Real-Time DXT Compression

May 20th 2006
J.M.P. van Waveren

© 2006, Id Software, Inc.

Abstract

S3TC also known as DXT is a lossy texture compression format with a fixed compression ratio. The DXT format is designed for real-time decompression in hardware on graphics cards, while compressing textures to DXT format may take a considerable amount of time. However, at the cost of a little quality a DXT compressor can be optimized to achieve real-time performance. The DXT compressor described in this paper is optimized using the Intel Multi Media Extensions and the Intel Streaming SIMD Extensions 2. The presented optimizations allow textures to be compressed real-time which is useful for textures created procedurally at run time or textures streamed from a different format. Furthermore a texture compression scheme is presented which can be used in combination with the real-time DXT compressor to achieve high quality real-time texture compression.
1. Introduction

Textures are digitized images drawn onto geometric shapes to add visual detail. In today's computer graphics a tremendous amount of detail is mapped onto geometric shapes during rasterization. Not only textures with colors are used but also textures specifying surface properties like specular reflection or fine surface details in the form of normal or bump maps. All these textures can consume large amounts of system and video memory. Fortunately compression can be used to reduce the amount of memory required to store textures. Most of today's graphics cards allow textures to be stored in a variety of compressed formats that are decompressed in real-time during rasterization. One such format which is supported by most graphics cards is S3TC also known as DXT compression \([1, 2]\).

The DXT format is designed for real-time decompression in hardware on the graphics card during rendering. DXT is a lossy compression format with a fixed compression ratio of 4:1 or 8:1. DXT compression is a form of Block Truncation Coding (BTC) where an image is divided into non-overlapping blocks and the pixels in each block are quantized to a limited number of values. The color values of pixels in a 4x4 pixel block are approximated with equidistant points on a line through color space. This line is defined by two end points and for each pixel in the 4x4 block an index is stored to one of the equidistant points on the line. The end points of the line through color space are quantized to 16-bit 5:6:5 RGB format and either one or two intermediate points are generated through interpolation. The DXT1 format allows an 1-bit alpha channel to be encoded by using only one intermediate point and specifying one color which is black and fully transparent. The DXT2 and DXT3 formats encode a separate explicit 4-bit alpha value for each pixel in a 4x4 block. The DXT4 and DXT5 formats store a separate alpha channel which is compressed similarly to the color channels where the alpha values in a 4x4 block are approximated with equidistant points on a line through alpha space. In the DXT2 and DXT4 formats it is assumed that the color values have been premultiplied by the alpha values. In the DXT3 and DXT5 formats it is assumed that the color values are not premultiplied by the alpha values. In other words the data and interpolation methods are the identical for DXT2 and DXT3, and DXT4 and DXT5 respectively. However, the format name specifies whether or not the color values are assumed to be premultiplied with the alpha values.

DXT compressed textures do not only require significantly less memory on the graphics card, they generally also render faster than uncompressed textures because of reduced bandwidth requirements. Some quality may be lost due to the DXT compression. However, when the same amount of memory is used on the graphics card there is generally a significant gain in quality compared to using textures that are only 1/4th or 1/8th the resolution.

The image on the left below is an original 256x256 RGB image (= 256 kB) of Lena. The image in the middle is first resized down to 128x128 (= 64 kB) and then resized back up to 256x256 with bilinear filtering. The image on the right is first compressed to DXT1 (=
32 kB) with the ATI Compressonator [3], and then decompressed with a regular DXT1 decompressor. The images clearly show that the DXT1 compressed image takes up less space and maintains a lot more detail than the 128x128 image.

DXT1 compression with the ATI Compressonator compared to low a resolution image

![Images showing compression comparison](image)

<table>
<thead>
<tr>
<th>Resolution</th>
<th>RGB</th>
<th>DXT1</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x256</td>
<td>256 kB</td>
<td>32 kB</td>
</tr>
<tr>
<td>128x128</td>
<td>64 kB</td>
<td></td>
</tr>
</tbody>
</table>

Most textures are typically created by artists, either drawn by hand or rendered from high detail models. These textures can be compressed off-line to a format supported by the graphics card. However, textures can also be created procedurally through scripts or mathematical formulas during graphics rendering. Such textures cannot be compressed off-line and either have to be sent to the graphics card uncompressed or have to be compressed on the fly.

The compression formats supported by today's graphics cards are designed for real-time decompression in hardware and may as such not result in the best possible compression ratio. Graphics applications may use vast amounts of texture data that is not displayed all at once but streamed from disk as the view point moves or the rendered scene changes. Different forms of compression (JPEG, MPEG-4, H.264) may be required to deal with such vast amounts of texture data to keep storage and bandwidth requirements within acceptable limits. Once these textures are streamed and decompressed they either have to be sent to the graphics card uncompressed or have to be compressed in real-time, just like procedurally created textures.

The DXT format is designed for real-time decompression while high quality compression may take a considerable amount of time. However, at the cost of a little quality a DXT compressor can be optimized to achieve real-time performance. The DXT compressor described in this paper is optimized using the Intel Multi Media Extensions (MMX) and the Intel Streaming SIMD Extensions 2 (SSE2). The presented optimizations allow textures to be compressed real-time.
1.1 Previous Work

There are several good DXT compressors available. Most notably there are the ATI Compressonator [3] and the nVidia DXT Library [4]. Both compressors produce high quality DXT compressed images. However, these compressors are not open source and they are optimized for high quality off-line compression and are too slow for real-time use. There is an open source DXT compressor available by Roland Scheidegger for the Mesa 3D Graphics Library [5]. Although this compressor is a little bit faster it is still too slow for real-time texture compression. Another good DXT compressor is Squish by Simon Brown [6]. This compressor is open source but it is also optimized for high quality off-line compression and is typically too slow for real-time use.
2. DXT1 Compression

DXT1 compression is block based and each color in a block can be considered a point in a 3-dimensional space of red, green and blue. DXT1 compression approximates the pixels in a 4x4 pixel block with equidistant points on a line through this color space.

First a 4x4 block of pixels is extracted from the texture for improved cache usage and to make it easier to implement optimized routines that work on a fixed size block as opposed to a variable sized texture. Next the compressor searches for the best line through color space which can be used to approximate the pixels in the 4x4 block. For each original pixel of the 4x4 block the compressor then searches for a point on the line that best approximates the pixel color and emits an index to that point. The following code gives an overview of the DXT1 compressor.

```c
typedef unsigned char byte;
typedef unsigned short word;
typedef unsigned int dword;

byte *globalOutData;

void CompressImageDXT1( const byte *inBuf, byte *outBuf, int width, int height, int &outputBytes ) {
    ALIGN16( byte block[64] );
    ALIGN16( byte minColor[4] );
    ALIGN16( byte maxColor[4] );

    globalOutData = outBuf;
    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {
            ExtractBlock( inBuf + i * 4, width, block );
            GetMinMaxColors( block, minColor, maxColor );
            EmitWord( ColorTo565( maxColor ) );
            EmitWord( ColorTo565( minColor ) );
            EmitColorIndices( block, minColor, maxColor );
        }
    }

    outputBytes = globalOutData - outBuf;
}
```

The above compressor expects 'inBuf' to point to an input image in 4-byte RGBA format. The 'outBuf' should point to a buffer large enough to store the DXT file. The 'width' and 'height' specify the size of the input image in pixels and 'outputBytes' will be set to the number of bytes written to the DXT file. The function 'ExtractBlock' extracts a 4x4 block from the texture and stores it in a fixed size buffer. This function is implemented as follows.

```c
void ExtractBlock( const byte *inPtr, int width, byte *colorBlock ) {
    for ( int j = 0; j < 4; j++ ) {
        memcpy( &colorBlock[j*4*4], inPtr, 4*4 );
        inPtr += width * 4;
    }
}
```
SIMD optimized versions of 'ExtractBlock' can be found in appendix A. The function 'GetMinMaxColors' finds the two end points of the line through color space. This function is described in section 2.1. The code below performs the conversion from 24-bit 8:8:8 RGB format to 16-bit 5:6:5 RGB format.

```c
word ColorTo565( const byte *color ) {
    return ( ( color[0] >> 3 ) << 11 ) | ( ( color[1] >> 2 ) << 5 ) | ( color[2] >> 3 );
}
```

The function 'EmitColorIndices' finds the best point on the line through color space for each original pixel in the 4x4 block and emits the indices. This function is described in section 2.2. The following functions are used to emit bytes, words and double words in little endian format to the DXT file in memory.

```c
void EmitByte( byte b ) {
    globalOutData[0] = b;
    globalOutData += 1;
}
void EmitWord( word s ) {
    globalOutData[0] = ( s >> 0 ) & 255;
    globalOutData[1] = ( s >> 8 ) & 255;
    globalOutData += 2;
}
void EmitDoubleWord( dword i ) {
    globalOutData[0] = ( i >> 0 ) & 255;
    globalOutData[1] = ( i >> 8 ) & 255;
    globalOutData[2] = ( i >> 16 ) & 255;
    globalOutData[3] = ( i >> 24 ) & 255;
    globalOutData += 4;
}
```

The above compressor does not encode an 1-bit alpha channel and always uses 2 intermediate points on the line through color space. There are no special cases for when 'ColorTo565( minColor ) >= ColorTo565( maxColor )' because 'GetMinMaxColors' is implemented such that 'ColorTo565( minColor )' can never be larger than 'ColorTo565( maxColor )' and if 'ColorTo565( minColor ) == ColorTo565( maxColor )' then 'EmitColorIndices' will still properly write out the indices.
2.1 Finding a Line Through Color Space

During DXT compression the basic problem is to find a best fit line through color space which can be used to approximate the pixels in a 4x4 block. Many different best fit line algorithms can be used to find such lines through color space. A technique called principle component analysis can find the direction along which the points in color space vary the most. This direction is called the principle axis and a line along this axis generally provides a good basis for approximating the points in color space. However, calculating the principal axis and points on this axis is expensive. Another approach is to use the two colors from the 4x4 block that are furthest apart as the end points of the line through color space. This approach is shown in the following code.

```c
int ColorDistance( const byte *c1, const byte *c2 ) {
    return  ( ( c1[0] - c2[0] ) * ( c1[0] - c2[0] ) ) +
}

void SwapColors( byte *c1, byte *c2 ) {
    byte tm[3];
    memcpy( tm, c1, 3 );
    memcpy( c1, c2, 3 );
    memcpy( c2, tm, 3 );
}

void GetMinMaxColors( const byte *colorBlock, byte *minColor, byte *maxColor ) {
    int maxDistance = -1;
    for ( int i = 0; i < 64 - 4; i += 4 ) {
        for ( int j = i + 4; j < 64; j += 4 ) {
            int distance = ColorDistance( &colorBlock[i], &colorBlock[j] );
            if ( distance > maxDistance ) {
                maxDistance = distance;
                memcpy( minColor, colorBlock+i, 3 );
                memcpy( maxColor, colorBlock+j, 3 );
            }
        }
    }
    if ( ColorTo565( maxColor ) < ColorTo565( minColor ) ) {
        SwapColors( minColor, maxColor );
    }
}
```

The above routine finds the two colors that are furthest apart based on the euclidean distance. Another approach is to take the two colors that are furthest apart based on the luminance which is considerably faster. The following routine shows how this can be implemented.
int ColorLuminance( const byte *color ) {
    return ( color[0] + color[1] * 2 + color[2] );
}

void GetMinMaxColors( const byte *colorBlock, byte *minColor, byte *maxColor ) {
    int maxLuminance = -1, minLuminance = MAX_INT;
    for ( i = 0; i < 16; i++ ) {
        int luminance = ColorLuminance( colorBlock+i*4 );
        if ( luminance > maxLuminance ) {
            maxLuminance = luminance;
            memcpy( maxColor, colorBlock+i*4, 3 );
        }
        if ( luminance < minLuminance ) {
            minLuminance = luminance;
            memcpy( minColor, colorBlock+i*4, 3 );
        }
    }
    if ( ColorTo565( maxColor ) < ColorTo565( minColor ) ) {
        SwapColors( minColor, maxColor );
    }
}

The above algorithms do generally not find the best fit line through color space and are still too slow for real-time use due to a lot of branching. The branches are typically hard to predict and result in numerous mispredictions and significant penalties on today's CPUs that implement a deep pipeline. When a branch is mispredicted, the misprediction penalty is usually equal to the depth of the pipeline [7, 8].

Yet another approach is to use the extents of the bounding box of the color space of the pixels in a 4x4 block for the end points of the approximating line. This is generally not the best fit line and some quality is lost, but in practice the points on the line through the bounding box extents give fairly good approximations of the colors in a 4x4 pixel block. Calculating the bounding box extents is very fast because it is no more than a sequence of min/max instructions. The following code shows how this is implemented in regular C.

#define INSET_SHIFT 4       // inset the bounding box with ( range >> shift )

void GetMinMaxColors( const byte *colorBlock, byte *minColor, byte *maxColor ) {
    int i;
    byte inset[3];
    for ( i = 0; i < 16; i++ ) {
        if ( colorBlock[i*4+0] < minColor[0] ) { minColor[0] = colorBlock[i*4+0]; }
        if ( colorBlock[i*4+1] < minColor[1] ) { minColor[1] = colorBlock[i*4+1]; }
        if ( colorBlock[i*4+0] > maxColor[0] ) { maxColor[0] = colorBlock[i*4+0]; }
        if ( colorBlock[i*4+1] > maxColor[1] ) { maxColor[1] = colorBlock[i*4+1]; }
    }
    inset[0] = ( maxColor[0] - minColor[0] ) >> INSET_SHIFT;
}
The above code does not only calculate the bounding box but also insets the bounding box with 1/16th of its size. This slightly improves the quality because it lowers the overall Root Mean Square (RMS) error. SIMD optimized code for 'GetMinMaxColors' can be found in appendix B. The above C code uses conditional branches but the SIMD optimized routines in appendix B use the MMX / SSE2 instructions 'pminub' and 'pmaxub' to get the minimum and maximum of the color channels and as such avoid all conditional branches.
2.2 Finding Matching Points On The Line Through Color Space

For each original pixel from the 4x4 block the compressor needs to emit the index to the point on the line through color space that best approximates the color of the original pixel. A straightforward approach is to calculate the squared euclidean distance between an original color and each of the points on the line through color space and choose the point that is closest to the original color. The following routine shows how this can be implemented.

```c
#define C565_5_MASK         0xF8    // 0xFF minus last three bits
#define C565_6_MASK         0xFC    // 0xFF minus last two bits

void EmitColorIndices( const byte *colorBlock, const byte *minColor, const byte *maxColor ) {
    byte colors[4][4];
    unsigned int indices[16];

    colors[0][0] = ( maxColor[0] & C565_5_MASK ) | ( maxColor[0] >> 5 );
    colors[0][1] = ( maxColor[1] & C565_6_MASK ) | ( maxColor[1] >> 6 );
    colors[0][2] = ( maxColor[2] & C565_5_MASK ) | ( maxColor[2] >> 5 );
    colors[1][0] = ( minColor[0] & C565_5_MASK ) | ( minColor[0] >> 5 );
    colors[1][1] = ( minColor[1] & C565_6_MASK ) | ( minColor[1] >> 6 );
    colors[1][2] = ( minColor[2] & C565_5_MASK ) | ( minColor[2] >> 5 );

    colors[2][0] = ( 2 * colors[0][0] + 1 * colors[1][0] ) / 3;
    colors[2][1] = ( 2 * colors[0][1] + 1 * colors[1][1] ) / 3;
    colors[2][2] = ( 2 * colors[0][2] + 1 * colors[1][2] ) / 3;
    colors[3][0] = ( 1 * colors[0][0] + 2 * colors[1][0] ) / 3;
    colors[3][1] = ( 1 * colors[0][1] + 2 * colors[1][1] ) / 3;
    colors[3][2] = ( 1 * colors[0][2] + 2 * colors[1][2] ) / 3;

    for ( int i = 0; i < 16; i++ ) {
        unsigned int minDistance = INT_MAX;
        for ( int j = 0; j < 4; j++ ) {
            unsigned int dist = ColorDistance( &colorBlock[i*4], &colors[j][0] );
            if ( dist < minDistance ) {
                minDistance = dist;
                indices[i] = j;
            }
        }
    }

    dword result = 0;
    for ( int i = 0; i < 16; i++ ) {
        result |= ( indices[i] << (unsigned int)( i << 1 ) );
    }

    EmitDoubleWord( result );
}
```

The above routine first calculates the 4 colors on the line through color space. The high order bits of the minimum and maximum color are replicated to the low order bits the same way the graphics card converts the 16-bit 5:6:5 RGB format to 24-bit 8:8:8 RGB format.

Instead of the euclidean distance some compressors use a weighted distance between colors to improve the perceived flat image quality. However, the DXT compressed textures are typically used for 3D rendering where filtering, blending and lighting
Calculations may considerably change the way images are perceived. As a result, improving the flat image quality may very well not improve the rendered image quality. Therefore it is often better to minimize the straight RGB error.

To calculate the squared euclidean distance between colors the MMX or SSE2 instruction 'pmaddwd' could be used. However, this instruction operates on words while the color channels are stored as bytes. The color bytes would first have to be converted to words and subtracted before using the 'pmaddwd' instruction. Instead of calculating the squared euclidean distance, the sum of absolute differences can be calculated directly with the 'psadbw' instruction because this instruction operates on bytes. This obviously does not produce the same result but using the sum of absolute differences also works well to minimize the overall RGB error and most of all it is fast.

The innermost loop in the 'EmitColorIndices' routine above suffers from a lot of branch mispredictions. These conditional branches, however, can be avoided by calculating a color index directly from the results of comparing the sums of absolute differences. To avoid the branches the innermost loop is unrolled and the following C code calculates the four sums of absolute differences.

```c
int c0 = colorBlock[i*4+0];
int c1 = colorBlock[i*4+1];
int c2 = colorBlock[i*4+2];
int d0 = abs( colors[0][0] - c0 ) + abs( colors[0][1] - c1 ) + abs( colors[0][2] - c2 );
int d1 = abs( colors[1][0] - c0 ) + abs( colors[1][1] - c1 ) + abs( colors[1][2] - c2 );
int d2 = abs( colors[2][0] - c0 ) + abs( colors[2][1] - c1 ) + abs( colors[2][2] - c2 );
int d3 = abs( colors[3][0] - c0 ) + abs( colors[3][1] - c1 ) + abs( colors[3][2] - c2 );
```

After calculating the four sums of absolute differences the results can be compared with each other. There are 4 sums that need to be compared which results in 6 comparisons as follows.

```c
int b0 = d0 > d2;
int b1 = d1 > d3;
int b2 = d0 > d3;
int b3 = d1 > d2;
int b4 = d0 > d1;
int b5 = d2 > d3;
```

A color index in the DXT format can take four different values and is represented by a 2-bit binary number. The following table shows the correlation between the results of the comparisons and the binary numbers.

<table>
<thead>
<tr>
<th>index</th>
<th>binary</th>
<th>expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>!b0 &amp; !b2 &amp; !b4</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>!b1 &amp; !b3 &amp; b4</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>b0 &amp; b3 &amp; !b5</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>b1 &amp; b2 &amp; b5</td>
</tr>
</tbody>
</table>
Using the above table, each individual bit of the 2-bit binary number can be derived from the results of the comparisons. This results in the following expression.

\[
\text{result} = ( \neg b3 \land b4 ) \lor ( b2 \land b5 ) \lor (( ( b0 \land b3 ) \lor ( b1 \land b2 ) ) \ll 1);
\]

Evaluating the above expression reveals that the sub expression \(( \neg b3 \land b4 )\) can be omitted because it does not significantly contribute to the final result.

\[
\text{result} = ( b2 \land b5 ) \lor (( ( b0 \land b3 ) \lor ( b1 \land b2 ) ) \ll 1);
\]

The above can then be rewritten to the following.

```c
int b0 = d0 > d3;
int b1 = d1 > d2;
int b2 = d0 > d2;
int b3 = d1 > d3;
int b4 = d2 > d3;
int x0 = b1 & b2;
int x1 = b0 & b3;
int x2 = b0 & b4;
result = x2 | ( ( x0 | x1 ) \ll 1 );
```

The end result is the following routine which calculates and emits the color indices in a single loop without conditional branches.

```c
void EmitColorIndices( const byte *colorBlock, const byte *minColor, const byte *maxColor ) {
    word colors[4][4];
    dword result = 0;
    colors[0][0] = ( maxColor[0] & C565_5_MASK ) | ( maxColor[0] >> 5 );
    colors[0][1] = ( maxColor[1] & C565_6_MASK ) | ( maxColor[1] >> 6 );
    colors[0][2] = ( maxColor[2] & C565_5_MASK ) | ( maxColor[2] >> 5 );
    colors[1][0] = ( minColor[0] & C565_5_MASK ) | ( minColor[0] >> 5 );
    colors[1][1] = ( minColor[1] & C565_6_MASK ) | ( minColor[1] >> 6 );
    colors[1][2] = ( minColor[2] & C565_5_MASK ) | ( minColor[2] >> 5 );
    colors[2][0] = ( 2 * colors[0][0] + 1 * colors[1][0] ) / 3;
    colors[2][1] = ( 2 * colors[0][1] + 1 * colors[1][1] ) / 3;
    colors[2][2] = ( 2 * colors[0][2] + 1 * colors[1][2] ) / 3;
    colors[3][0] = ( 1 * colors[0][0] + 2 * colors[1][0] ) / 3;
    colors[3][1] = ( 1 * colors[0][1] + 2 * colors[1][1] ) / 3;
    colors[3][2] = ( 1 * colors[0][2] + 2 * colors[1][2] ) / 3;

    for ( int i = 15; i >= 0; i-- ) {
        int c0 = colorBlock[i*4+0];
        int c1 = colorBlock[i*4+1];
        int c2 = colorBlock[i*4+2);
        int d0 = abs( colors[0][0] - c0 ) + abs( colors[0][1] - c1 ) + abs( colors[0][2] - c2 );
        int d1 = abs( colors[1][0] - c0 ) + abs( colors[1][1] - c1 ) + abs( colors[1][2] - c2 );
        int d2 = abs( colors[2][0] - c0 ) + abs( colors[2][1] - c1 ) + abs( colors[2][2] - c2 );
    }
}
```
```c
int d3 = abs( colors[3][0] - c0 ) + abs( colors[3][1] - c1 ) + abs( colors[3][2] - c2 );
int b0 = d0 > d3;
int b1 = d1 > d2;
int b2 = d0 > d2;
int b3 = d1 > d3;
int b4 = d2 > d3;
int x0 = b1 & b2;
int x1 = b0 & b3;
int x2 = b0 & b4;
result |= ( x2 | ( ( x0 | x1 ) << 1 ) ) << ( i << 1 );
}
EmitDoubleWord( result );
```

SIMD optimized code for 'EmitColorIndices' can be found in appendix C. To calculate the two intermediate points on the line through color space the above function uses integer divisions by three. However, there is no instruction for integer divisions in the MMX or SSE2 instruction sets. Instead of using a division the same result can be calculated with a fixed point multiplication [9]. This is also known as a weak form of operator strength reduction [10]. The 'pmulhw' instruction can be used to multiply two word integers while only the high word of the double word result is stored. The instruction 'pmulhw x, y' is equivalent to the following code in C.

```c
x = ( x * y ) >> 16;
```

The division 'x / 3' can be calculated by setting:

```c
y = ( 1 << 16 ) / 3 + 1;
```

The 'psadbw' instruction is used to calculate the sums of absolute differences. The 'pcmpgtw' instruction is used to compare the sums of absolute differences and several 'pand' and 'por' instruction are used to calculate the index from the results of the comparisons. The MMX routine calculates four indices per iteration while the SSE2 version calculates eight indices per iteration.
3. DXT5 Compression

In the DXT5 format an additional alpha channel is compressed separately in a similar way as the DXT1 color channels. For each 4x4 block of pixels the compressor searches for a line through alpha space and for each pixel in the block the alpha channel value is approximated by one of the equidistant points on the line. The end points of the line through alpha space are stored as bytes and either 4 or 6 intermediate points are generated through interpolation. For the case with 4 intermediate points two additional points are generated, one for fully opaque and one for fully transparent.

```c
byte *globalOutData;

void CompressImageDXT5( const byte *inBuf, byte *outBuf, int width, int height, int &outputBytes ) {
    ALIGN16( byte block[64] );
    ALIGN16( byte minColor[4] );
    ALIGN16( byte maxColor[4] );

    globalOutData = outBuf;
    for ( int j = 0; j < height; j += 4, inBuf += width * 4*4 ) {
        for ( int i = 0; i < width; i += 4 ) {
            ExtractBlock( inBuf + i * 4, width, block );
            GetMinMaxColors( block, minColor, maxColor );
            EmitByte( maxColor[3] );
            EmitByte( minColor[3] );
            EmitAlphaIndices( block, minColor[3], maxColor[3] );
            EmitWord( ColorTo565( maxColor ) );
            EmitWord( ColorTo565( minColor ) );
            EmitColorIndices( block, minColor, maxColor );
        }
    }
    outputBytes = globalOutData - outBuf;
}
```

The function 'GetMinMaxColors' is extended to not only find the two end points of the line through color space, but also the end points of the line through alpha space. The function 'EmitAlphaIndices' finds the best point on the line through alpha space for each original pixel in the 4x4 block and emits the indices. The compression of the color channels is the same as the DXT1 compression described in the previous section. The above compressor does not explicitly encode extreme values for fully opaque and fully transparent. The compressor always uses 6 intermediate points on the line through alpha space. There are no special cases for when 'minColor[3] >= maxColor[3]' because 'GetMinMaxColors' is implemented such that 'minColor[3]' can never be larger than 'maxColor[3]' and if 'minColor[3] == maxColor[3]' then 'EmitAlphaIndices' will still properly write out the indices.
3.1 Finding a Line Through Alpha Space

Just like for colors the bounding box extents of alpha space are used for the end points of the line through alpha space for a 4x4 block. However, the alpha channel is a one-dimensional space and as such this approach does not introduce any additional loss of quality for the alpha channel. The following implementation of 'GetMinMaxColors' has been extended to also calculate the minimum and maximum alpha value for a 4x4 block of pixels.

```c
#define INSET_SHIFT 4       // inset the bounding box with ( range >> shift )

void GetMinMaxColors( const byte *colorBlock, byte *minColor, byte *maxColor ) {
    int i;
    byte inset[4];


    for ( i = 0; i < 16; i++ ) {
        if ( colorBlock[i*4+0] < minColor[0] ) { minColor[0] = colorBlock[i*4+0]; }
        if ( colorBlock[i*4+1] < minColor[1] ) { minColor[1] = colorBlock[i*4+1]; }
        if ( colorBlock[i*4+3] < minColor[3] ) { minColor[3] = colorBlock[i*4+3]; }
        if ( colorBlock[i*4+0] > maxColor[0] ) { maxColor[0] = colorBlock[i*4+0]; }
        if ( colorBlock[i*4+1] > maxColor[1] ) { maxColor[1] = colorBlock[i*4+1]; }
    }

    inset[0] = ( maxColor[0] - minColor[0] ) >> INSET_SHIFT;

    minColor[0] = ( minColor[0] + inset[0] <= 255 ) ? minColor[0] + inset[0] : 255;

    maxColor[0] = ( maxColor[0] >= inset[0] ) ? maxColor[0] - inset[0] : 0;
}
```

SIMD optimized code for GetMinMaxColors can be found in appendix B. The MMX / SSE2 instructions 'pminub' and 'pmaxub' are used to get the minimum and maximum respectively.
3.2 Finding Matching Points On The Line Through Alpha Space

For each original pixel from the 4x4 block the compressor needs to emit the index to the point on the line through alpha space that best approximates the alpha value of the original pixel. The alpha space is a one-dimensional space and it is straightforward to choose the point on the line through alpha space that is closest to the original alpha value. The following routine shows how this can be implemented.

```c
void EmitAlphaIndices( const byte *colorBlock, const byte minAlpha, const byte maxAlpha )
{
    byte indices[16];
    byte alphas[8];

    alphas[0] = maxAlpha;
    alphas[1] = minAlpha;
    alphas[2] = ( 6 * maxAlpha + 1 * minAlpha ) / 7;
    alphas[3] = ( 5 * maxAlpha + 2 * minAlpha ) / 7;
    alphas[4] = ( 4 * maxAlpha + 3 * minAlpha ) / 7;
    alphas[5] = ( 3 * maxAlpha + 4 * minAlpha ) / 7;
    alphas[6] = ( 2 * maxAlpha + 5 * minAlpha ) / 7;
    alphas[7] = ( 1 * maxAlpha + 6 * minAlpha ) / 7;

colorBlock += 3;

    for ( int i = 0; i < 16; i++ ) {
        int minDistance = INT_MAX;
        byte a = colorBlock[i*4];

        for ( int j = 0; j < 8; j++ ) {
            int dist = abs( a - alphas[j] );
            if ( dist < minDistance ) {
                minDistance = dist;
                indices[i] = j;
            }
        }
    }

    EmitByte( (indices[ 0] >> 0) | (indices[ 1] << 3) | (indices[ 2] << 6) );
}
```

The above routine suffers from a lot of branch mispredictions. Because the alpha space is a one-dimensional space and the points on the line through alpha space are equidistant the closest point for each original alpha value could be calculated through a division. However, integer division is rather slow and there are no MMX or SSE2 instructions available for integer division. Calculating the division with a fixed point multiplication with the 'pmulhw' instruction would require a lookup table because the divisor is a variable and not a constant. Furthermore the 'pmulhw' and similar instructions operate on words which means only 4 operations are executed in parallel using MMX code and 8 using SSE2 code. Instead of branching or an integer division it is also possible to
calculate the correct index to the closest point on the line through alpha space by comparing the original alpha value with a set of crossover points and adding the results of the comparisons.

The alpha values in the 4x4 block are approximated by 8 equidistant points on the line through alpha space. There are 7 crossover points between these 8 equidistant points where a given alpha value goes from being closest to one point to another. These crossover points can be calculated from the minimum and maximum alpha value that define the line through alpha space as follows.

```c
byte mid = ( maxAlpha - minAlpha ) / ( 2 * 7 );
byte ab1 = minAlpha + mid;
byte ab2 = ( 6 * maxAlpha + 1 * minAlpha ) / 7 + mid;
byte ab3 = ( 5 * maxAlpha + 2 * minAlpha ) / 7 + mid;
byte ab4 = ( 4 * maxAlpha + 3 * minAlpha ) / 7 + mid;
byte ab5 = ( 3 * maxAlpha + 4 * minAlpha ) / 7 + mid;
byte ab6 = ( 2 * maxAlpha + 5 * minAlpha ) / 7 + mid;
byte ab7 = ( 1 * maxAlpha + 6 * minAlpha ) / 7 + mid;
```

A given alpha value can now be compared to these crossover points on the line through alpha space.

```c
int b1 = ( a <= ab1 );
int b2 = ( a <= ab2 );
int b3 = ( a <= ab3 );
int b4 = ( a <= ab4 );
int b5 = ( a <= ab5 );
int b6 = ( a <= ab6 );
int b7 = ( a <= ab7 );
```

An index can be derived by adding the results of these comparisons. However, the points on the line through alpha space are not listed in natural order in the DXT format. For the case with 8 interpolated values the maximum and minimum values are listed first and then the 6 intermediate points from the maximum to the minimum are listed as follows.

```
max  min  6  5  4  3  2  1
```

Adding the results of the comparisons results in the following order.

```
max  6  5  4  3  2  1  min
```

Adding one and clipping the index with a logical 'and' with 7 results in the following order.

```
min  max  6  5  4  3  2  1
```

Now index value 0 maps to the minimum value and index value 1 maps to the maximum value. However, in the DXT format index value 0 maps to the maximum and index value
1 maps to the minimum. An 'exclusive or' with the result of the comparison (2 > index) can be used to swap index value 0 and index value 1. This results in the correct order and the following expression.

```c
int index = (b1 + b2 + b3 + b4 + b5 + b6 + b7 + 1) & 7;
index = index ^ (2 > index);
```

The end result is the following routine which calculates and emits the alpha indices.

```c
void EmitAlphaIndices(const byte *colorBlock, const byte minAlpha, const byte maxAlpha) {
    assert(maxAlpha > minAlpha);

    byte indices[16];
    byte mid = (maxAlpha - minAlpha) / (2 * 7);
    byte ab1 = minAlpha + mid;
    byte ab2 = (6 * maxAlpha + 1 * minAlpha) / 7 + mid;
    byte ab3 = (5 * maxAlpha + 2 * minAlpha) / 7 + mid;
    byte ab4 = (4 * maxAlpha + 3 * minAlpha) / 7 + mid;
    byte ab5 = (3 * maxAlpha + 4 * minAlpha) / 7 + mid;
    byte ab6 = (2 * maxAlpha + 5 * minAlpha) / 7 + mid;
    byte ab7 = (1 * maxAlpha + 6 * minAlpha) / 7 + mid;

    colorBlock += 3;
    for (int i = 0; i < 16; i++) {
        byte a = colorBlock[i*4];
        int b1 = (a <= ab1);
        int b2 = (a <= ab2);
        int b3 = (a <= ab3);
        int b4 = (a <= ab4);
        int b5 = (a <= ab5);
        int b6 = (a <= ab6);
        int b7 = (a <= ab7);
        int index = (b1 + b2 + b3 + b4 + b5 + b6 + b7 + 1) & 7;
        indices[i] = index ^ (2 > index);
    }
}
```

SIMD optimized code for EmitAlphaIndices can be found in appendix D. Just like the division by 3 as described in section 2.2 the division by 7 is calculated with a fixed point multiplication. The 'pmulhw' instruction is used with a constant of ((1 << 16) / 7 + 1). The division by 14 is also calculated with the 'pmulhw' instruction but with a constant of ((1 << 16) / 14 + 1). The inner loop in the above routine only operates on bytes which allows maximum parallelism to be exploited through SIMD instructions. The MMX version operates on 8 pixels from the 4x4 block at a time. The SSE2 version operates on all 16 pixels from the 4x4 block in one pass.
Unfortunately there are no instructions in the MMX or SSE2 instruction sets for comparing unsigned bytes. Only the 'pcmpeqb' instruction will work on both signed and unsigned bytes. However, the 'pminub' instruction does work with unsigned bytes. To evaluate a less than or equal relationship the 'pminub' instruction can be used followed by the 'pcmpeqb' instruction because the expression \( x \leq y \) is equivalent to the expression \( \min(x, y) == x \).
4. High Quality DXT Compression

The DXT5 format can be used in many ways for different purposes. A well known example is DXT5 compression of swizzled normal maps [11, 12]. The DXT5 format can also be used for high quality compression of color images by using the YCoCg color space. The YCoCg color space was first introduced for H.264 compression [13, 14]. Conversion to the YCoCg color space provides a coding gain over the RGB or the YCbCr color space by reducing the dynamic range with minimal computational complexity.

High quality DXT5 compression can be achieved by converting the RGB data to CoCg_Y. In other words the luminance (Y) is stored in the alpha channel and the chrominance (CoCg) is stored in the first two of the 5:6:5 color channels. For color images this results in a 3:1 or 4:1 compression ratio (compared to 8:8:8 RGB or 8:8:8:8 RGBA respectively) and the quality is very good and generally better than 4:2:0 JPEG at the highest quality setting.

The image on the left below is an original 256x256 RGB image (= 256 kB) of Lena. The image in the middle is first resized down to 128x128 (= 64 kB) and then resized back up to 256x256 with bilinear filtering. The image on the right is first compressed to CoCg_Y DXT5 (= 64 kB) with a custom high quality DXT5 compressor, and then decompressed with a regular DXT5 decompressor.
The CoCg_Y DXT5 compressed image shows no noticeable loss in quality and consumes one fourth the amount of the memory of the original image. The CoCg_Y DXT5 also looks a lot better than the lower resolution image of 128x128 which consumes the same amount of memory. The image above is compressed with a custom high quality DXT5 compressor but the SIMD optimized DXT5 compressor described above can also be used for this kind of high quality DXT compression.

Obviously CoCg_Y color data is retrieved in a fragment program and some work is required to perform the conversion back to RGB. However, the conversion to RGB is rather simple:

\[
\begin{align*}
R &= Y + Co - Cg \\
G &= Y + Cg - 0.5 \\
B &= Y - Co - Cg + 1
\end{align*}
\]

This conversion should be no more than three instructions in a fragment program. Furthermore, filtering and other calculations can often be done in YCoCg space.
5. Results

The following tables shows the performance of the SIMD optimized DXT compressors on an Intel 2.8 GHz dual-core Xeon and an Intel 2.9 GHz Core 2 Duo. The 256x256 Lena image used throughout this paper is also used for all the following performance tests. The different compressors are setup to compress the Lena image to either DXT1 or DXT5 as fast as possible without generating mip maps. For the DXT5 compression the blue channel from the Lena image is replicated to the alpha channel.

The following table shows the number of Mega Pixels that can be compressed to DXT1 format per second (higher MP/s = better). The table also shows the Root Mean Square (RMS) error of the RGB channels of the compressed image compared to the original image.

<table>
<thead>
<tr>
<th>compressor</th>
<th>RMS</th>
<th>MP/s 1</th>
<th>MP/s 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATI Compressionator Library</td>
<td>5.67</td>
<td>0.17</td>
<td>0.34</td>
</tr>
<tr>
<td>nVidia DXT Library</td>
<td>6.32</td>
<td>2.27</td>
<td>1.23</td>
</tr>
<tr>
<td>Mesa S3TC Compression Library</td>
<td>5.31</td>
<td>5.46</td>
<td>10.63</td>
</tr>
<tr>
<td>Squish DXT Compression Library</td>
<td>5.57</td>
<td>1.80</td>
<td>4.03</td>
</tr>
<tr>
<td>C optimized</td>
<td>5.28</td>
<td>22.22</td>
<td>35.45</td>
</tr>
<tr>
<td>MMX optimized</td>
<td>5.28</td>
<td>96.37</td>
<td>194.53</td>
</tr>
<tr>
<td>SSE2 optimized</td>
<td>5.28</td>
<td>112.05</td>
<td>200.62</td>
</tr>
</tbody>
</table>

1 Intel 2.8 GHz Dual-Core Xeon ("Paxville" 90nm NetBurst microarchitecture)
2 Intel 2.9 GHz Core 2 Extreme ("Conroe" 65nm Core 2 microarchitecture)

The following table shows the number of Mega Pixels that can be compressed to DXT5 format per second (higher MP/s = better). The table also shows the Root Mean Square (RMS) error of the RGBA channels of the compressed image compared to the original image.

<table>
<thead>
<tr>
<th>compressor</th>
<th>RMS</th>
<th>MP/s 1</th>
<th>MP/s 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATI Compressionator Library</td>
<td>4.21</td>
<td>0.21</td>
<td>0.46</td>
</tr>
<tr>
<td>nVidia DXT Library</td>
<td>4.74</td>
<td>1.79</td>
<td>1.16</td>
</tr>
<tr>
<td>Mesa S3TC Compression Library</td>
<td>4.08</td>
<td>3.58</td>
<td>8.03</td>
</tr>
<tr>
<td>Squish DXT Compression Library</td>
<td>4.21</td>
<td>1.86</td>
<td>4.17</td>
</tr>
<tr>
<td>C optimized</td>
<td>4.04</td>
<td>15.50</td>
<td>26.18</td>
</tr>
<tr>
<td>MMX optimized</td>
<td>4.04</td>
<td>56.93</td>
<td>116.08</td>
</tr>
<tr>
<td>SSE2 optimized</td>
<td>4.04</td>
<td>66.43</td>
<td>127.55</td>
</tr>
</tbody>
</table>

1 Intel 2.8 GHz Dual-Core Xeon ("Paxville" 90nm NetBurst microarchitecture)
2 Intel 2.9 GHz Core 2 Extreme ("Conroe" 65nm Core 2 microarchitecture)
The ATI Compressonator Library 1.27 [3] (March 2006) is used with the following options.

```cpp
ATT_TC_CompressOptions::bUseChannelWeighting = false;
ATT_TC_CompressOptions::fWeightingRed = 1.0;
ATT_TC_CompressOptions::fWeightingGreen = 1.0;
ATT_TC_CompressOptions::fWeightingBlue = 1.0;
ATT_TC_CompressOptions::bUseAdaptiveWeighting = false;
ATT_TC_CompressOptions::bDXT1UseAlpha = false;
ATT_TC_CompressOptions::nAlphaThreshold = 255;
```

The nVidia DXT Library [4] (April 19th 2006) from the nVidia DDS Utilities April 2006 is used with the following compression options.

```cpp
nvCompressionOptions::quality = kQualityFastest;
nvCompressionOptions::bForceDXT1FourColors = true;
nvCompressionOptions::mipMapGeneration = kNoMipMaps;
```

The Squish DXT Compression Library [6] (April 2006) is compiled without SIMD code because the SIMD code does not actually seem to improve the performance. Furthermore the following options are used to get the best performance.

```cpp
squish::kColourRangeFit | squish::kColourMetricUniform
```

No additional parameters are set for the Mesa S3TC Compression Library [5] (May 2006). Note that the compression options for each compressor are chosen in an attempt to achieve the best performance, possibly at the cost of some quality. All compressors are compiled into one executable and except for the closed source libraries all code is generated with the same compiler optimizations.
The image on the left below is an original 256x256 RGB image (= 256 kB) of Lena. The image in the middle is first resized down to 128x128 (= 64 kB) and then resized back up to 256x256 with bilinear filtering. The image on the right is first compressed to DXT1 (= 32 kB) with the SIMD optimized DXT1 compressor described in this paper, and then decompressed with a regular DXT1 decompressor.

DXT1 compression with SIMD optimized compressor compared to low a resolution image

256x256 RGB = 256 kB
128x128 RGB = 64 kB
256x256 DXT1 = 32 kB

The DXT1 compressed image consumes 1/8th the memory of the original image. There is some loss in quality compared to a (slow) off-line compressor but overall the quality is still a lot better than the lower resolution image of 128x128 that consumes twice the amount of memory.
The image on the left below is an original 256x256 RGB image (= 256 kB) of Lena. The image in the middle is first resized down to 128x128 (= 64 kB) and then resized back up to 256x256 with bilinear filtering. The image on the right is first compressed to CoCg_Y DXT5 (= 64 kB) with the SIMD optimized DXT5 compressor described in this paper, and then decompressed with a regular DXT5 decompressor.

CoCg_Y DXT5 compression with SIMD optimized compressor compared to low a resolution image

256x256 RGB = 256 kB  
128x128 RGB = 64 kB  
256x256 CoCg_Y DXT5 = 64 kB

The CoCg_Y DXT5 compressed image consumes 1/4th the memory of the original image. There are a few color artifacts but no noticeable loss of detail. The CoCg_Y DXT5 image obviously maintains a lot more detail than the lower resolution image of 128x128 which consumes the same amount of memory.
6. Conclusion

DXT compression can provide improved texture quality by significantly lowering the memory requirements and as such allowing much higher resolution textures to be used. At the cost of a little quality a DXT compressor can be optimized to achieve real-time performance which is useful for textures created procedurally at run time or textures streamed from a different format. The SIMD optimized DXT compressor presented in this paper is a magnitude faster than currently available compressors. Furthermore CoCg_Y DXT5 compression can be used to trade memory for improved compressed texture quality while still keeping the overall memory requirements to a minimum.

7. Future Work

The CoCg_Y DXT5 compression trades memory for quality. It is also possible to trade performance for quality. Instead of using the bounding box extents for the end points of the line through color space an SIMD optimized algorithm could be used to calculate the principal axis. This is slower than calculating the bounding box extents but for some applications it may be worthwhile to trade the performance for quality.

8. References

1. S3 Texture Compression
   Pat Brown
   NVIDIA Corporation, November 2001
   Available Online: http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt

2. Compressed Texture Resources
   Microsoft Developer Network
   DirectX SDK, April 2006

3. ATI Compressor Library
   Seth Sowerby, Daniel Killebrew
   ATI Technologies Inc, The Compressor version 1.27.1066, March 2006
   Available Online: http://www.ati.com/developer/compressor.html

4. nVidia DXT Library
   nVidia
   nVidia DDS Utilities, April 2006
   Available Online: http://developer.nvidia.com/object/nv_texture_tools.html

5. Mesa S3TC Compression Library
   Roland Scheidegger
   libtxc_dxt version 0.1, May 2006
   Available Online: http://homepage.ghi.de/rscheidegger/dri_experimental/s3tc_index.html
6. Squish DXT Compression Library
   Simon Brown
   Squish version 1.8, September 2006
   Available Online: http://sjbrown.co.uk/?code=squish

7. Avoiding the Cost of Branch Misprediction
   Rajiv Kapoor
   Intel, December 2002

8. Branch and Loop Reorganization to Prevent Mispredicts
   Jeff Andrews
   Intel, January 2004

9. Division by Invariant Integers using Multiplication
   T. Granlund, P.L. Montgomery
   SIGPLAN Notices, Vol. 29, page 61, June 1994
   Available Online: http://www.swox.com/~tege/

10. Operator Strength Reduction
    K. Cooper, T. Simpson, C. Vick
    Programming Languages and Systems, n. 5, Vol. 23, pp. 603-625, September 1995
    Technical Report CRPC-TR95-635-S, Rice University, October 1995
    Available Online: http://www.cs.rice.edu/~keith/EMBED/OSR.pdf

11. Bump Map Compression
    Simon Green
    nVidia Technical Report, October 2001

12. Normal Map Compression
    ATI Technologies Inc
    ATI, August 2003
    Available Online: http://www.ati.com/developer/NormalMapCompression.pdf

13. Transform, Scaling & Color Space Impact of Professional Extensions
    H. S. Malvar, G. J. Sullivan

14. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range
    H. S. Malvar, G. J. Sullivan
    Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, Document No. JVT-I014r3, July 2003
void ExtractBlock_MMX( const byte *inPtr, int width, byte *colorBlock ) {
    __asm {
        mov         esi, inPtr
        mov         edi, colorBlock
        mov         eax, width
        shl         eax, 2
        movq        mm0, [esi+0]
        movq        [edi+ 0], mm0
        movq        mm1, [esi+8]        // + 4 * width
        movq        [edi+ 8], mm1
        movq        mm2, [esi+eax+0]        // + 4 * width
        movq        [edi+16], mm2
        movq        mm3, [esi+eax+8]        // + 4 * width
        movq        [edi+24], mm3
        movq        mm4, [esi+eax*2+0]       // + 8 * width
        movq        [edi+32], mm4
        movq        mm5, [esi+eax*2+8]       // + 8 * width
        add         esi, eax
        movq        [edi+40], mm5
        movq        mm6, [esi+eax*2+16]      // + 12 * width
        movq        [edi+48], mm6
        movq        mm7, [esi+eax*2+24]      // + 12 * width
        movq        [edi+56], mm7
        emms
    }
}

void ExtractBlock_SSE2( const byte *inPtr, int width, byte *colorBlock ) {
    __asm {
        mov         esi, inPtr
        mov         edi, colorBlock
        mov         eax, width
        shl         eax, 2
        movdqa      xmm0, [esi]
        movdqa      [edi+ 0], xmm0
        movdqa      xmm1, [esi+eax]       // + 4 * width
        movdqa      [edi+16], xmm1
        movdqa      xmm2, [esi+eax*2]     // + 8 * width
        add         esi, eax
        movdqa      [edi+32], xmm2
        movdqa      xmm3, [esi+eax*2]     // + 12 * width
        movdqa      [edi+48], xmm3
    }
}
/* SIMD Optimized Calculation of Line Through Color Space Copyright (C) 2006 Id Software, Inc. Written by J.M.P. van Waveren This code is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. */

#define ALIGN16( x )                __declspec(align(16)) x
#define R_SHUFFLE_D( x, y, z, w )   (( (w) & 3 ) << 6 | ( (z) & 3 ) << 4 | ( (y) & 3 ) << 2 | ( (x) & 3 ))

ALIGN16( static byte SIMD_MMX_byte_0[8] ) = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };  

#define INSET_SHIFT         4       // inset the bounding box with ( range >> shift )

void GetMinMaxColors_MMX( const byte *colorBlock, byte *minColor, byte *maxColor ) {
  __asm {
    mov         eax, colorBlock
    mov         esi, minColor
    mov         edi, maxColor
    // get bounding box
    pshufw      mm0, qword ptr [eax+ 0], R_SHUFFLE_D( 0, 1, 2, 3 )
    pshufw      mm1, qword ptr [eax+ 0], R_SHUFFLE_D( 0, 1, 2, 3 )
    pminub      mm0, qword ptr [eax+ 8]
    pmaxub      mm1, qword ptr [eax+ 8]
    pminub      mm0, qword ptr [eax+16]
    pmaxub      mm1, qword ptr [eax+16]
    pminub      mm0, qword ptr [eax+24]
    pmaxub      mm1, qword ptr [eax+24]
    pminub      mm0, qword ptr [eax+32]
    pmaxub      mm1, qword ptr [eax+32]
    pminub      mm0, qword ptr [eax+40]
    pmaxub      mm1, qword ptr [eax+40]
    pminub      mm0, qword ptr [eax+48]
    pmaxub      mm1, qword ptr [eax+48]
    pminub      mm0, qword ptr [eax+56]
    pmaxub      mm1, qword ptr [eax+56]
    pshufw      mm6, mm0, R_SHUFFLE_D( 2, 3, 2, 3 )
    pshufw      mm7, mm1, R_SHUFFLE_D( 2, 3, 2, 3 )
    pminub      mm0, mm6
    pmaxub      mm1, mm7
    // inset the bounding box
    punpcklbw   mm0, SIMD_MMX_byte_0
    punpcklbw   mm1, SIMD_MMX_byte_0
    movq        mm2, mm1
    psubw       mm2, mm0
    psrlw       mm2, INSET_SHIFT
    paddw       mm0, mm2
    psubw       mm1, mm2
    packuswb    mm0, mm0
    packuswb    mm1, mm1
    // store bounding box extents
    movd        dword ptr [esi], mm0
  }
}
ALIGN16( static byte SIMD_SSE2_byte_0[16] ) = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }; 

void GetMinMaxColors_SSE2( const byte *colorBlock, byte *minColor, byte *maxColor ) { 
    __asm { 
        mov eax, colorBlock 
        mov esi, minColor 
        mov edi, maxColor 

        // get bounding box 
        movdq xmm0, qword ptr [eax] 
        movdq xmm1, qword ptr [eax] 
        pminub xmm0, qword ptr [eax+16] 
        pmaxub xmm1, qword ptr [eax+16] 
        pminub xmm0, qword ptr [eax+32] 
        pmaxub xmm1, qword ptr [eax+32] 
        pminub xmm0, qword ptr [eax+48] 
        pmaxub xmm1, qword ptr [eax+48] 
        pshufd xmm3, xmm0, R_SHUFFLE_D( 2, 3, 2, 3 ) 
        pshufd xmm4, xmm1, R_SHUFFLE_D( 2, 3, 2, 3 ) 
        pminub xmm0, xmm3 
        pmaxub xmm1, xmm4 
        pshufld xmm6, xmm0, R_SHUFFLE_D( 2, 3, 2, 3 ) 
        pshufld xmm7, xmm1, R_SHUFFLE_D( 2, 3, 2, 3 ) 
        pminub xmm0, xmm6 
        pmaxub xmm1, xmm7 

        // inset the bounding box 
        punpcklbw xmm0, SIMD_SSE2_byte_0 
        punpcklbw xmm1, SIMD_SSE2_byte_0 
        movdq xmm2, xmm1 
        psubb xmm2, xmm0 
        psrlw xmm2, INSET_SHIFT 
        paddw xmm0, xmm2 
        psubb xmm1, xmm2 
        packuswb xmm0, xmm0 
        packuswb xmm1, xmm1 

        // store bounding box extents 
        movd dword ptr [esi], xmm0 
        movd dword ptr [edi], xmm1 
    } 
}
Appendix C

/*
SIMD Optimized Calculation of Color Indices
Copyright (C) 2006 Id Software, Inc.
Written by J.M.P. van Waveren

This code is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This code is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.
*/

ALIGN16( static word SIMD_MMX_word_0[4] ) = { 0x0000, 0x0000, 0x0000, 0x0000};
ALIGN16( static word SIMD_MMX_word_1[4] ) = { 0x0001, 0x0001, 0x0001, 0x0001};
ALIGN16( static word SIMD_MMX_word_2[4] ) = { 0x0002, 0x0002, 0x0002, 0x0002};
ALIGN16( static word SIMD_MMX_word_div_by_3[4] ) = { (1<<16)/3+1, (1<<16)/3+1,
(1<<16)/3+1, (1<<16)/3+1};
ALIGN16( static byte SIMD_MMX_byte_colorMask[8] ) = { C565_5_MASK, C565_6_MASK,
C565_5_MASK, 0x00, 0x00, 0x00, 0x00, 0x00};

void EmitColorIndices_MMX( const byte *colorBlock, const byte *minColor, const byte *maxColor ) {

ALIGN16( byte color0[8] );
ALIGN16( byte color1[8] );
ALIGN16( byte color2[8] );
ALIGN16( byte color3[8] );
ALIGN16( byte result[8] );

__asm {
   mov         esi, maxColor
   mov         edi, minColor
   pxor        mm7, mm7
   movq        result, mm7
   movd        mm0, [esi]
pand        mm0, SIMD_MMX_byte_colorMask
   punpcklbw   mm0, mm7
   pshufw      mm4, mm0, R_SHUFFLE_D( 0, 3, 2, 3 )
pshufw      mm5, mm0, R_SHUFFLE_D( 3, 1, 3, 3 )
   psrlw       mm4, 5
   psrlw       mm5, 6
   por         mm0, mm4
   por         mm0, mm5
   movq        mm2, mm0
   packuswb    mm2, mm7
   movq        color0, mm2
   movd        mm1, [edi]
pand        mm1, SIMD_MMX_byte_colorMask
   punpcklbw   mm1, mm7
   pshufw      mm4, mm1, R_SHUFFLE_D( 0, 3, 2, 3 )
pshufw      mm5, mm1, R_SHUFFLE_D( 3, 1, 3, 3 )
   psrlw       mm4, 5
   psrlw       mm5, 6
   por         mm1, mm4
   por         mm1, mm5
   movq        mm3, mm1
   packuswb    mm3, mm7
   movq        color1, mm3
   ...
}
movq mm6, mm0
paddw mm6, mm0
paddw mm6, mm1
pmulhw mm6, SIMD_MMX_word_div_by_3 // * ( ( 1 << 16 ) / 3 + 1 ) ) >> 16
packuswb mm6, mm7
movq color2, mm6

paddw mm1, mm1
paddw mm0, mm1
pmulhw mm0, SIMD_MMX_word_div_by_3 // * ( ( 1 << 16 ) / 3 + 1 ) ) >> 16
packuswb mm0, mm7
movq color3, mm0

mov eax, 48
mov esi, colorBlock
loop1: // iterates 4 times
movd mm3, dword ptr [esi+eax+0]
movd mm5, dword ptr [esi+eax+4]
movq mm0, mm3
movq mm6, mm5
psadbw mm0, color0
psadbw mm6, color0
packssdw mm0, mm6
movq mm1, mm3
movq mm6, mm5
psadbw mm1, color1
psadbw mm6, color1
packssdw mm1, mm6
movq mm2, mm3
movq mm6, mm5
psadbw mm2, color2
psadbw mm6, color2
packssdw mm2, mm6
movq mm4, mm3
movq mm5, mm5
psadbw mm4, color3
psadbw mm5, color3
packssdw mm4, mm5
movd mm4, dword ptr [esi+eax+8]
movd mm5, dword ptr [esi+eax+12]
movq mm6, mm4
movq mm7, mm5
psadbw mm6, color0
psadbw mm7, color0
packssdw mm6, mm7
packssdw mm0, mm6 // d0
movq mm6, mm4
movq mm7, mm5
psadbw mm6, color1
psadbw mm7, color1
packssdw mm6, mm7
packssdw mm1, mm6 // d1
movq mm6, mm4
movq mm7, mm5
psadbw mm6, color2
psadbw mm7, color2
packssdw mm6, mm7
packssdw mm2, mm6 // d2
psadbw mm4, color3
psadbw mm5, color3
packssdw mm4, mm5
packssdw mm3, mm4 // d3
movq mm7, result
pslld mm7, 8
movq mm4, mm0
movq mm5, mm1
pcmpgtw mm0, mm3    // b0
pcmpgtw mm1, mm2    // b1
pcmpgtw mm4, mm2    // b2
pcmpgtw mm5, mm3    // b3
pcmpgtw mm2, mm3    // b4
pand mm4, mm1      // x0
pand mm5, mm0      // x1
pand mm2, mm0      // x2
por mm4, mm5
pand mm2, SIMD_MMX_word_1
pand mm4, SIMD_MMX_word_2
por mm2, mm4

psubw mm5, mm2, R_SHUFFLE_D( 2, 3, 0, 1 )
punpcklwd mm2, SIMD_MMX_word_0
punpcklwd mm5, SIMD_MMX_word_0
pslld mm5, 4
por mm7, mm5
por mm7, mm2
movq result, mm7

sub eax, 16
jge loop1

mov esi, globalOutData
movq mm6, mm7
psrlq mm6, 32-2
por mm7, mm6
movd dword ptr [esi], mm7
emms

globalOutData += 4;
}

ALIGN16( static word SIMD_SSE2_word_0[8] ) = { 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000 };
ALIGN16( static word SIMD_SSE2_word_1[8] ) = { 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001 };
ALIGN16( static word SIMD_SSE2_word_2[8] ) = { 0x0002, 0x0002, 0x0002, 0x0002, 0x0002, 0x0002, 0x0002, 0x0002 };
ALIGN16( static word SIMD_SSE2_word_div_by_3[8] ) = { (1<<16)/3+1, (1<<16)/3+1, (1<<16)/3+1, (1<<16)/3+1, (1<<16)/3+1, (1<<16)/3+1, (1<<16)/3+1, (1<<16)/3+1 };
ALIGN16( static byte SIMD_SSE2_byte_colorMask[16] ) = { C565_5_MASK, C565_6_MASK, C565_5_MASK, 0x00, 0x00, 0x00, 0x00, 0x00, C565_5_MASK, C565_6_MASK, C565_5_MASK, 0x00, 0x00, 0x00, 0x00 };

void EmitColorIndices_SSE2( const byte *colorBlock, const byte *minColor, const byte *maxColor ) {
    ALIGN16( byte color0[16] );
    ALIGN16( byte color1[16] );
    ALIGN16( byte color2[16] );
    ALIGN16( byte color3[16] );
    ALIGN16( byte result[16] );

    __asm {
        mov esi, maxColor
        mov edi, minColor
        pxor xmm7, xmm7
        movdqa result, xmm7
        movd xmm0, [esi]
pand xmm0, SIMD_SSE2_byte_colorMask
        punpcklwb xmm0, xmm7
        psubflw xmm4, xmm0, R_SHUFFLE_D( 0, 3, 2, 3 )
pshufld xmm5, xmm0, R_SHUFFLE_D( 3, 1, 3, 3 )
pshufld xmm4, xmm5, 5
        psslq xmm5, 6
        por xmm0, xmm4
        por xmm0, xmm5
    }
movd     xmm1, [edi]
pand     xmm1, SIMD_SSE2_byte_colorMask
punpcklbw xmm1, xmm7
pshufbw  xmm4, xmm1, R_SHUFFLE_D( 0, 3, 2, 3 )
pshufbw  xmm5, xmm1, R_SHUFFLE_D( 3, 1, 3, 3 )
psrlw    xmm4, 5
psrlw    xmm5, 6
por      xmm1, xmm4
por      xmm1, xmm5
movdqa   xmm2, xmm0
packuswb xmm2, xmm7
pshufd   xmm2, xmm2, R_SHUFFLE_D( 0, 1, 0, 1 )
movdqa   color0, xmm2
movdqa   xmm6, xmm0
paddw    xmm6, xmm0
paddw    xmm6, xmm1
pmulhw   xmm6, SIMD_SSE2_word_div_by_3   // * ( ( 1 << 16 ) / 3 + 1 ) ) >> 16
packuswb xmm6, xmm7
pshufd   xmm6, xmm6, R_SHUFFLE_D( 0, 1, 0, 1 )
movdqa   color2, xmm6
movdqa   xmm3, xmm1
packuswb xmm3, xmm7
pshufd   xmm3, xmm3, R_SHUFFLE_D( 0, 1, 0, 1 )
movdqa   color1, xmm3
paddw    xmm1, xmm1
paddw    xmm0, xmm1
pmulhw   xmm0, SIMD_SSE2_word_div_by_3   // * ( ( 1 << 16 ) / 3 + 1 ) ) >> 16
packuswb xmm0, xmm7
pshufd   xmm0, xmm0, R_SHUFFLE_D( 0, 1, 0, 1 )
movdqa   color3, xmm0
mov      eax, 32
mov      esi, colorBlock

loop1:   // iterates 2 times
movq     xmm3, qword ptr [esi+eax+0]
pshufd   xmm3, xmm3, R_SHUFFLE_D( 0, 2, 1, 3 )
movq     xmm5, qword ptr [esi+eax+8]
pshufd   xmm5, xmm5, R_SHUFFLE_D( 0, 2, 1, 3 )
movdqa   xmm0, xmm3
movdqa   xmm6, xmm5
psadbw   xmm0, color0
psadbw   xmm6, color0
packssdw xmm0, xmm6
movdqa   xmm1, xmm3
movdqa   xmm6, xmm5
psadbw   xmm1, color1
psadbw   xmm6, color1
packssdw xmm1, xmm6
movdqa   xmm2, xmm3
movdqa   xmm6, xmm5
psadbw   xmm2, color2
psadbw   xmm6, color2
packssdw xmm2, xmm6
psadbw   xmm3, color3
psadbw   xmm5, color3
packssdw xmm3, xmm5
movq     xmm4, qword ptr [esi+eax+16]
pshufd   xmm4, xmm4, R_SHUFFLE_D( 0, 2, 1, 3 )
movq     xmm5, qword ptr [esi+eax+24]
pshufd   xmm5, xmm5, R_SHUFFLE_D( 0, 2, 1, 3 )
movdqa   xmm6, xmm4
movdqa   xmm7, xmm5
psadbw  xmm6, color0
psadbw  xmm7, color0
packssdw xmm6, xmm7
packssdw xmm0, xmm6 // d0
movdqa  xmm6, xmm4
movdqa  xmm7, xmm5
psadbw  xmm6, color1
psadbw  xmm7, color1
packssdw xmm6, xmm7
packssdw xmm1, xmm6 // d1
movdqa  xmm6, xmm4
movdqa  xmm7, xmm5
psadbw  xmm6, color2
psadbw  xmm7, color2
packssdw xmm6, xmm7
packssdw xmm2, xmm6 // d2
psadbw  xmm4, color3
psadbw  xmm5, color3
packssdw xmm4, xmm5
packssdw xmm3, xmm4 // d3
movdqa  xmm7, result
pslld   xmm7, 16
movdqa  xmm4, xmm0
movdqa  xmm5, xmm1
pcmpgtw xmm0, xmm3 // b0
pcmpgtw xmm1, xmm2 // b1
pcmpgtw xmm4, xmm2 // b2
pcmpgtw xmm5, xmm3 // b3
pcmpgtw xmm2, xmm3 // b4
pand    xmm4, xmm1 // x0
pand    xmm5, xmm0 // x1
pand    xmm2, xmm0 // x2
por     xmm4, xmm5
pand    xmm2, SIMD_SSE2_word_1
pand    xmm4, SIMD_SSE2_word_2
por     xmm2, xmm4
pshufd  xmm5, xmm2, R_SHUFFLE_D( 2, 3, 0, 1 )
punpcklwd xmm2, SIMD_SSE2_word_0
punpcklwd xmm5, SIMD_SSE2_word_0
pslld   xmm5, 8
por     xmm7, xmm5
por     xmm7, xmm2
movdqa  result, xmm7
sub     eax, 32
jge     loop1
mov     esi, globalOutData
pshufd  xmm4, xmm7, R_SHUFFLE_D( 1, 2, 3, 0 )
pshufd  xmm5, xmm7, R_SHUFFLE_D( 2, 3, 0, 1 )
pshufd  xmm6, xmm7, R_SHUFFLE_D( 3, 0, 1, 2 )
pshufd  xmm4, 2
pslld   xmm5, 4
pslld   xmm6, 6
por     xmm7, xmm4
por     xmm7, xmm5
por     xmm7, xmm6
movd    dword ptr [esi], xmm7
}
globalOutData += 4;
}
SIMD Optimized Calculation of Alpha Indices
Copyright (C) 2006 Id Software, Inc.
Written by J.M.P. van Waveren

This code is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

ALIGN16( static byte SIMD_MMX_byte_1[8] ) = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 }; ADMKX
ALIGN16( static byte SIMD_MMX_byte_2[8] ) = { 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02 }; ADMKX
ALIGN16( static byte SIMD_MMX_byte_7[8] ) = { 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07 }; ADMKX
ALIGN16( static word SIMD_MMX_word_div_by_7[4] ) = { (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1 }; ADMKX
ALIGN16( static word SIMD_MMX_word_div_by_14[4] ) = { (1<<16)/14+1, (1<<16)/14+1, (1<<16)/14+1, (1<<16)/14+1 }; ADMKX
ALIGN16( static word SIMD_MMX_word_scale654[4] ) = { 6, 5, 4, 0 }; ADMKX
ALIGN16( static word SIMD_MMX_word_scale123[4] ) = { 1, 2, 3, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask0[2] ) = { 7<<0, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask1[2] ) = { 7<<3, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask2[2] ) = { 7<<6, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask3[2] ) = { 7<<9, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask4[2] ) = { 7<<12, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask5[2] ) = { 7<<15, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask6[2] ) = { 7<<18, 0 }; ADMKX
ALIGN16( static dword SIMD_MMX_dword_alpha_bit_mask7[2] ) = { 7<<21, 0 }; ADMKX

void EmitAlphaIndices_MMX( const byte *colorBlock, const byte minAlpha, const byte maxAlpha ) {

ALIGN16( byte alphaBlock[16] ); ADMKX
ALIGN16( byte ab1[8] ); ADMKX
ALIGN16( byte ab2[8] ); ADMKX
ALIGN16( byte ab3[8] ); ADMKX
ALIGN16( byte ab4[8] ); ADMKX
ALIGN16( byte ab5[8] ); ADMKX
ALIGN16( byte ab6[8] ); ADMKX
ALIGN16( byte ab7[8] ); ADMKX

__asm {
    mov     esi, colorBlock
    movq    mm0, [esi+ 0]
    movq    mm5, [esi+ 8]
    psrld   mm0, 24
    psrld   mm5, 24
    packuswb mm0, mm5
    packuswb mm0, mm6
    movq    mm6, [esi+16]
    movq    mm4, [esi+24]
    psrld   mm6, 24
    packuswb mm4, mm6
}
movq        alphaBlock+0, mm0
movq        mm0, [esi+32]
movq        mm5, [esi+40]
psrld       mm0, 24
psrld       mm5, 24
packuswb    mm0, mm5
movq        mm6, [esi+48]
movq        mm4, [esi+56]
psrld       mm6, 24
psrld       mm4, 24
packuswb    mm6, mm4
packuswb    mm0, mm6
movq        alphaBlock+8, mm0
movzx       ecx, maxAlpha
movd        mm0, ecx
pshufw      mm0, mm0, R_SHUFFLE_D( 0, 0, 0, 0 )
movq        mm1, mm0
movzx       edx, minAlpha
movd        mm2, edx
pshufw      mm2, mm2, R_SHUFFLE_D( 0, 0, 0, 0 )
movq        mm3, mm2
movq        mm4, mm0
psubw       mm4, mm2
pmulhw      mm4, SIMD_MMX_word_div_by_14        // * ( ( 1 << 16 ) / 14 + 1 ) )

>> 16

movq        mm5, mm2
paddw       mm5, mm4
packuswb    mm5, mm5
movq        ab1, mm5
pmullw      mm0, SIMD_MMX_word_scale654
pmullw      mm1, SIMD_MMX_word_scale123
pmullw      mm2, SIMD_MMX_word_scale123
pmullw      mm3, SIMD_MMX_word_scale654
paddw       mm0, mm2
paddw       mm1, mm3
pmulhw      mm0, SIMD_MMX_word_div_by_7        // * ( ( 1 << 16 ) / 7 + 1 ) ) >>

16

pmulhw      mm1, SIMD_MMX_word_div_by_7        // * ( ( 1 << 16 ) / 7 + 1 ) ) >>

16

paddw       mm0, mm4
paddw       mm1, mm4
pshufw      mm2, mm0, R_SHUFFLE_D( 0, 0, 0, 0 )
pshufw      mm3, mm0, R_SHUFFLE_D( 1, 1, 1, 1 )
pshufw      mm4, mm0, R_SHUFFLE_D( 2, 2, 2, 2 )
packuswb    mm2, mm2
packuswb    mm3, mm3
packuswb    mm4, mm4
movq        ab2, mm2
movq        ab3, mm3
movq        ab4, mm4
pshufw      mm2, mm1, R_SHUFFLE_D( 0, 0, 0, 0 )
pshufw      mm3, mm1, R_SHUFFLE_D( 1, 1, 1, 1 )
pshufw      mm4, mm1, R_SHUFFLE_D( 2, 2, 2, 2 )
packuswb    mm2, mm2
packuswb    mm3, mm3
packuswb    mm4, mm4
movq        ab5, mm2
movq        ab6, mm3
movq        ab7, mm4
pshufw      mm0, alphaBlock+0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm1, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm2, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm3, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pminub  mm1, ab1
pminub  mm2, ab2
pminub  mm3, ab3
pminub  mm4, ab4
pminub  mm5, ab5
pminub  mm6, ab6
pminub  mm7, ab7
pcmpeqb  mm1, mm0
pcmpeqb  mm2, mm0
pcmpeqb  mm3, mm0
pcmpeqb  mm4, mm0
pcmpeqb  mm5, mm0
pcmpeqb  mm6, mm0
pcmpeqb  mm7, mm0
pand    mm1, SIMD_MMX_byte_1
pand    mm2, SIMD_MMX_byte_1
pand    mm3, SIMD_MMX_byte_1
pand    mm4, SIMD_MMX_byte_1
pand    mm5, SIMD_MMX_byte_1
pand    mm6, SIMD_MMX_byte_1
pand    mm7, SIMD_MMX_byte_1
pshufw  mm0, SIMD_MMX_byte_1, R_SHUFFLE_D( 0, 1, 2, 3 )
paddusb mm0, mm1
paddusb mm0, mm2
paddusb mm0, mm3
paddusb mm0, mm4
paddusb mm0, mm5
paddusb mm0, mm6
paddusb mm0, mm7
pand    mm0, SIMD_MMX_byte_7
pshufw  mm1, SIMD_MMX_byte_2, R_SHUFFLE_D( 0, 1, 2, 3 )
pcmpeqtb mm1, mm0
pand    mm1, SIMD_MMX_byte_1
pxor     mm0, mm1
pshufw  mm1, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm2, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm3, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psrlq   mm1,  8- 3
psrlq   mm2, 16- 6
psrlq   mm3, 24- 9
pshufw  mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pshufw  mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psrlq   mm4, 32-12
psrlq   mm5, 40-15
psrlq   mm6, 48-18
psrlq   mm7, 56-21
pand    mm0, SIMD_MMX_dword_alpha_bit_mask0
pand    mm1, SIMD_MMX_dword_alpha_bit_mask1
pand    mm2, SIMD_MMX_dword_alpha_bit_mask2
pand    mm3, SIMD_MMX_dword_alpha_bit_mask3
pand    mm4, SIMD_MMX_dword_alpha_bit_mask4
pand    mm5, SIMD_MMX_dword_alpha_bit_mask5
pand    mm6, SIMD_MMX_dword_alpha_bit_mask6
pand    mm7, SIMD_MMX_dword_alpha_bit_mask7
por     mm0, mm1
por     mm2, mm3
por     mm4, mm5
por     mm6, mm7
por     mm0, mm2
por     mm4, mm6
por     mm0, mm4
mov     esi, outPtr
movd        [esi+0], mm0

psubw       mm0, alphaBlock+8, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm1, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm2, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm3, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
pminub     mm1, ab1
pminub     mm2, ab2
pminub     mm3, ab3
pminub     mm4, ab4
pminub     mm5, ab5
pminub     mm6, ab6
pminub     mm7, ab7
pcmpeqb     mm1, mm0
pcmpeqb     mm2, mm0
pcmpeqb     mm3, mm0
pcmpeqb     mm4, mm0
pcmpeqb     mm5, mm0
pcmpeqb     mm6, mm0
pcmpeqb     mm7, mm0

pand       mm1, SIMD_MMX_byte_1
pand       mm2, SIMD_MMX_byte_1
pand       mm3, SIMD_MMX_byte_1
pand       mm4, SIMD_MMX_byte_1
pand       mm5, SIMD_MMX_byte_1
pand       mm6, SIMD_MMX_byte_1
pand       mm7, SIMD_MMX_byte_1
psubw       mm0, SIMD_MMX_byte_1, R_SHUFFLE_D( 0, 1, 2, 3 )
paddusb    mm0, mm1
paddusb    mm0, mm2
paddusb    mm0, mm3
paddusb    mm0, mm4
paddusb    mm0, mm5
paddusb    mm0, mm6
paddusb    mm0, mm7

pand       mm0, SIMD_MMX_byte_7
psubw       mm1, SIMD_MMX_byte_2, R_SHUFFLE_D( 0, 1, 2, 3 )
pcmpegb     mm1, mm0

pxor        mm0, mm1
pxor        mm2, mm3
pxor        mm4, mm5
pxor        mm6, mm7

psrlq       mm1, 8 - 3
psrlq       mm2, 16 - 6
psrlq       mm3, 24 - 9
psubw       mm4, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm5, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm6, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )
psubw       mm7, mm0, R_SHUFFLE_D( 0, 1, 2, 3 )

psrlq       mm4, 32 - 12
psrlq       mm5, 40 - 15
psrlq       mm6, 48 - 18
psrlq       mm7, 56 - 21

pand       mm0, SIMD_MMX_dword_alpha_bit_mask0
pand       mm1, SIMD_MMX_dword_alpha_bit_mask1
pand       mm2, SIMD_MMX_dword_alpha_bit_mask2
pand       mm3, SIMD_MMX_dword_alpha_bit_mask3
pand       mm4, SIMD_MMX_dword_alpha_bit_mask4
pand       mm5, SIMD_MMX_dword_alpha_bit_mask5
pand       mm6, SIMD_MMX_dword_alpha_bit_mask6
pand       mm7, SIMD_MMX_dword_alpha_bit_mask7
por        mm0, mm1
por        mm2, mm3
por        mm4, mm5
por        mm6, mm7
por        mm0, mm2
por mm4, mm6
por mm0, mm4
movd [esi+3], mm0
emms
}
globalOutData += 6;
}
ALIGN16( static byte SIMD_SSE2_byte_1[16] ) = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01 };
ALIGN16( static byte SIMD_SSE2_byte_2[16] ) = { 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02 };
ALIGN16( static byte SIMD_SSE2_byte_7[16] ) = { 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07 };
ALIGN16( static word SIMD_SSE2_word_div_by_7[8] ) = { (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1, (1<<16)/7+1 };
ALIGN16( static word SIMD_SSE2_word_div_by_14[8] ) = { (1<<16)/14+1, (1<<16)/14+1, (1<<16)/14+1, (1<<16)/14+1, (1<<16)/14+1, (1<<16)/14+1, (1<<16)/14+1 };
ALIGN16( static word SIMD_SSE2_word_scale66554400[8] ) = { 6, 6, 5, 5, 4, 4, 0, 0 };
ALIGN16( static word SIMD_SSE2_word_scale11223300[8] ) = { 1, 1, 2, 2, 3, 3, 0, 0 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask0[4] ) = { 7<<0, 7<<0, 7<<0, 7<<0 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask1[4] ) = { 7<<3, 7<<3, 7<<3, 7<<3 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask2[4] ) = { 7<<6, 7<<6, 7<<6, 7<<6 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask3[4] ) = { 7<<9, 7<<9, 7<<9, 7<<9 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask4[4] ) = { 7<<12, 7<<12, 7<<12, 7<<12 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask5[4] ) = { 7<<15, 7<<15, 7<<15, 7<<15 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask6[4] ) = { 7<<18, 7<<18, 7<<18, 7<<18 };
ALIGN16( static dword SIMD_SSE2_dword_alpha_bit_mask7[4] ) = { 7<<21, 7<<21, 7<<21, 7<<21 };

void EmitAlphaIndices_SSE2( const byte *colorBlock, const byte minAlpha, const byte maxAlpha ) {
__asm {
mov esi, colorBlock
movdqa xmm0, [esi+0]
movdqa xmm5, [esi+16]
pshld xmm0, 24
pshld xmm5, 24
packuswb xmm0, xmm5
movdqa xmm6, [esi+32]
movdqa xmm4, [esi+48]
pshld xmm6, 24
pshld xmm4, 24
packuswb xmm6, xmm4
movzx ecx, maxAlpha
mov xmm5, ecx
psubflw xmm5, xmm5, R_SHUFFLE_D( 0, 0, 0, 0 )
psubfd xmm5, xmm5, R_SHUFFLE_D( 0, 0, 0, 0 )
movdqa xmm7, xmm5
movzx edx, minAlpha
movd xmm2, edx
psubflw xmm2, xmm2, R_SHUFFLE_D( 0, 0, 0, 0 )
psubfd xmm2, xmm2, R_SHUFFLE_D( 0, 0, 0, 0 )
movdqa xmm3, xmm2
movdqa xmm4, xmm5
psubw xmm4, xmm4
pmullw xmm4, SIMD_SSE2_word_div_by_14 // * ( ( 1 << 16 ) / 14 + 1 ) ) >> 16
movdqa xmm1, xmm2
padw xmm1, xmm4
packuswb xmm1, xmm1 // ab1
pmullw xmm5, SIMD_SSE2_word_scale66554400
pmullw xmm7, SIMD_SSE2_word_scale11223300
}
pmulw xmm2, SIMD_SSE2_word_scale12223300
pmulw xmm3, SIMD_SSE2_word_scale66554400
paddw xmm5, xmm2
paddw xmm7, xmm3
pmulhw xmm5, SIMD_SSE2_word_div_by_7  // * ( ( 1 << 16 ) / 7 + 1 ) ) >> 16
pmulhw xmm7, SIMD_SSE2_word_div_by_7  // * ( ( 1 << 16 ) / 7 + 1 ) ) >> 16
paddw xmm5, xmm4
paddw xmm7, xmm4
pshufd xmm2, xmm5, R_SHUFFLE_D( 0, 0, 0, 0 )
pshufd xmm3, xmm5, R_SHUFFLE_D( 1, 1, 1, 1 )
pshufd xmm4, xmm5, R_SHUFFLE_D( 2, 2, 2, 2 )
packuswb xmm2, xmm2  // ab2
packuswb xmm3, xmm3  // ab3
packuswb xmm4, xmm4  // ab4
packuswb xmm0, xmm6  // alpha values
packuswb xmm5, xmm5  // ab5
packuswb xmm6, xmm6  // ab6
packuswb xmm7, xmm7  // ab7
pminub xmm1, xmm0
pminub xmm2, xmm0
pminub xmm3, xmm0
pcmpeqb xmm1, xmm0
pcmpeqb xmm2, xmm0
pcmpeqb xmm3, xmm0
pcmpeqb xmm4, xmm0
pcmpeqb xmm5, xmm0
pcmpeqb xmm6, xmm0
pcmpeqb xmm7, xmm0
pand xmm1, SIMD_SSE2_byte_1
pand xmm2, SIMD_SSE2_byte_1
pand xmm3, SIMD_SSE2_byte_1
pand xmm4, SIMD_SSE2_byte_1
pand xmm5, SIMD_SSE2_byte_1
pand xmm6, SIMD_SSE2_byte_1
pand xmm7, SIMD_SSE2_byte_1
movdqa xmm0, SIMD_SSE2_byte_1
paddusb xmm0, xmm1
paddusb xmm2, xmm3
paddusb xmm4, xmm5
paddusb xmm6, xmm7
paddusb xmm0, xmm2
paddusb xmm4, xmm6
paddusb xmm0, xmm4
pand xmm0, SIMD_SSE2_byte_7
movdqa xmm1, SIMD_SSE2_byte_2
pcmpeqb xmm1, xmm0
pand xmm1, SIMD_SSE2_byte_1
pxor xmm0, xmm1
movdqa xmm1, xmm0
movdqa xmm2, xmm0
movdqa xmm3, xmm0
movdqa xmm4, xmm0
movdqa xmm5, xmm0
movdqa xmm6, xmm0
movdqa xmm7, xmm0
psrlq xmm1, 8 - 3
psrlq xmm2, 16 - 6
psrlq xmm3, 24 - 9

41
psrlq        xmm4, 32-12
psrlq        xmm5, 40-15
psrlq        xmm6, 48-18
psrlq        xmm7, 56-21
pand        xmm0, SIMD_SSE2_dword_alpha_bit_mask0
pand        xmm1, SIMD_SSE2_dword_alpha_bit_mask1
pand        xmm2, SIMD_SSE2_dword_alpha_bit_mask2
pand        xmm3, SIMD_SSE2_dword_alpha_bit_mask3
pand        xmm4, SIMD_SSE2_dword_alpha_bit_mask4
pand        xmm5, SIMD_SSE2_dword_alpha_bit_mask5
pand        xmm6, SIMD_SSE2_dword_alpha_bit_mask6
pand        xmm7, SIMD_SSE2_dword_alpha_bit_mask7
por         xmm0, xmm1
por         xmm2, xmm3
por         xmm4, xmm5
por         xmm6, xmm7
por         xmm0, xmm2
por         xmm4, xmm6
por         xmm0, xmm4
mov         esi, outPtr
movd        [esi+0], xmm0
pshufd      xmm1, xmm0, R_SHUFFLE_D( 2, 3, 0, 1 )
movd        [esi+3], xmm1

globalOutData += 6;