

Project3: BMP Image Processing

Contents

1. BMP 图像初窥	2.
1.1. BMP 文件格式	2.
1.2. BMP 文件的编码与解码	7.
2. 程序功能展示	10.
2.1. 帮助	10.
2.2. 图像混合	11.
2.3. 卷积	13.
2.4. 色彩空间转换	14.
2.5. 傅里叶变换	14.
2.6. 基本变换	15.
2.7. 噪点产生	16.
2.8. 图像裁剪	17.
2.9. 图像绘制	17.
2.10. 图像 base64 编码	17.
2.11. ASCII 字符画	18.
2.12. 图像信息	18.
3. 项目思考	19.
3.1. 色彩空间	20.
3.1.1. RGB 与 BGR	20.
3.1.2. HSV	20.
3.1.3. Gray	21.
3.2. 色彩空间转换	22.
3.2.1. RGB 与 BGR	22.

3.2.2. RGB 与 HSV	22.
3.2.3. HSV 与 RGB	24.
3.2.4. RGB 与 Gray	25.
3.3. 卷积	25.
3.4. 傅里叶变换	28.
3.5. 仿射变换	31.
3.5.1. 齐次坐标系	32.
3.5.2. 仿射变换	32.
3.6. 形状绘制	36.
3.6.1. 直线绘制	36.
3.6.2. 圆形绘制	37.
3.7. ASCII 字符画	38.
4. 总结	39.
Bibliography	40.

§1. BMP 图像初窥

BMP(Bitmap)是一种位置无关的位图格式，也称为 DIB(Device Independent Bitmap)，它是由微软公司开发的图像文件格式。BMP 文件通常包含一个位图头部和一个位图数据部分。位图头部包含了图像的元信息，比如文件大小、图像类型、图像大小与位深度等。位图数据部分则按字节为单位存储了图像的像素数据。

一般来说，.bmp 默认是不压缩的 24 位的位图格式，不压缩意味着位图保存每个位置对应像素原始的像素值，24 位的意思是每个像素用 24 位来表示颜色信息，分别对应红色、绿色和蓝色三个通道，这三个通道混合在一起就形成了我们所看到的颜色。由于未经压缩，BMP 文件的体积通常较大，尤其是对于高分辨率的图像来说。不过，BMP 格式的优点在于它的简单性和易于处理，这也是项目文档为什么要求选择 24bit-BMP 格式作为主要处理对象的原因之一。

§1.1. BMP 文件格式

目前主流的 BMP 文件格式包含两个数据头部，一个是位图文件头(BITMAPFILEHEADER)，另一个是位图信息头(BITMAPINFOHEADER)。位图文件头包含了 BMP 文件的基本信息，比如文件大小、位图数据的偏移量等。位图信息头则包含了更详细的信息，比如图像的宽度、高度、颜色深度等。

Bitmap File Header			
位置 (hex/dec)		尺寸 (byte)	描述
00		0	2 头文件字段 对于常见的 bmp 文件，内容为 0x424D (对应 ASCII 码 BM)
02		2	4 整个 .bmp 文件的大小 (little endian)
06		6	2 预留字段，通常为 0
08		8	2
0A		10	4 图片信息的开始位置

上面这张图¹描述了BITMAPFILEHEADER的格式及字段含义，其中最开始的两个字节是文件类型标识符，分别是0x42和0x4D，ascii码对应的字符是B和M，合在一起就是BMP的缩写。这个标识符用于标识文件是否为BMP格式，通常情况下BMP文件的前两个字节都是0x424d，也就是BM。如果不是这个标识符，那么就不是BMP格式的文件了。

实际上除了BM，不同的位图类型还有不同的标识符，比如BA表示OS/2 Bitmap Array，CI表示OS/2 Color Icon，CP表示OS/2 Color Pointer，IC表示OS/2 Icon，PT表示OS/2 Pointer等等，我们在本项目处理BMP文件时只需要关注BM即可。

接下来的是一个4字节的文件大小字段，表示整个BMP文件的大小，包括位图文件头、位图信息头和位图数据部分。这个字段的大小是4字节，表示文件的大小，单位是字节。而后面连续的4个字节是保留字段，通常情况下我们可以忽略它们。接下来是一个4字节的位图数据偏移量字段，表示位图数据在文件中的起始位置。通常这个值为54，因为标准的位图文件头为14字节，位图信息头为40字节，所以位图数据的偏移量就是14+40=54字节。

¹Both the two images of header are the translated version of according item from Wikipedia.

Bitmap Information Header

位置 (hex/dec)	尺寸 (byte)	描述
0E	14	DIB header 的大小 通常为 40 bytes 即 0x28
12	18	图像宽度 (little endian)
16	22	图像高度 (little endian)
1A	26	色彩平面 (color plane) 的数量 必须为 1
1C	28	每像素用多少 bit 来表示
1E	30	采用何种压缩方式 通常不压缩, 即 BI_RGB, 对应值为 0
22	34	图片大小 (原始位图数据的大小) 对于不压缩的图片, 通常表示为 0
26	38	横向分辨率 (像素/米)
2A	42	纵向分辨率 (像素/米)
2E	46	调色板中颜色数量 通常为 0 (不表示没有颜色)
32	50	重要颜色的数量 (通常被忽略) 通常为 0, 表示每种颜色都重要

上面这张图则是关于了 BITMAPINFOHEADER 的格式及字段含义。这个头部包含了比较关键的信息。从 0E 位置起始的两个字节时 BITMAPINFOHEADER 的大小, 通常是 40 字节。

为什么要特别记录这个字段呢? 因为 BMP 文件格式是可以扩展的, BMP 位图信息头的大小并不是固定的, 在不同的颜色深度下, 位图信息头的大小可能会有所不同。比如在 OS/2 位图格式中, 位图信息头的大小为 12 字节, 而在 Windows 位图格式中, 位图信息头的大小为 40 字节。这个字段的大小可以帮助我们判断 BMP 文件的版本和颜色深度。

后面的 4 个字节分别是图像的宽度与高度, 单位为像素。宽度和高度都是有符号整数, 和一般的认知不同的是, 如果高度为正数, 图像的存储方式是从下到上(即最低字节的像素表示原图像左下角的数据), 从左到右; 如果高度为负数, 图像的存储方式是从上到下, 从左到右。我们需要在转换时特别处理这个问题, 以保证图像在处理时的参考系是一致的。

色彩平面数时一个 2 字节的字段, 表示图像的色彩平面数, 在大多数平台上, BMP 文件的色彩平面数为 1, 所以这个字段通常可以忽略。

色深字段是一个 2 字节的字段, 表示图像的颜色深度, 通常为 24 位, 也就是每个像素用 24 位来表示颜色信息。这个字段的值可以是 1、4、8、16、24 或 32, 分别表示 1 位、4 位、8 位、16 位、24 位和 32 位的颜色深度。我们在本项目中只需要处理 24 位的 BMP 文件, 所以这个字段的值要求为 24。

压缩类型字段是一个 4 字节的字段, 表示图像的压缩类型。BMP 文件可以是未压缩的, 也可以是压缩的。常见的压缩类型有 BI_RGB(未压缩)、BI_RLE8(8 位 RLE 压缩)、BI_RLE4(4 位 RLE 压缩)等。我们在本项目中只需要处理未压缩的 BMP 文件, 所以这个字段的值要求为 0。

图像大小字段是一个 4 字节的字段，表示图像数据的大小，单位为字节。这个字段的值可以帮助我们判断图像数据的大小，以便在读取和处理图像时分配足够的内存。

最后的几个字段分别是水平分辨率、垂直分辨率、颜色数和重要颜色数。水平分辨率和垂直分辨率是 4 字节的字段，表示图像的水平和垂直分辨率，单位为像素/米。颜色数是一个 4 字节的字段，表示图像的颜色数。重要颜色数是一个 4 字节的字段，表示图像中重要颜色的数量。这些字段无需要特别处理，通常可以忽略。

在这两个信息头后就是原始位图数据。拿最常见的 24BPP RGB（24 比特每像素，红绿蓝三通道）位图来说，每种颜色需要 8 比特，或者说 1 字节，来存储。在二进制文件中，通常情况下，RGB 按照蓝、绿、红的顺序依次表示图片中的像素点，而 RGBA 则按照蓝、绿、红、透明的顺序（从左下开始，横向逐行向上扫描）。特殊时候，也会出现顺序与上述情况不同的特例，这时色彩顺序会写在 DIB Header 的 Bit Fields 中，以不同色彩通道的 Mask 的形式进行规定。由于 BI_BITFIELDS 也是一种压缩方式，而通常 BMP 不采用任何压缩方式，所以绝大多数时候，我们都是按照前面说的顺序进行排序。

不过，在处理行数据的时候要注意一点，由于早期主流的 CPU 每次从内存中读取并处理数据块的大小(chunk)通常为 32bit(4 字节)，为了提升读取效率，位图数据行的大小通常会被填充到 4 字节的整数倍。比如说，假设我们有一个 1*3 的位图数据，RGB 的顺序是 BGR，那么它的理想位图数据应该是这样的：

```
FF 00 FF
0C 0C 0C
FF 0C 0C
```

但是由于行数据的大小需要补零填充到 4 字节的整数倍，所以实际存储的位图数据应该是这样的：

```
FF 00 FF 00
0C 0C 0C 00
FF 0C 0C 00
```

这就需要我们在读取位图数据时，需要预先计算每行数据大小与实际存储的位图数据大小之间的差值，然后在读取时跳过每行数据的填充部分。这个差值可以通过以下公式计算：

$$\text{padding} = \left(4 - \left(\text{width} * \frac{\text{BitsPerPixel}}{8} \bmod 4 \right) \right) \bmod 4$$

其中，“BitsPerPixel”表示每个像素的位数，“width”表示图像的宽度，“padding”表示每行数据的填充大小。这个公式的意思是：先计算出每行数据的实际大小，然后再计算出每行数据的填充大小。

说完了理论，我们可以来具体看一个 BMP 文件的例子。下面是一个用 vscode-hexeditor 插件打开的 BMP 文件的截图：

project3 > img > bgr.bmp		Data Inspector
Bitmap	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Bitmap
File	42 4D 56 AA 44 00 00 00 00 00 36 00 00 00 28 00	Binary 01000010
Header	00 00 DC 05 00 00 E8 03 00 00 01 00 18 00 00 00	Info 102
Header	00 00 20 AA 44 00 60 0F 00 00 60 0F 00 00 00 00	Header 66
00000030	00 00 00 00 00 00 FF 00 00 FF 00 00 FF 00 00 FF 00	int8 66
00000040	FF 00 00 00 FF 00	uint16 19778
00000050	FF 00 00	int16 19778
00000060	FF 00 00	uint24 5655874
00000070	FF 00 00	int24 5655874
00000080	FF 00 00	uint32 2857782594
00000090	FF 00 00	int32 -1437184702
000000A0	FF 00 00	uint64 294915558722
000000B0	FF 00 00	int64 294915558722
000000C0	FF 00 00	ULEB128 66
000000D0	FF 00 00	SLEB128 -62
000000E0	FF 00 00	float16 21.03125
000000F0	FF 00 00	bfloat16 203423744
00000100	FF 00 00	float32 -1.9033822369791953e-13
		float64 1.457076459886e-312

用 hexeditor 打开 BMP 文件后，我们可以看到文件每行为 16 个字节。其中绿色方框内的前 14 个字节为位图文件头。可以清楚地看到前两个字节为 0x42 和 0x4D，也就是 BM。后面 4 个字节为文件大小字段，为 0x0044AA56，也就是 4500054 个字节。后面四个保留字段为全零也符合我们的预期。接下来是位图数据偏移量字段，为 0x00000036，也就是 54 个字节，这个值也是标准的 BMP 文件头的大小。

接下来是位图信息头，前两个字节为 0x00000028，也就是 40 个字节。后面 4 个字节为图像的宽度和高度，分别为 0x0000005DC 和 0x0000003E8，也就是 1000*1500 的图像。接下来是色彩平面数字段，为 1，色深字段为 24，代表是 24 位的色彩空间。接下来是压缩类型字段，为 0，表示未压缩。图像大小字段为 0x0044AA20，表示图像数据的大小为 4500000 bytes。最后是水平分辨率、垂直分辨率、颜色数和重要颜色数。

水平分辨率和垂直分辨率为 0，表示不限制分辨率。颜色数为 0，表示使用默认的颜色数。重要颜色数为 0，表示所有颜色都是重要颜色。

接下来是位图数据部分。从此处开始三个字节分别对应 B、G、R 三个通道的值。按照小端序阅读顺序，我们可以看到前几个像素的值都为为 0x0000FF，对应的颜色为蓝色，这与原图像左下角的颜色一致。

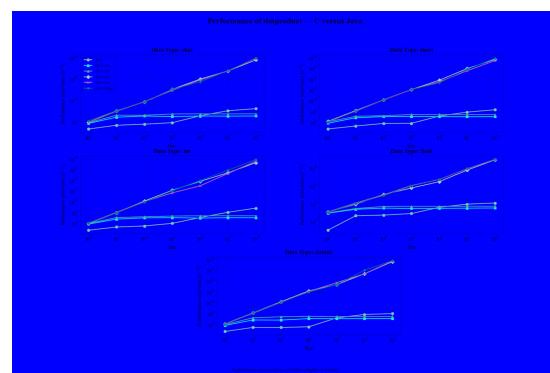


Figure 4: 打开的 BMP 文件的原图

为了观察到字节对齐的效果，我选择一个了一个比较小的图像并用 hexeditor 打开。我们可以看到，图像的宽度为 6，高度为 2，每行数据的大小原本应该是 $6 \times 3 = 18$ 字节，但是由于需要填充到 4 字节的整数倍，所以实际存储的位图数据大小为 20 字节。也就是每行数据后面多了 2 个字节的填充数据。



Figure 5: 2*6 的 BMP 图像数据，可以看到每行数据后面多了 2 个字节的填充数据

因此，我们从实践角度来看，以上的解析格式是正确的。接下来我们就可以开始实现 BMP 文件的编码与解码模块了。

§1.2. BMP 文件的编码与解码

为了方便表示不同长度的字节数据段，我事先利用 `stddef.h` 中的无符号整数类型定义了以下几个新类型作为存储字段的别名：

type	description	name in <code>stddef.h</code>
byte	8-bit data	<code>uint8_t</code>
hword	16-bit data	<code>uint16_t</code>
word	32-bit data	<code>uint32_t</code>
dword	64-bit data	<code>uint64_t</code>

这样一来，我们可以比较简单的定义 BMP 文件头部的结构体了。下面是位图文件头的结构体定义：

```
typedef struct PACKED BitmapFileHeader_ {
    hword bf_type;
    word bf_size;
    word bf_reserve;
    word bf_offset;
} BFHeader;
```

在这里 `PACKED` 是编译器扩展 `_attribute__((__packed__))` 的宏定义，表示这个结构体是紧凑存储的，也就是在内存里应连续存储，这样做的好处是我们在复制 bmp 文件字节流到内置结构体时可以整读整取，避免了字节对齐的问题。

接下来是位图信息头的结构体定义：

```
typedef struct PACKED BitmapInfoHeader_ {
    word bi_size;
    word bi_width;
    word bi_height;
    hword bi_planes;
```

```

    hword bi_bitcnt;
    word bi_compress;
    word bi_size_image;
    word pixels_per_meter_h;
    word pixels_per_meter_v;
    word bi_clr_used;
    word bi_clr_important;
} BIHeader;

```

由于各字段名称比较清晰明了，所以我就不再赘述了。接下来为了存储位图像素数据，我们需要定义像素矩阵的结构体：

```

typedef struct Matu8_2d_ {
    size_t rows;
    size_t cols;
    uint8_t *bytes;
} Matu8_2d, Mat;

```

为了避免整数符号问题，我将行列数都定义为 `size_t` 类型。这个结构体的作用是存储位图像素数据的矩阵，`rows` 和 `cols` 分别表示矩阵的行数和列数，`bytes` 是一个指向字节数据的指针，用于存储位图像素数据。至于为什么使用一维数组来存储二维矩阵的数据，主要是为了提高内存访问的效率。因为在内存中，二维数组的读取需要进行两次地址计算，在大规模数据处理时会影响性能。

综上所述，我们可以定义一个 BMP 文件的结构体来存储 BMP 文件的所有信息：

```

typedef struct BitmapImage {
    BFHeader bfHeader;
    BIHeader biHeader;
    size_t psize;

    Mat r;
    Mat g;
    Mat b;
} BMPImage;

```

这个结构体包含了位图文件头、位图信息头、像素数据大小和 RGB 三个通道的像素数据。`r`、`g` 和 `b` 分别表示红色、绿色和蓝色通道的像素数据，都是一个 `Mat` 类型的结构体。`psize` 表示像素数据的大小，单位为字节。这个字段的值可以帮助在遍历像素数据时判断是否到达了数据的末尾。

定义好了结构体后，我们就可以开始实现 BMP 文件的编码与解码过程了。不过，在此之前，我们需要考虑到该过程中需要涉及到文件 I/O 过程。为了便于处理文件字节流到内存的转换，图像处理库 opencv 在处理各种格式的图像数据前在 `imgcodecs/src/bitstrm.hpp` 模块预先定义了的字节流输入输出流 `RLByteStream/WLByteStream`。

```

// class RLByteStream - uchar-oriented stream.
// l in prefix means that the least significant uchar of a multi uchar value goes
first
class RLByteStream : public RBaseStream
{
public:
    virtual ~RLByteStream();

    int     getByte();
    int     getBytes( void* buffer, int count );
    int     getWord();
    int     getDWord();
}

```

```

};

// class WLByteStream - uchar-oriented stream.
// l in prefix means that the least significant uchar of a multi-byte value goes
// first
class WLByteStream : public WBaseStream
{
public:
    virtual ~WLByteStream();

    bool putByte( int val );
    bool putBytes( const void* buffer, int count );
    bool putWord( int val );
    bool putDWord( int val );
};

```

可以看到，RLByteStream 和 WLByteStream 分别是字节流的输入流和输出流，继承自 RBaseStream 和 WBaseStream。它们分别实现了读取和写入字节、字、双字的方法。我们可以借鉴这个设计思路，来实现一个简单的文件输入输出流。下面是文件输入输出流的定义：

```

typedef struct Stream_ {
    uint64_t ptr;
    uint64_t capacity;
    uint64_t size;
    bool has_closed;
    byte *buffer;
} Stream, IStream, OStream;

```

其中 `ptr` 表示当前指针位置，`capacity` 表示缓冲区的大小，`size` 表示缓冲区中实际存储的数据大小，`has_closed` 表示流是否关闭，`buffer` 是一个指向字节数据的指针，用于存储缓冲区的数据。由于输入与输出的逻辑在本质上是相同的，所以这个结构体既是输入流也是输出流。在标识上我们使用 `IStream` 和 `OStream` 来分别表示输入流和输出流。

这个输入输出流会绑定到文件指针上，使用时需要传入文件指针和缓冲区大小。在文件流和内存流之间进行转换时，我们可以使用 `fread` 和 `fwrite` 函数来实现。读取 BMP 文件时，我们使用 `IStream_readBytes` 函数来读取流中的字节数据，写入 BMP 文件时，我们使用 `OStream_writeBytes` 函数来写入流中的字节数据。下面是这两个函数的实现：

```

uint64_t IStream_readBytes(IStream *istream, byte* buffer, uint64_t num_bytes)
{
    if (!istream || !buffer || !istream->buffer) return 0ULL;
    uint64_t available_bytes = IStream_availableBytes(istream);
    uint64_t read_bytes = (available_bytes >= num_bytes) ? num_bytes :
available_bytes;
    memcpy(buffer, istream->buffer + istream->ptr, read_bytes);
    istream->ptr += read_bytes;
    return read_bytes;
}
uint64_t OStream_writeBytes(OStream *ostream, const byte *bytes, uint64_t num_bytes)
{
    if (!ostream || !ostream->buffer || !bytes) return 0ULL;
    if (ostream->has_closed)
    {
        WARNING("Stream has been closed!");
        return 0ULL;
    }
    uint64_t available_bytes = OStream_availableBytes(ostream);
    if (available_bytes < num_bytes)

```

```

{
    uint64_t capacity = OStream_dilate(ostream, (ostream->size<<1) + 1);
    if (capacity == 0ULL)
    {
        WARNING("The memory fails to reallocate!");
        return 0ULL;
    }
    available_bytes = OStream_availableBytes(ostream);
}
uint64_t write_bytes = (available_bytes >= num_bytes) ? num_bytes :
available_bytes;
memcpy(ostream->buffer + ostream->ptr, bytes, write_bytes);
ostream->ptr += write_bytes;
ostream->size = ostream->ptr;
return write_bytes;
}

```

如此一来我们在编解码的时候使用这两个函数来进行稳定的字节存取操作，如果读取时读取到的字节数小于请求的字节数，那么就相应的在这个位置进行错误处理。同时，将正确的字节数据复制到 BMP 结构体的相应字段中。写入的过程也是类似的，只不过是将字节数据写入到文件中。

完成读取后，我们需要像 `fclose` 函数一样关闭输入输出流，同时释放缓冲区的内存，确保所有资源及时释放。

§2. 程序功能展示

Note (推荐编译选项).

```
gcc -Wall -lm -O3 submission_ver.c -o cimage
```

这个项目实现了较为完善的 24-bit BMP 图像的处理功能，包括图像的指定模式混合、色彩空间转换、图像的缩放、旋转、翻转、裁剪、傅里叶变换、图像卷积、形状绘制、base64 编码等功能。下面是一些功能的展示：

§2.1. 帮助

```

> ./cimage
>>> CImageProcess - BMP Image Processing CLI
Supported formats: 24-bit BMP (RGB, BGR, HSV, grayscale)

Commands:
  cimage op [src1.ext] + [src2.ext] -o [dest.ext] -r [rule] [--wl=xxx] [--w2=xxx]
  cimage conv [src1.ext] x [src2.ext] -o [dest.ext] [--no-padding]
  cimage cvtclr -i [src.ext] -o [dest.ext] -m [mode]
  cimage fft -i [src.ext] -o [dest.ext] [--no-shift]
  cimage scale -i [src.ext] -o [dest.ext] --sx=[scaleX] --sy=[scaleY]
  cimage rotate -i [src.ext] -o [dest.ext] -a [angle]
  cimage flip -i [src.ext] -o [dest.ext] --x --y
  cimage trans -i [src.ext] -o [dest.ext] --tx=[transX] --ty=[transY]
  cimage noise -i [src.ext] -o [dest.ext] -p [probability]
  cimage art -i [src.ext] -c [color]

```

```
cimage line -i [src.ext] -o [dest.ext] -c [color] --x0=[xxx] --y0=[xxx] --x1=[xxx]
--y1=[xxx]
cimage rect -i [src.ext] -o [dest.ext] -c [color] --x=[xxx] --y=[xxx] --dx=[xxx] --
dy=[xxx]
cimage circle -i [src.ext] -o [dest.ext] -c [color] --r=[xxx] --rx=[xxx] --ry=[xxx]
cimage clip -i [src.ext] -o [dest.ext] --x=[xxx] --y=[xxx] --dx=[xxx] --dy=[xxx]
cimage b64 -i [src.ext] -o [dest.ext]
cimage info -i [src.ext]
```

For detailed help on each command, use: cimage [command] --help

直接运行 cimage 命令可以查看帮助信息。可以看到，支持的命令有：

leading	needed arguments	optional arguments	description	remark
op	src1, src2, dest	rule, w1, w2	对两个图像的三个通道进行混合，可指定混合规则	w1 和 w2 分别表示两个图像的权重，默认值为 0.5，在混合规则为 weight 时有效
conv	src1, src2, dest	no-padding	对两个图像的三个通道进行卷积	-no-padding 表示不进行边缘填充
cvtclr	src, dest	mode	对图像进行色彩空间转换	
fft	src, dest	no-shift	对图像进行傅里叶变换	-no-shift 表示不进行频谱中心化处理
scale	src, dest	scaleX, scaleY	对图像进行缩放	-sx 和 -sy 分别表示 x 轴和 y 轴的缩放比例，默认值为 1
rotate	src, dest, angle		对图像进行旋转	
flip	src, dest, angle	x, y	对图像进行翻转	-x 表示水平翻转，-y 表示上下翻转
noise	src, dest, probability		对图像进行椒盐噪声处理	-p 表示噪声概率
art	src, color		将图像转换为灰度字符画	-c 表示字符画的颜色，默认值为 0xFFFFFFFF，也就是白色
line	src, dest	color, x0, y0, x1, y1	在图像上绘制直线	
rect	src, dest	color, x, y, dx, dy	在图像上绘制矩形	
circle	src, dest	color, r, rx, ry	在图像上绘制圆形	
clip	src, dest	x, y, dx, dy	裁剪图像	
b64	src, dest		对图像进行 base64 编码	
info	src		查看图像信息	

§2.2. 图像混合

我们使用如下指令可以均等混合 arisu.bmp 和 mask.bmp 两幅图像：

```
./cimage op arisu.bmp + mask.bmp -o mask_arisu.bmp
```



“+”



经过混合后，我们可以看到 arisu.bmp 的图像因为 mask.bmp 的纯白色而显得更淡了一些。

我们选择权重混合规则，并指定权重分别为为 0.8 和 0.2，分别对应 arisu.bmp 和 mask.bmp 两幅图像：

```
./cimage op arisu.bmp + mask.bmp -o mask_arisu.bmp -r weight --w1=0.8 --w2=0.2
```



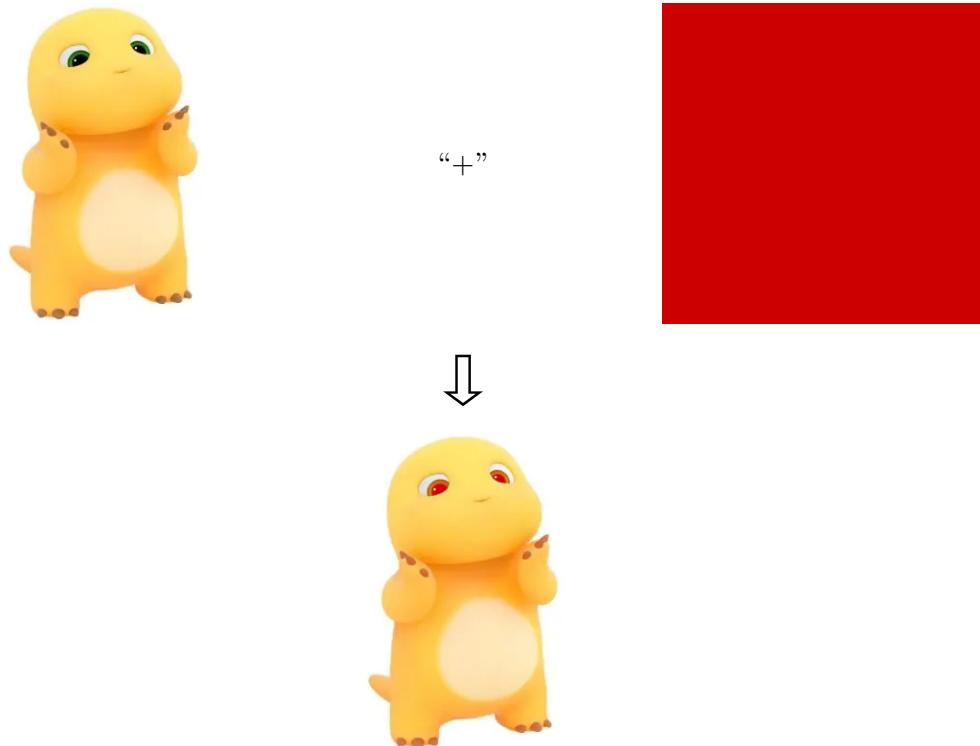
“+”



经过混合后，我们可以看到，由于 mask 权重被降低，混合后的图像的明度更接近于 arisu.bmp 的原图像。

基于最大值的混合规则也可以使用，命令如下：

```
./cimage op nailoong.bmp + mask1.bmp -r max -o mask_nailoong.bmp
```



根据简单的色彩知识，我们知道白色背景色的RGB值为(255, 255, 255)，而红色遮罩图像的RGB值为(255, 0, 0)，所以在混合时，只有人物瞳色发生了明显的变化，其他部分的颜色基本不变。

Warning (注意).

混合时要求两幅图像的形状必须一致，否则会报错。

§2.3. 卷积

我们使用如下指令可以对基 aya.bmp 和卷积核 nk.bmp 进行卷积操作：

```
./cimage conv aya.bmp x nk.bmp -o conv_aya.bmp
```





经过卷积后，我们可以看到图像被染上了卷积核的颜色，同时图像的细节也被模糊化了。这个效果类似于均值模糊，卷积核的大小和权重会影响模糊的程度。

--no-padding 参数表示不进行边缘零填充，相比原图像，卷积后的图像会变小。我们可以使用如下指令进行卷积。

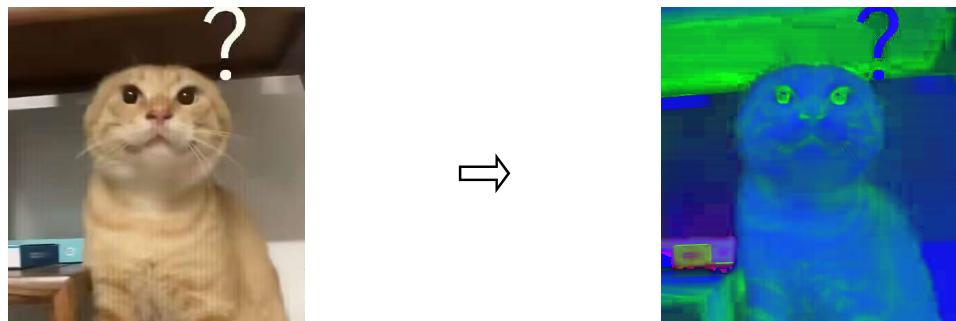
Warning (注意).

卷积时要求卷积核的大小必须为奇数，否则会报错。

§2.4. 色彩空间转换

我们使用如下指令可以对 hachimi.bmp 进行 rgb 到 hsv 的色彩空间转换，其中 -m 参数用于指定转换模式，

```
./cimage cvtclr -i hachimi.bmp -o hsv.bmp -m rgb2hsv
```

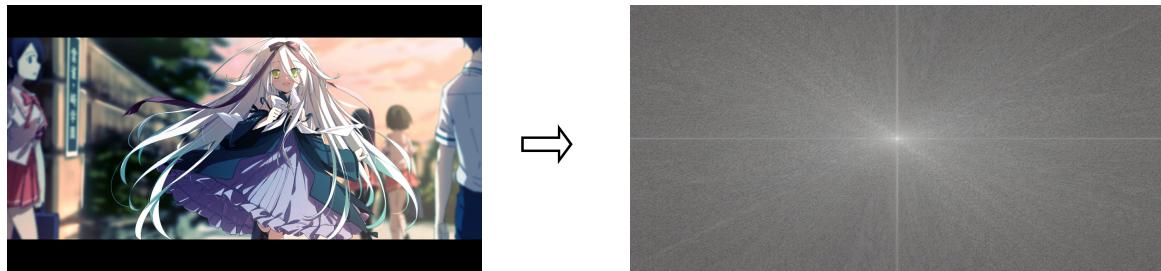


事实上，我们支持多种色彩空间转换，包括 rgb2hsv、hsv2rgb、rgb2bgr、bgr2rgb、rgb2gray、hsv2bgr、bgr2hsv 等。

§2.5. 傅里叶变换

傅里叶变换是图像处理中的一种重要技术，可以将图像从空间域转换到频率域。我们使用如下指令可以对 meia.bmp 进行傅里叶变换：

```
./cimage fft -i meia.bmp -o fft.bmp
```

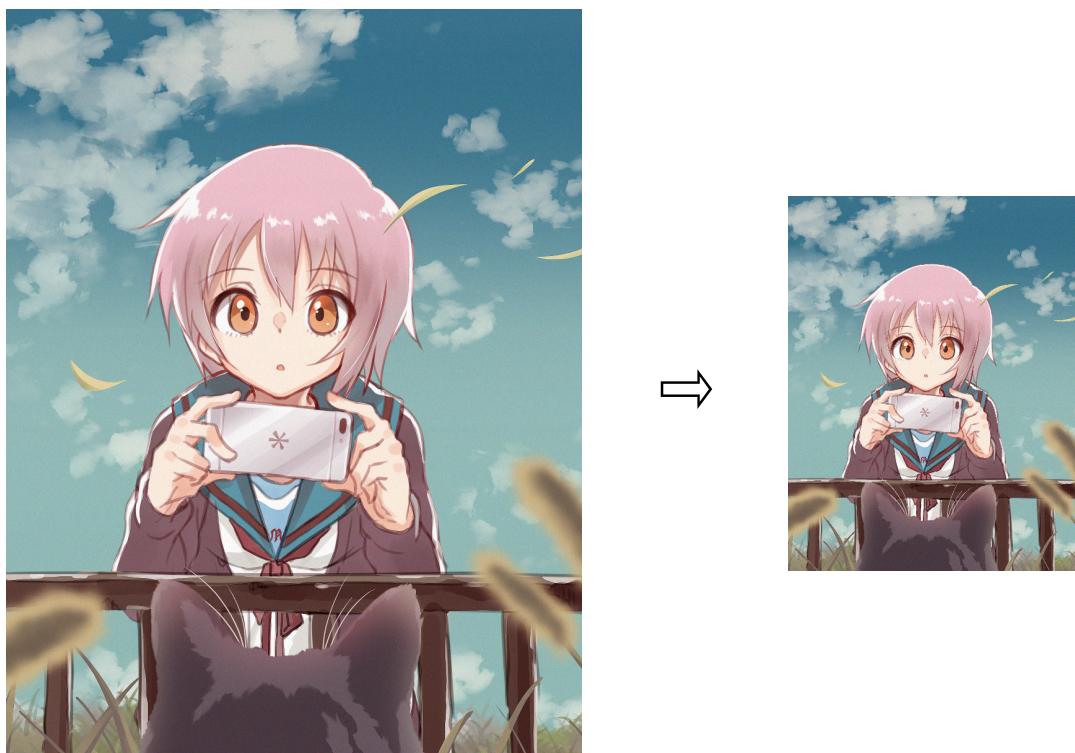


--no-shift 参数表示不进行频谱中心化处理，傅里叶变换后的频谱图像高频信号会分布在图像的四个角落，低频信号分布在图像的中心。

§2.6. 基本变换

使用 scale 命令可以对图像进行缩放，sx 和 sy 参数分别是用于指定 x 轴和 y 轴的缩放比例，这个值需要为正数，样例命令如下：

```
./cimage scale -i yuki.bmp -o scale.bmp --sx=0.5 --sy=0.5
```



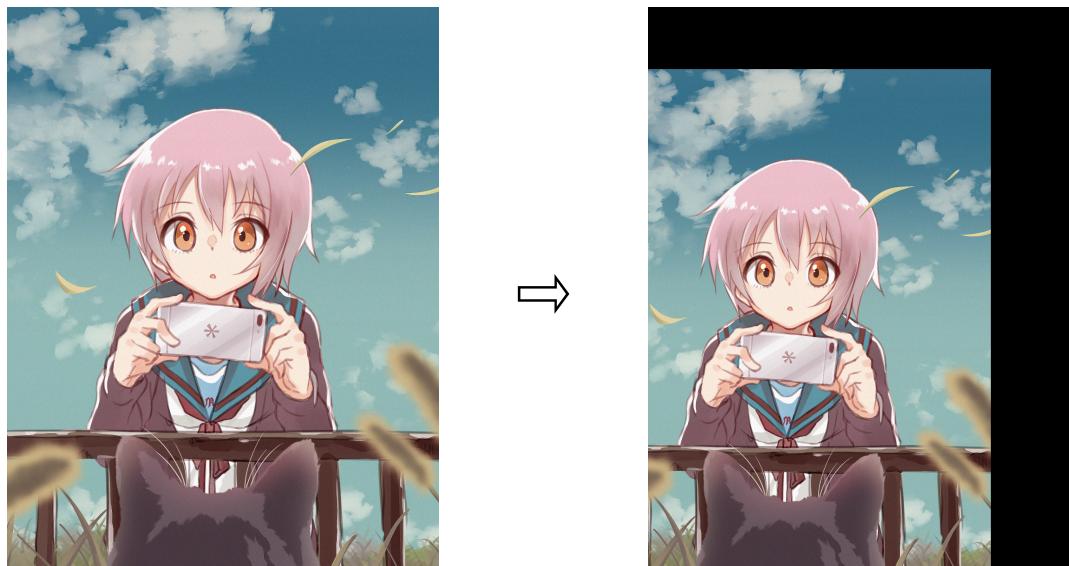
使用 rotate 命令可以对图像绕中心进行顺时针旋转，-a 参数用于指定旋转角度，单位为度，样例命令如下：

```
./cimage rotate -i yuki.bmp -o rot50.bmp -a 50
```



使用 `trans` 命令可以对图像进行平移, `tx` 和 `ty` 参数分别是用于指定 x 轴和 y 轴的平移距离, 样例命令如下:

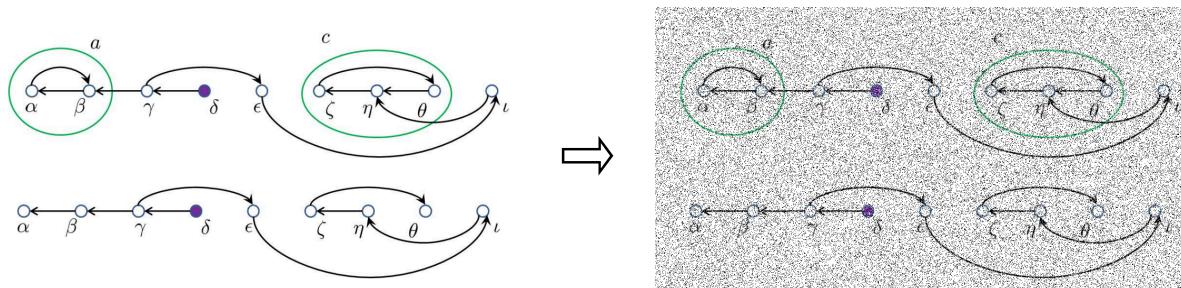
```
./cimage trans -i yuki.bmp -o trans.bmp --tx=-100 --ty=150
```



§2.7. 噪点产生

使用 `noise` 命令可以对图像进行椒盐噪声处理, `-p` 参数用于指定噪声概率, 样例命令如下:

```
./cimage noise -i chuliu.bmp -o noise.bmp -p 0.3
```



§2.8. 图像裁剪

使用 `clip` 命令可以对图像进行裁剪，`x` 和 `y` 参数分别是用于指定裁剪区域的左上角坐标，`dx` 和 `dy` 参数分别是用于指定裁剪区域的宽度和高度，样例命令如下：

```
./cimage clip -i arisu.bmp -o clip_arisu.bmp --x=500 --y=100 --dx=480 --dy=436
```



§2.9. 图像绘制

使用 `line` 命令可以在图像上绘制直线，`x0`、`y0`、`x1` 和 `y1` 参数分别是用于指定直线的起点和终点坐标，`-c` 参数用于指定直线的颜色，接受十六进制格式颜色，样例命令如下：

```
./cimage line -i mami.bmp -o line.bmp --x0=16 --y0=172 --x1=278 --y1=172 -c ff0000
```



使用 `rect` 和 `circle` 命令可以在图像上绘制矩形和圆形，其用法与 `line` 命令类似。`rect` 命令的 `x` 和 `y` 参数分别是用于指定矩形的左上角坐标，`dx` 和 `dy` 参数分别是用于指定矩形的宽度和高度；而 `circle` 命令的 `r`、`rx` 和 `ry` 参数分别是用于指定圆形的半径、圆心 `x` 坐标和圆心 `y` 坐标。

§2.10. 图像 base64 编码

使用 `b64` 命令可以对图像进行 base64 编码，输出格式为 base64 字符串，命令如下：

```
./cimage b64 -i Ereshkigal.bmp -o base64.txt
```



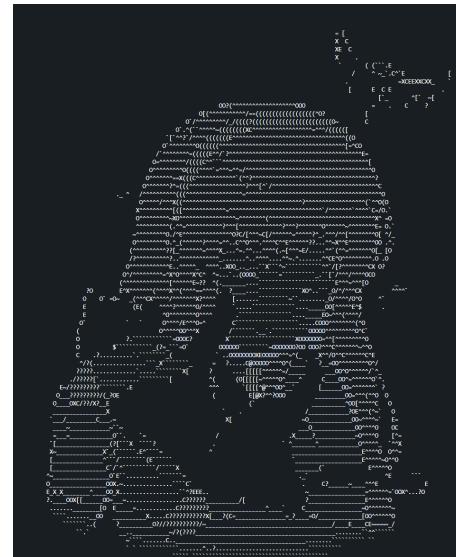
“Qk12WQcAAAAAADYAAAoAAAA...”
base64 string

Base64 编码适用于 60KB 大小以下的图像在网络中传输。

§2.11. ASCII 字符画

使用 art 命令可以将图像转换为 ASCII 字符画输出在终端, -c 参数用于指定字符画的颜色, 接受十六进制格式颜色, 样例命令如下:

```
./cimage art -i yukina.bmp -c 000000
```



推荐设置终端或文本编辑器的字体为等宽字体(如 Consolas), 这样可以更好地显示字符画效果。

§2.12. 图像信息

使用 info 命令可以查看图像信息, 命令如下:

```
./cimage info -i arisu.bmp
```

输出信息如下:

```
[NOTICE]: ./src/main.c:2157: at function 'image_info'
└─Displaying information for image arisu.bmp
```

```
==== BMP File Header ====
Type: 0x4D42
Size: 6266934 bytes
Reserved: 0x00000000
Offset: 54 bytes
```

```

==== BMP Info Header ====
Size: 40 bytes
Width: 1920 pixels
Height: 1088 pixels (bottom-up)
Planes: 1
Bit Count: 24 bits
Compression: 0
Image Size: 6266880 bytes
X Resolution: 3778 pixels/meter
Y Resolution: 3778 pixels/meter
Colors Used: 0
Important Colors: 0

==== Image Data ====
Dimensions: 1088 x 1920 pixels

==== RGB Sample (top-left corner) ====
Pixel[0,0]: R= 39 G= 56 B=107
Pixel[0,1]: R= 34 G= 54 B=104
Pixel[0,2]: R= 35 G= 56 B=102
Pixel[0,3]: R= 38 G= 57 B=108
Pixel[0,4]: R= 35 G= 54 B=105
Pixel[1,0]: R= 34 G= 53 B=100
Pixel[1,1]: R= 36 G= 56 B=104
Pixel[1,2]: R= 32 G= 53 B=100
Pixel[1,3]: R= 33 G= 54 B=101
Pixel[1,4]: R= 33 G= 55 B=103
Pixel[2,0]: R= 31 G= 51 B=100
Pixel[2,1]: R= 37 G= 58 B=105
Pixel[2,2]: R= 33 G= 52 B=100
Pixel[2,3]: R= 34 G= 54 B=102
Pixel[2,4]: R= 35 G= 56 B=104
Pixel[3,0]: R= 33 G= 53 B=100
Pixel[3,1]: R= 34 G= 54 B=100
Pixel[3,2]: R= 38 G= 60 B=106
Pixel[3,3]: R= 32 G= 54 B=102
Pixel[3,4]: R= 35 G= 57 B=102
Pixel[4,0]: R= 36 G= 56 B=106
Pixel[4,1]: R= 33 G= 51 B=101
Pixel[4,2]: R= 35 G= 55 B=104
Pixel[4,3]: R= 31 G= 50 B= 99
Pixel[4,4]: R= 34 G= 55 B=101

==== Image Statistics ====
Averages: R=35.59 G=53.00 B=91.15

```

该命令会输出 BMP 文件头、位图信息头的各字段信息，以及图像的宽度和高度，部分像素数据的 RGB 值，以及图像的平均 RGB 值。

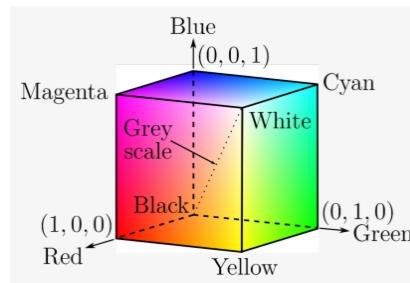
§3. 项目思考

由于已经有了 BMP 的基本框架，一部分简单操作的实现实际上并没有太大难度。我只对这个程序开发中一些值得一提的实现细节进行详细的介绍。

§3.1. 色彩空间

§3.1.1. RGB 与 BGR

RGB 图像是我们最熟知的色彩空间之一。它是一种加色模型，通过将红色、绿色和蓝色三种颜色的光线以不同的强度混合来产生其他颜色。如下图所示，RGB 色彩的三原色分别是色彩空间的三个维度，所有色彩都均匀分布在这个 $255 \times 255 \times 255$ 立方体的表面上。原点到白色的对角线是代表 RGB 色彩灰度的变化线。



在 RGB 色彩空间中，一些色彩的性质可以描述如下：

- 色彩变化：三个坐标轴 RGB 最大分量顶点与黄紫青 YMC 色顶点的连线
- 深浅变化：RGB 顶点和 CMY 顶点到原点和白色顶点的中轴线的距离
- 明暗变化：中轴线的点的位置，到原点，就偏暗，到白色顶点就偏亮

RGB 色彩也是最接近人类视觉感知原理的混合模式，比如说我们看到的白色光线是由红色、绿色和蓝色三种颜色的光线以相同的强度混合而成的。

BGR 色彩则是在存储时将 R 通道和 B 通道的顺序进行了交换。其他的性质和 RGB 是一样的。BGR 色彩空间是 OpenCV 默认的色彩空间格式，这和其他图像处理库的 RGB 色彩空间格式相反的，出于 OpenCV 在图像处理领域的重要地位，我们在此特别提及。

§3.1.2. HSV

HSV(Hue, Saturation, Value)是根据颜色的直观特性，由 A. R. Smith 在 1978 年创建的一种颜色空间，也称六角锥体模型(Hexcone Model)。这个模型中颜色的参数分别是：色调(H)，饱和度(S)，明度(V)。

1. 色调 Hue

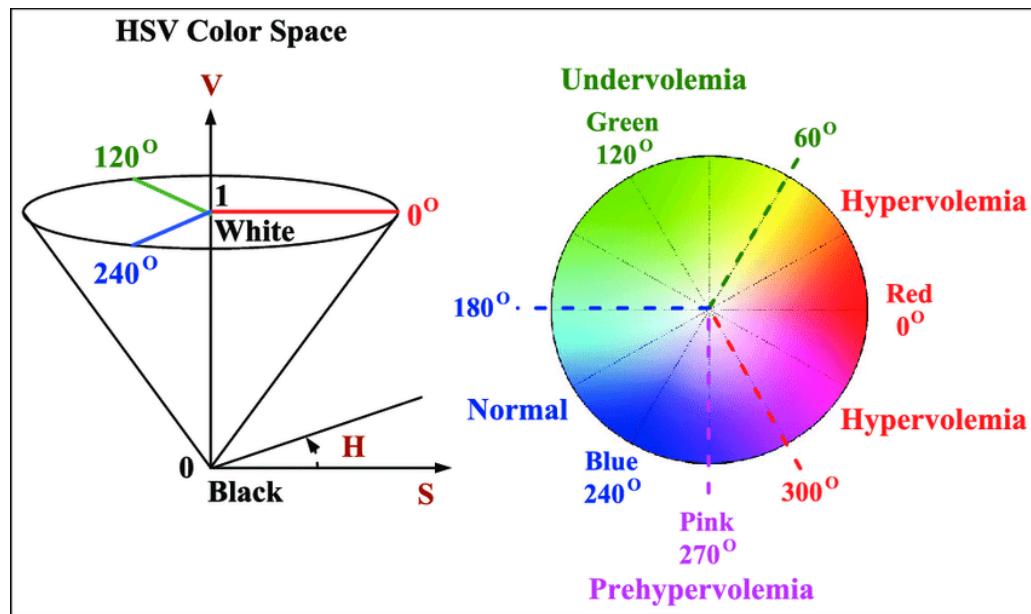
在 HSV 色彩空间中，色调是指颜色的种类，它们分布在一个圆环上。色调的值范围是 0 到 360 度。从红色开始按逆时针方向计算，红色为 0° ，绿色为 120° ，蓝色为 240° 。它们的补色是：黄色为 60° ，青色为 180° ，品红为 300° ；

2. 饱和度 Saturation

饱和度 S 表示颜色接近光谱色的程度。一种颜色，可以看成是某种光谱色与白色混合的结果。其中光谱色所占的比例愈大，颜色接近光谱色的程度就愈高，颜色的饱和度也就愈高。饱和度高，颜色则深而艳。光谱色的白光成分为 0，饱和度达到最高。通常取值范围为 0%~100%，值越大，颜色越饱和。

3. 明度 Value

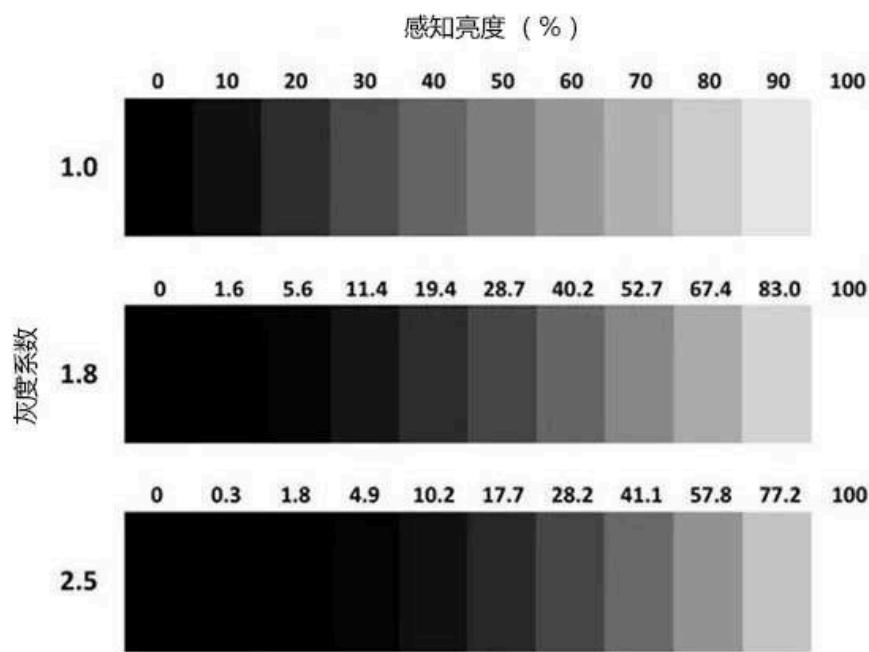
明度表示颜色明亮的程度，对于光源色，明度值与发光体的光亮度有关；对于物体色，此值和物体的透射比或反射比有关。通常取值范围为 0%（黑）到 100%（白）。



一般来说，HSV 色彩的混合模式更符合直觉，因为我们很难判断出靛蓝与橙色混合会产生怎样的颜色；但如果是对品红色调低明度或者增加饱和度，我们就能很容易的想象出它的颜色变化。因此 HSV 色彩空间在图像处理、计算机视觉等领域得到了广泛的应用。

§3.1.3. Gray

灰度图像，或者叫灰阶图像，是一种只有亮度信息的图像。它是通过将 RGB 图像中的红、绿、蓝三种颜色的亮度值进行加权平均得到的。灰度图像通常使用 8 位表示，每个像素的值范围是 0 到 255，其中 0 表示黑色，255 表示白色，中间的值表示不同的灰色。下图是一个描述不同灰度系数下的灰度阶调的图像：



在 OpenCV 文档 *Color conversions* 一节²中将 RGB 转为灰度图像的公式描述为：

²see https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html

$$\text{RGB}[A] \text{ to Gray: } Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

这对应与 OpenCV 的 `cv::cvtColor` 函数中的 `cv::COLOR_BGR2GRAY` 和 `cv::COLOR_RGB2GRAY` 转换模式。在这个函数中，`cv::COLOR_BGR2GRAY` 表示将 BGR 格式的图像转换为灰度图像，而 `cv::COLOR_RGB2GRAY` 表示将 RGB 格式的图像转换为灰度图像。

灰度图相对于 RGB 图像来说处理更加方便，在图像二值化、边缘检测、特征提取等操作中，灰度图像通常是首选的输入格式。在图像处理中，灰度图像通常用于图像的预处理和特征提取。由于灰度图像只包含亮度信息，因此它们在存储和处理时比彩色图像更高效。灰度图像也可以用于边缘检测、形态学操作等任务，因为这些操作通常只依赖于亮度信息，而不需要颜色信息。

§3.2. 色彩空间转换

§3.2.1. RGB 与 BGR

这个操作非常简单，只需要交换 R 通道和 B 通道的值即可。在实现上是通过交换 R 通道的 Mat 数组指针和 B 通道的 Mat 数组指针来实现的。

§3.2.2. RGB 与 HSV

在转换 RGB 到 HSV 之前，我们需要明确 H、S、V 三个分量的对应到 RGB 的关系。

明度 V 是最容易得出的分量，它表示颜色的亮度。一个 RGB 颜色的亮度即为三个通道的最大值：

$$V = \max(R, G, B)$$

饱和度 S 表示颜色的纯度。它是指颜色中最小分量与最大分量的差值与最大分量的比值：

$$S = \frac{\max(R, G, B) - \min(R, G, B)}{\max(R, G, B)}$$

确定 H 分量比较复杂。首先需要根据最大分量对应的通道来判断大致的色调范围：

- 如果最大分量是 R 通道，则色调范围应在 -60° 到 60° 之间；
- 如果最大分量是 G 通道，则色调范围应在 60° 到 180° 之间；
- 如果最大分量是 B 通道，则色调范围应在 180° 到 300° 之间；

接着我们从分区中点开始，根据其他分量的值来计算 H 分量的值。

具体的公式如下图所示，最大分量为 R 时最后得到的结果可能会出现负数，我们将其加上 360° 即可。

■ colour cone

- $H = \text{hue} / \text{colour in degrees} \in [0, 360]$

- $S = \text{saturation} \in [0, 1]$

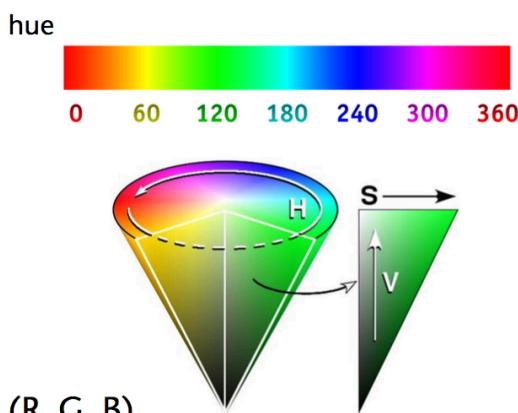
- $V = \text{value} \in [0, 1]$

■ conversion RGB \rightarrow HSV

- $V = \max = \max(R, G, B), \min = \min(R, G, B)$

- $S = (\max - \min) / \max \quad (\text{or } S = 0, \text{ if } V = 0)$

$$\begin{aligned} H &= 60 \times \begin{cases} 0 + (G - B) / (\max - \min), & \text{if } \max = R \\ 2 + (B - R) / (\max - \min), & \text{if } \max = G \\ 4 + (R - G) / (\max - \min), & \text{if } \max = B \end{cases} \\ H &= H + 360, \text{ if } H < 0 \end{aligned}$$



当然,为了能在8-bit的范围内存储,我们按照OpenCV的规范化方法将HSV的范围伸缩到0-255之间。

$$H \leftarrow \frac{H}{2}, V \leftarrow V * 255, S \leftarrow S * 255$$

C 代码如下:

```
color cvtclr_rgb2hsv(color clr)
{
    double rn = (double)clr.rgb[2] / 0xFF;
    double gn = (double)clr.rgb[1] / 0xFF;
    double bn = (double)clr.rgb[0] / 0xFF;

    double cmax = MAX(rn, gn, bn);
    double cmin = MIN(rn, gn, bn);
    double delta = cmax - cmin;

    double hn = 0., sn = 0., vn = 0.;

    /* calculate h value */
    if (delta < FLT_EPSILON) { // 避免浮点精度问题
        hn = 0;
    } else if (cmax == rn) {
        hn = 60 * (((gn - bn) / delta) + (gn < bn ? 6 : 0)); // 修正公式
    } else if (cmax == gn) {
        hn = 60 * (((bn - rn) / delta) + 2); // 修正公式
    } else { // cmax == bn
        hn = 60 * (((rn - gn) / delta) + 4); // 修正公式
    }

    /* calculate s value */
    sn = (cmax < FLT_EPSILON) ? 0 : (delta / cmax);
```

```

/* calculate v value */
vn = cmax;

color _clr;
hsv(_clr, (uint8_t)(hn / 2), // 这里除以2是为了映射到0-180的范围
     (uint8_t)(sn * 255),
     (uint8_t)(vn * 255));
// 标准HSV: H范围0-360, S范围0-1, V范围0-1
return _clr;
}

```

在 OpenCV 中，`cv::cvtColor` 函数中的 `cv::COLOR_BGR2HSV` 和 `cv::COLOR_RGB2HSV` 转换模式分别对应 BGR 和 RGB 格式的图像转换为 HSV 格式的图像。

§3.2.3. HSV 与 RGB

HSV 到 RGB 的转换公式和 RGB 到 HSV 的转换公式是互逆的。我们可以通过以下公式将 HSV 转换为 RGB：

$$\begin{aligned}
 h_i &= \lfloor \frac{H}{60} \rfloor \\
 f &= \frac{H}{60} - h_i \\
 p &= V \times (1 - S) \\
 q &= V \times (1 - f \times S) \\
 t &= V \times (1 - (1 - f) \times S)
 \end{aligned}$$

则相应的 RGB 颜色向量 (r, g, b) 为

$$(r, g, b) := \begin{cases} (v, t, p) & \text{if } h_i = 0 \\ (q, v, p) & \text{if } h_i = 1 \\ (p, v, t) & \text{if } h_i = 2 \\ (p, q, v) & \text{if } h_i = 3 \\ (t, p, v) & \text{if } h_i = 4 \\ (v, p, q) & \text{if } h_i = 5 \end{cases}$$

用 C 语言实现如下：

```

color cvtclr_hsv2rgb(color clr)
{
    double hn = clr.hsv[2] * 2.;
    double sn = (double)clr.hsv[1] / 0xFF;
    double vn = (double)clr.hsv[0] / 0xFF;

    int hi = (int)(hn/60);
    double f = hn/60-hi;
    double p = vn * (1-sn);
    double q = vn * (1-f*sn);
    double t = vn * (1-(1-f)*sn);

    color _clr;
    switch (hi) {
        case 0:
            rgb(_clr, (uint8_t)(vn * 255),
                 (uint8_t)(t * 255),

```

```

        (uint8_t)(p * 255));
    break;
case 1:
    rgb(_clr, (uint8_t)(q * 255),
        (uint8_t)(vn * 255),
        (uint8_t)(p * 255));
    break;
case 2:
    rgb(_clr, (uint8_t)(p * 255),
        (uint8_t)(vn * 255),
        (uint8_t)(t * 255));
    break;
case 3:
    rgb(_clr, (uint8_t)(p * 255),
        (uint8_t)(q * 255),
        (uint8_t)(vn * 255));
    break;
case 4:
    rgb(_clr, (uint8_t)(t * 255),
        (uint8_t)(p * 255),
        (uint8_t)(vn * 255));
    break;
default:
    rgb(_clr, (uint8_t)(vn * 255),
        (uint8_t)(p * 255),
        (uint8_t)(q * 255));
    break;
}
return _clr;
}

```

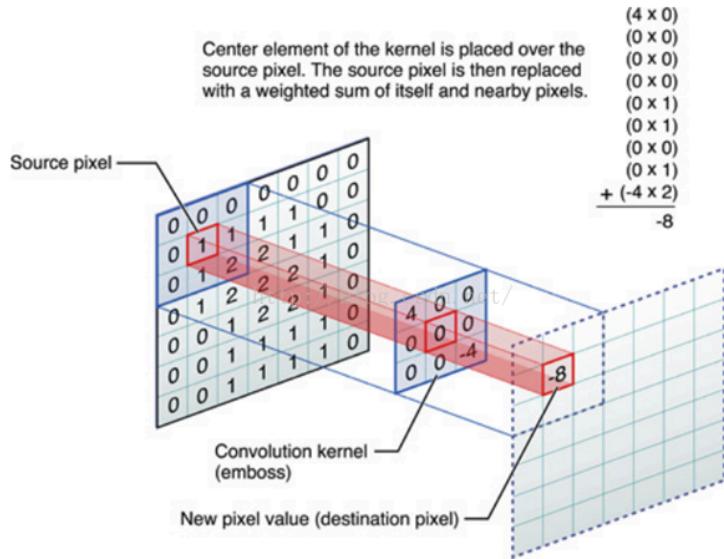
在 OpenCV 中，`cv::cvtColor` 函数中的 `cv::COLOR_HSV2BGR` 和 `cv::COLOR_HSV2RGB` 转换模式分别对应 HSV 格式的图像转换为 BGR 和 RGB 格式的图像。

§3.2.4. RGB 与 Gray

这个操作也非常简单，只需要将 RGB 图像中的红、绿、蓝三种颜色的亮度值进行加权平均即可。在前面已经提到过相应的公式，在此不再赘述。

§3.3. 卷积

卷积是图像处理中最常用的操作之一，它可用于图像的平滑、锐化、边缘检测等操作。卷积操作是通过将一个小的滤波器（也称为卷积核）在图像上滑动，并计算滤波器与图像的重叠区域的加权和来实现的。



卷积操作的公式如下：

$$I'(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k I(x+i, y+j) * K(i+k, j+k)$$

常用于图像处理的卷积核有很多种，比如高斯模糊、均值模糊、Sobel 算子、Laplace 算子等。高斯模糊针对高频噪声的处理效果比较显著，Sobel 算子和 Laplace 算子则是用于提取图像的边缘信息。不同的卷积核大小以及权重分配也会影响卷积的效果。

在这个项目中，我们可以使用 conv 命令对图像进行卷积操作。通过上面的介绍，我们会发现得到的卷积结果实际上比原图像小一圈，这时候我们需要提前对图像进行填充操作。一般来说，我们会在图像的四周进行零填充，这样卷积操作后得到的图像大小和原图像一样。

卷积在实际图像处理有很多经典应用，比如利用均值模糊卷积核对图像进行模糊处理可以显著降低图像的椒盐噪声。接下来，我们就用这个程序来完成这个操作，观察一下效果。

```
./cimage noise -i chuliu.bmp -o noise.bmp -p 0.01
./cimage conv noise.bmp x mean9.bmp -o chuliu_recv9.bmp
./cimage conv noise.bmp x mean13.bmp -o chuliu_recv13.bmp
./cimage conv noise.bmp x mean25.bmp -o chuliu_recv25.bmp
```

以上是对图像进行椒盐噪声处理后，分别使用 9, 13, 25 大小的均值模糊卷积核对图像进行模糊处理的命令。我们可以看到，随着卷积核的增大，图像的噪声去除效果越来越明显。

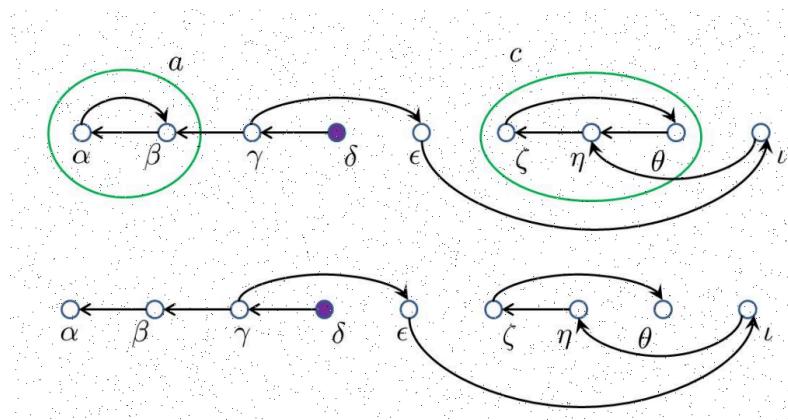
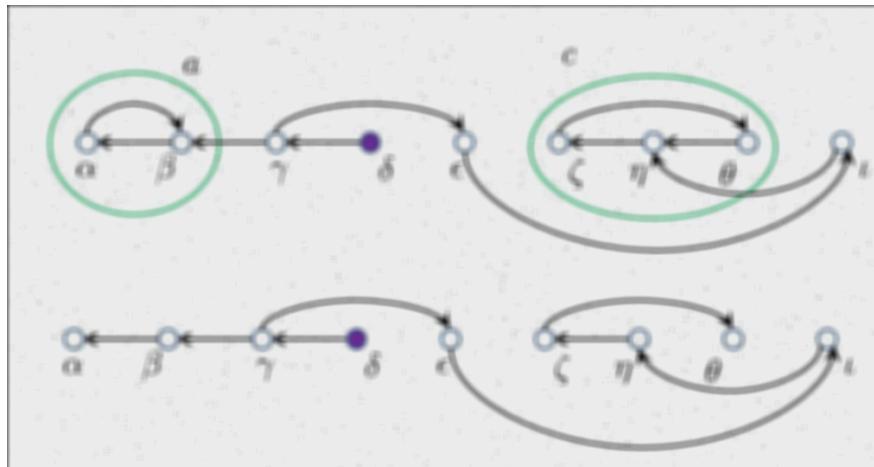
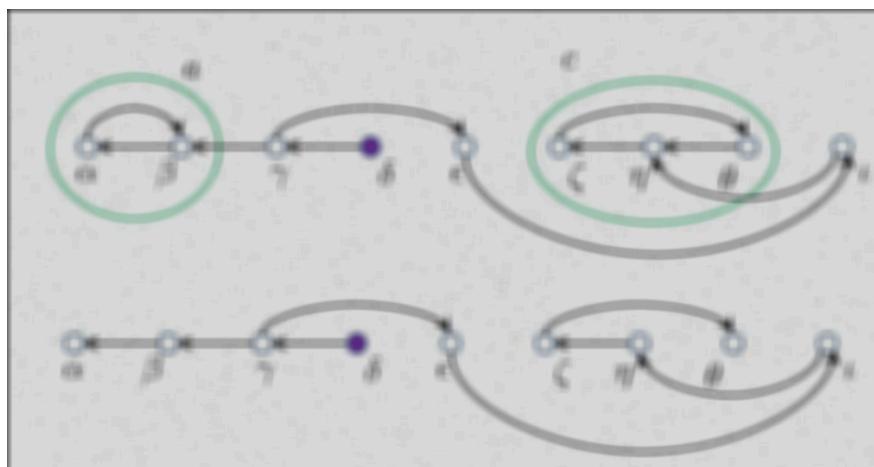


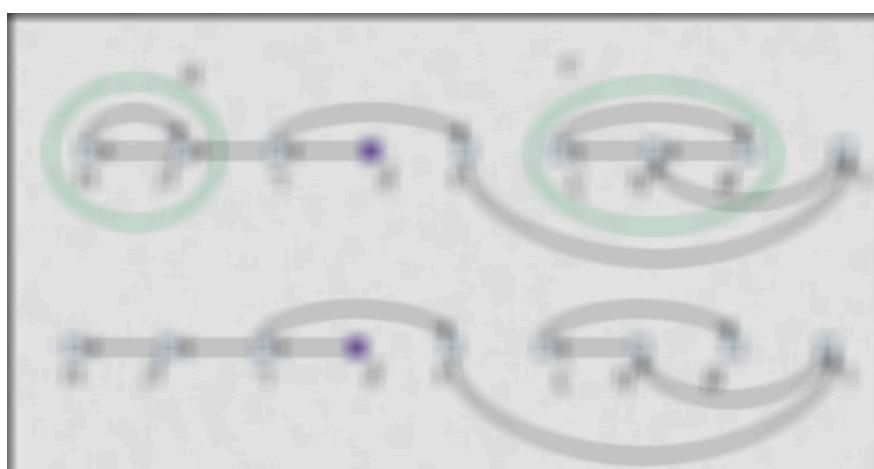
Figure 6: 带有噪点的图像



kernel size = 9



kernel size = 13

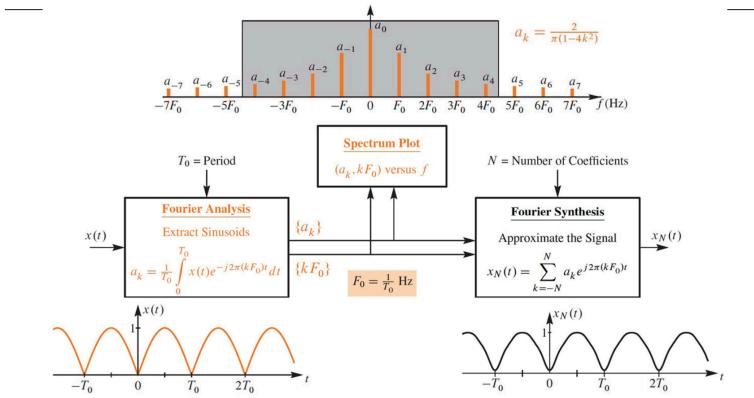


kernel size = 25

§3.4. 傅里叶变换

傅里叶变换实际上是来自于信号处理领域的一个重要工具，它可以将信号从时域转换到频域。在此前的 Project 1 中，我们借助快速傅里叶变换 FFT 将大数乘法的复杂度降低到了 $O(n \log n)$ 。除此之外，傅里叶变换还有其他有意思的应用，比如如果将图像视作沿空间维度变化的信号，那么傅里叶变换就可以将图像从空间域转换到频率域。

Fourier Series Synthesis



Aug 2016

© 2003-2016, JH McClellan & RW Schafer

14

推导二维傅里叶过程的思路如下^[1]: 假设一个大小为 $M \times N$ 的矩阵上进行变换，那么索引为 (u, v) 的频率分量的傅里叶变换公式为：

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \exp \left[-2\pi i \left(\frac{xu}{M} + \frac{yu}{N} \right) \right]$$

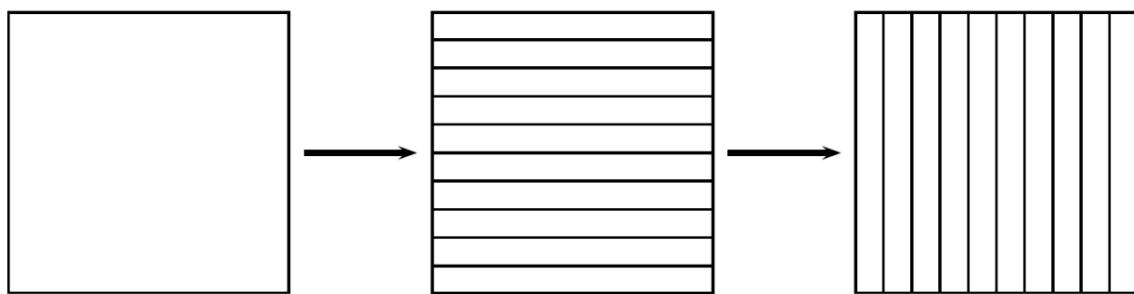
注意到指数项可以分离为两个部分，分别对应于 x 和 y 方向的傅里叶变换：

$$\exp \left[-2\pi i \left(\frac{xu}{M} + \frac{yu}{N} \right) \right] = \exp \left[-2\pi i \frac{xu}{M} \right] \exp \left[-2\pi i \frac{yu}{N} \right]$$

因此，2D 的傅里叶变换就转化为分别在 x 和 y 方向上进行 1D 的傅里叶变换，即

$$\mathbb{F}_{M \times N} = F_M \cdot X_{M \times N} \cdot F_N^T$$

下图形象化解释了 2D 傅里叶变换的流程



(a) Original image

(b) DFT of each row of (a)

(c) DFT of each column of (b)

这样得到的频域图像是复数形式的，为了显示每个像素的幅度和相位，我们需要将其转换为实数形式。我们可以使用如下公式来计算幅度^[2]：

$$A(u, v) = \sqrt{\text{Re}^2 + \text{Im}^2}$$

$$A(u, v) = \log(1 + A(u, v))$$

最后结果需要取对数阶来将幅度值差异缩小到可视化范围内。

在实现上，1D 的傅里叶变换基本思路与 Project 1 中的 FFT 思路相同，不过为了提高图像处理的效率，我将位置换函数使用位操作优化并且采用蝶形变换优化递归操作。

```
void fft_change(double complex *a, uint32_t n, int k) {
    for (uint32_t i = 0; i < n; ++i) {
        // 使用Stanford Bit Twiddling Hacks中的位反转技巧
        uint32_t t = i;
        t = ((t >> 1) & 0x55555555) | ((t & 0x55555555) << 1);
        t = ((t >> 2) & 0x33333333) | ((t & 0x33333333) << 2);
        t = ((t >> 4) & 0x0F0F0F0F) | ((t & 0x0F0F0F0F) << 4);
        t = ((t >> 8) & 0x00FF00FF) | ((t & 0x00FF00FF) << 8);
        t = (t >> 16) | (t << 16);
        // 调整位宽，只反转k位
        t >>= (32 - k);

        // 只在需要时交换，避免重复交换
        if (i < t) {
            double complex temp = a[i];
            a[i] = a[t];
            a[t] = temp;
        }
    }
}

void fft1d(double complex *data, size_t n, bool inverse) {
    // 计算k = log2(n)
    int k = log2_uint32(n);

    // 位反转排序
    fft_change(data, n, k);

    // 蝴蝶运算
    for (size_t len = 2; len <= n; len <= 1) {
        double angle = 2 * M_PI / len * (inverse ? -1 : 1);
        complex double wn = cos(angle) + I * sin(angle);

        for (size_t i = 0; i < n; i += len) {
            complex double w = 1.0 + 0.0 * I;
            for (size_t j = 0; j < len / 2; j++) {
                complex double u = data[i + j];
                complex double t = w * data[i + j + len / 2];

                data[i + j] = u + t;
                data[i + j + len / 2] = u - t;

                w *= wn;
            }
        }
    }
}
```

```

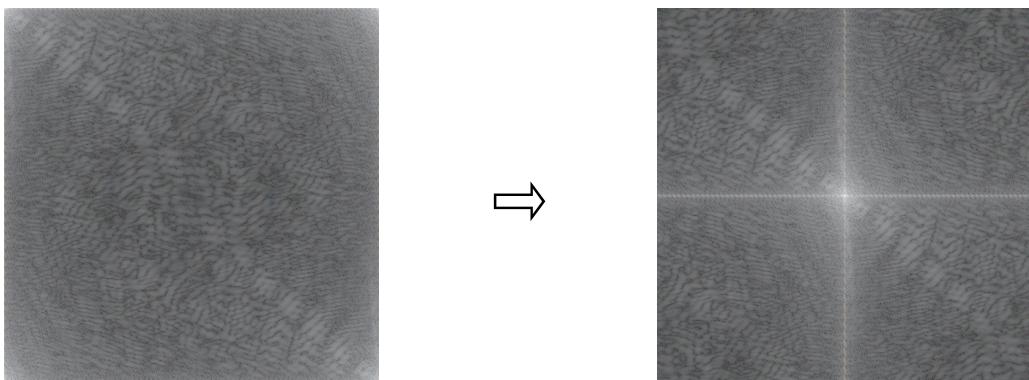
    }

    // 如果是逆变换，需要除以n
    if (inverse) {
        for (size_t i = 0; i < n; i++) {
            data[i] /= n;
        }
    }
}

```

优化后的位置换操作时间复杂度由 $O(n \log n)$ 降低到 $O(n)$ ，而蝶形变换的时间复杂度仍然是 $O(n \log n)$ 。

这样处理得到的频域图像，特征是频域图像的中心部分对应于原图像的低频信号，边缘部分对应于高频信号。为了便于观察，我们可以将频域图像进行中心化处理，将频域图像的中心部分移动到左上角，边缘部分移动到右下角，也就是将图像的四个象限进行交换。



关于图像处理中的傅里叶变换实际上有不少应用，基于傅里叶变换的低通滤波器和高通滤波器就是其中之一。对一个图像来说，低频信号对应画面里的平坦区域和缓慢变化的部分，而高频信号对应画面里的细节和噪声。而经过傅里叶变换后，图像的低频信号和高频信号分别对应于频域图像的中心和边缘部分。结合之前的卷积操作，如果我们对频域图像中心部分加入白色遮罩，边缘部分加入黑色遮罩，那么就可以通过逆傅里叶变换来提取出图像的低频信号。这就是理想低通滤波器的原理。

$$m(x, y) := \begin{cases} 1 & \text{if } \sqrt{x^2 + y^2} < d_0 \\ 0 & \text{otherwise} \end{cases}$$

其中， $m(x, y)$ 是频域图像的掩膜， d_0 是截止频率。截止频率是决定滤波效果的一个重要参数，截止频率越大，滤波器的通带越宽，低通滤波后的图像越模糊；截止频率越小，滤波器的通带越窄，低通滤波后的图像越清晰。下面这张图展示了不同截止频率下的低通滤波器的效果：

Ideal Low Pass Filtering

Applying
low pass filter
to DFT
Cutoff D = 15

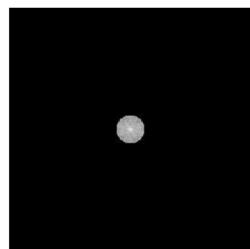
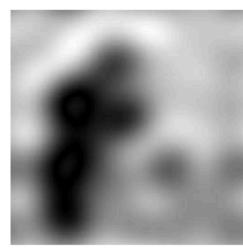


Image after
inversion



low pass filter
Cutoff D = 5



low pass filter
Cutoff D = 30

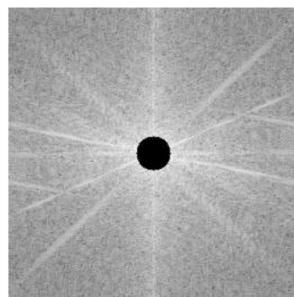
Note: Sharp filter
Cutoff causes
ringing



而高通滤波器的原理和低通滤波器是相反的，我们可以通过对频域图像的边缘部分加入白色遮罩，中心部分加入黑色遮罩来提取出图像的高频信号。

Ideal High Pass Filtering

- Opposite of low pass filtering: eliminate center (low frequency values), keeping others
- High pass filtering causes image **sharpening**
- If we use circle as cutoff again, size affects results
 - Large cutoff = More information removed



DFT of Image after
high pass Filtering



Resulting image
after inverse DFT

经过高通滤波器处理后的图像在形状边缘等信号分立的地方会有明显的增强，而在平坦区域和缓慢变化的部分则会被削弱。因此可以观察到比较明显的边缘效果。

§3.5. 仿射变换

如果想要一个个实现图像的旋转、缩放、平移等操作，那未免有些麻烦且难以进行扩展。但如果熟悉线性代数的相关知识，这些基础的图像操作实际上可以视作二维空间上的线性变换。在具体推导这些线性变换之前，我们需要知道图形学中齐次坐标系的概念。

§3.5.1. 齐次坐标系

齐次坐标系是为了解决平移操作无法在经典坐标系下以线性变换的形式表达而引入的。具体来说，在二维空间中，对于平移变换，我们维持上面对于原图形和变换后图形坐标的表示不变，将对应 X 和 Y 方向上的平移分量分别记作 T_X 和 T_Y ，那么我们可以得到对应点坐标的等式关系为

$$X' = X + T_X$$

$$Y' = Y + T_Y$$

尝试规约到线性变换的形式

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} + \begin{bmatrix} T_X \\ T_Y \end{bmatrix}$$

会发现无论如何变形，在二维坐标下都无法避免这个非线性项 $\begin{bmatrix} T_X \\ T_Y \end{bmatrix}$ 的影响。但是如果我们将二维向量增广一个维度，引入新参数 W ，就可以解决这个问题。

$$\begin{bmatrix} X \\ Y \end{bmatrix} \rightarrow \begin{bmatrix} X \\ Y \\ W \end{bmatrix}$$

规定二维空间中的向量提升到三维齐次坐标系时， $W = 1$ ；将三维齐次坐标系下的向量表示转化到二维空间，则需要对所有维度都除以 W 。

Homogeneous coordinates

Conversion

Converting to homogeneous coordinates

$$(x, y) \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

homogeneous image
coordinates

$$(x, y, z) \Rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

homogeneous scene
coordinates

Converting from homogeneous coordinates

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} \Rightarrow (x/w, y/w)$$

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \Rightarrow (x/w, y/w, z/w)$$

知乎 @阿轩00

利用齐次坐标就可以顺利得到平移操作的线性变换为

$$\begin{bmatrix} X' \\ Y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_X \\ 0 & 1 & T_Y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

§3.5.2. 仿射变换

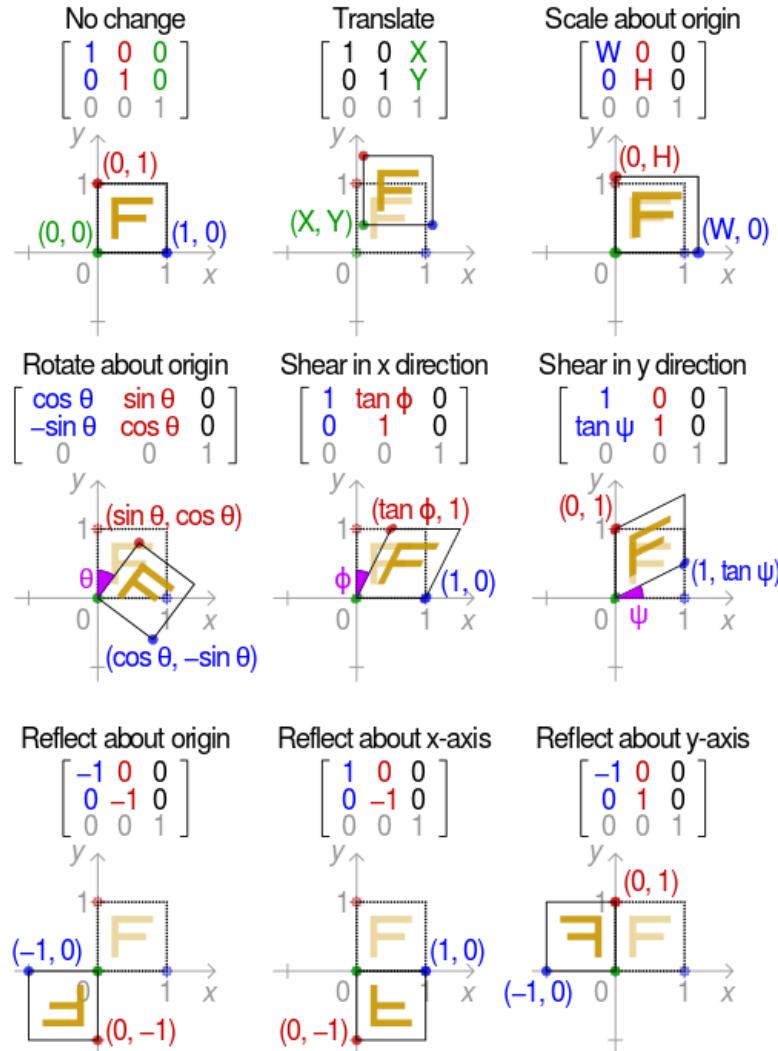
类似地, 利用几何线性代数的知识, 我们分别可以得到其他变换在齐次坐标系下对应的线性变换算子:

1. Scaling: $\begin{bmatrix} S_X & 0 & 0 \\ 0 & S_Y & 0 \\ 0 & 0 & 1 \end{bmatrix}$

2. Rotation: $\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

3. Translation $\begin{bmatrix} 1 & 0 & T_X \\ 0 & 1 & T_Y \\ 0 & 0 & 1 \end{bmatrix}$

这张图可以解释各个变换的来源



由两个线性变换的复合变换也是线性的这一性质可以得出, 如果我们同时复合这三个线性变换, 我们将得到一个能够表示以上所有操作的基变换 $a \equiv s \circ r \circ t$, 这一变换即称为仿射变换。用矩阵表示来讲, 就是把这三个矩阵表示相乘, 可以得到仿射变换的矩阵表示

$$f(S_X, S_Y, \theta, T_X, T_Y) = \begin{bmatrix} S_X & 0 & 0 \\ 0 & S_Y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & T_X \\ 0 & 1 & T_Y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_X \cos \theta & -S_X \sin \theta & T_X \\ S_Y \sin \theta & S_X \cos \theta & T_Y \\ 0 & 0 & 1 \end{bmatrix}$$

由于第三行的0和1不参与变换，因此我们可以将其省略掉，得到仿射变换的矩阵表示为

$$\mathbb{M}(S_X, S_Y, \theta, T_X, T_Y) = \begin{bmatrix} S_X \cos \theta & -S_X \sin \theta & T_X \\ S_Y \sin \theta & S_X \cos \theta & T_Y \\ 0 & 0 & 1 \end{bmatrix}$$

因此对于一个点 (x, y) ，我们可以通过矩阵乘法来得到变换后的点 (x', y') :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbb{M} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

我们定义如下结构体来表示仿射变换的矩阵:

```
typedef struct Matf64_2x3_ {
    double a, b, c; // x' = ax + by + c
    double d, e, f; // y' = dx + ey + f
} Matf64_2x3, AffineMat;
```

在实际处理图像时，我们已知变换后的坐标 (x', y') ，需要求解变换前的坐标 (x, y) 及其对应的像素值，我们可以通过仿射矩阵的逆来得到变换前的坐标 (x, y) :

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbb{M}^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

计算矩阵的逆矩阵可以使用伴随矩阵法，具体的实现如下:

```
AffineMat AffineMat_inv(const AffineMat *m) {
    AffineMat inv;

    double det = m->a * m->e - m->b * m->d;
    if (fabs(det) < FLT_EPSILON) {
        WARNING("Affine matrix is singular, cannot compute inverse.");
        return *m; // 返回原矩阵，表示错误
    }

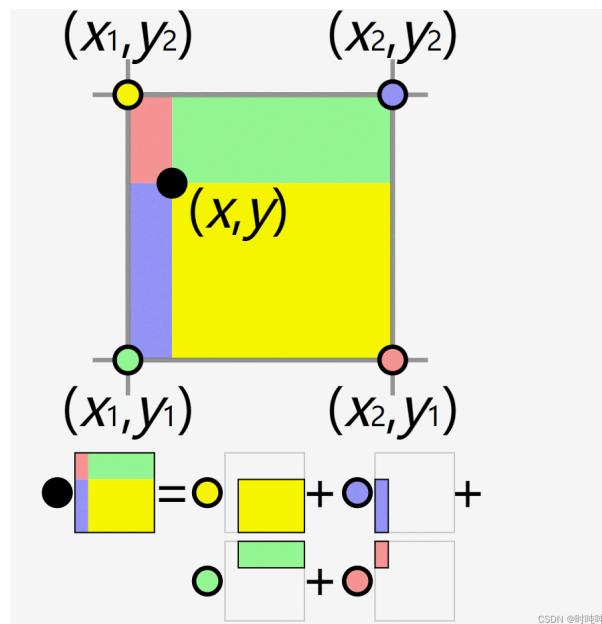
    inv.a = m->e / det;
    inv.b = -m->b / det;
    inv.c = (m->b * m->f - m->e * m->c) / det;

    inv.d = -m->d / det;
    inv.e = m->a / det;
    inv.f = (m->d * m->c - m->a * m->f) / det;

    return inv;
}
```

有了这些，我们基本上解决了基本图像变换问题的一大部分。然而在涉及到图像大小缩放时，会出现一个问题：我们在进行仿射变换时，可能会出现变换后的多个分数坐标 (x', y') 经过取整后，对应到原图像同一个像素点 (x, y) 的情况。说人话就是放大后的目标图像可能会出现锯齿状的边缘而显得不平滑。为了增加采样的平滑度，插值的方法经常被用来解决这个问题。其中最常用

的插值方法是双线性插值法，它的基本思路是通过对四个相邻像素进行加权平均来计算目标像素的值。



具体的计算方法可见 Wojik-Bilinear interpolation 这篇博客，应用了双线性插值法的仿射变换函数如下：

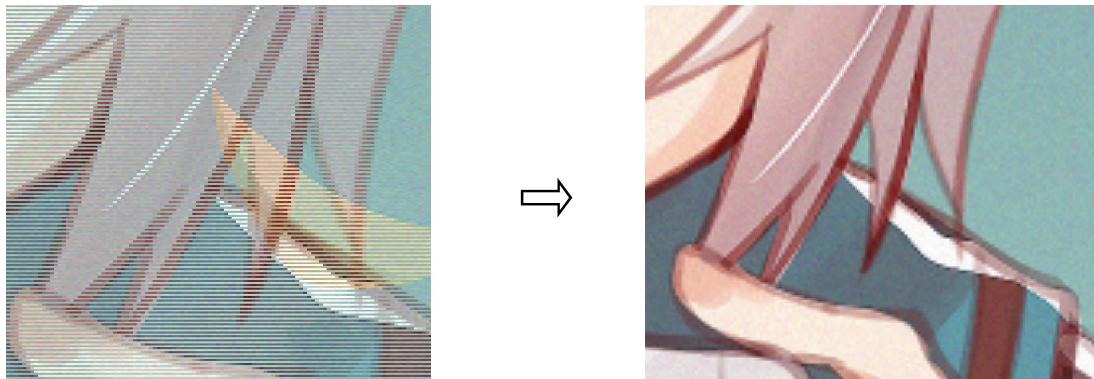
```
double bilinearInterpolation(const Mat *channel, double x, double y) {
    if (!channel || !channel->bytes) {
        WARNING("Channel can't be null!");
        return 0;
    }
    int x1 = (int)x;
    int y1 = (int)y;
    int x2 = x1 + 1;
    int y2 = y1 + 1;

    // 确保坐标在图像范围内
    if (x1 < 0 || y1 < 0 || x2 >= channel->cols || y2 >= channel->rows) {
        return 0; // 超出边界返回0值
    }

    double dx = x - x1;
    double dy = y - y1;

    // 进行双线性插值
    double value = (1 - dx) * (1 - dy) * channel->bytes[y1 * channel->cols + x1] +
                  dx * (1 - dy) * channel->bytes[y1 * channel->cols + x2] +
                  (1 - dx) * dy * channel->bytes[y2 * channel->cols + x1] +
                  dx * dy * channel->bytes[y2 * channel->cols + x2];
    return value;
}
```

对比未使用双线性插值法的仿射变换函数，我们可以看到，使用双线性插值法后，图像的边缘会更加平滑，锯齿状的边缘会被平滑掉。



§3.6. 形状绘制

在图像处理中，形状绘制是一个非常常见的操作。我们可以通过一些简单的函数来绘制基本的几何形状，比如直线、矩形、圆形等。我们在纸上画直线时，只需要定一个起点和终点，然后把两点连接起来就是一条直线，你将会得到一条笔直的直线。

但是，这个简单的过程，在计算机上却不容易。首先计算机的屏幕是一个一个的离散的 led。每一个白块都是一个灯管，当我们绘制一条直线时，就需要判断在这条直线上，需要点亮哪些灯管来模拟这条直线。

一种经典的、广泛应用的绘制算法是 Bresenham 算法^[3]，它是基于误差修正的方法来绘制直线的。Bresenham 算法的基本思路是通过计算直线在每个像素上的误差来决定下一个像素的位置，从而实现高效的直线绘制。

§3.6.1. 直线绘制

一个从点 $P(x_0, y_0)$ 到点 $P(x_1, y_1)$ 的直线可以用以下公式表示：

$$(x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0) = 0$$

处在直线上的点 (x, y) 都满足上面的公式。而如果等号左边结果不为 0，则说明点 (x, y) 离直线有一定的偏差，这就是 Bresenham 算法的中定义的误差项 err。误差 err 计算如下：

$$\text{err} = (x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0) = (y - y_0)\text{dx} - (x - x_0)\text{dy}$$

而我们在像素矩阵上遍历时，每一步最大不超过 1 个像素，因此我们定义对角方向的误差 $e_{xy} = (y + 1 - y_0) \text{dx} - (x + 1 - x_0) \text{dy} = e + \text{dx} - \text{dy}$ ；x 方向的误差 $e_x = (y + 1 - y_0) \text{dx} - (x - x_0) \text{dy} = e_{xy} + \text{dy}$ ；y 方向的误差 $e_y = (y - y_0) \text{dx} - (x + 1 - x_0) \text{dy} = e_{xy} - \text{dx}$ 。

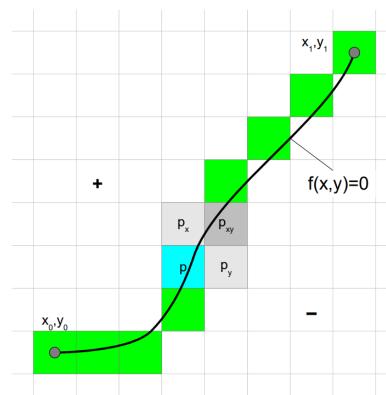


Figure 2: Pixel grid of curve $f(x,y)=0$

结合上面这张图说明一下，假设我们现在已经步入了点 p ，那么按线条斜率方向，我们可以选择 p_x, p_y, p_{xy} 作为下一个点的坐标。为了让下一个点更接近直线，我们需要比较这三个点的误差值，

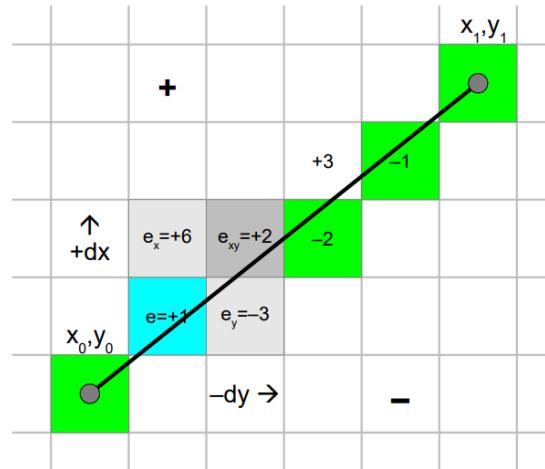


Figure 3: line with error values ($dx=5, dy=4$)

Bresenham 的基本思想是，如果当前点对应的 $|e_{xy}|$ 比 $|e_x|$ 更小，那么 x 方向的行进值就增加 1；如果当前点对应的 $|e_{xy}|$ 比 $|e_y|$ 更小，那么 y 方向的行进值就增加 1。以上图为例， p 点对应的 $|e_{xy}|$ 比 $|e_x|, |e_y|$ 都小，那么 x 方向和 y 方向的行进值都增加 1，也就是选择 p_{xy} 作为下一个点的坐标。据此，我们可以得到 Bresenham 算法的伪代码如下：

```

1: function BRESEHAM-DRAW-LINE( $P, x_0, y_0, x_1, x_2$ )
2:   Set up error variable  $e_{xy}$  for  $P(x_0 + 1, y_0 + 1)$ 
3:   while  $(x, y)$  not equals  $(x_1, y_1)$  do
4:     Set pixel at  $(x, y)$ 
5:     if  $e_x + e_{xy} > 0$  then
6:       Increment  $x$  by 1
7:       Update  $e_x$ 
8:     if  $e_y + e_{xy} < 0$  then
9:       Increment  $y$  by 1
10:      Update  $e_y$ 
```

§3.6.2. 圆形绘制

圆形绘制也是借助误差项计算过渡像素的明度值。一个用于绘制反锯齿填充圆形的算法的示例代码如下：

```

void plotFilledCircleAA(int xm, int ym, int r)
{ /* draw a filled anti-aliased circle */
  int x = r, y = 0, err = 0, i;

  for (r = 2*r+1; x > 0; err += ++y*2-1) { /* y step */
    if (err+2*x+1 < r) err += ++x*2-1; /* one anti-aliased step back */
    for ( ; err > 0; err -= --x*2+1) { /* x step */
      i = 255*err/r; /* set anti-aliased values of pixel */
      setPixelAA(xm+x, ym+y, i); setPixelAA(xm-y, ym+x, i);
      setPixelAA(xm-x, ym-y, i); setPixelAA(xm+y, ym-x, i);
    }
    for (i = x; i > 0; i--) { /* fill pixel inside circle */
      setPixel(xm+i, ym+y); setPixel(xm-y, ym+i);
      setPixel(xm-i, ym-y); setPixel(xm+y, ym-i);
    }
  }
}
```

```

    }
}

setPixel(xm, ym); /* set center pixel */
}

```

最后，这个绘制算法得到的效果如下，可以观察到圆形的边缘是平滑的，色彩过渡也很自然。

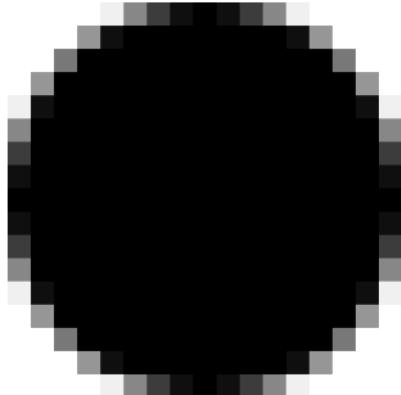


Figure 36: Filled circle with anti-aliasing

§3.7. ASCII 字符画

ASCII 字符画是一种主要依靠电脑 ASCII 字符来表达图像的艺术形式，最早于 1982 年美国卡内基梅隆大学出现，它早期在前端技术不太发达的年代被广泛使用在 BBS 上绘制网站头图等。ASCII 图案一般使用定宽字体来绘制，如 Courier New、Consolas 等。

```

team@itsfoss:~/Downloads$ jp2a --output=tux.txt --colors Tux.png
team@itsfoss:~/Downloads$ cat tux.txt
:
:
;
;kkk ddkK
k .d: K; .ox
x:x000K0ccK
dx000KKK0xx:
loxxxkkkxdxx: ;
@koloodxOKNN;
@WX0000XNWMMMW:
NMMMWWMWMMMWMMMW
XMMMWWMWMMMWNNNO
NWMMMWWMWMMMWNNXXO
oMMMMMMMWMMMWMMMWNN
oMMMMMMMWMMMWMMMWMMX
MMMMMMMWMMMWMMMWMMMWMM
@MMMMMMMWMMMWMMMWMMMWMM
MMMMMMMWMMMWMMMWMMMWMM
cc, @MMMMMMMWMMMWMMMWMMMWMMNW
;x000d dWMWWMMMWMMMWMMMWMMMWMMWx0:
cxxxxkkk00000k, :@MMMMMMMWMMMWMMMWx0d . ,d00:
o00000000000001 XMMMMMMMWMMMWMMNKxx0xxxxk0000c
ck0000000000000d KMMMMMMMWMMMWMM@ ,x0000000000000o'
ok0000000000000kkWMWWMMMWMMMNx :x0000000000000Kxd
:dk0000000000000d: 'oxkkxo: cx000000kxd
ldxxxkk00000kd: .:oxkxxxol
.oddol; ,:cc:'

```

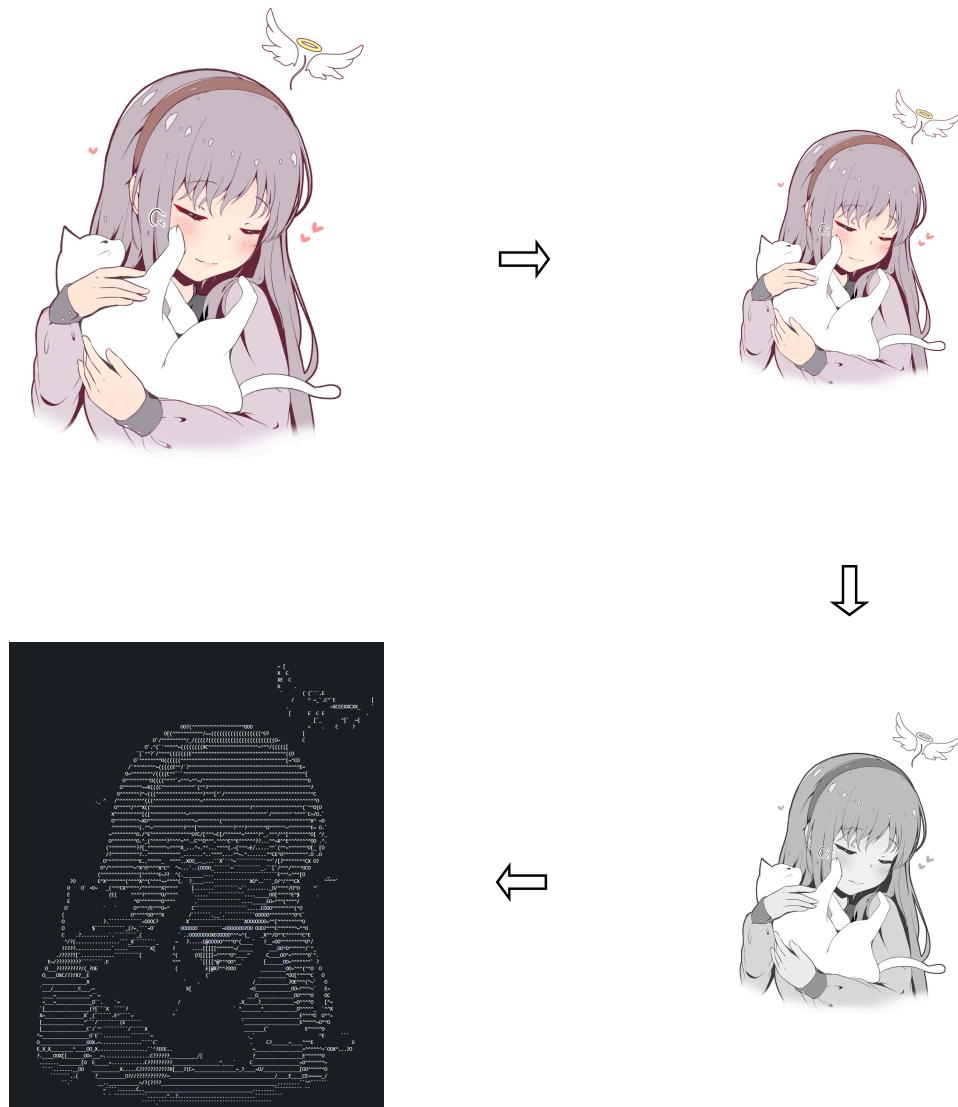
为了绘制 ASCII 字符画，我们需要将图像转换为灰度图像，然后根据灰度值划分到相应的区间，替换为相应的 ASCII 字符来展示灰度。

在实现上，一个合适的 ASCII 字符集表示不同的灰度值是必须的，比如使用(space)表示最亮的部分，使用@表示最暗的部分。这里我使用的 ASCII 字符集如下，灰度值从低到高依次为：

```
@W#$0EXC[( /?=^~_.` (space)
```

为了保证 ASCII 字符画最后打印到终端时不会变形，我们需要将字符画的宽高比进行调整。在终端中，字符的宽高比通常是 2:1，因此我们需要用已经实现的缩放操作来调整字符画的宽高比。

一个 ASCII 字符画的转换过程展示如下：



而在终端中调色可以接着用 Project 1 中提及的 ANSI 转义序列来实现，此处不再赘述。

§4. 总结

这个小项目针对 24-bit BMP 图像，实现了比较丰富的图像处理功能。在实现过程中，我主要以 OpenCV 的各种模块作为参考，不断地查阅相关文档，在实践中加强了计算机视觉中学习的各种图像处理知识。在这个项目的开发里，我深切感受到从零开始造各种轮子的痛点和乐趣，毫无疑问 C 语言的语法不够灵活给很多实现带来了切实的麻烦。同时，作为语言特色的指针和内存管理也让代码编写在后期愈发复杂。不过，相比于在 Python 脚本里傻瓜式地敲下已经封装好的顶层函数，用 C 语言用最基础的语句和容器来实现这些功能，确实让我对图像处理的核心原理和巧妙之处有了更深刻的理解。尤其是将线性代数与图像的几何变换结合起来的仿射变换，把缩放、旋转、平移等操作都统一到一个线性变换的框架下，体现了逐层抽象的编程思想。最后将这些积木块拼接起来，组成一个完整度较高的 ImgProc 项目，并且用这个工具进行一些具体的图像处理实验，确实是一个非常 awesome 的过程。



对比目前功能最为齐全、最为流行的开源图像处理库 OpenCV，这个项目显然只是个小玩具。翻阅 OpenCV 的源代码，只是一个基础的 BMP 图像解编码模块就有上千行的代码，其中也体现了很多优秀的设计模式和编程技巧。不仅如此，OpenCV 对底层计算的优化也做到了极致。比如说，复杂度高达 $O(n^2r^2)$ 的卷积操作，现在主流的 method 是使用分离卷积的方式将其复杂度降低到 $O(n^2 + r^2)$ ，而且还针对 OpenCL/CUDA 等不同的硬件平台进行了分别优化。最后才能达到大多数卷积操作只需要几百毫秒就能完成的效果。这个项目里由于时间原因，我并没有进行比较有效的优化，在处理较大卷积核时速度就相形见绌了。

正如高德纳所言：“计算机科学中没有银弹”，每一行代码都是对性能与优雅的权衡。当我用 C 语言尝试构建这座微型 OpenCV 时，我不仅是在编写程序，更是在与像素对话，在矩阵中舞蹈。虽然与专业库相比尚有差距，但正如艾伦·图灵提出：“我们只能看到短距离的前方，但我们能看到那里有很多事情需要完成。”在未来的优化与拓展中，我将继续怀揣“算法之道，存乎一心”的信念，在计算与艺术的交界处，寻找二进制世界的更多奥秘。

Bibliography

- [1] Emmanuel Agu, “Digital Image Processing (CS/ECE 545) Lecture 10 Discrete Fourier Transform(DFT).” [Online]. Available: <https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture10.pdf>
- [2] OpenCV, “Discrete Fourier Transform-OpenCV.” [Online]. Available: https://docs.opencv.org/3.4/d8/d01/tutorial_discrete_fourier_transform.html
- [3] Alois Zingl", “A Rasterizing Algorithm for Drawing Curves.”