项目三报告

12313605 吕梓翀

2025 年 4 月 13 日

1 项目简述

在这个本次的项目中，我完成了一个可以读取bmp文件，并进行处理的程序。我完成了图片亮度调整，图片混合，图片切割，还有图片放缩四个功能。在AI的帮助下生成了基本的框架内容，后来又在用指针代替数组，用移位代替乘除法等等方面进行优化，以提高处理的速度。最后，我又尝试了SIMD指令对程序进行进一步的优化。

2 代码简述

2.1 BMP文件简述与结构体设置

BMP文件包含四个部分：文件头，位图信息头，调色板，位图数据。其中，文件头共有14个字节，位图信息头有40个字节组成，因为本次项目针对24位深的无压缩BMP图片，所以不存在调色板。位图数据的大小取决于位图信息头中的宽度以及高度。文件头中包含文件类型，文件大小，两个保留位，以及位图数据的偏移量。位图信息头包含了图片的长宽，水平垂直分辨率，颜色位深，图片是否被压缩等重要信息。而位图数据就是构成图片的数据了。本次项目的24位深的色彩，分配到红绿蓝三原色，每个颜色对应8位，所以共可以构成16777216种不同的颜色。根据上述内容，我构建了结构体用来储存需要的处理的BMP图像。

```
#pragma pack(push, 1) // 确保结构体紧凑排列
typedef struct
{
```

```
 char signature[2]; // "BM"
 int file_size;     // 文件大小
 short reserved1;   // 保留
 short reserved2;   // 保留
 int offset;        // 像素数据偏移量
} BMPFileHeader;

typedef struct
{
 int header_size;      // 信息头大小
 int width;            // 图像宽度
 int height;           // 图像高度
 short planes;         // 颜色平面数
 short bits_per_pixel; // 每像素位数
 int compression;      // 压缩方式
 int image_size;       // 图像数据大小
 int x_pixels_per_m;   // 水平分辨率
 int y_pixels_per_m;   // 垂直分辨率
 int colors_used;      // 使用的颜色数
 int important_colors; // 重要颜色数
} BMPInfoHeader;

typedef struct
{
 BMPFileHeader file_header;
 BMPInfoHeader info_header;
 unsigned char *pixel_data; // 像素数据
} BMPImage;
#pragma pack(pop)
```

这样模块化的设置结构体可以增加可读性。在结构体的最后设置了指针 unsigned char *pixel_data 用来存入的位图数据的地址。我没有将位图数据全部存入结构体，这样会让结构体变得非常庞大，我只是存入了一个指针，而将真正的位图数据重新分配另一块内存进行储存。此外，结构体内需要

一个固定的内存大小的数组，他不可以根据图像大小变化，这样的会造成内存的大量浪费，以及对可处理的图像大小也有限制。具体的读取方式我将在下一部分叙述。

```
#pragma pack(push, 1)
#pragma pack(pop)
```

这样的办法禁止了结构体储存过程中的内存填充，保证了结构体排列紧凑。我使用这样方式进行优化的原因是，在这个BMP图像处理的过程中，I/O占了很多时间，这样的排列方式可以与原文件排列格式对齐，这样就可以使用fread()大批量的快速读取文件，大大减少了I/O过程的耗时[1].

2.2 BMP文件的读取与写入

在BMP文件读取的过程，我先使用了fopen打开要读取的文件，然后开始依次读取内容，我先分配一部分内存给文件头，然后使用fread读取文件头信息储存下来，然后再读取位图信息头，分配内存储存。最后是位图数据。我按照刚刚读取的数据中，位图信息头的图片长宽信息，分配出一块内存，并让结构体内的指针指向这块内存，然后使用fread一次性将全部的数据读入内存中。

文件读取的过程中，我还会会进行安全性检查。我会检查文件是否存在并且可以被打开，输入的文件是否是24位深的无压缩图片，是否是bmp类型的图片，能否成功读取文件，内存分配是否成功。确保了异常情况下程序正常的退出。（详细内容请参考源代码40-129行，read_bmp 函数）此外，我检查了一般图片可能出现的像素，并针对这个进行了计算。在24位深的无压缩BMP图像的情况下，当一个图片的像素到达6亿的时候，图片的大小就来到了恐怖的1.7GB左右，这个图片实在是太大了，也实在很不常见，同时也有着非常高的处理难度。而且我了解到，32位计算机系统下堆的大小一般只有2GB左右，如果不加限制，那么在32位计算机系统下，我的程序肯定无法处理大数量级的图片。为了确保可拓展性，我将结构体中像素的长宽数据类型设置为了int。接着我设置了一个检查，当输入图片的像素大小的三倍，即像素宽高大小的积超过int类型的范围时，我将拒绝处理如此庞大的图片。为什么是像素大小的三倍呢？因为三倍刚好是位图数据占据

[1] 1

的堆的大小。这样设置方便后续操作，我可以不用担心后续整数溢出的问题。

BMP文件的写入是非常简单的，只需要将新生成的结构体写入即可。使用fwrite就可以快速的完成了，在此处就不过多赘述了。（详细内容请参考源代码132-168行，write_bmp 函数）

2.3 内存释放函数

```
void free_bmp(BMPImage *image)
{
 if (image)
 {
  free(image->pixel_data);
  free(image);
 }
}
```

在这个函数中，我分别释放了图像和位图数据的储存空间。这个函数会在程序异常或正常结束后被调用一次。

2.4 主函数和功能函数

在这一部分我将简要讲述我代码的主函数（处理输入的命令行参数），功能函数（实现调整亮度，混合图像，裁剪，调整大小四个功能的函数）。

2.4.1 主函数

我的主函数会自左向右读取内容，然后根据以读取的内容通过分支，继续进行接下来的操作。整体的参数处理逻辑是这样的

```
for (int i = 1; i < argc; i++) {
 if (strcmp(argv[i], "-i") == 0) {
  // 处理输入文件
 } else if (strcmp(argv[i], "-o") == 0) {
  // 处理输出文件
```

```
  } else if (strcmp(argv[i], "-op") == 0) {
   // 处理操作类型及参数
  }
 }
```

主函数中也有相应的错误处理机制。对于内存分配错误，输入错误，文件操作错误都有相应的方法来保证没有错误出现。针对输入错误，我还增加了输入格式提醒

```
  printf("您输入了不支持或错误的操作\n");
  printf("使用格式:\n");
  printf("调整亮度: ./bmpedit -i input.bmp -o output.bmp -op add value\n");
  printf("图像混合: ./bmpedit -i input1.bmp -i input2.bmp -o output.bmp -op average\n")
  printf("图像剪裁: ./bmpedit -i input.bmp -o output.bmp -op crop x y width height\n");
  printf("图像缩放: ./bmpedit -i input.bmp -o output.bmp -op resize width height\n");
  free_bmp(image1);
  return 1;
```

当操作正确且全程没有其他错误出现时，我会返回0，然后输出操作成功!而有错误时，我会返回1，并描述错误原因，方便使用者进行更改。具体的代码内容很长，不在此处展示了，在源代码的368-527行可以详细的看到。

2.4.2 功能函数

首先我要简述第一个功能，亮度调整功能，这个功能是最基础最好实现的。在此板块我不会讲述代码优化部分，但是展示的部分是优化过的。这将在后面的板块仔细地讲述。

```
 void adjust_brightness(BMPImage *image, int value)
 {
  int row_size = (image->info_header.width * 3 + 3) & ~3; // 每
行实际字节数
  int row_width = image->info_header.width * 3;           // 每
行有效像素数据字节数（去除填充位）
```

```
 unsigned char *pixel = image->pixel_data;

 for (int y = 0; y < image->info_header.height; y++)
 {
  unsigned char *row_start = pixel; // 当前行的起始位置
  for (int x = 0; x < row_width; x++)
  {
   int new_value = *pixel + value;
   *pixel = (new_value > 255) ? 255 : ((new_value < 0) ? 0 : new_value);
   pixel++;
  }
  pixel = row_start + row_size; // 跳过填充位，移动到下一行
 }
}
```

这个代码中，我先获得了结构体中用来储存位图数据的指针，这个指针指向了位图数据的第一个像素点的R信息字节。我在读取BMP图片函数的描述部分中提到了我的前置处理，所以此处 image-¿info_header.width * image-¿info_header.height * 3 不会引起溢出，从而影响后续的操作。接着我获取了位图数据截至处的地址，然后开始进行循环，当当前像素点的色彩点数据在调整亮度后超过255，将被设为255，小于0将被设置为0。这个地方我使用了三目运算符，让我的代码更加简洁。此外，我使用了一个嵌套的for循环来保证了增加亮度操作不会影响到填充位，进而影响图像。在计算增加过填充位后的行长度时，我使用了位运算int row_size = (image-¿info_header.width * 3 + 3) & 3，这让计算的过程得到了一定程度上的加速。后续涉及的相关处理我也是使用了相同的处理方法。

其次我要讲的函数是，图像混合函数。

```
// 混合两幅图像
void blend_images(BMPImage *img1, BMPImage *img2)
{
 // 检查图像尺寸是否相同
 if (img1->info_header.width != img2->info_header.width ||
 img1->info_header.height != img2->info_header.height)
```

```
 {
  fprintf(stderr, "图像尺寸不匹配\n");
  return;
 }

 unsigned char *p1 = img1->pixel_data;
 unsigned char *p2 = img2->pixel_data;
 unsigned char *end = p1 + img1->info_header.width * img1->info_header.height * 3;

 while (p1 < end)
 {
  int sum = (int)*p1 + (int)*p2;
  *p1 = (unsigned char)(sum >> 1);
  p1++;
  p2++;
 }
}
```

这个函数中，首先我要检查的就是图像的尺寸问题。不同尺寸的图像混合我找不到一个合理的处理方法，没有合理的对齐的基准点。之后类似调整亮度的操作，我对他进行了加和取平均值的操作。这里同样的，可能遇到加法后溢出的情况，但可以确保的是，取完平均值后一定小于255，所以我先将他们转为int，进行处理之后再转回无符号整数。这样我就完成了我的混合操作。混合操作没有必要顾及填充位，两个尺寸相同的图片必然有相同的填充位，那么两个0取平均值仍然是0，没有必要再做处理。接下来讲述裁剪功能，这个功能也相对比较简单。由于裁剪后的图被改变了像素数量，所以需要建立一个新的结构体来储存。虽然这样加大了内存消耗，但是处理起来更快。

```
BMPImage *crop_image(BMPImage *image, int x, int y, int width, int height)
{
 // 检查剪裁范围是否有效
 if (x < 0 || y < 0 || width <= 0 || height <= 0 || x + width > image->info_header.wi
 {
```

```
 fprintf(stderr, "剪裁范围无效\n");
 return NULL;
}

// 创建新图像
BMPImage *result = (BMPImage *)malloc(sizeof(BMPImage));
if (!result)
return NULL;

// 复制头部信息并修改尺寸
memcpy(&result->file_header, &image->file_header, sizeof(BMPFileHeader));
memcpy(&result->info_header, &image->info_header, sizeof(BMPInfoHeader));
result->info_header.width = width;
result->info_header.height = height;

// 计算新图像的行大小（包括填充位）
int new_row_size = (width * 3 + 3) & ~3;
result->info_header.image_size = new_row_size * height;
result->file_header.file_size = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader) + resu

// 分配新的像素数据内存
result->pixel_data = (unsigned char *)malloc(result->info_header.image_size);
if (!result->pixel_data)
{
 free(result);
 return NULL;
}

// 计算原图行大小
int old_row_size = (image->info_header.width * 3 + 3) & ~3;
unsigned char *src = image->pixel_data + y * old_row_size + x * 3;
unsigned char *dst = result->pixel_data;
```

```
    // 复制像素数据（自动保留填充位的0值）
    for (int i = 0; i < height; i++)
    {
     memcpy(dst, src, width * 3);
     memset(dst + width * 3, 0, new_row_size - width * 3);
     src += old_row_size;
     dst += new_row_size;
    }
    return result;
  }
```

最后，我要来讲一讲最难的函数，图像缩放函数。这个函数的想法来自对图像混合的处理。在处理图像混合时，我拒绝了处理不同像素大小。所以我就设置了一个图像缩放函数用来处理尺寸不同的图像的混合问题，我们可以缩放后再进行混合。下面我将简单讲述一下代码的逻辑。我参考了OpenCV中对于图像大小重置的相关函数，他们默认的一种高效的处理方法叫做双线性插值法。 这样的操作比简单的单线性插值要好一些，他会

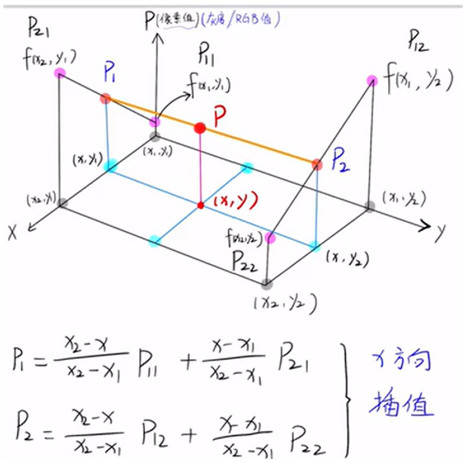

图 1: 双线性插值原理图

让生成的图片锯齿感不那么明显，以及对边界处理的问题的表现也很不错，不会出现边界模糊的情况。在这部分代码中，我依然检测了目标图像的大小，避免生成太大的图像。其余部分与裁剪部分的代码雷同，我特别展示

一下双线性插值的过程

```
    // 计算缩放比例
float x_ratio = (float)old_width / new_width;
float y_ratio = (float)old_height / new_height;

// 计算原图的行大小（包括填充位）
int old_row_size = (old_width * 3 + 3) & ~3;

// 双线性插值算法
for (int y = 0; y < new_height; y++)
{
 float src_y = y * y_ratio;
 int y1 = (int)src_y;
 int y2 = (y1 + 1 < old_height) ? y1 + 1 : y1;
 float y_diff = src_y - y1;

 unsigned char *dst_row = dst + y * new_row_size;  // 当前目标行
的起始位置

 for (int x = 0; x < new_width; x++)
 {
  float src_x = x * x_ratio;
  int x1 = (int)src_x;
  int x2 = (x1 + 1 < old_width) ? x1 + 1 : x1;
  float x_diff = src_x - x1;

  // 获取四个像素点的指针
  unsigned char *p11 = src + y1 * old_row_size + x1 * 3;
  unsigned char *p12 = src + y1 * old_row_size + x2 * 3;
  unsigned char *p21 = src + y2 * old_row_size + x1 * 3;
  unsigned char *p22 = src + y2 * old_row_size + x2 * 3;

  // 对 R、G、B 三个通道分别进行插值
```

```
  unsigned char *dst_pixel = dst_row + x * 3;
  while (dst_pixel < dst_row + x * 3 + 3)
  {
   float r1 = (*p11) * (1 - x_diff) + (*p12) * x_diff;
   float r2 = (*p21) * (1 - x_diff) + (*p22) * x_diff;
   *dst_pixel = (unsigned char)(r1 * (1 - y_diff) + r2 * y_diff);
   p11++;
   p12++;
   p21++;
   p22++;
   dst_pixel++;
  }
 }

 // 填充当前行的尾部
 memset(dst_row + new_width * 3, 0, new_row_size - new_width * 3);
}
```

用这个方法就很快速的实现了对于图像的扩充和缩小。以上就是我本次项目的基本代码内容了。

3 代码运行演示

3.1 生成图片用于演示

在这个部分，我将逐个展示我的代码的运行结果和输入方式。由于互联网上图片本身质量不佳，以及难以体现代码实现的结果是否准确，我让deepseek帮我写了一个python程序，调用了pillow包生成24位深无压缩的BMP图像。

```
from PIL import Image
import numpy as np

# 创建一个新的24位BMP图像（RGB模式）
```

```
width, height = 256, 256  # 图像尺寸
image = Image.new("RGB", (width, height))

# 获取像素访问对象
pixels = image.load()

# 填充像素数据（生成渐变效果）
for y in range(height):
for x in range(width):
# 计算渐变值（0-255）
gradient = int((x + y) * 255 / (width + height))

# 设置RGB颜色（可以调整不同通道的渐变来改变颜色效果）
r = gradient          # 红色通道渐变
g = 0                 # 绿色通道固定
b = 255 - gradient    # 蓝色通道反向渐变

pixels[x, y] = (r, g, b)

# 保存为24位BMP文件
image.save("gradient.bmp", "BMP")

print("渐变效果的24位BMP图片已生成: gradient.bmp")
```

然后我对代码稍加调整，又生成了一个出了斜对角线全部是黑色的图片，下面我先展示一下我的两个图片。

3.2 亮度改变

下面演示第一部分代码，也就是改变图片亮度的部分

```
./bmpedit -i input_24bit.bmp -o out.bmp -op add 100
./bmpedit -i input_24bit.bmp -o out.bmp -op add -100
```

图 2: 演示图片1

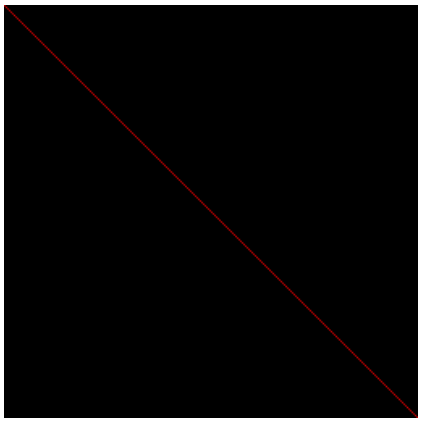

图 3: 演示图片2

图 4: 亮度增加

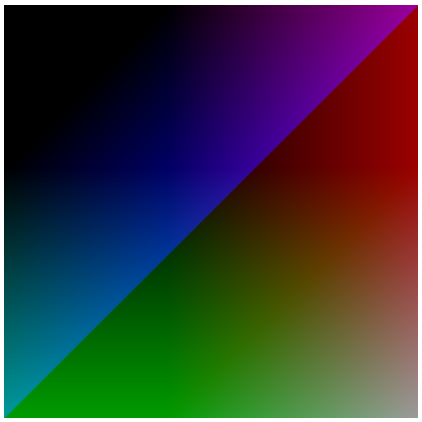

图 5: 亮度减小

3.3 图像混合

接下来演示第二部分，也就是混合图片的代码

```
./bmpedit -i input_24bit_2.bmp -i input_24bit.bmp -o out.bmp -op average
```

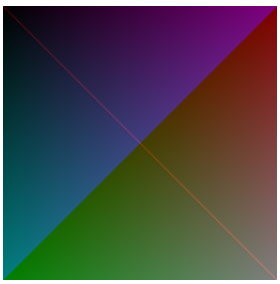

图 6: 混合结果

结合这个图片，我们可以很清晰的看到，图片整体变暗了，并用有一道很明显的红线从左上角到达右下角。这符合我们对图2和图3混合的预期。

3.4 图像裁剪

接下来演示第三部分，也就是图片截取的部分

```
./bmpedit -i input_24bit_2.bmp -o out.bmp -op crop 0 0 150 150
./bmpedit -i input_24bit_2.bmp -o out.bmp -op crop 35 28 78 59
```

3.5 图像缩放

最后一部分是图片的像素放大和缩小

```
./bmpedit -i input_24bit.bmp -o out.bmp -op resize 3 3
./bmpedit -i input_24bit.bmp -o out.bmp -op resize 10000 10000
```

对于缩小的操作，在下面左侧的图片中我们可以看到，整体图片的样貌差不多可以得到存留，但是这样小的缩小确实丢失了很多信息。对于放大操

图 7: 图片裁剪

图 8: 图片裁剪

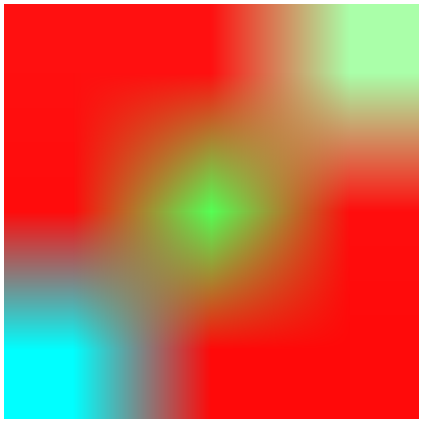

图 9: 图片缩小

图 10: 图片放大

作，我把这个256x256的图像放大了非常大的倍数，图片中，基本没有出现失真的现象，而且图片没有出现明显的锯齿化，此外图像的边界点也都得到了很好的处理。

通过上述的图片，我比较全面的展示了我的运行结果。篇幅有限，我没有再展示我的错误处理效果，但是请相信我，我做了比较全面的考虑。此外，在进行代码演示的时候，我使用了valgrind，检测了我的内存是否出现泄露，检查结果显示均没有内存泄漏的可能出现。

4 优化

4.1 I/O优化

关于I/O的优化我在介绍代码部分简单的提到了一些。我要求结构体排列紧凑，这样就可以直接使用fread，fwrite进行I/O操作，非常的快[2]。我并未对此特性展开实验探究，我在互联网上搜索到了一些相关实验来作证。这样的结构体设置不仅优化了I/O操作，同时也优化了内存的分配。但是，由于不对齐可能造成一些处理上的降速，但是fopen，fwrite对于I/O的操作的优化要更加显著一些，尤其是在面对大数据输入时。此外，结构体中需要多次访问的位图数据内容本身就是对齐的，所以即时我不让他对齐，在处理时也不会有影响。

4.2 操作符优化

这一部分优化起到的作用可能并不显著，我使用了位移操作来大量代替了乘法操作并用三目运算符代替了if判断。不过我后续了解到，可能当前版本的编译器，在编译过程中就针对这些操作有所优化。所以这个改动可能并不会显著影响操作时间。

4.3 指针优化

在AI工具为我生成的初代代码中，以及后续的修改中，他们都很喜欢使用数组代替指针。这会让耗时加大，因为数组每次都要有一次搜索的过程，这远没有直接移动指针效果好。所以我将全部的数组操作都替换成了

[2] 1

指针的操作。尤其是针对宽度很长的数据，这样的更改操作会让程序的时间优化效果更加的明显。

4.4 内存内容优化

在前两个函数体中，AI生成的代码都设置了一个新的结构体来储存，但这是冗余的，我选择直接把数据覆盖在一个原有的图像上。

4.5 使用SDIM优化

我尝试了使用SDIM对代码进行优化，我分别尝试了使用SDIM处理8字节，16字节的数据块。在小规模输入的操作中，他们的效果都不很明显，甚至说几乎和源情况类似。但是在针对大规模输入的时候，都表现出了极好的效果。并且指令集中，还有自动的饱和加法，平均值计算，这都极大程度上的加速了程序的进程。经过实验单独检测计算部分，这样的方法让计算部分的时间减少了十倍，非常惊人的效果。我在使用python调用了 opencv的resize对大规模输入进行操作并计时后与我优化后的代码进行比对，我的代码的性能可以接近opencv效率的百分之八十。通过这个过程也使我了解到了，原来一些python语言的库的底层是用C/C++实现的。

5 总结

5.1 AI工具的使用

其实本次项目中，AI的表现不能说很好，首先他给我生成的代码完全不管填充位，他认为位图数据不需要对齐。我在最初的了解时，我了解到行字节数是需要填充成4的倍数的。但是在AI的误导下，我在相当的一段时间内相信了他。直到我在构建剪切和缩放函数的时候，我发现输出的奇数边长的图片文件出现了问题，被系统识别为损坏的bmp文件。这让我十分苦恼。我调用了opencv，让他进行了相同的操作，然后储存下图片，并使用powershell以十六进制的格式读取文件，这才让我突然明白，填充是必要的。不过总体上AI工具在生成代码框架和代码优化时都给了我很多帮助。我并不是很了解python语言，所以当我想要使用python程序的时候，我就会求助AI，事实上他生成的内容也相当的完美。