多媒体Project3说明文档

by 黄子豪 18302010034

JPEG编码的详细说明

JPEG编码步骤为：

1、FDCT正向离散余弦变换

2、量化

3、Z字形编码

4、使用DPCM对直流系数进行编码

5、使用RLE对交流系数进行编码

6、熵编码

7、组成JPEG位数据流

jpeg编码的主题函数为*jpeg_encode*，先进行头部信息的一些写入，接下来介绍主体压缩编码的几个操作流程。

1、色彩空间的转换

把rgb格式转换为YCbCr色彩系统，这通过*rgb_to_ycbcr*函数实现：

```
void rgb_to_ycbcr(UINT8 *rgb_unit, ycbcr_unit *ycc_unit, int x, int w) {
    ycbcr_tables *tbl = &ycc_tables;
    UINT8 r, g, b;

    int src_pos = x * 3;
#ifdef REVERSED
    src_pos += w * (DCTSIZE - 1) * 3;
#endif
    int dst_pos = 0;
    int i, j;
    for (j = 0; j < DCTSIZE; j++) {
        for (i = 0; i < DCTSIZE; i++) {
            b = rgb_unit[src_pos];
            g = rgb_unit[src_pos + 1];
```

```
            r = rgb_unit[src_pos + 2];
            ycc_unit->y[dst_pos] = (INT8) ((UINT8)
                                                ((tbl->r2y[r] + tbl->g2y[g] +
tbl->b2y[b]) >> 16) - 128);
            ycc_unit->cb[dst_pos] = (INT8) ((UINT8)
                    ((tbl->r2cb[r] + tbl->g2cb[g] + tbl->b2cb[b]) >> 16));
            ycc_unit->cr[dst_pos] = (INT8) ((UINT8)
                    ((tbl->r2cr[r] + tbl->g2cr[g] + tbl->b2cr[b]) >> 16));
            src_pos += 3;
            dst_pos++;
        }
#ifdef REVERSED
        src_pos -= (w + DCTSIZE) * 3;
#elif
        src_pos += (w - DCTSIZE) * 3;
#endif
    }
}
```

想要用JPEG基本压缩法处理全彩色图像，得先把RGB颜色模式图像数据，转换为YCbCr颜色模式的数据。Y代表亮度，Cb和Cr则代表色度、饱和度。通过下列计算公式可完成数据转换：

$$Y = 0.2990R + 0.5870G + 0.1140B$$
$$Cb = -0.1687R - 0.3313G + 0.5000B + 128$$
$$Cr = 0.5000R - 0.4187G - 0.0813B + 128$$

人类的眼睛对低频的数据比对高频的数据具有更高的敏感度，事实上，人类的眼睛对亮度的改变也比对色彩的改变要敏感得多，也就是说Y成份的数据是比较重要的。既然Cb成份和Cr成份的数据比较相对不重要，就可以只取部分数据来处理。以增加压缩的比例。

这样转换对以后后续的压缩、增强和恢复会更加方便。

2、分块

把数字图像分割成8X8大小的像素块，方便DCT变换，这只需要通过像素点下标的递增就可以实现：

```
for (x = 0; x < binfo->width; x += 8){
    /*my code here*/
}
```

3、进行DCT变换

主要是为了将8 × 8图像的空间表达是转换为频率域，只需要少量的数据点来表示图像，由于此部分在本文档第三部分有详细解释，故此处不再赘述。

DCT变换使用了以下公式实现：

$$F(u,v) = \frac{1}{4}C(u)C(v)[\sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j)\cos\frac{(2i+1)u\pi}{16}\cos\frac{(2j+1)v\pi}{16}]$$

4、量化

对变换后的矩阵进行量化，目的是减小非0系数的幅度以及增加0值系数的数目：

```
void jpeg_quant(ycbcr_unit *ycc_unit, quant_unit *q_unit) {
    quant_tables *tbl = &q_tables;
    float q_lu, q_ch;
    int x, y, i = 0;
    for (x = 0; x < DCTSIZE; x++) {
        for (y = 0; y < DCTSIZE; y++) {
            q_lu = (float) (1.0 / ((double) tbl->lu[ZIGZAG[i]] * \
                                   AAN_SCALE_FACTOR[x] * AAN_SCALE_FACTOR[y] *
8.0));
            q_ch = (float) (1.0 / ((double) tbl->ch[ZIGZAG[i]] * \
                                   AAN_SCALE_FACTOR[x] * AAN_SCALE_FACTOR[y] *
8.0));

            q_unit->y[i] = (INT16) ((ycc_unit->y[i] * q_lu + 16384.5) - 16384);
            q_unit->cb[i] = (INT16) ((ycc_unit->cb[i] * q_ch + 16384.5) -
16384);
            q_unit->cr[i] = (INT16) ((ycc_unit->cr[i] * q_ch + 16384.5) -
16384);

            i++;
        }
    }
}
```

量化实现了压缩数据，在一定的主观保真的前提下，丢掉那些对视觉效果影响不大的信息，使用了以下公式：

$$\hat{F}(u,v) = round(\frac{F(u,v)}{Q(u,v)})$$

5、Huffman编码

DCT本身并不能实现压缩的效果，必须对于处理后的矩阵数据使用Huffman编码压缩，由于本文档第二部分有详细介绍Huffman编码，故这里对原理不过多介绍，留至第二部分。

6、补齐最后的不足一个字节的部分

这一功能使用*set_bits*实现：

```
void set_bits(BITS *bits, INT16 data) {
    /*******************************************************/
    /*                                                     */
    /*             finish the missing codes                */
    /*                                                     */
    /*******************************************************/
```

```
    UINT16 pos;
    int i;
    pos = (UINT16) (data < 0 ? ~data + 1 : data);
    for (i = 15; i >= 0; i--)
        if ((pos & (1 << i)) != 0)
            break;
    bits->len = (UINT8) (i + 1);
    bits->val = (UINT16) (data < 0 ? data + (1 << bits->len) - 1 : data);
}
```

最后再写入一个代表文件结尾的字符即可。

Huffman编码的基本原理

- 依据信源字符出现的概率（频率）大小来构造代码
- 对出现概率（频率）较大的信源字符，给予较短码长
- 对于出现概率（频率）较小的信源字符，给予较长的码长
- 最后使得编码的平均码字最短，这样得到的熵值会是最小的

具体步骤：

1、将信源符号按概率递减顺序排列；
2、把两个最小的概率加起来，作为新符号的概率；
3、重复前两步，直到概率和达到1为止，构建出最优哈夫曼树；
4、在每次合并消息时，将被合并的消息赋予1和0或0和1；
5、寻找从每一信源符号到概率为1的路径，记录下路径上的1和0；
6、对每一符号写出从码树的根到中节点1、0序列

实现细节：

```
void jpeg_compress(compress_io *cio,
                   INT16 *data, INT16 *dc, BITS *dc_htable, BITS *ac_htable) {
    INT16 zigzag_data[DCTSIZE2];
    BITS bits;
    INT16 diff;
    int i, j;
    int zero_num;
    int mark;

    /* zigzag encode */
    for (i = 0; i < DCTSIZE2; i++)
        zigzag_data[ZIGZAG[i]] = data[i];

    /* write DC */
    diff = zigzag_data[0] - *dc;
    *dc = zigzag_data[0];

    if (diff == 0)
        write_bits(cio, dc_htable[0]);
    else {
        set_bits(&bits, diff);
        write_bits(cio, dc_htable[bits.len]);
        write_bits(cio, bits);
    }
```

```
    /* write AC */
    int end = DCTSIZE2 - 1;
    while (zigzag_data[end] == 0 && end > 0)
        end--;
    for (i = 1; i <= end; i++) {
        j = i;
        while (zigzag_data[j] == 0 && j <= end)
            j++;
        zero_num = j - i;
        for (mark = 0; mark < zero_num / 16; mark++)
            write_bits(cio, ac_htable[0xF0]);
        zero_num = zero_num % 16;
        set_bits(&bits, zigzag_data[j]);
        write_bits(cio, ac_htable[zero_num * 16 + bits.len]);
        write_bits(cio, bits);
        i = j;
    }

    /* write end of unit */
    if (end != DCTSIZE2 - 1)
        write_bits(cio, ac_htable[0]);
}
```

用到的建Huffman树的辅助函数：

```
void set_huff_table(UINT8 *nrcodes, UINT8 *values, BITS *h_table) {
    int i, j, k;
    j = 0;
    UINT16 value = 0;
    for (i = 1; i <= 16; i++) {
        for (k = 0; k < nrcodes[i]; k++) {
            h_table[values[j]].len = (UINT8) i;
            h_table[values[j]].val = value;
            j++;
            value++;
        }
        value <<= 1;
    }
}

void init_huff_tables() {
    huff_tables *tbl = &h_tables;
    set_huff_table(STD_LU_DC_NRCODES, STD_LU_DC_VALUES, tbl->lu_dc);
    set_huff_table(STD_LU_AC_NRCODES, STD_LU_AC_VALUES, tbl->lu_ac);
    set_huff_table(STD_CH_DC_NRCODES, STD_CH_DC_VALUES, tbl->ch_dc);
    set_huff_table(STD_CH_AC_NRCODES, STD_CH_AC_VALUES, tbl->ch_ac);
}
```

DCT变换的原理

进行DCT变换之前，需要把输入图像划分为8 × 8（或者16 × 16）的块，然后利用以下正交余弦变换公式：

$$F(u,v)=\frac{1}{4}C(u)C(v)[\sum_{i=0}^{7}\sum_{j=0}^{7}f(i,j)\cos\frac{(2i+1)u\pi}{16}\cos\frac{(2j+1)v\pi}{16}]$$

主要目的是把空间域表示的图8 × 8（或者16 × 16）图像变换成频率域表示的图，使空间域的能量重新分布，把能量集中在矩阵左上角少数几个系数上，降低图像的相关性，这样就只需要少量的数据点来表示图像。

```
void jpeg_fdct(float *data)
{
  float tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
  float tmp10, tmp11, tmp12, tmp13;
  float z1, z2, z3, z4, z5, z11, z13;
  float *dataptr;
  int ctr;

  /* Pass 1: process rows. */

  dataptr = data;
  for (ctr = 0; ctr < DCTSIZE; ctr++) {
    /* Load data into workspace */
    tmp0 = dataptr[0] + dataptr[7];
    tmp7 = dataptr[0] - dataptr[7];
    tmp1 = dataptr[1] + dataptr[6];
    tmp6 = dataptr[1] - dataptr[6];
    tmp2 = dataptr[2] + dataptr[5];
    tmp5 = dataptr[2] - dataptr[5];
    tmp3 = dataptr[3] + dataptr[4];
    tmp4 = dataptr[3] - dataptr[4];

    /* Even part */

    tmp10 = tmp0 + tmp3;    /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    /* Apply unsigned->signed conversion */
    dataptr[0] = tmp10 + tmp11; /* phase 3 */
    dataptr[4] = tmp10 - tmp11;

    z1 = (tmp12 + tmp13) * ((float) 0.707106781); /* c4 */
    dataptr[2] = tmp13 + z1;    /* phase 5 */
    dataptr[6] = tmp13 - z1;

    /* Odd part */

    tmp10 = tmp4 + tmp5;    /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;

    /* The rotator is modified from fig 4-8 to avoid extra negations. */
    z5 = (tmp10 - tmp12) * ((float) 0.382683433); /* c6 */
    z2 = ((float) 0.541196100) * tmp10 + z5; /* c2-c6 */
```

```
    z4 = ((float) 1.306562965) * tmp12 + z5; /* c2+c6 */
    z3 = tmp11 * ((float) 0.707106781); /* c4 */

    z11 = tmp7 + z3;        /* phase 5 */
    z13 = tmp7 - z3;

    dataptr[5] = z13 + z2;  /* phase 6 */
    dataptr[3] = z13 - z2;
    dataptr[1] = z11 + z4;
    dataptr[7] = z11 - z4;

    dataptr += DCTSIZE;     /* advance pointer to next row */
  }

  /* Pass 2: process columns. */

  dataptr = data;
  for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
    tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
    tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
    tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
    tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
    tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
    tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
    tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
    tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];

    /* Even part */

    tmp10 = tmp0 + tmp3;    /* phase 2 */
    tmp13 = tmp0 - tmp3;
    tmp11 = tmp1 + tmp2;
    tmp12 = tmp1 - tmp2;

    dataptr[DCTSIZE*0] = tmp10 + tmp11; /* phase 3 */
    dataptr[DCTSIZE*4] = tmp10 - tmp11;

    z1 = (tmp12 + tmp13) * ((float) 0.707106781); /* c4 */
    dataptr[DCTSIZE*2] = tmp13 + z1; /* phase 5 */
    dataptr[DCTSIZE*6] = tmp13 - z1;

    /* Odd part */

    tmp10 = tmp4 + tmp5;    /* phase 2 */
    tmp11 = tmp5 + tmp6;
    tmp12 = tmp6 + tmp7;

    /* The rotator is modified from fig 4-8 to avoid extra negations. */
    z5 = (tmp10 - tmp12) * ((float) 0.382683433); /* c6 */
    z2 = ((float) 0.541196100) * tmp10 + z5; /* c2-c6 */
    z4 = ((float) 1.306562965) * tmp12 + z5; /* c2+c6 */
    z3 = tmp11 * ((float) 0.707106781); /* c4 */

    z11 = tmp7 + z3;        /* phase 5 */
    z13 = tmp7 - z3;

    dataptr[DCTSIZE*5] = z13 + z2; /* phase 6 */
    dataptr[DCTSIZE*3] = z13 - z2;
```

```
    dataptr[DCTSIZE*1] = z11 + z4;
    dataptr[DCTSIZE*7] = z11 - z4;

    dataptr++;          /* advance pointer to next column */
  }
}
```

在高频系数被舍弃之后就可以对余下的矩阵系数进行量化，减少数据量后再使用Huffman编码完成压缩。