1 绘图

1.1 光栅化和直线绘制

1.1.1 光栅化

定义 1.1 光栅化 将图形转换为像素的过程称为**光栅化 (rasterization)**.

光栅化的思想在三维绘制 (即渲染) 中也会用到.

1.1.2 直线绘制

假定我们有一块分辨率为 $W \times H$ 的屏幕, 并且希望在屏幕上绘制直线 $y = kx + b(x_0 \leqslant x_1)$, 起止点为 $(x_0, y_0), (x_1, y_1)$. 为了简化问题, 我们假设 $0 \leqslant k \leqslant 1$.

基于浮点运算的直线绘制法 最简单的想法是遍历所有 $x \in [x_0, x_1]$, 计算对应的 y 值, 然后将 (x, y) 处的像素点填上对应的颜色.

```python
def draw_line(x0:int, y0:int, k:float, b:float, x1:int):
    for x in range(x0, x1 + 1):
        y = k * x + b
        draw(x,round(y))
```

然而, 浮点数的运算开销是较大的. 为了避免浮点数的乘法, 又由于我们的直线是等间隔采样的, 因此可以使用累加法.

```python
def draw_line(x0:int, y0:int, x1:int, y1:int):
    k = (y1 - y0) / (x1 - x0)
    y = y0
    for x in range(x0, x1 + 1):
        draw(x, round(y))
        y += k
```

这就是直线绘制的 **DDA 算法 (Digital Differential Analyzer)**.

但是, 上面的算法中仍然包含了浮点数的加法.

Bresenham 直线算法 Bresenham 教授在 1962 年提出了著名的**布雷森汉姆直线算法 (Bresenham's Line Algorithm)**, 能够在只计算整数加减法的情况下获得和 DDA 算法相同的结果.

令直线的起终点为 P_0, P_1. 令 $\overrightarrow{P_0P_1}$ 顺时针旋转 $90°$ 形成的向量 $\boldsymbol{n} = (y_1 - y_0, x_0 - x_1)$, 即直线的法向量, 那么直线上任意一点 P 就满足

$$F(P) = \boldsymbol{n} \cdot \overrightarrow{P_0P} = 0$$

这就是直线的隐式方程. 给出任意一点 Q, 判断 Q 在直线的上方还是下方, 只需判断 $F(Q)$ 的符号即可. 不难得出, 如果 $F(Q)$ 为负, 那么 Q 在直线上方, 反之则在直线下方.

假定已经绘制了直线上的一点 (x_i, y_i). 对于斜率 $k \in (0,1)$ 的直线, 下一个像素的坐标仅有两种可能: (x_i+1, y_i) 或 (x_i+1, y_i+1). Bresenham 认为如果上述两点的中点 $\left(x_i+1, y_i+\dfrac{1}{2}\right)$ 在直线上方, 就绘制 (x_i+1, y_i), 否则绘制 (x_i+1, y_i+1). 这种判断方法显然比较符合直觉.

现在我们考虑具体的实现方法. 对于已经绘制的点 $P_i\left(x_i, y_i\right)$, 需要用点 $P_i'\left(x_i+1, y_i+\dfrac{1}{2}\right)$ 判断下一个像素. 为了避免直接计算 $F\left(P_i'\right)$(否则我们又要计算乘法), 我们考虑下面的递推方法:

$$F\left(P_i'\right) = \boldsymbol{n} \cdot \overrightarrow{P_{i-1}'P_i'} = \boldsymbol{n} \cdot \left(\overrightarrow{P_0P_{i-1}'} + \overrightarrow{P_{i-1}'P_i'}\right) = F\left(P_{i-1}'\right) + \boldsymbol{n} \cdot \boldsymbol{\delta}$$

其中 $\boldsymbol{\delta} = (1,0)$ 或 $(1,1)$(判断点的移动和绘制点的移动显然是同步的). 这样, 画完之后我们可以每次更新 $F\left(P_i'\right)$ 的值以进行下一个像素位置的判断.$\boldsymbol{n} \cdot \boldsymbol{\delta}$ 可以在循环前就算好, 不必重复. 递推的初值为

$$F\left(P_0'\right) = F\left(P_0\right) + \boldsymbol{n} \cdot \left(1, \frac{1}{2}\right) = (y_1 - y_0) - \frac{1}{2}(x_1 - x_0)$$

为了避免 1/2 带来的浮点运算, 我们可以把 F 放大 2 倍.

总结而言, 我们可以将上面的方法写成下面的程序:

```python
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(x, y)
        if F < 0: # F(P_i') < 0
            F += dy # F(P_{i+1}') = F(P_i') + n * delta_0
        else: # F(P_i') >= 0
            y += 1
            F += dydx # F(P_{i+1}') = F(P_i') + n * delta_1
```

所有直线的绘制 我们在前面只讲了斜率 $k \in [0,1]$ 的直线的绘制. 对于其他斜率的直线, 通过适当的变换也可以做到一样的效果.

首先根据起终点计算直线的斜率 k, 如果 $|k| \leqslant 1$, 那么遍历 x 坐标并按照上述算法更新 y 坐标 (如果 $k<0$, 更新 y 时需将其递减); 如果 $|k|>1$, 那么交换前述 x 和 y 的角色即可. 代码实现如下:

```python
def draw_line(x0: int, y0: int, x1: int, y1: int):
    f = (y1 - y0) < (x1 - x0)
    if not f: x0, y0, x1, y1 = y0, x0, y1, x1
    if x0 > x1: x0, y0, x1, y1 = x1, y1, x0, y0
    y, sy = y0, 1 if (y1 > y0) else -1
```

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

1.2 多边形填充

1.2.1 多边形的光栅化

三角形是最简单的多边形, 也是多边形的基本组成部分. 最简单的办法是枚举所有可能的像素点, 判断其是否在三角形内.

```
def draw_triangle(x0: int, y0: int, x1: int, y1: int, x2: int, y2: int):
    xmin, xmax = min(x0, x1, x2), max(x0, x1, x2)
    ymin, ymax = min(y0, y1, y2), max(y0, y1, y2)
    for x in range(xmin, xmax + 1):
        for y in range(ymin, ymax + 1):
            if inside(x, y, x0, y0, x1, y1, x2, y2):
            draw(x, y)
```

要判断点 Q 是否在三角形内, 可以将边按逆时针顺序排列, 并将边逆时针旋转 $90°$ 得到的法向量记作 $\boldsymbol{N}_i(i=0,1,2)$. 各 $\boldsymbol{N}_i$ 都指向三角形内部, 于是只要各 $F_i(Q)=\boldsymbol{N}_i\cdot\overrightarrow{P_iQ}>0$ 即可说明 Q 在三角形内.

然而, 上面的方法对于每个点都要计算三次点积, 效率不高. 注意到我们每次总是在按行填充像素, 因此只要维护每行的起点 x_L 和终点 x_R 即可. 我们也不必通过点积计算点的位置, 只要根据边的斜率和上一行的起终点更新即可 (即使用 DDA 算法更新起终点). 这就是**扫描线算法 (Scanline Algorithm)**.

对于多边形, 也可以用扫描线算法绘制, 但需要注意非凸的情形. 此时我们需要根据交点的数目按奇偶规则填充像素. 假定交点为 $x_1,\cdots,x_n$, 那么需要填充的部分为 x_{2k-1} 到 x_{2k} 的部分. 这可以通过简单的画图证明. 当然, 也可以将多边形分解成多个三角形进行分别绘制.

扫描线算法相比于简单的三角形算法更高效, 然而在现代计算机上, 其实使用的最多的是并行版的简单算法. 由于三角形在图形学中的重要性, 现代 GPU 搭载了用于三角形内外检测的专门模块, 从而在硬件上实现并行的简单算法.

1.2.2 多边形的插值

颜色插值 在绘制多边形时, 我们有时希望多边形的颜色是渐变的. 在绘制线段时, 假定两个顶点 (x_0, y_0) 和 (x_1, y_1) 的颜色为 c_0 和 c_1, 我们可以通过线性插值得到线段上 (x, y) 的颜色:

$$c = \frac{(x_1 - x)c_0 + (x - x_0)c_1}{x_1 - x_0} = c_0(1 - t) + c_1 t, t = \frac{x - x_0}{x_1 - x_0}$$

对多边形进行插值时, 我们可以应用扫描线算法, 每一行内的颜色由两个端点插值得到, 而端点的颜色又可以由对应的边的端点插值得到. 上述插值方法即**双线性插值 (Bilinear Interpolation)**.

在不使用扫描线算法的情况下, 我们需要单独确定每个点的颜色. 我们可以用**重心坐标 (Barycentric Coordinate)** 来做插值. 设三角形的顶点为 A, B, C, 那么对于三角形内的任意一点 P, 令

$$\alpha = \frac{S_{\triangle PBC}}{S_{\triangle PBC}} \qquad \beta = \frac{S_{\triangle PAC}}{S_{\triangle PBC}} \qquad \gamma = \frac{S_{\triangle PAB}}{S_{\triangle PBC}}$$

根据中学所学的平面向量知识不难得出

$$\alpha + \beta + \gamma = 1 \qquad \alpha\overrightarrow{PA} + \beta\overrightarrow{PB} + \beta\overrightarrow{PC} = \mathbf{0}$$

于是我们可以把 P 处的颜色表示为

$$c = \alpha c_A + \beta c_B + \gamma c_C$$

有关上面两种插值方式有以下结论.

定理 1.2 重心插值和双线性插值的等价性 重心插值是线性的, 即在任意一处发生相同位移时颜色的改变值一样. 特别地, 三角形的重心插值和双线性插值等价.

证明. 仍然设三角形的顶点为 $A\left(x_0, y_0\right), B\left(x_1, y_1\right), C\left(x_2, y_2\right)$, 并设 $P\left(x, y\right)$ 在三角形内. 重心坐标的表示要求

$$\begin{cases} \alpha_P(x_0 - x) + \beta_P(x_1 - x) + \gamma_P(x_2 - x) = 0 \\ \alpha_P(y_0 - y) + \beta_P(y_1 - y) + \gamma_P(y_2 - y) = 0 \\ \alpha_P + \beta_P + \gamma_P = 1 \end{cases}$$

这一线性方程组具有唯一解

$$\begin{cases} \alpha_P = \dfrac{\left(y - y_1\right)\left(x_2 - x_1\right) - \left(x - x_1\right)\left(y_2 - y_1\right)}{\left(y_0 - y_1\right)\left(x_2 - x_1\right) - \left(x_0 - x_1\right)\left(y_2 - y_1\right)} \\ \beta_P = \dfrac{\left(y - y_2\right)\left(x_0 - x_2\right) - \left(x - x_2\right)\left(y_0 - y_2\right)}{\left(y_1 - y_2\right)\left(x_0 - x_2\right) - \left(x_1 - x_2\right)\left(y_0 - y_2\right)} \\ \gamma_P = \dfrac{\left(y - y_0\right)\left(x_1 - x_0\right) - \left(x - x_0\right)\left(y_1 - y_0\right)}{\left(y_2 - y_0\right)\left(x_1 - x_0\right) - \left(x_2 - x_0\right)\left(y_1 - y_0\right)} \end{cases}$$

α_P, β_P 和 γ_P 都是关于 x, y 的线性函数, 因此 P 处的颜色 c_P 可以写为如下形式:

$$c_P = k_1 x + k_2 y + C$$

其中 k_1, k_2, C 与 x, y 无关. 因此, 重心插值是线性的.

现在来证明重心插值和双线性插值等价. 如果将颜色 c 作为三维空间中的 z 坐标, 那么这两种插值方式都在空间中描绘了一个平面. 它们都过三角形的顶点对应的 (x_i, y_i, c_i), 因而对应同一平面, 从而等价. □

多边形的拉伸 我们以图片的拉伸作为多边形插值的例子. 考虑一张四边形图片, 顶点为 A, B, C, D, 现将这张图片拉伸到屏幕上的四边形 $A'B'C'D'$. 我们可以在原图建立直角坐标系, 每个顶点对应 (u_i, v_i), 在拉伸后顶点和边上的坐标值不变 (这可以先由边的线性插值得到), 然后对四边形内的点进行双线性插值. 上面的方法称作 **UV 坐标映射 (UV Mapping)**.

尽管三角形的重心插值和双线性插值等价, 但四边形没有唯一的插值方法. 直接对其双线性插值和分解成两个三角形进行插值得到的结果不同.