VCI

sand

From September 2025 to December 2025

目录

1 Introduction

课程名为《可视计算与交互》(Visual Computing & Interaction)，使人听之即可明白它总览性的特征。本课程以计算机图形学为主线，安排了下面几个板块：

- 二维图形、图像的相关知识
- 几何建模，主要指三维
- 渲染（from 三维 to 电脑图像）
- 动态生成
- 可视化与交互

具体应用广泛，相关分支有计算机图形、计算机视觉、可视化、虚拟/增强现实、人机交互，等等等等。

1. 虚拟现实：沉浸感

2. 增强现实：（例子“Pokerman Go”游戏）

3. 计算机图形与计算机视觉，前者强调让计算机“画画”，后者强调从图片转换成计算机内部表达/数字化。

4. 形体 + 物理 + 控制 = 数字世界

1.1 智能各领域顶会

1. 计算机视觉（Computer Vision）

代表会议：CVPR（IEEE Conference on Computer Vision and Pattern Recognition）

2. 计算机图形学（Computer Graphics）

代表会议：SIGGRAPH（ACM Special Interest Group on Computer Graphics and Interactive Techniques）

3. 自然语言处理（Natural Language Processing）

代表会议：ACL（Annual Meeting of the Association for Computational Linguistics）

4. 机器学习与人工智能基础（Machine Learning / AI Theory）

代表会议：NeurIPS（Conference on Neural Information Processing Systems）

5. 强化学习与规划（Reinforcement Learning / Planning）

代表会议：AAAI（Association for the Advancement of Artificial Intelligence）

6. 数据挖掘与知识发现（Data Mining & Knowledge Discovery）

代表会议：KDD（ACM SIGKDD Conference on Knowledge Discovery and Data Mining）

7. 信息检索与推荐系统（Information Retrieval & Recommendation Systems）

代表会议：SIGIR（ACM Special Interest Group on Information Retrieval）

8. 机器人学（Robotics）

代表会议：ICRA（IEEE International Conference on Robotics and Automation）

9. 人机交互与可视计算（Human–Computer Interaction / Visual Computing）

代表会议：CHI（ACM Conference on Human Factors in Computing Systems）

10. 虚拟现实与增强现实（Virtual / Augmented Reality）

代表会议：IEEE VR（IEEE Conference on Virtual Reality and 3D User Interfaces）

11. 医学人工智能与生物信息学（Medical AI / Biomedical Informatics）

代表会议：MICCAI（Medical Image Computing and Computer-Assisted Intervention）

12. 知识图谱与语义网（Knowledge Graphs & Semantic Web）

代表会议：ISWC（International Semantic Web Conference）

13. 多智能体系统与群体智能（Multi-Agent Systems / Swarm Intelligence）

代表会议：AAMAS（International Conference on Autonomous Agents

and Multiagent Systems）

14. 语音识别与信号处理（Speech & Audio Processing）

代表会议：ICASSP（IEEE International Conference on Acoustics, Speech and Signal Processing）

15. 通用人工智能与跨模态 AI（General / Multimodal AI）

代表会议：ICLR（International Conference on Learning Representations）

1.2 传送门

课程网页：http://vcl.pku.edu.cn/course/vci

答疑网页：https://piazza.com/class/mfcn4clnh8z3wv/post/8

教材：https://vcl-pku.github.io/vci-book/

北大教学网：https://course.pku.edu.cn

线下答疑：

- 时间：周五下午 2-4 点
- 地点：理科 2 号楼 2110

2 Color, color perception and visualization

2.1 起源与基础知识

一些研究人员认为，我们的灵长类动物祖先可能为了检测森林中的水果而进化出色觉。

颜色的本质区分来自光的波段大小，因此不妨把这个称为颜色的本质。颜色包含的信息：温度，情感，文化信息。

光的两种结合方式：

- 自身发光：以加法方式结合，RGB 合在一起是白光，也是电脑显示屏的显色原理。
- 反射光成色：减法结合，三色结合近乎黑色。主要体现于日常颜料和印刷。（因此颜料三原色是指青色（Cyan）、品红色（Magenta）和黄色（Yellow），与 RGB 不同）

生理机制：视锥细胞（三种，集中于中央凹，分辨颜色）和视杆细胞（外围多，分辨明暗）。

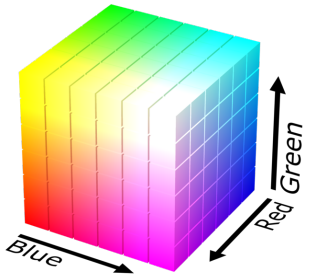


图 1: RGB

其他有趣的色觉相关机制：

- Chromostereopsis （色彩立体视觉）：不同色光折射程度不同。通常黄色聚焦最清晰。
- 色温。
- 颜色恒定性：人脑特殊的处理机制，不会因为光线明暗导致的灰度一致而错误辨别颜色的现象。
- 不同曝光的一组照片信息叠加。
- 色域（不同技术系统所能呈现的颜色范围，如 adobe 等）的互相转换、色诱导、边界效应、色盲.....

2.2 Representational Color Spaces

- RGB 色彩空间：如Figure 1所示。
- HSV 色彩空间：见Figure 2。HSL 对应色相、饱和度、亮度（hue，saturation，lightness）。HSV 对应色相、饱和度、明度（hue，saturation，value）。使用柱坐标最大的好处，在于符合人们对色彩的经验直觉，但重复颜色过多。去除冗余色彩后，我们可以看到的色彩空间如Figure 3所示。HSL 是类似的，这里不多赘述，感兴趣者参见https://en.wikipedia.org/wiki/HSL_and_HSV。

可见光谱的颜色与显示器、颜料所能呈现的颜色范围互有重叠与不重叠之处。

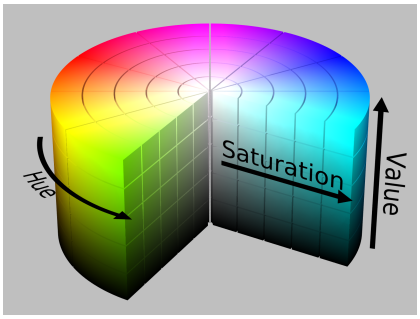


图 2: HSV in cylinder

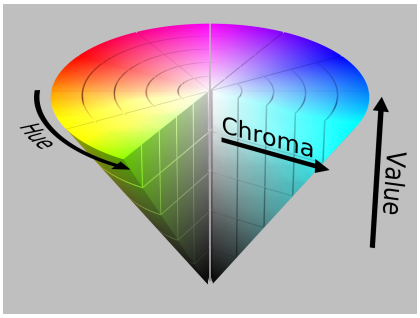


图 3: HSV in Cone

2.3 颜色在数据可视化中的应用

重叠区域的几种手段:

- 颜色混合（RGB 混合法）
- 噪点

3 display

硬件设备如何“显示”数字化的图像?

3.1 稳定显示

首先发明的是矢量显示。早期的主流电子显示器之一是阴极射线管（CRT），通过电子枪发射电子束在磁场中的偏折、打到荧光层的显示屏上显示一个亮点。最初的 CRT 是单色的，以同样的原理，人们后来还开发出了彩色 CRT。

所谓的矢量显示（vector display），更重要的是“线”的表示，技术处理上通过控制“点”按照一定的轨迹快速移动实现。矢量显示的问题在于绘制图形的复杂度十分受限，填充内容物的颜色十分困难，绘制时间长，颜色不准确等等。

新登场的，也是现在占据主流显示的是像素显示，我们通过控制每个像素的开关和颜色，且任意图形复杂度下显示的开销不变（需要控制所有像素）。对于上面提到的 CRT 显示器，我们可以通过逐行扫描的方式，将其

改造为像素显示。像素显示的问题在于“锯齿”现象。
接下来介绍一些常见的显示变体：

- LCD（液晶）：直接发光体（发光二极管等），通过偏振器改变颜色。液晶分子在电压的控制下会形成不同的排列方式，所以可以控制光线的透过，形成不同色光比例的混合和不同明暗程度的画面。有机发光二极管（OLED）屏幕则直接使用红绿蓝三种颜色的有机二极管发出对应颜色的光，并直接控制每种色光的强弱。

- HDR（高动态范围成像）：亮度有个比特数，它支持亮度的更大范围。人眼对暗的更敏感，因此比特分布是非线性的，比如 log（在拍照的时候，往往调低曝光更清晰）。通过拍摄多张不同曝光的图像，在浮点空间中合成一张高动态范围图像，保存更多亮部和暗部信息。由于显示器仍然只能显示有限亮度范围，需要进行色调映射。

- GigaPixel（超高分辨率）：通常为多设备拍摄，经过计算得到更高像素图像。

- 立体（stereo）显示原理：双目视觉；偏振光 3D；裸眼 3D（利用光学结构（如透镜、光栅等）在显示屏上让左右眼分别看到不同的图像，从而在大脑中产生深度感。微柱透镜有个固定的 sweet spot；光屏障式）

- 3D 显示：强调人眼可变，动态跟踪能得到更多新画面，不一定是立体显示。例如，裸眼 3D（显示屏高速刷新，通常牺牲分辨率）；光场显示：通过大量不同聚焦的照片构成一个“光场”，在屏幕上可选聚焦；全息投影（光干涉）；其他（用于使你对数字世界的产物感到真实的方法）：力反馈、触觉、嗅觉、体感互动显示……

3.2 刷新 *（refresh）

一些基础概念：

- 像素（pixel）：每个可由电子束点亮的屏幕点
- 分辨率（resolution）：屏幕能显示的像素个数
- 纵横比（perspective ratio）：像素列数除以行数
- 帧（frame）：屏幕所显示的一个画面
- 帧率（frame rate）：每秒屏幕刷新的帧数

- 隔行扫描：保证屏幕亮度均衡，提高帧率

撕裂（screen tearing）：显示器总是以固定的频率刷新页面，刷新的内容从帧缓存（frame buffer）中读取。帧缓存是一片内存空间，软件通过更新帧缓存的内容更新屏幕显示的内容，但软件更新频率是不一定的，这两种更新频率之间的错位就可能造成画面的撕裂。最常见的解决做法是多重缓冲 + 垂直同步（没更新完不切换）。

图形处理器（俗称显卡）GPU 有成百上千个简单的计算单元，可以在同一时间并行运行大量程序，比如计算每个像素的颜色，而 CPU 一般只有个位数的可并行的计算单元。

3.3 颜色显示的伽马校正 *

其实是一个历史遗留问题：图像上的数值转化为 CRT 显示器的电压信号之后，最后显示屏的亮度 I 与电压 V 之间并不是线性关系：

$$I \propto V^{\gamma} \tag{1}$$

这里的 γ 一般在 2.2 左右。这种幂函数关系会压低暗部加强亮部。因此我们需要先把图片的像素做一次相反的变换：

$$c \to c^{1/\gamma} \tag{2}$$

尽管 CRT 已成为历史，这个步骤仍被沿用了下来。sRGB 颜色空间的在标准中直接规定各种显示器（γ 值可能不同）就需要进行模拟来适配这个标准。因此，现在也常将 sRGB 颜色空间称为伽马颜色空间，与之相对的就是线性颜色空间。

特别地，渲染（模拟现实光照）应该使用线性颜色空间，画面显示出来前则需要再做一次伽马校正。

4 画图

Scan Conversion, a.k.a.（also known as）Rasterization
把连续的图形转化成离散的像素点称为光栅化。

4.1 画直线

由离散的像素点近似而成。

由于像素通常以正方形的形式出现，为了简化问题，我们约定直线的倾斜角在 $[0, \pi/2]$ 之间，则绘图时下一个填充的方格可以视为 (x+1, y) (x+1, y+1) 中的一个，需要决定的是每次选择哪个像素，直观的做法是从直线方程中得到目标点，如：

Listing 1: 最简单的直线光栅化算法

```
def draw_line(x0: int, x1: int, m: float, b: float):
    for x in range(x0, x1 + 1):
        y = m * x + b
        draw_pixel(x, Round(y))
```

如果都填充会发生什么？在这里的讨论中，我们默认线的宽度为一个像素，怎么画更自然的“粗线”？

- 几何扩展。画“一条窄长矩形”
- 多像素覆盖（Anti-Aliasing 抗锯齿）。对于直线穿过的每个像素，计算直线覆盖该像素的面积比例。颜色按比例混合（比如黑到浅灰）。
- 加粗。比如两层 Bresenham，相位对齐。

这种方法需要多次浮点数乘法，简单的优化手段是用累加来替代（DDA 算法）：

Listing 2: 最简单的直线光栅化算法

```
def draw_line(x0: int, x1: int, m: float, b: float):
    y = y0
    m = (y1 - y0) / (x1 - x0)
    for x in range(x0, x1 + 1):
        draw_pixel(x, Round(y))
        y += m
```

整数运算往往比浮点数更快，我们只需要判断 y 坐标是否 +1。Bresenham 选择的判据是（x+1, y+1/2）在直线的哪侧，利用法向量，判断与法

向量的夹角。因此有：

$$
\begin{aligned}
P = (x, y), \quad \Delta &= (1, 1/2) \\
F(P + \Delta) &= N \cdot (P + \Delta - P_0) \\
&= N \cdot (P - P_0) + N \cdot \Delta \\
&= F(P) + N \cdot \Delta \\
F(P_{next}) &= F(P + \Delta) \pm N \cdot (0, 1/2)
\end{aligned}
$$

F 代表直线的隐式方程的值，N 表示直线的 $P_1 - P_0$ 这个矢量顺时针旋转 $90°$ 得到的法向向量，$N = (dy, -dx)$。使用 Δ 作为累加项 $(1, 1/2)$，进而我们发现 $N \cdot \Delta$ 是定值，可以用于 DDA。实际上为了减少 $1/2$ 带来的计算开销，保持整数，会一开始就把法向量 N 乘以 2。

Listing 3: 布雷森汉姆直线算法

```
def draw_line(x0: int, y0: int, x1: int, y1: int):
    y = y0
    dx, dy = 2 * (x1 - x0), 2 * (y1 - y0)
    dydx, F = dy - dx, dy - dx // 2
    for x in range(x0, x1 + 1):
        draw_pixel(x, y)
        if F < 0:
            F += dy
        else:
            y += 1
            F += dydx
```

在我的 lab1 实现中，利用 $(dy, -dx)(sx, sy) -> (|dy|, -|dx|) \cdot s_x \cdot s_y$，又根据几何知识，$s_x \cdot s_y$ 的两种情况 $F >= 0$ 时靠近斜率为一的直线，从而实现统一。

另一个实现是利用误差绝对值，误差 err += F，与前一种方法不同，此方法采取的思想是通过下一个选点“修正”偏移，前一种思想则通过“判断”选点本身的误差大小。我们定义 $err = (x - x_0) \cdot (y_1 - y_0) + (x_1 - x_0) \cdot (y - y_0)$。
根据误差定义，点 (x + 1,y) 的误差值: e - dy
点 (x, y + 1) 的误差值: e + dx
点 (x + 1,y + 1) 的误差值: e + dx - dy
如果 e > dy, 选择点 (x + 1 ,y)(仅移动 x); 如果 e < -dx, 选择点 (x, y+1)(仅移动 y); 否则，选择点 (x + 1, y + 1)(同时移动 x 和 y)。

下为一个不需要分类讨论的 Bresenham 画线算法的伪代码实现:

Listing 4: 不需要分类讨论的 Bresenham 画线算法

```
void plotLine(int x,int y,int x1,int y1)
    int dx = abs(x1-x0);
    int dy = abs(y1-y0);
    int sx = x < x1 ? 1 : -1;
    int sy = y < y1 ? 1 : -1;
    int err = 0:
    for(;;){
        setPixel(x0,y0);
        if(err >= 0){
            if(x0 == x1)break;
            err-= dy; x  += SX;}
        if(err <= 0){
        if(y0 == y1) break;err += dx;y0 += sy;}
    }
```

扩展：圆的 Bresenham 算法。只要画 1/8 圆弧，然后利用对称性复制即可。从圆的最上方 (0,r) 往右下走。每次移动时，判断下一个像素应该是正右边的点还是右下角的点。通过一个“判别式”(decision variable) 来决定。

1. $F(x+1,y) = F + 2x + 1$, $F(x+1,y-1) = F + 2x - 2y + 2$，考虑中点误差

$$f(x+1, y-0.5) = (x+1)^2 + (y-0.5)^2 - r^2$$

决策参数 d 是中点误差的缩放整数版本，初始化为

$$d = 3 - 2r$$

并通过增量更新来模拟误差变化。

2. 选择的原则是考察精确值 y 是靠近 yi 还是靠近 yi-1，计算式为

$$y^2 = r^2 - (x_i + 1)^2$$

$$d_1 = y_i^2 - r^2 + (x_i + 1)^2 = y_i^2 - y^2$$

$$d_2 = r^2 - (x_i + 1)^2 - (y_i - 1)^2 = y^2 - (y_i - 1)^2$$

令 $\Delta p_i = d_1 - d_2$，并代入 d_1 和 d_2，则有:

$$p_i = 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2r^2$$

这里我们称 p_i 为误差

如果 $p_i < 0$ 则 $y_{i+1} = y_i$，否则 $y_{i+1} = y_i - 1$。p_i 的递推式为：

$$p_{i+1} = p_i + 4x_i + 6 + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i)$$

Listing 5: Bresenham 画圆算法

```
void draw_circle(int radius) {
    int x = 0, y = r;
    int d = 3 - 2 * r;
    while (x <= y) {
        if (d < 0) { /* select East point next */
            d += 4 * x + 6;
        } else { /* select South-East point next */
            d += 4 * (x - y) + 10;
            y--;
        }
        x++;
        draw_8_pts(x, y); /* draws point in each octant */
    }
}
```

更进一步，有 Bresenham 椭圆算法推导与实现。

椭圆的标准方程为：$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$

与圆不同，椭圆在 x 和 y 方向有不同的曲率，因此需要将椭圆按照斜率绝对值与 1 的大小关系分为两个区域.

在区域 1，我们从点 (0,b) 开始，x 递增，y 可能递减。定义误差函数：

$$f(x, y) = b^2x^2 + a^2y^2 - a^2b^2$$

在点 (x_k, y_k)，考虑下一个中点 $(x_k + 1, y_k - 0.5)$：$f(M) = b^2(x_k + 1)^2 + a^2(y_k - 0.5)^2 - a^2b^2$

增量更新：

选择 E 点时：$p1_{k+1} = p1_k + 2b^2x_k + 3b^2$

选择 SE 点时：$p1_{k+1} = p1_k + 2b^2x_k + 3b^2 - 2a^2y_k + 2a^2$

对称地，有：

选择 S 点时：$p2_{k+1} = p2_k - 2a^2y_k + 3a^2$

选择 SE 点时：$p2_{k+1} = p2_k + 2b^2x_k + 2b^2 - 2a^2y_k + 3a^2$

阶段一：基本中点法（浮点版）

核心思想：通过判断中点 $(x+1, y-\frac{1}{2})$ 在椭圆内外的位置，决定下一像素绘制位置。

$$F(x,y) = b^2x^2 + a^2y^2 - a^2b^2$$
$$p_{1,0} = b^2 - a^2b + \tfrac{1}{4}a^2$$

更新规则：

- 若 $p_1 < 0$（中点在椭圆内）⇒ 选择 **E** 点：

$$p_{1,k+1} = p_{1,k} + 2b^2x + 3b^2$$

- 若 $p_1 \geq 0$（中点在椭圆外）⇒ 选择 **SE** 点：

$$p_{1,k+1} = p_{1,k} + 2b^2x + 3b^2 - 2a^2y + 2a^2$$

伪代码：

```
while (b²x <= a²y) {
    draw(x, y);
    if (p < 0)
        p += 2*b²*x + 3*b²;
    else {
        p += 2*b²*x + 3*b² - 2*a²*y + 2*a²;
        y--;
    }
    x++;
}
```

阶段二：乘 4 整数化（去浮点）

改进点：将所有含 $\frac{1}{2}$、$\frac{1}{4}$ 的小数项整体乘 4，消除浮点。

$$P_{1,0} = 4b^2 - 4a^2b + a^2$$

$$P_{1,k+1} = \begin{cases} P_{1,k} + 8b^2x + 12b^2, & (E) \\ P_{1,k} + 8b^2x + 12b^2 - 8a^2y + 8a^2, & (SE) \end{cases}$$

特点：所有变量为整数，加减运算即可。

阶段三：引入 dx, dy, err

核心改进：用 $dx = 2b^2x$、$dy = 2a^2y$ 保存偏导值，避免重复乘法。

$$dx_{next} = dx + 2b^2, \quad dy_{next} = dy - 2a^2$$

整数化伪代码：

```
err = 4*b² - 4*a²*b + a²;
dx  = 0;
dy  = 2*a²*b;

while (dx <= dy) {
    draw(x, y);
    if (err < 0) {
        x++; dx += 2*b²;
        err += dx + b²;
    } else {
        x++; y--;
        dx += 2*b²; dy -= 2*a²;
        err += dx - dy + a² + b²;
    }
}
```

优点：去掉所有乘法，仅保留加法递推。

阶段四：统一循环与矩形适配（高性能整数版）

改进点：

1. 将区域 1、区域 2 合并为统一循环；
2. 引入 $e2 = 2 * err$ 比较判断；
3. 适配任意矩形输入 (x_0, y_0, x_1, y_1)；
4. 加入奇偶高修正：$b1 = b\&1$。

Listing 6: Bresenham 椭圆绘制算法伪代码

```
void plotEllipseRect(int x0, int y0, int x1, int y1)
{
    int a = abs(x1-x0), b = abs(y1-y0), b1 = b&1;
    long dx = 4*(1-a)*b*b, dy = 4*(b1+1)*a*a;
    long err = dx+dy+b1*a*a, e2;

    if (x0 > x1) { x0 = x1; x1 += a; }
    if (y0 > y1) y0 = y1;
    y0 += (b+1)/2; y1 = y0-b1;
    a *= 8*a; b1 = 8*b*b;

    do {
        setPixel(x1, y0); /* I. Quadrant */
        setPixel(x0, y0); /* II. Quadrant */
        setPixel(x0, y1); /* III. Quadrant */
        setPixel(x1, y1); /* IV. Quadrant */
        e2 = 2*err;
        if (e2 <= dy) {
            y0++;
            y1--;
            err += dy += a; }
        if (e2 >= dx || 2 * err > dy) {
            x0++;
            x1--;
            err += dx += b1; }
    } while ((y0 - y1 < b)){
        setPixel(x0 - 1, y0);
        setPixel(x1 + 1, y0++);
        setPixel(x0 - 1, y1);
        setPixel(x1 + 1, y1--);};
}
```

整体对比总结表

阶段	名称	主要改进	特点
1	浮点中点法	中点判定	最直观，含小数
2	×4 整数化	消除浮点	纯整数形式
3	dx/dy/err 引入	去乘法	用增量更新误差
4	统一循环 + 奇偶修正	完整高性能实现	图形库常用版本

4.2 多边形

凸多边形：判断点是否在内部，使用 Bresenham 算法。更快捷的是扫描线算法（查找顶部和底部顶点，左右列出边缘）。

凹多边形：同样用扫描线算法，但有复杂的奇偶规则；

化繁为简：把凹多边形切成若干个凸多边形。（编程原则：用多个简单明确的程序实现而非复杂程序）

最简单的凸多边形：三角形。这在图形学中称为“三角化”。

4.3 颜色插值（Color Interpolations 非纯色）

插值（interpolation）= 根据已知点，推算中间点。

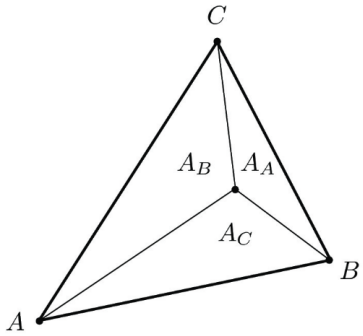
扫描线算法：线性插值，进行加权平均。如Figure 5所示。

- 一维线性插值（linear interpolation）：在两点之间插。
- 二维双线性插值（bilinear interpolation）：在四个像素格子间插。图像一般是双线性插值，对 R、G、B 三个通道分别做插值，最后合并成一个颜色。
- 三维三线性插值（trilinear interpolation）：在 3D 体素（voxel，体积像素）里，对 8 个格点插值。

在不使用扫描线时，我们直接绘制每个单独的像素，就需要确定每个像素相对于三角形三个顶点的权重，构造方法称为**重心坐标法**，如Figure 4。

$$c = \alpha c_0 + \beta c_1 + \gamma c_2, \quad \alpha + \beta + \gamma = 1$$

图像变形其实是同样的原理：坐标映射 + 插值。由于新坐标通常不是整数坐标，有最近邻插值（nearest neighbor，直接取最近的像素）、线性插值等做法。



$$\alpha = \frac{A_A}{A_A + A_B + A_C}$$
$$\beta = \frac{A_B}{A_A + A_B + A_C}$$
$$\gamma = \frac{A_C}{A_A + A_B + A_C}$$

图 4: 重心坐标插值

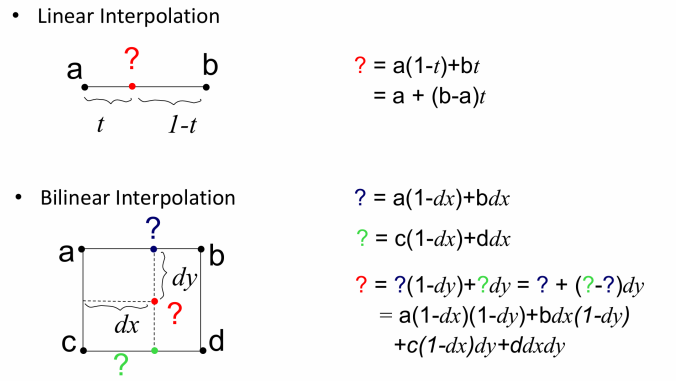


图 5: Interpolation

非透视正确插值 (Non-perspectively correct warping)：直接线性插值，出现弯曲过度。
透视正确插值 (Perspectively correct warping)：在这种方法里，插值时会考虑深度信息。正确做法是：先在三维空间里按比例插值（例如用重心坐标），再通过透视投影除以深度得到屏幕坐标。

4.4 合成 *

主要问题：1. 如何处理前后遮挡关系；2. 如何处理半透明的颜色。

4.4.1 深度缓存

画家算法（Painter’s Algorithm）：给每个形状一个额外的深度属性 z，然后对所有形状的深度进行排序，按照 z 从大到小的顺序进行绘制。但每个物体的深度不是固定值，所有我们为每个像素引入一个深度值，屏幕上所有像素点的深度值构成深度缓存（depth buffer），这个概念与3.2中介绍的帧缓存相对应。

深度值也可以进行插值。

我们也不再需要对所有深度进行排序，而是每个像素独立检测，如果发现等待绘制的图形上的深度小于当前像素的深度，则覆盖当前像素并更新最小深度；否则表示图形被遮挡，不更新屏幕像素。（这是空间换时间复杂度的做法）

Listing 7: 深度缓存算法

```
depth_buffer = initialize_depth_buffer()

for shape in shapes:
    for pixel in shape:
        depth = get_shape_depth(pixel)
        if depth < depth_buffer[pixel]:
            draw(shape, pixel)
            depth_buffer[pixel] = depth
```

4.4.2 半透明

我们用不透明度（opacity）来表达物体透明的程度，一般用 α 表示，因此也可以直接叫 alpha 值。对背景颜色 c_0 叠加半透明颜色 c_1 那么就有：

$$c = \alpha c_1 + (1-\alpha)c_0$$

需要画家算法 + 深度缓存。

4.5 补充 *

设有向面积记作$S(X,Y,Z) = \frac{1}{2}(Y-X)^{\perp}\cdot(Z-X)$　它对每个自变量都是线性的。

已知重心坐标的向量定义：

$$P = \alpha A + \beta B + \gamma C, \qquad \alpha+\beta+\gamma = 1.$$

对固定的 B, C，用对第一个参数的线性性展开：

$$\begin{aligned} S(P,B,C) &= S(\alpha A + \beta B + \gamma C,\ B,\ C) \\ &= \alpha S(A,B,C) + \beta S(B,B,C) + \gamma S(C,B,C) \\ &= \alpha S(A,B,C) \quad (S(B,B,C) = S(C,B,C) = 0). \end{aligned}$$

两边同除以 $S(A,B,C)$（非退化三角形下不为 0），得：

$$\alpha = \frac{S(P,B,C)}{S(A,B,C)}.$$

同理可得：

$$\beta = \frac{S(A,P,C)}{S(A,B,C)}, \qquad \gamma = \frac{S(A,B,P)}{S(A,B,C)}.$$

$$S(P,B,C) = \tfrac{1}{2}(B-C)^{\perp}\cdot(P-C) \quad \Rightarrow \quad \alpha(P) = \frac{(B-C)^{\perp}\cdot(P-C)}{(B-C)^{\perp}\cdot(A-C)}.$$

因此重心坐标是线性的：权重关于位置的导数是一个常数。

由于不透明度是物理量，定义于线性颜色空间中，但在3.3中的伽马校正后，不透明度应当先解码到线性空间、计算，再重新做伽马校正。

5 曲线 (Curves)

5.1 SVG 与表达方式

SVG (Scalable Vector Graphics) 是一种矢量图格式，便于缩放和保持连续性。曲线表示的核心难题：**平衡离散表达与连续表达**，即在保证光滑连续的同时，兼顾高效的操作能力。

常见曲线表达方式：

- **显式表达**：点的信息直接给出，例如 $y = f(x)$。缺点是表达能力不足，例如圆需要分段才能近似。
- **隐式表达**：隐式方程 $f(x, y) = 0$；体数据（多维数组插值定位）；分形递归。优点是便于区分内部/外部，缺点是难以采样。
- **参数表达**：$(x, y) = (f(t), g(t))$。支持维度扩展，表达能力强，取样与可视化方便，并可快速计算切线斜率（通过对 t 求导）。常见参数化方式：角度参数化、弧长参数化等。

5.2 高阶与分段

高阶曲线往往数值不稳定：对控制点的微小变化可能导致曲线大幅震荡。因此常用 **分段参数化**（piecewise），尤其是分段低阶多项式（例如线性、三次）。

折线可以视为两点之间的线性插值（lerp, linear interpolation）：

$$P(t) = (1 - t)P_0 + tP_1, \quad 0 \le t \le 1$$

其中插值的本质就是一种比例关系。

绘图网站推荐：Desmos

5.3 Bézier 曲线

Bézier 曲线源自工程实践。常见性质：

1. **光滑性**：两条相连 Bézier 曲线在连接点处光滑过渡的条件是：公共端点两侧的控制点在一条直线上。三次 Bézier 曲线有 4 个控制点，恰好可以自由控制两端的切线，因此广泛使用。

2. **凸包性质**：由于 Bézier 曲线的基函数非负，且权重和为 1，曲线始终位于控制点凸包之内。

光滑性的更严格区分：

- C^1 连续：切向量在连接处方向和大小都一致。
- C^2 C^1 + 二阶导数一致。
- G^1 连续：切线方向处处一致，但大小可以不同。
- G^2 曲率处处连续。

杂谈：一条分段曲线整体的光滑度，取决于最差的连接处。C 相比 G 多要求一个切向量大小，切向量大小可理解为点在曲线上运动的速率。由于速率本质上还是通过决定下一个点的切线方向来影响曲线形状，如果两条曲线上所有的切线方向都一致，那切线向量大小也一定是一样的。

高阶的函数关系：以二阶 Bézier 曲线为例，曲线可写为

$$S = (1-t)Q_0+tQ_1 = (1-t)\big((1-t)P_0+tP_1\big)+t\big((1-t)P_1+tP_2\big) = (1-t)^2P_0+2t(1-t)P_1+t^2P_2. \tag{3}$$

不难发现，上式中各控制点的权重系数正好是二项式 $(t+(1-t))^2$ 展开后的项。更一般地，n 阶 Bézier 曲线的数学表达式为：

$$S(t) = \sum_{k=0}^{n} B_{n,k}(t)P_k = \sum_{k=0}^{n} C_n^k (1-t)^{\,n-k} t^k \, P_k, \tag{4}$$

其中系数多项式为

$$B_{n,k}(t) = C_n^k (1-t)^{\,n-k} t^k, \tag{5}$$

称为 n 阶 **伯恩斯坦多项式** (Bernstein polynomials)。

5.3.1 de Casteljau 算法

de Casteljau 算法是一种通过多次线性插值来构造 Bézier 曲线的算法。以二次 Bézier 曲线为例，给定三个控制点 P_0, P_1, P_2，算法步骤如下：

1. 第一步：对相邻控制点作线性插值

$$Q_0 = \mathrm{lerp}(P_0, P_1, t) = (1-t)P_0 + tP_1,$$

$$Q_1 = \mathrm{lerp}(P_1, P_2, t) = (1-t)P_1 + tP_2.$$

2. 第二步：再对 Q_0, Q_1 作线性插值

$$S = \mathrm{lerp}(Q_0, Q_1, t) = (1-t)Q_0 + tQ_1.$$

当参数 t 在区间 $[0,1]$ 变化时，点 S 的轨迹即为由 P_0, P_1, P_2 控制的二次 Bézier 曲线。

de Casteljau 算法的优点是：效率较高，数值稳定性好，编程实现方便，因此在计算机图形学中被广泛使用。类似地，三次 Bézier 曲线可通过三轮线性插值得到。

5.3.2 Bézier 曲面 (Bézier Patches)

Bézier 曲线在二维参数空间的推广：

$$S(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} B_i^m(u)\, B_j^n(v)\, P_{i,j}$$

注意控制点们只保证中间那一块是 Bézier 曲面。

5.4 样条曲线 (Splines)

样条曲线是由一系列点定义的平滑曲线。

- **三次样条 (Cubic spline)**：在每段用三次多项式拟合，并附加两个条件（例如两端点二阶导数为零）。
- **三次 Hermite 样条**：分段三次多项式，但不要求二阶导数连续，而是允许直接指定每个顶点的一阶导数。大的好处在于不需要解方程来确定参数，并且具有局部性：我们只需要区间两端的位置和导数就能唯一确定区间内部曲线的形状，与其他控制点的位置无关。

5.4.1 B 样条 (B-spline)

B 样条是一类基于递归基函数的分段曲线（不是插值曲线）：

- **Knots (节点)**：将参数区间划分为若干段。
- **基函数**：递归定义，每个基函数只在有限区间内非零。
- 三次 B 样条基函数跨越 4 段，保证曲线达到 C^2 光滑。

与 Bézier 曲线的区别：

- Bézier：控制点数 = 次数 +1，控制点多 → 高次多项式。
- B 样条：控制点多 → 分段多，但次数固定（通常三次）。

插值越多次数越大：多项式（如 bezier）；若是分段则会分更多段，次数不变。

NURBS (Non-Uniform Rational B-Spline)：- 非均匀：节点间隔不等距。- 有理：每个控制点带有权值 w_i，当 $w_i = 1$ 时退化为普通 B 样条。

Tips: 由于显卡使用矩阵加速，所以常常把多项式变成矩阵表达来加速。

5.5 曲线的光栅化

常见方法：

- **自适应分段线性近似**。
- **DDA 算法与 Bresenham 变体**：
  1. 将曲线转为隐式方程，偏差值定义为隐式方程 $F(x, y)$，在曲线上偏差 = 0。
  2. 细分曲线，保证分段单调。
  3. 已绘制当前点后，比较 x 方向或 y 方向步进的偏差值：
     - 若偏差 $< 0 \rightarrow$ 向 x 方向走；
     - 若偏差 $> 0 \rightarrow$ 向 y 方向走。

5.6 扫描线算法填充

扫描线与曲线交点处理：

- 奇偶规则：奇数交点到偶数交点间填充，偶数到奇数交点不填充（外部或空洞）。
- **环绕数规则 (TTF 字体)**：
  - 外轮廓：顺时针 (+1)。
  - 内轮廓：逆时针 (-1)。

- 从某像素点发射射线：自左向右跨越边界 $\Rightarrow +1$，自右向左跨越边界 $\Rightarrow -1$。
- 环绕数 $\neq 0 \rightarrow$ 点在内部；环绕数 $= 0 \rightarrow$ 点在外部。

6 图像处理

图像：光的能量在二维的分布。

存储 两种存储：1. 矢量图（连续性，任意分辨率，适合线条图）适配的是：硬件-早期电子束显示器 + 现在的软件（如 pdf）2. 光栅化（存储空间小）

现在的图像：像素的 2D 数组。

绝大多数帧缓存（存储像素值二维数组的空间）存储在独立显卡中，在集成显卡里才存在内存里。显示器支持不同类型（黑白 1 位，彩色多位，如 24 位 RGB 各 8 位：0 255，称为**色深**。）。颜色存储方法：直接颜色数据；color map+ 下标值。

常见的图片格式：bmp（直接存储，无损，文件大）；JPEG（压缩，性价比高，有损）；png（无损压缩，可存透明度）tiff（标签化灵活存储，支持多张共存，有无损均可）。

帧缓存额外的位的作用：不透明度（像素有额外的 alpha 通道，通常也是 8bit）；深度值；多缓冲。

点处理 b[x,y] = f(a[x,y])，f 单独变换每个像素值，可以方便地调整对比度（回忆：伽马校正）。

滤波 对“频率”作用。1. 模糊，不改变分辨率，加滤镜 kernel。其他：边滤波。

卷积本指相乘函数的积分，离散化后成为叠加（图像处理中指邻域的加权求和）。卷积可交换，但在离散化以后为了减少计算通常用小的 kernel 在大的图像上作用。卷积是线性运算，滤波器在整张图像上都是一样的，不会因为位置不同而改变，所以这类滤波器叫做**线性、平移不变滤波器 (Linear Shift-Invariant, LSI)**。

sobel 卷积核分别提取水平和竖直，然后叠加开方，也就是提取出梯度的大小而不管它的方向。非线性。

了解：图像补全（剃刀法则：不增加额外信息；梯度尽量小；局部相似）；图像融合（泊松编辑，泛函优化等求解的手段）；有限差分作二阶微分的近似；图像修复 (Inpainting) 或滤波时：边界像素值固定不动 → 就是 Dirichlet 边界条件。绿幕（背景用一种特别鲜艳、和主体颜色差异很大的颜色（通常是绿色，也有用蓝色）。拍摄时，幕布的光会反射到演员身上，尤其是边缘、头发、白衣服等地方。结果：演员边缘区域带上了一点“绿色/蓝色光晕”。这就叫绿色溢出（green spill）或蓝色溢出（blue spill），统称颜色溢出。

7 lab

7.1 lab0:Build the Codebase

也就是俗称的配环境。具体操作步骤参见https://vcl.pku.edu.cn/course/vci/lab0。

7.2 lab1

使用 vscode 找不到头文件，但用 xmake 能正常编译的话可以不用理会。但如果想使用转到定义之类的功能就需要。最后发现问题在于 vscode 如果只打开了子目录（比如 src/VCX/Labs/1-Drawing2D）或者更高一级的目录，那 VSCode 就会在当前目录下找 compile_commands.json，自然报错。所以解决方法是从项目根目录进入。

第七题贝塞尔曲线只需要计算单个点，不需要画整条线。

第六题输入大小不一定等于输出。