Appendix K

RGTC Compressed Texture Format

This appendix will describe the RG-texture compression format, used for compressing one- and two-component textures. It has the following major sections:

- "RGTC Overview"

RGTC Overview

The RGTC compressed texture format is a standardized compression format for one- and two-component textures. It is a block-based format where collections of 4 × 4 texel squares that are compressed into a 64-bit encodings for each component. The first two bytes of an encoding for a channel represent representative values, which may be linearly combined to compute the color of any pixel in the 4 × 4 block. The remaining 48-bits represent 16 three-bit codes used to determine the appropriate linear combination of the two representative values.

Textures compressed using the RGTC format must have a width and height that are multiples of four pixels. Any pixels that fall outside of the multiple-of-four width or height are ignored on decompression.

The RGTC format includes four encodings depending on the number of components in the image, and the sign of the texel data, as shown in Table K-1.

RGTC Format	Description
GL_COMPRESSED_RED_RGTC1	Single-channel 8-bit unsigned image data
GL_COMPRESSED_SIGNED_RED_RGTC1	Single-channel 8-bit signed image data
GL_COMPRESSED_RG_RTGC2	Two-channel 8-bit per component unsigned image data
GL_COMPRESSED_SIGNED_RG_RGTC2	Two-channel 8-bit per component signed image data

Table K-1 The RTGC Compressed Image Formats

Encoding RGTC Images

The following routine will encode a one-component 4 × 4 block of GLubyte pixels into an RGTC bit-field representing the compressed pixels, as described in *The OpenGL Graphics System: A Specification, Appendix D (Version 3.1)*.

```
GLubyte*
RGTCEncode( const GLubyte pixels[4][4] )
{
    int       i, j, r, c, idx;
    GLubyte   min = 0xff,  max = 0x00;  /* range for current 4x4
    block */
    GLubyte   codes[4][4];
    GLbyte    group = 0;

    static GLubyte   encoded[8];

    unsigned long long  *bits;

    /* Note which group of codes were using
    **  in our encoding.
    **
    **   Group[0] ==> red_0 > red_1
    **   Group[1] ==> red_0 <= red_1
    */
    const GLubyte  Group[2] = { 0x00, 0x80 };
    const GLubyte  Mask[2]  = { 0x07, 0x38 };
    const GLubyte  Shift[2] = { 0, 3 };

    /* compute min & max pixel values */
    for ( i = 0; i < 4; ++i ) {
        for ( j = 0; j < 4; ++j ) {
          GLubyte val = pixels[i][j];
          if ( min > val )  min = val;
          if ( max < val )  max = val;
        }
    }

    if ( min == max ) {
        /* Constant color across the block - we'll use the
        **   first set of codes (red_0 > red_1), and set
        **   all bits to be code 0 */
        encoded[0] = min;
        encoded[1] = max;
        encoded[2] = 0x00;
        encoded[3] = 0x00;
        encoded[4] = 0x00;
        encoded[5] = 0x00;
        encoded[6] = 0x00;
        encoded[7] = 0x00;
    }
```

```
else {
    GLfloat d1 = (max - min) / 7.0;
    GLfloat d2 = (max - min) / 5.0;

    GLubyte *pptr = (GLubyte*) pixels;
    GLubyte *cptr = (GLubyte*) codes;

    for ( i = 0; i < 16; ++i, ++pptr, ++cptr ) {
      GLfloat v1 = (*pptr - min) / d1;
      GLfloat v2 = (*pptr - min) / d2;

      GLubyte ip1 = (GLubyte) v1;
      GLubyte ip2 = (GLubyte) v2;

      GLfloat fp1 = v1 - ip1;
      GLfloat fp2 = v2 - ip2;

      enum { MinCode = 0, MaxCode = 1 };

      if ( fp1 > 0.5 ) {
         fp1 -= 0.5;
         ip1 += 1;
      }



      if ( fp2 > 0.5 ) {
         fp2 -= 0.5;
         ip1 += 1;
      }

      if ( ip1 == 0 ) {
         *cptr = MinCode;
      }
      else if ( ip1 == 7 ) {
         *cptr = MaxCode;
      }
      else {
         *cptr = (ip2 << Shift[1]) | (ip1 << Shift[0]);
      }

      if ( fp1 < fp2 ) {
         /* The current pixel's closer to one of the
         **   seventh range partitoning values */
         *cptr |= Group[0];
         ++group;
      }
```

```
            else {
               /* Here, the current pixel's closer to
               **   one of the fifth range partitonings
               **   values */
               *cptr |= Group[1];
               --group;
            }
         }
      }

      group = (group >= 0) ? 0 : 1;

      bits = (unsigned long long*) encoded;

      for ( i = 0; i < 4; ++i ) {
         for ( j = 0; j < 4; ++j ) {
            GLubyte shift = 3*(4*j+i);
            GLubyte code;
            unsigned long long mask;

            code = codes[i][j] & Mask[group];
            code >>= Shift[group];
            mask = code << shift;
            *bits |= mask;
         }
      }

      if ( group == 0 ) {
            encoded[0] = max;
            encoded[1] = min;
         }
         else {
            encoded[0] = min;
            encoded[1] = max;
         }
      }

      return encoded;
}
```