彩色空间转换实验

问题和程序设计

- 分析问题的能力
  - 从计算和程序的角度分析问题
  - 从问题出发，通过逐步分析和分解，把原问题转化为可用程序方式解决的问题。在此过程中设计出一个解决方案。
- 掌握所用的程序语言，熟悉语言中各种结构（包括其形式和意义）
  - 一定要反复实践，开动脑筋想办法处理遇到的各种情况

问题和程序设计

- 学会写程序
  - 确定适用的程序结构
  - 写出的程序是否结构良好，清晰，易于阅读和理解
  - 当有些条件或要求改变时，程序是否容易修改去满足新的要求
- 检查程序错误的能力
  - 语言错误 逻辑错误 算法错误
- 熟悉所用工具和编程环境

本项目的开发进程

- 基本原理
  - RGB和YUV彩色空间的基础知识
  - 数据类型的分析
- 相关编程知识的掌握
  - 文件 缓冲区的开辟与释放 指针的操作
- 熟悉编程环境
- 问题的分析与实现
  - 流程分析
  - 代码实现
- 实验总结

彩色空间转换实验

- 如何新建一个工程
- C相关知识回顾
  - 文件读写，动态数组，指针
- RGB2YUV文件转换流程
- RGB2YUV文件转换的实现

在VC6中新建一个工程

1. FILE—>New—>Project，选择
   Win32 Console Application
2. 选择工程的路径，输入工程的名称(如test1)。选择An empty project。随即打开一个空白的project
3. FILE—>New—>File，选择C++ source file。将Add to Project选中。输入文件名。
4. FILE—>New—>File，选择C/C++ header file。将Add to Project选中。输入文件名。

在Visual Studio中新建一个工程

1. 新建——项目，选择Visual C++模板，选择
   Win32 Console Application（win32控制台应用程序）
2. 选择工程的路径，输入工程的名称(如test1)。
3. 新建源文件和头文件，并进行编辑。

新建一个工程

程序文件一般分为三部分：

头文件：包括与结构（类）的声明和使用这些结构（类）的函数的原型

源代码文件：包含与结构（类）有关的函数的代码，即函数的定义

源代码文件：包含调用这些函数的代码

新建一个工程

头文件常包含的内容

- 函数原型
- 使用#define或const定义的符号常量
- 结构（类）声明
- 模板声明
- 内联函数

不要将函数的定义或变量的声明放在头文件中

新建一个工程

头文件的管理

在同一文件中只能包含同一头文件一次。如何避免？

C/C++技术可以避免这种情况的产生。基于编译预处理命令

```
#ifndef RGB2YUV_H_
#define RGB2YUV_H_
…
#endif
```

指针数组作为main函数的形参

- 不带参数的主函数定义

  void main()

- 带参数的主函数

  定义：**void main(int argc, char *argv[])**

  调用：命令名 参数1 参数2…参数n

  参数值： argc=n+1; argv[0]=命令名 argv[1]=参数1，argv[2]=参数2，….argv[n]=参数 n

指针数组作为main函数的形参

- 例, rgb2yuv工程

在Project—>Settings，右侧Debug条中设定可执行文件所在的路径和工作路径。在Program arguments输入down.rgb down.yuv 256 256

```
void main(int argc, char *argv[])
{
   …  //见例程
}
```

执行：rgb2yuv.exe down.rgb down.yuv 256 256

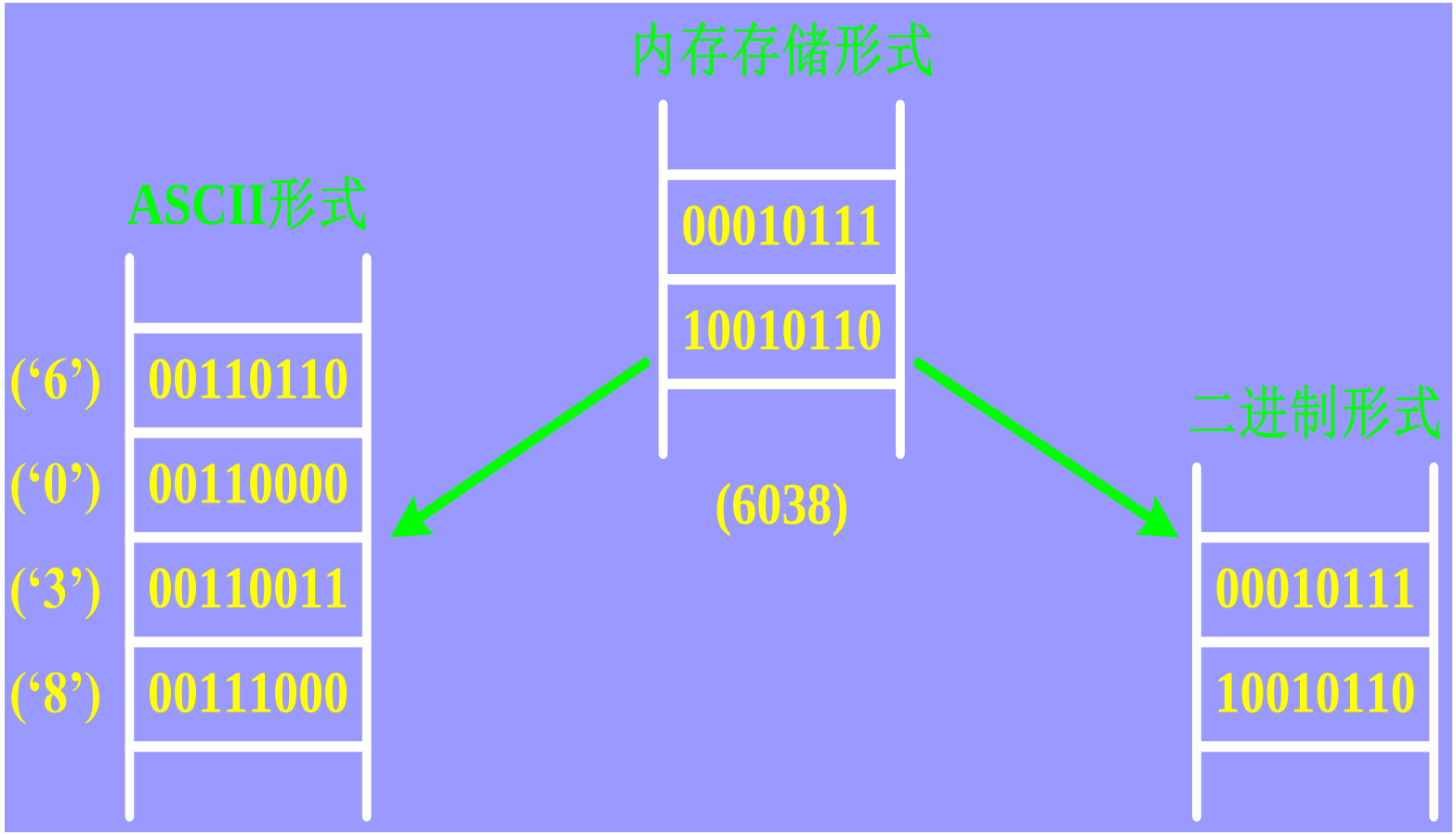
C语言相关知识回顾

- 文件概述
- 文件类型指针
- 文件的打开和关闭
- 文件的读写
- 文件的定位

文件 (File)

- C语言中的文件
  - C语言把文件看作一个字节的序列
  - C语言对文件的存取是以字节为单位的
- 文本文件(ASCII文件)
  - 按数据的ASCII形式存储
- 二进制文件
  - 按数据在内存中的二进制形式存储

文本文件和二进制文件


内存存储形式
00010111
10010110
(6038)
ASCII形式
('6') 00110110
('0') 00110000
('3') 00110011
('8') 00111000
二进制形式
00010111
10010110

文件类型指针

- **FILE**类型
  - 保存被使用的文件的有关信息
  - 所有的文件操作都需要**FILE**类型的指针
  - **FILE**是库文件中定义的**结构体的别名**
  - 注意不要写成**struct FILE**
- 举例
  - **FILE *fp;**

FILE类型

```
typedef struct {
    short          level;  /*缓冲区满空程度*/
    unsigned       flags;  /*文件状态标志*/
    char           fd;     /*文件描述符*/
    unsigned char  hold;   /*无缓冲则不读取字符*/
    short          bsize;  /*缓冲区大小*/
    unsigned char *buffer; /*数据缓冲区*/
    unsigned char *curp;   /*当前位置指针*/
    unsigned       istemp; /*临时文件指示器*/
    short          token;  /*用于有效性检查*/
} FILE;
```

文件的打开 (fopen函数)

- 函数原型
  - **FILE *fopen(char *filename,**
    char *mode);
- 参数说明
  - **filename**:要打开的文件路径
  - **mode** :打开模式
- 返回值
  - 若成功，返回指向被打开文件的指针
  - 若出错，返回空指针**NULL(0)**

文件的打开模式

打开模式	描　　述
r	只读，打开已有文件，不能写
w	只写，创建或打开，覆盖已有文件
a	追加，创建或打开，在已有文件末尾追加
r+	读写，打开已有文件
w+	读写，创建或打开，覆盖已有文件
a+	读写，创建或打开，在已有文件末尾追加
t	按文本方式打开 (缺省)
b	按二进制方式打开

文件的打开举例

```
FILE* rgbFile = NULL;
FILE* yuvFile = NULL;
rgbFile = fopen(rgbFileName, "rb");
if (rgbFile == NULL)
{
        printf("cannot find rgb file\n");
        exit(1);}
yuvFile = fopen(yuvFileName, "wb");
if (yuvFile == NULL)
{
        printf("cannot find yuv file\n");
        exit(1);}
```

文件的关闭 (fclose函数)

- 函数原型
  - **int fclose(FILE *fp);**
- 参数说明
  - **fp:**要关闭的文件指针
- 返回值
  - 若成功，返回**0**
  - 若出错，返回**EOF(-1)**
- 不用的文件应关闭，防止数据破坏丢失

文件的关闭举例

```
FILE *fp;
char file[]="D:\\USER\\STUDENTS.DAT";
if (!(fp=fopen(file, "rb+"))) {
    printf("Open file %s error!\n", file);
    exit(0);
    }
... ...
fclose(fp);
```

feof函数

- 函数原型
  - **int feof(FILE *fp);**
- 参数
  - **fp:**文件指针
- 返回值
  - 若文件结束，返回非零值
  - 若文件尚未结束，返回**0**

文件的读写

函数	功能	函数	功能
fputc	输出字符	**fprintf**	格式化输出
fgetc	输入字符	**fscanf**	格式化输入
putc	输出字符	**putw**	输出一个字
getc	输入字符	**getw**	输入一个字
fwrite	输出数据块	**fputs**	输出字符串
fread	输入数据块	**fgets**	输入字符串

fputc/putc函数

- 函数原型
  - **int fputc(int c, FILE *fp);**
  - **int putc(int c, FILE *fp);**
- 参数
  - **c :**要输出到文件的字符
  - **fp:**文件指针
- 返回值
  - 若成功，返回输出的字符
  - 若失败，返回**EOF**

fgetc/getc函数

- 函数原型
  - **int fgetc(FILE *fp);**
  - **int getc(FILE *fp);**
- 参数
  - **fp:**文件指针
- 返回值
  - 若成功，返回输入的字符
  - 若失败或文件结束，返回**EOF**

fputc和fgetc函数举例

```
FILE *fp1, *fp2;
char c;
fp1 = fopen("file.in", "r");
fp2 = fopen("file.out", "w");
while(!feof(fp1)) {
    c = fgetc(fp1);
    fputc(c, fp2);
    }
fclose(fp1);
fclose(fp2);
```

fwrite和fread函数 (1)

- 函数原型
  - **size_t fwrite(void *buffer,**
    size_t size,
    size_t count,
    FILE *fp);
  - **size_t fread (void *buffer,**
    size_t size,
    size_t count,
    FILE *fp);

fwrite和fread函数 (2)

- 参数
  - **buffer:**要读/写的数据块地址
  - **size :**要读/写的每个数据项的字节数
  - **count :**要读/写的数据项数量
  - **fp :**文件指针
- 返回值
  - 若成功，返回实际读/写的数据项数量
  - 若失败，一般返回**0**

fwrite和fread函数举例

```
rgbBuf = (u_int8_t*)malloc(frameWidth*frameHeight * 3);
yBuf = (u_int8_t*)malloc(frameWidth * frameHeight);
uBuf = (u_int8_t*)malloc((frameWidth * frameHeight) / 4);
vBuf = (u_int8_t*)malloc((frameWidth *frameHeight) / 4);
fread(rgbBuf, 1, frameWidth * frameHeight * 3, rgbFile) );
fwrite(yBuf, 1, frameWidth * frameHeight, yuvFile);
fwrite(uBuf, 1, (frameWidth * frameHeight) / 4, yuvFile);
fwrite(vBuf, 1, (frameWidth * frameHeight) / 4, yuvFile);
```

fprintf和fscanf函数

- 函数原型
  - **int fscanf(FILE *fp,**
    char *format[,address,...]);
  - **int fprintf(FILE *fp,**
    char *format[,argument,...]);
- 说明
  - 与**printf**和**scanf**函数类似
  - 从文件输入或输出到文件

fputs函数

- 函数原型
  - **int fputs(char *s, FILE *fp);**
- 返回值
  - 若成功，返回输出字符个数(或最后的字符)
  - 若失败，返回**EOF**
- 说明
  - 字符串的结束标志**'\0'**不会输出到文件
  - 也不会在字符串末尾自动添加换行符

fgets函数

- 函数原型
  - **char *fgets(char *s, int n,**
    FILE *fp);
- 返回值
  - 若成功，返回**s**首地址；若失败，返回**NULL**
- 说明
  - 从**fp**输入字符串到**s**中
  - 输入**n-1**个字符，或遇到换行符或**EOF**为止
  - 读完后自动在字符串末尾添加**'\0'**

文件的定位

- 文件位置指针
  - 位置指针指向当前读写的位置
  - 每次读写文件，位置指针都会相应移动
  - 可以通过相关函数强制修改位置指针
- 相关函数
  - **rewind**函数
  - **fseek**函数
  - **ftell**函数

rewind函数

- 函数原型
  - **void rewind(FILE *fp);**
- 参数
  - **fp:**文件指针
- 功能
  - 使文件位置指针重新返回文件开头

fseek函数 (1)

- 函数原型
  - **int fseek(FILE *fp, long offset,**
    int whence);
- 参数
  - **fp** **:**文件指针
  - **offset:**偏移量
  - **whence:**起始位置
- 功能
  - 随机改变文件的位置指针

fseek函数 (2)

- 起始位置
  - **SEEK_SET(0):**文件开始
  - **SEEK_CUR(1):**文件当前位置
  - **SEEK_END(2):**文件末尾
- 举例
  - **fseek(fp, 100L, SEEK_SET);**
  - **fseek(fp, -10L, SEEK_CUR);**
  - **fseek(fp, -20L, SEEK_END);**

ftell函数

- 函数原型
  - **long ftell(FILE *fp);**
- 参数
  - **fp:**文件指针
- 功能
  - 返回**fp**所指向文件中的读写位置

C语言回顾 - 动态数组和指针

- 申请空间
- 头文件
- 赋值
- 访问
- 空间回收

动态数组 - 申请空间

- 用如下方法可以定义一个数组：
  - char name[20];//定义一个类型为char，长度为20的静态数组
- 在申请动态数组之前，先定义指向动态数组的指针，然后用malloc()函数申请动态数组：
  - char *name;
  - name = (char *)malloc(20);

动态数组 - 申请空间

```
float *pf;
pf = (float *)malloc(sizeof(float)*20);
```

//申请一个长度为20的数组，数组元素为float类型。

动态数组 - 头文件

- 使用前要在文件的开始包含<malloc.h>文件。例如：

```
#include <malloc.h>
void main( )
{
    double *pd;
    pd = (double *)malloc(sizeof(double)*50);
    ……
}
```

动态数组 - 赋值（指针的运用）

```
int *pi;
pi = (int *)malloc(sizeof(int) * 100);
//100个数组元素，每个元素类型为int
*pi = 0;    //将数组的第1个元素赋值为0
*(pi+1)=1; //将数组的第2个元素赋值为1
*(pi+2)=2; //将数组的第3个元素赋值为2
```

动态数组 - 赋值（指针的运用）

- 与静态数组相同，动态数组的元素下标也是从0开始的。所以对上面pi所指的数组可以用如下的循环将全部元素赋初值为0：

```
int j;
for(j=0; j<100; j++)
    *(pi+j) = 0;
```

动态数组 – 例

```
    for (i = 0; i < x_dim; i ++)
{
    g = b + 1;
    r = b + 2;
    *y = (unsigned char)(  RGBYUV02990[*r] + RGBYUV05870[*g] +
    RGBYUV01140[*b]);
    *u = (unsigned char)(- RGBYUV01684[*r] - RGBYUV03316[*g] +
    (*b)/2          + 128);
    *v = (unsigned char)(  (*r)/2 - RGBYUV04187[*g] -
    GBYUV00813[*b] + 128);
    b += 3;
    y ++;
    u ++;
    v ++;            }
```

动态数组 - 空间回收

- 静态数组是数组变量定义时分配空间的，它的空间在该变量失效时由系统自动回收。
- 动态数组是用malloc()函数动态申请的，需要在程序中调用free()函数主动释放。如：

```
char *pc;  //定义字符指针pc
pc = (int *)malloc(sizeof(char) * 2); //两个数组元素，每个元素类型为char
……
free(pc);
```

RGB2YUV文件转换流程分析

1. 程序初始化（打开两个文件、定义变量和缓冲区等）
2. 读取RGB文件，抽取RGB数据写入缓冲区
3. 调用RGB2YUV的函数实现RGB到YUV数据的转换
4. 写YUV文件
5. 程序收尾工作（关闭文件，释放缓冲区）

RGB2YUV文件转换流程分析

调用RGB2YUV的函数实现RGB到YUV数据的转换

采用部分查找表法，提高运行效率

```
void initLookupTable()
{
        for (int i=0;i<256;i++)
        {
                RGBYUV02990[i] = (float)0.2990 * i;
                RGBYUV05870[i] = (float)0.5870 * i;
                RGBYUV01140[i] = (float)0.1140 * i;
                RGBYUV01684[i] = (float)0.1684 * i;
                RGBYUV03316[i] = (float)0.3316 * i;
                RGBYUV04187[i] = (float)0.4187 * i;
                RGBYUV00813[i] = (float)0.0813 * i;
        }
}
```

YUV2RGB文件转换流程分析

1. 程序初始化（打开两个文件、定义变量和缓冲区等）
2. 读取YUV文件，抽取YUV数据写入缓冲区
3. 调用YUV2RGB的函数实现YUV到RGB数据的转换
4. 写RGB文件
5. 程序收尾工作（关闭文件，释放缓冲区）

YUV2RGB缓冲区分析

1. 两个缓冲区/四个缓冲区

Y(176*144)，U(88*72)，V(88*72)，rgb_out(3*176*144)

Stride_y=width, stride_uv=width/2

stride_out=3*width

4个基本指针，指向正在操作的行

Y，U，V，OUT

5个移动指针，用于操作当前数据

pY，pY2，pU，pV，pOUT，pOUT2

YUV2RGB缓冲区分析

```
for(y=0; y<height; y+=2)
{   unsigned char *pY=Y;
    unsigned char *pY2=Y+stride_y;
    unsigned char *pU=U;
    unsigned char *pV=V;
    unsigned char *pOUT=OUT;
    unsigned char *pOUT2=OUT+stride_out;
    for(x=0; x<width; x+=2)
    {       根据当前的*pY，*pY2，*pU和*pV计算rgb值
            将此rgb值填入*pOUT指向的6个字节（重复两次）
            将此rgb值填入*pOUT2指向的6个字节（重复两次）
            pU++；pV++；pY+=2； pY2+=2
     }
    Y+=2*stride_y;
    U+=stride_u;  V+=stride_v;
    OUT+=2*stride_out;
}
```