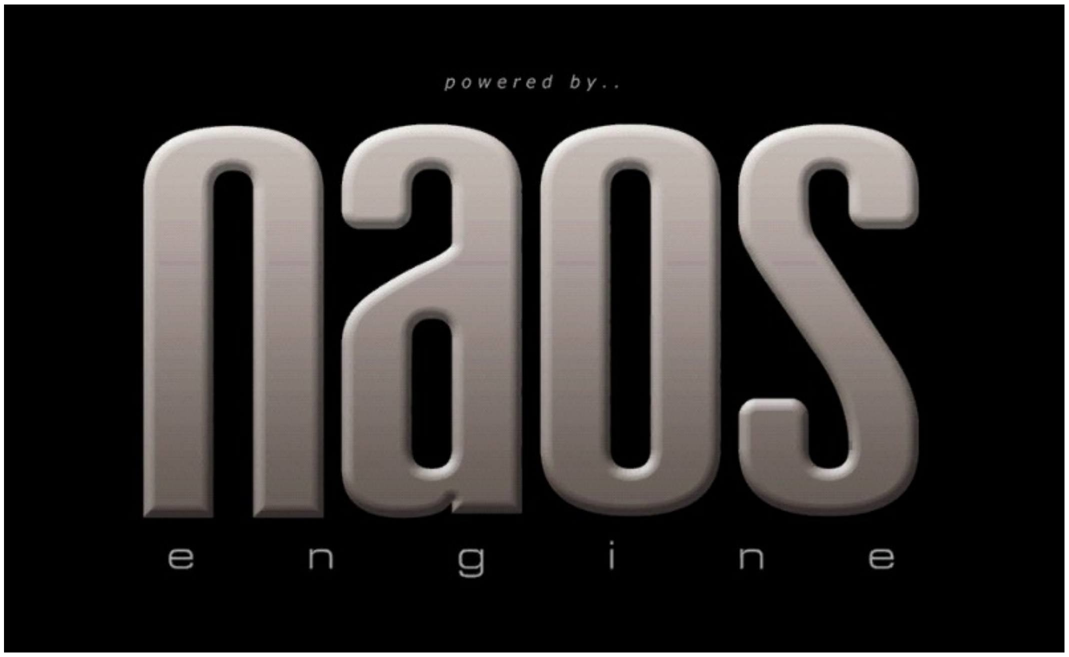
6. Conversion Artifact Examples

We consider the resulting artifacts to be effectively minimal and well within the bounds of most game fidelity. However, there is some banding present in smooth gradients. For most other cases, the differences are invisible. The following logo graphic displays the worst-case-scenario that we have seen for conversion artifacts:

Natively loaded DXTC texture, 16bit rendering, NVIDIA Tegra 3 (T30):



Converted DXTC texture, 16bit rendering, Qualcomm Snapdragon MSM8255 (Adreno 205):



7. Conclusion

We find the above methodology to be of significant use for our title, *Vendetta Online*. Its utility is somewhat specific to the Android platform, but in cases where a DXTC asset pack must be utilized, it can result in access to a valuable expanded market share of devices. We hope that widespread, patent-free acceptance of DXTC will eventually supplant techniques like this one.

For those who do not require DXTC asset usage, we currently (March 1st, 2012) recommend conversion to the Ericsson Texture Compression (ETC1) standard, as this is far more broadly applicable across the Android platform, including GPUs that support no other compression methodology (ARM Mali, etc).

For further information, we can be reached via our company website:

www.guildsoftware.com

GUILD SOFTWARE