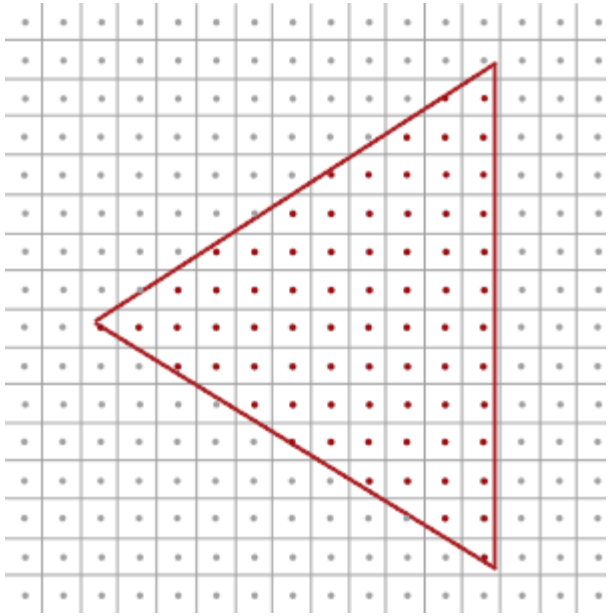
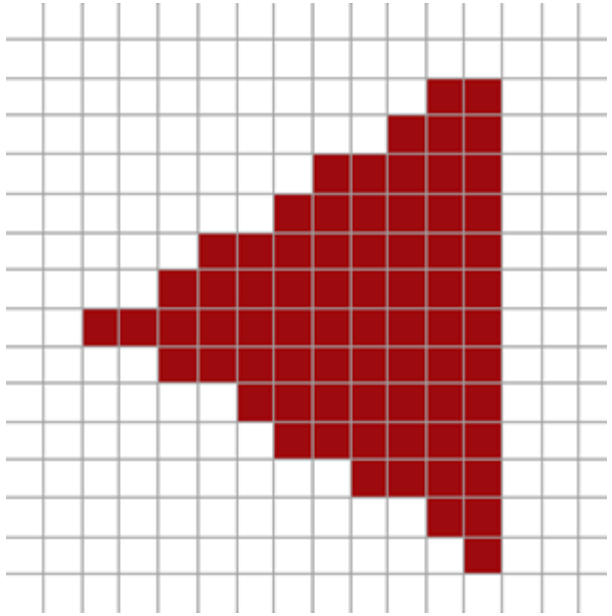
Super Sampling Anti Aliasing (SSAA)

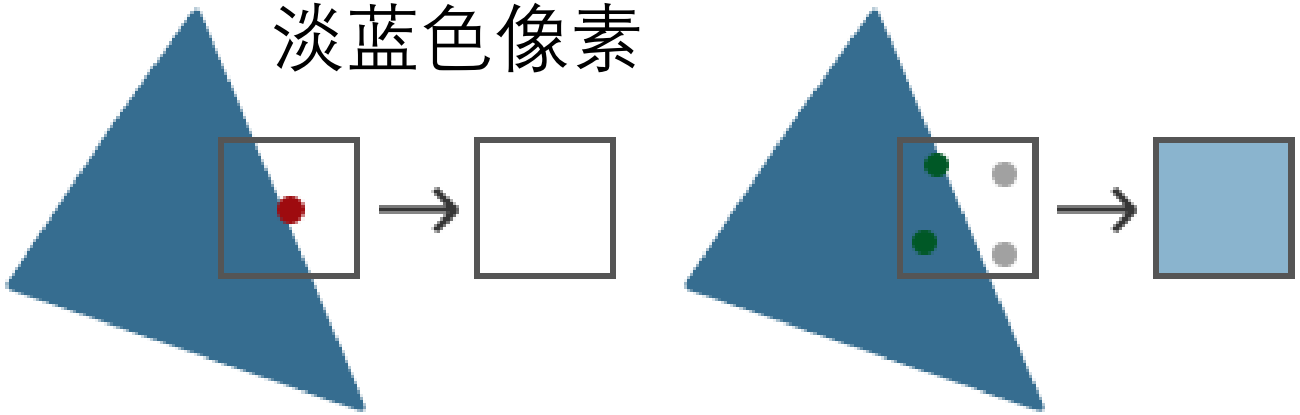
- 考虑这样一个图形的抗锯齿，首先我们考虑它最基本画出来的形态（右图）
- 由于屏幕像素总量的限制，有些边缘的像素能够被渲染出来，而有些则不会，结果就是我们使用了不光滑的边缘来渲染图元，导致之前讨论到的锯齿边缘





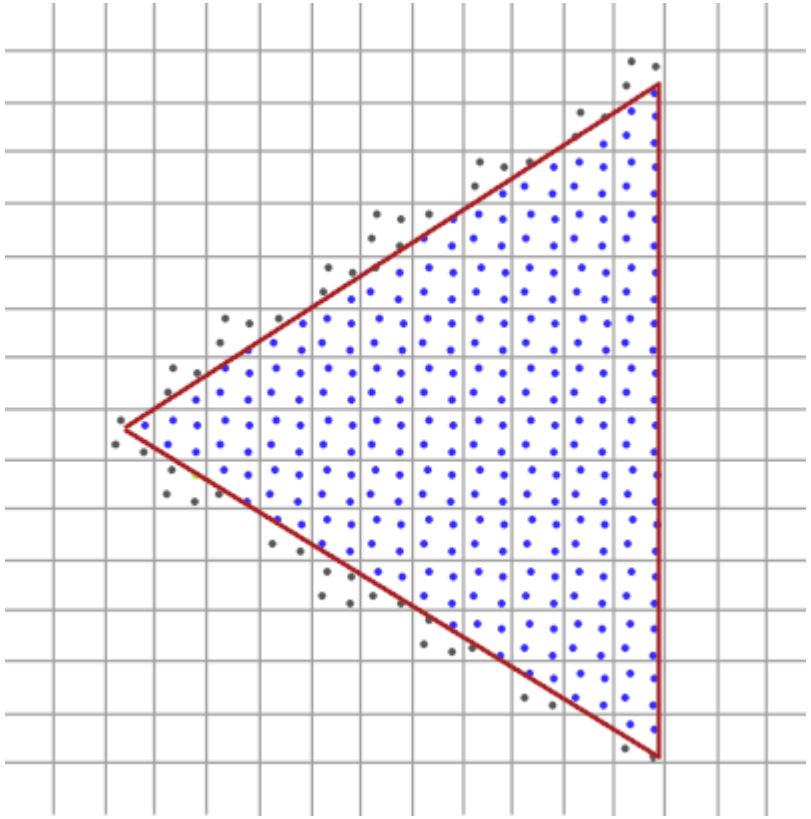
Super Sampling Anti Aliasing (SSAA)

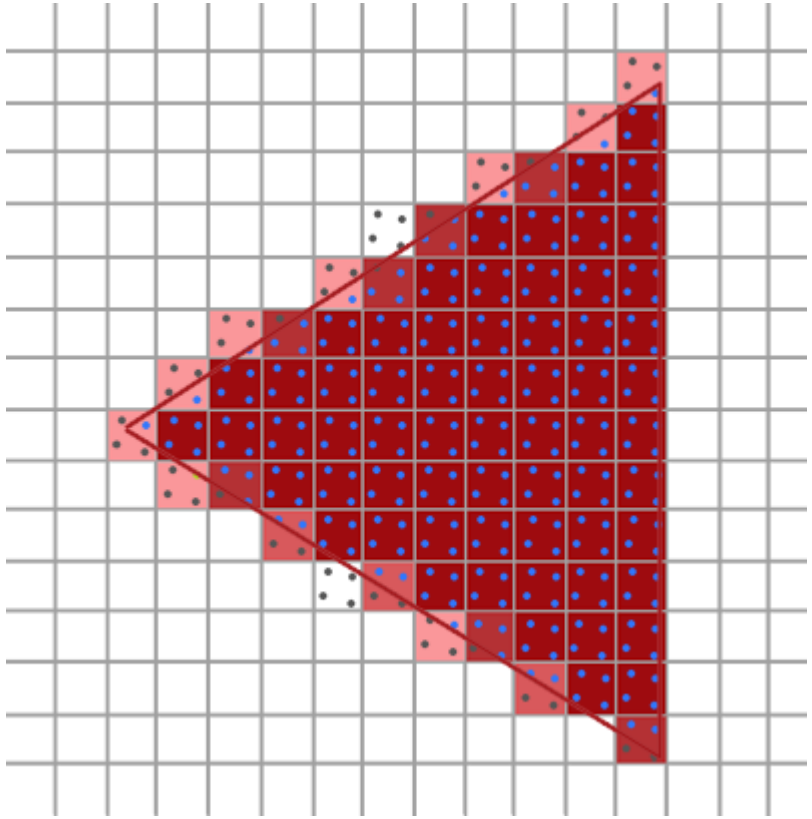
- 原始的像素是非0则1的，即每个像素采样的时候只询问是渲染这个颜色还是不渲染这个颜色
- 实际上我们进行一个很简单的小操作就可以较好的解决这个问题，我们可以为每个像素增加几个采样点，如下图，有两个采样点返回了有颜色
- 再回到这个像素来看，那么最终的目标颜色就是1/2的原色彩，也就是最终我们得到了一个淡蓝色像素



Super Sampling Anti Aliasing (SSAA)

- 最后我们用新的方法对原始的图元进行采样，就可以得到抗锯齿后的图形





Super Sampling Anti Aliasing (SSAA)

- 代码实现也非常的简单，下面给出伪代码的算法
- 即目标(des)的每个像素点的值为源(src)中的均值

```
function:
    For x from 0 to src_width:
        For y from 0 to src_height:
            sum = 0
            For i from 0 to SSAA_times:
                For j from 0 to SSAA_times:
                    sum += src[y * SSAA_times + i][x * SSAA_times + j]
            sum /= SSAA_times * SSAA_times
            des[y][x] = sum
```

Super Sampling Anti Aliasing (SSAA)

- 对于在easyx中如何建立一块超采样的画布，可以考虑以下代码

```
IMAGE *ssaa_img = new IMAGE(src_width * SSAA_times, src_height * SSAA_times);
 // 将ssaa画布设置为绘图设备
SetWorkingImage(ssaa_img);
// 以下绘图操作都会绘制在 ssaa_img 对象上面
line(0, 100, 200, 100);
line(100, 0, 100, 200);
circle(100, 100, 50);
// 设置绘图目标恢复为默认的绘图窗口
SetWorkingImage(NULL);
// 自行实现一个算法将SSAA画布绘制到当前窗口中
SSAA_putimage(ssaa_img);
```

Super Sampling Anti Aliasing (SSAA)

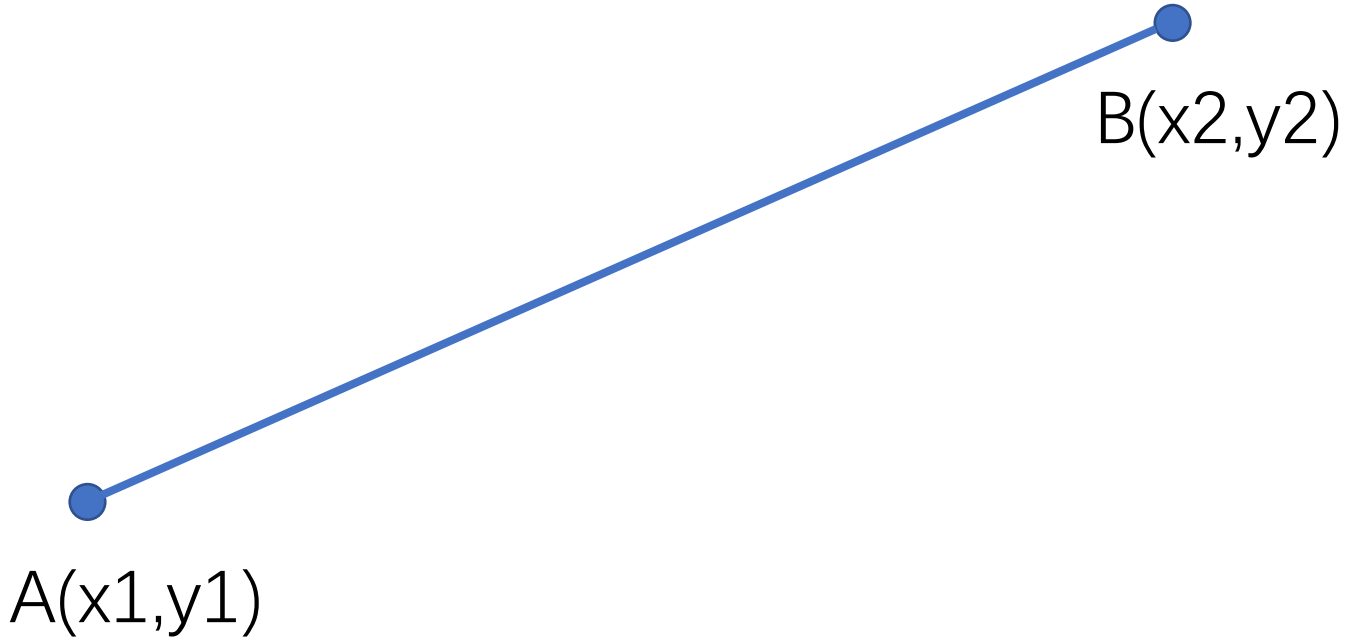
- 最终效果，可见左边（原始）具有大量的锯齿，中间（2xSSAA）的锯齿情况改善较好，右边（4xSSAA）和中间基本差不多一致

Signed Distance Filed (SDF)

- 超采样它虽然简单而且很香，只用很短的代码很简单的逻辑就实现了抗锯齿，但是他有很糟糕的性能问题
- 例如刚刚的demo，实际上运行时间超过了10秒三条直线才绘制完成，这完全不能满足时钟这样一秒需要更新一次的程序
- 所以我们需要考虑一个更高效的方案，观察抗锯齿过后的图形，以及刚刚SSAA的想法，我们不难发现，只要实现像素平滑的过渡，我们就能实现抗锯齿，进一步观察，我们发现越靠近边缘的图案，他的颜色越淡
- 那么有什么样的方式可以实现当像素越接近边缘的时候，它的颜色越淡呢？
- 于是，可微分的有符号距离场(SDF) 就这么被提出了

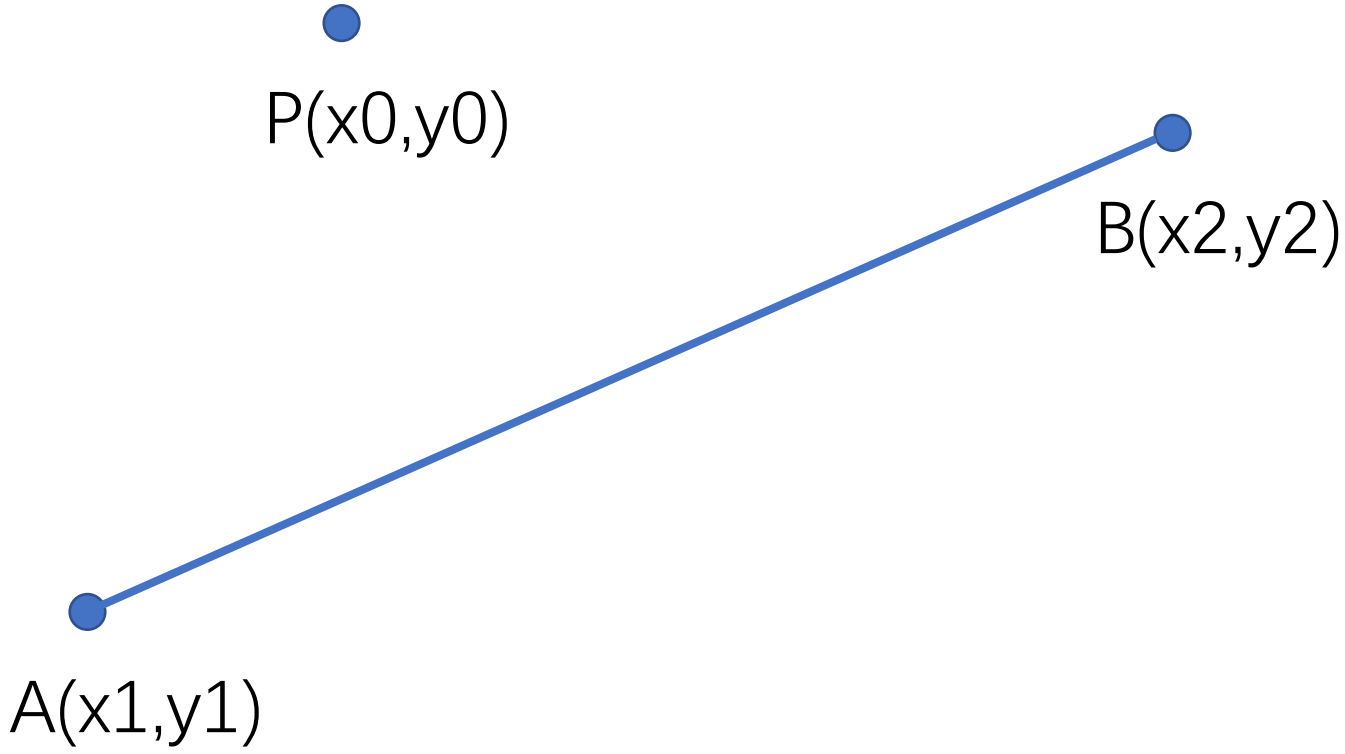
Signed Distance Filed (SDF)

- 依然以一个简单的案例，还是画直线，设A(x1,y1) B(x2,y2)两点，作一条直线（线段），并考虑抗锯齿



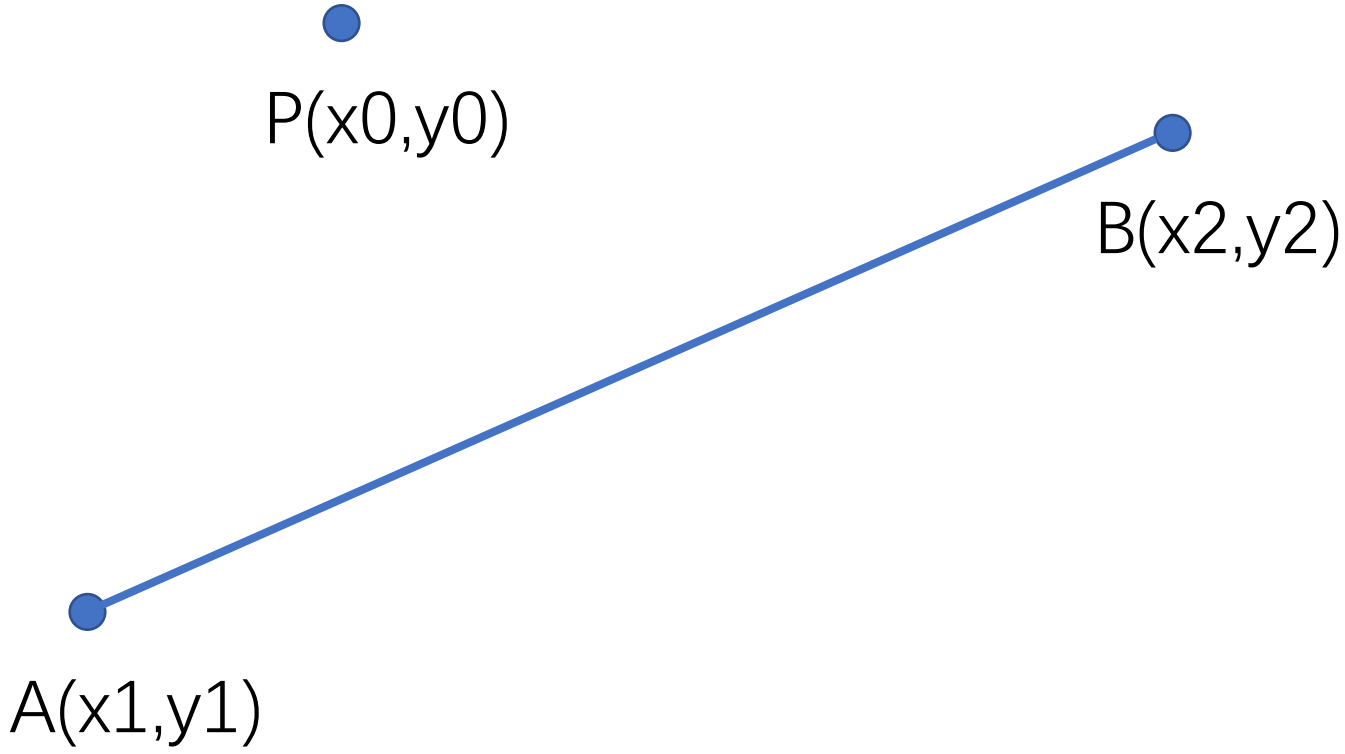
Signed Distance Filed (SDF)

- 问题：此时给出像素点p(x0,y0)，求出该像素点的颜色



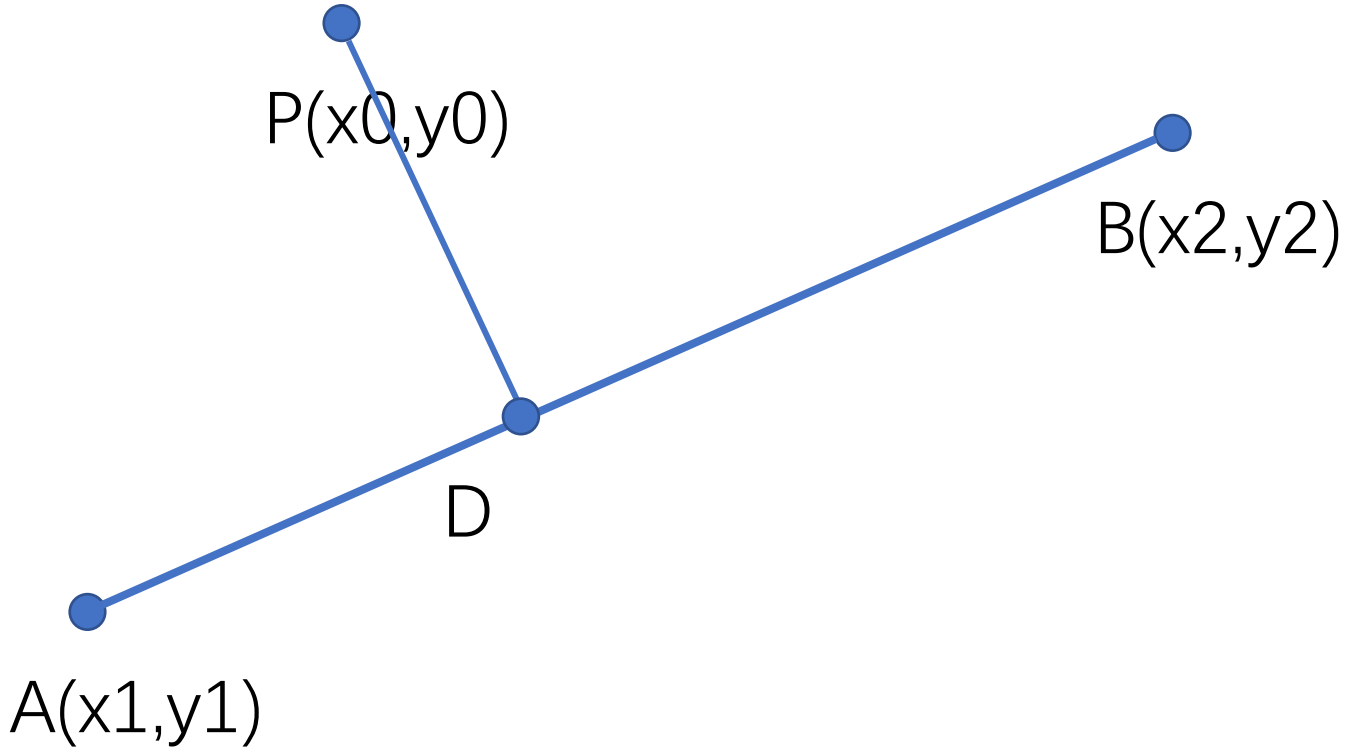
Signed Distance Filed (SDF)

- 考虑：我们需要让他越靠近边缘时颜色越淡，离开直线区域则颜色为空，在直线内部则颜色为纯色



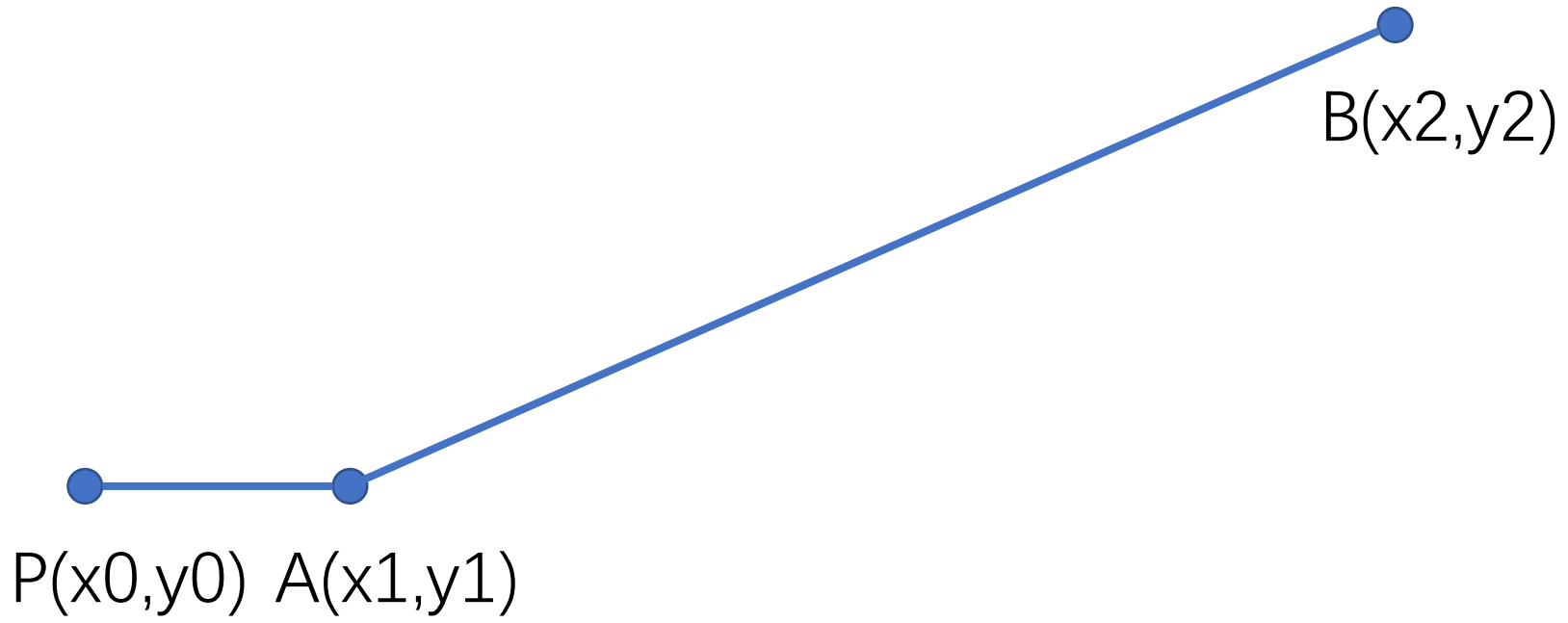
Signed Distance Filed (SDF)

- 求点P到AB的距离PD即可，有了距离后，我们就可以根据距离判断该像素的颜色深浅，与背景应该如何混合



Signed Distance Filed (SDF)

- 需要注意的是，这里不能简单地用点到直线的距离公式，因为我们的AB是一条线段，我们需要求的应该是点P到AB的最短距离，例如下面P在端点的话，最短距离就是PA

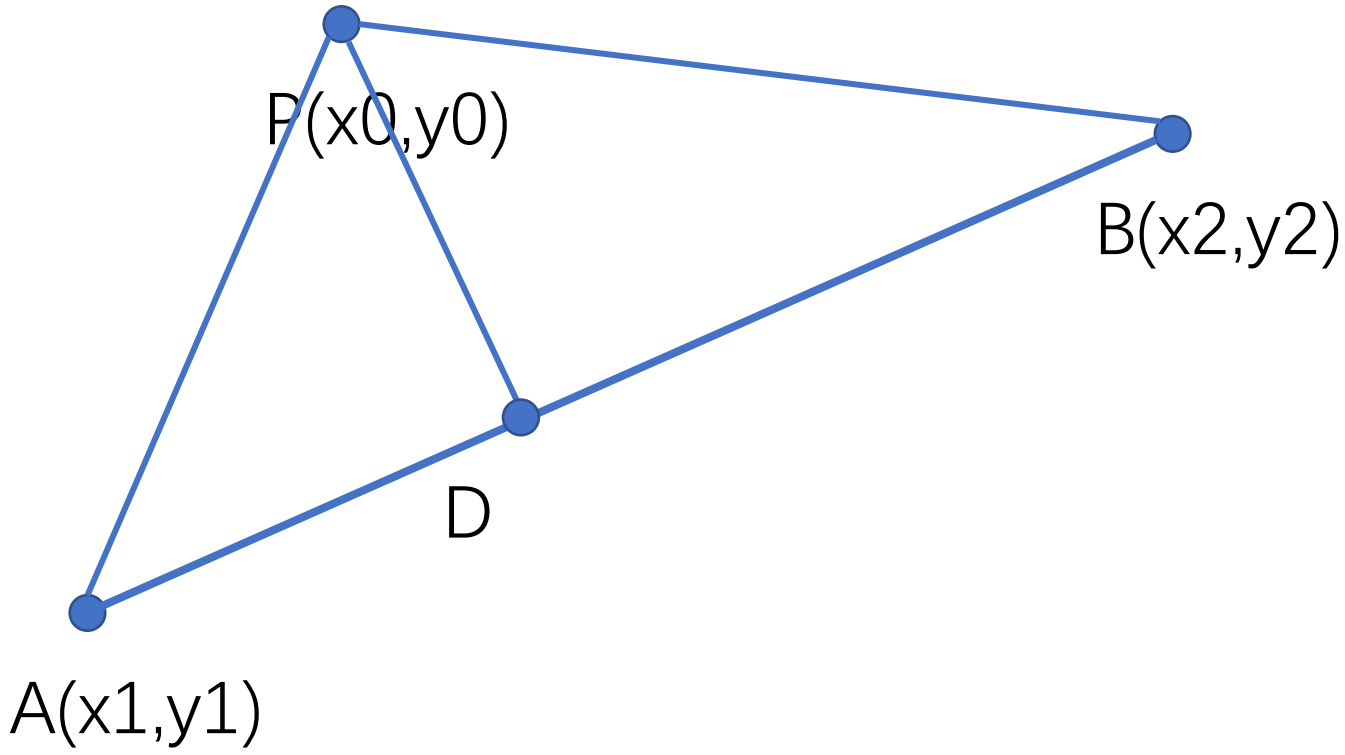


Signed Distance Filed (SDF)

- 这里我们给出一个通用的公式

$$h = \frac{\overrightarrow{PA} \cdot \overrightarrow{BA}}{\overrightarrow{BA}^2} \quad (0 \le h \le 1)$$

$$d = \sqrt{\overrightarrow{PA}^2 - (\overrightarrow{BA} \cdot h)^2}$$

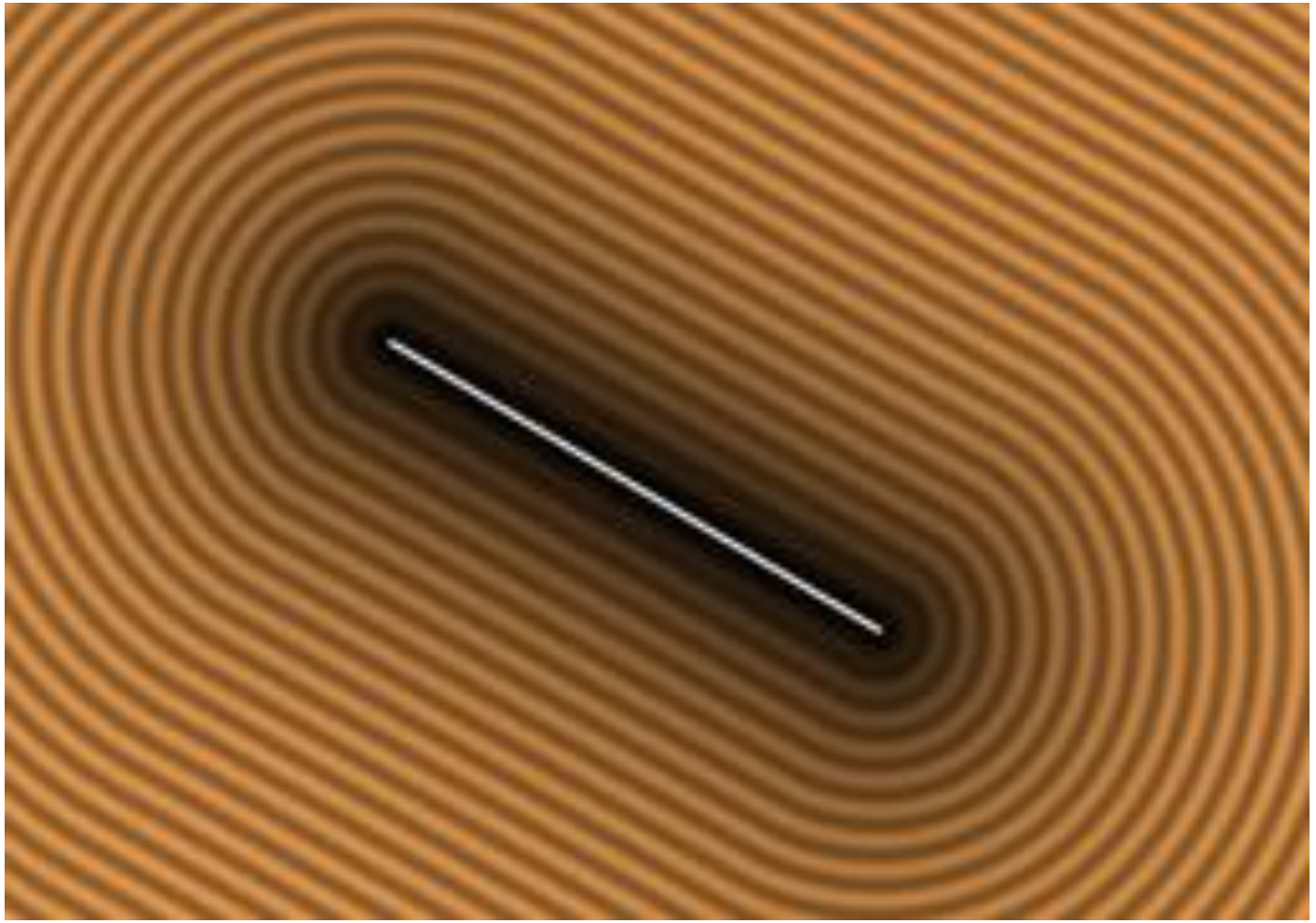


Signed Distance Filed (SDF)

$$h = \frac{\overrightarrow{PA} \cdot \overrightarrow{BA}}{\overrightarrow{BA}^2} \ (0 \le h \le 1)$$

$$d = \sqrt{\overrightarrow{PA}^2 - (\overrightarrow{BA} \cdot h)^2}$$

- 通过这个公式我们将求出一个距离场
- 该距离场指示越靠近直线的地方d值越小（中间黑色甚至亮点处），反之d值越大
- 另外这个距离场公式的特性就是，在直线内部会在0附近，在边缘会在0-1之间，且是可以微分的，因为这是一个连续的公式，那么这个公式就符合我们最开始的预期

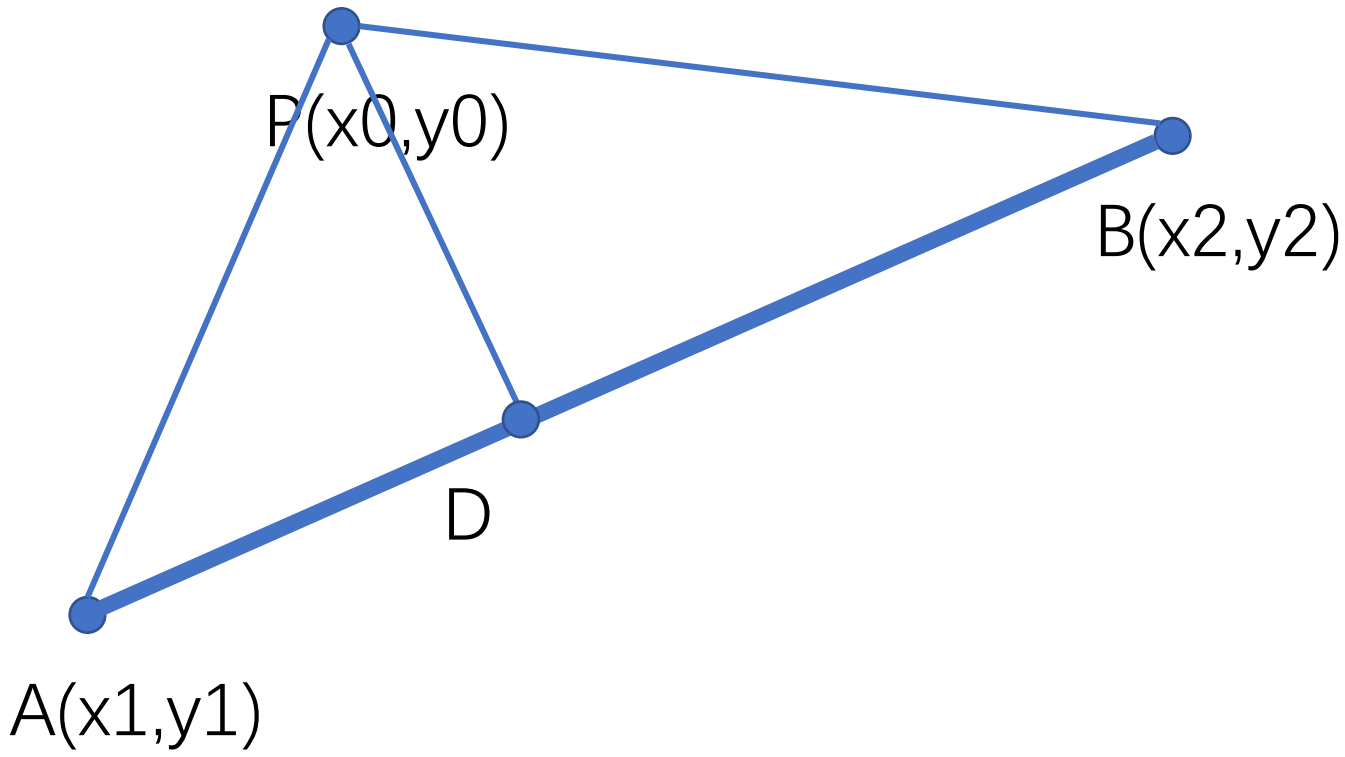


Signed Distance Filed (SDF)

- 更进一步，如果希望直线拥有体积，可以让答案减去至直线的半径r

$$h = \frac{\overrightarrow{PA} \cdot \overrightarrow{BA}}{\overrightarrow{BA}^2} \ (0 \le h \le 1)$$

$$d = \sqrt{\overrightarrow{PA}^2 - (\overrightarrow{BA} \cdot h)^2} - r$$

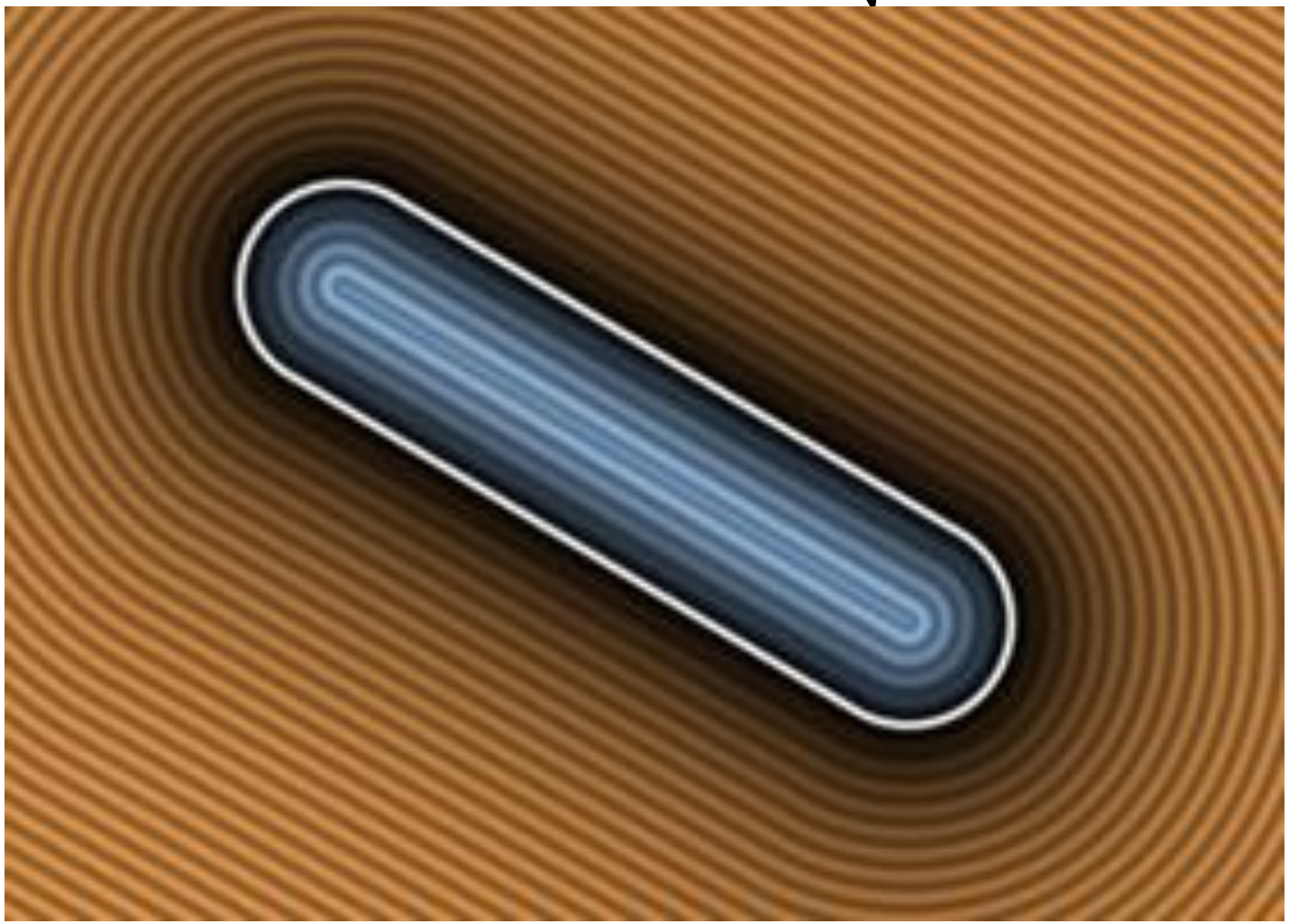


Signed Distance Filed (SDF)

$$h = \frac{\overrightarrow{PA} \cdot \overrightarrow{BA}}{\overrightarrow{BA}^2} \ (0 \le h \le 1)$$

$$d = \sqrt{\overrightarrow{PA}^2 - (\overrightarrow{BA} \cdot h)^2} - r$$

- 带体积的可视化或许会让你对这个公式更加明白一些



Signed Distance Filed (SDF) + Alpha Blending

- 想要求出该点具体的颜色，我们还需要引入一个新概念：Alpha Blending
- 不难想象，我们刚刚公式求出来的是一个Alpha值，代表这个像素颜色的深浅，我们有深浅后再回想一下在SSAA中我们是否对四个像素的颜色都求了一次平均
- 也就是说我们这个点的颜色不仅要考虑直线的颜色，还需要考虑他下面已经有的像素颜色，故我们需要如下的公式对背景色和前景色进行混合，参见下面公式

$$alpha = 0.5 - d\ (0 \leq alpha \leq 1)$$

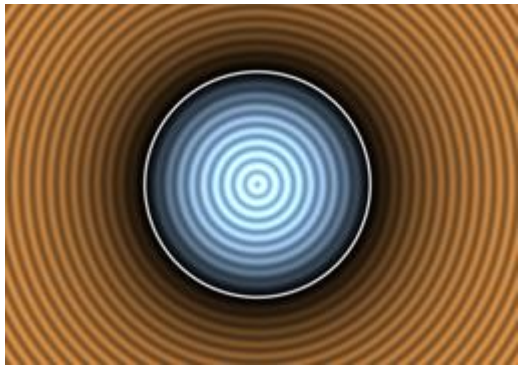
$$final_{color} = origin_{color} * (1 - alpha) + target_{color} * alpha$$

Signed Distance Filed (SDF) + Alpha Blending

- 最终效果，可见和4xSSAA质量相当，但是执行速度快了近10倍

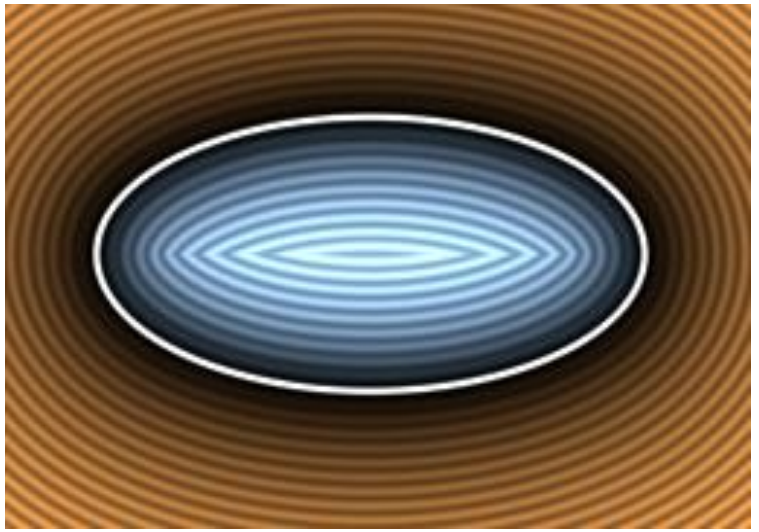
Signed Distance Filed (SDF)

- SDF方程的重点是推导一个点到一个点/直线/形状的最短距离的方程，由于它被方程描述所以是一个连续的、可微分的方程，所以我们可以利用方程进行抗锯齿计算
- 更多的，例如圆的SDF方程就相当简单： $d = |\overrightarrow{PA}| - r$
- 而椭圆的SDF方程相当复杂：



$$k_0 = \sqrt{\left(\frac{x_0 - x}{A}\right)^2 + \left(\frac{y_0 - y}{B}\right)^2} \quad k_1 = \sqrt{\left(\frac{x_0 - x}{A^2}\right)^2 + \left(\frac{y_0 - y}{B^2}\right)^2}$$

$$d = \frac{k_0(k_0 - 1)}{k_1}$$

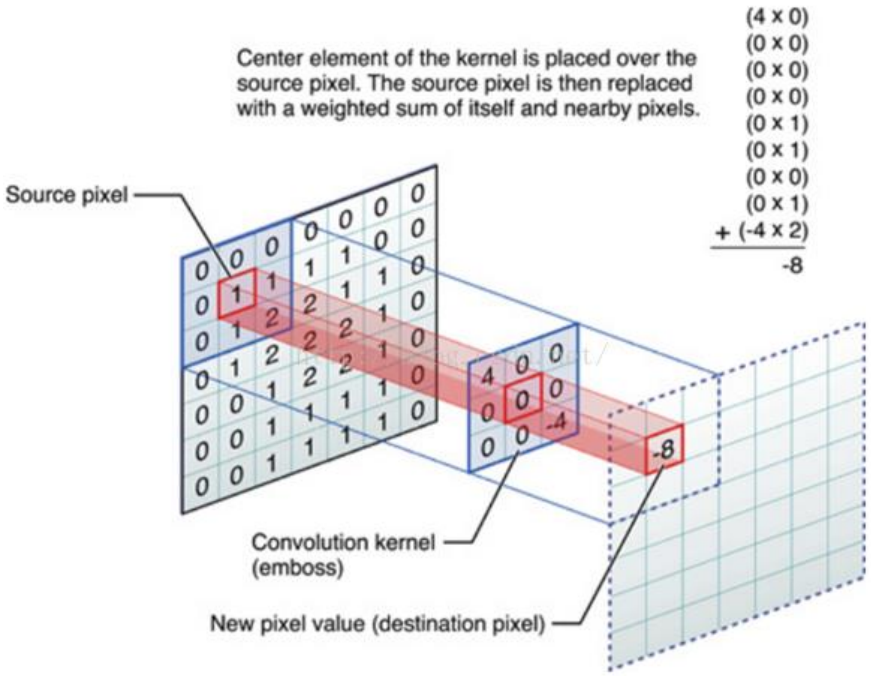


Advanced Graphics Algorithms

- 介绍几个经典的图形学算法，但不一定简单
- 高斯模糊
- 贝塞尔曲线
- Alpha通道叠加

Gaussian Blur

- 高斯模糊，听起来很高大上，但是不妨回忆一下上学期有关图像处理的大作业，是不是做了个卷积
- 高斯模糊的实质就是卷积，考察以下的高斯卷积核

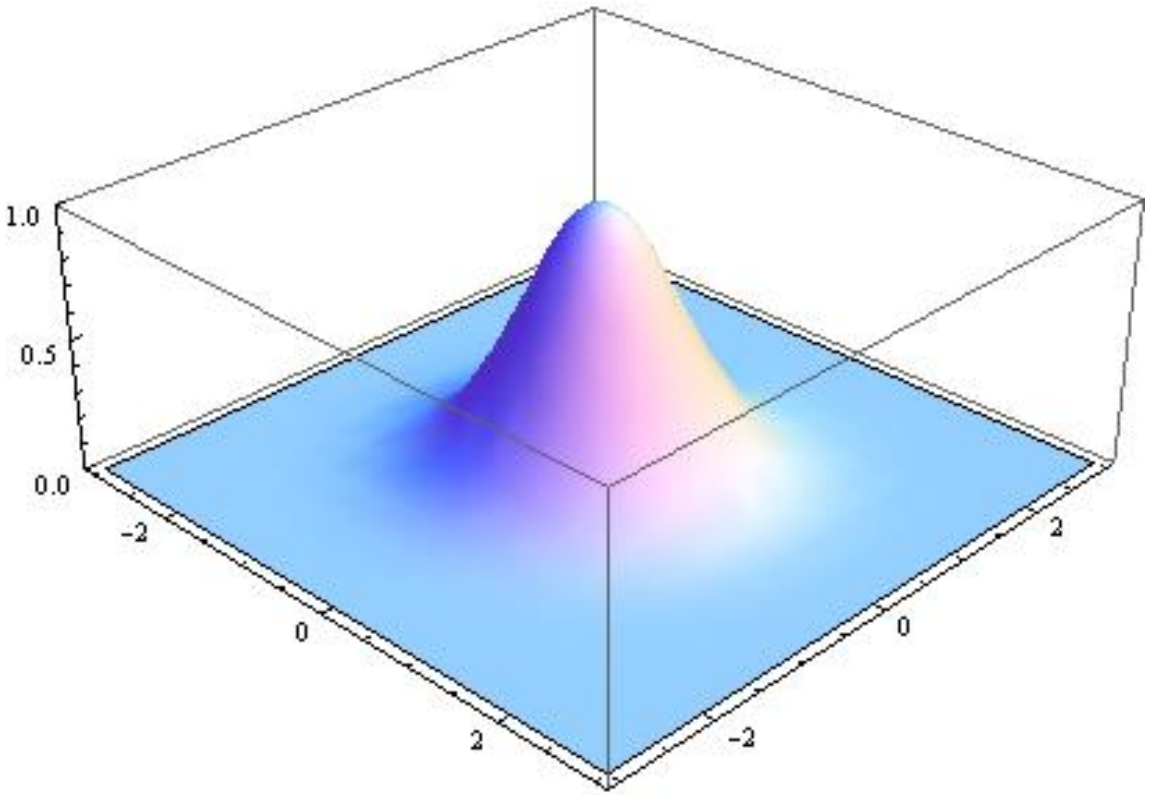


0.0947416	0.118318	0.0947416
0.118318	0.147761	0.118318
0.0947416	0.118318	0.0947416

Gaussian Blur

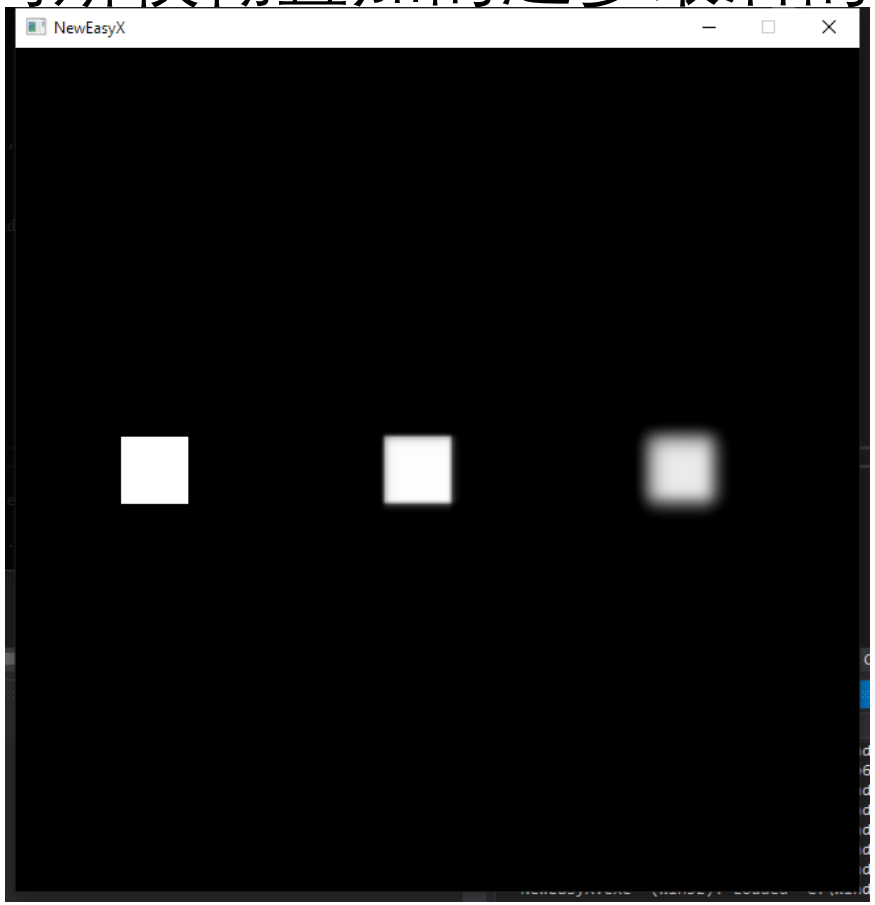
- 高斯卷积核其实是一个符合正态分布的矩阵，他的三维图像长这样，不妨想象下这样的卷积核卷出来的效果是怎么样的
- 正态分布的密度函数如下，可以用这个生成高斯卷积核的参数，通常地，σ取1.5-2
- 一维形式 $$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$
- 二维形式

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$



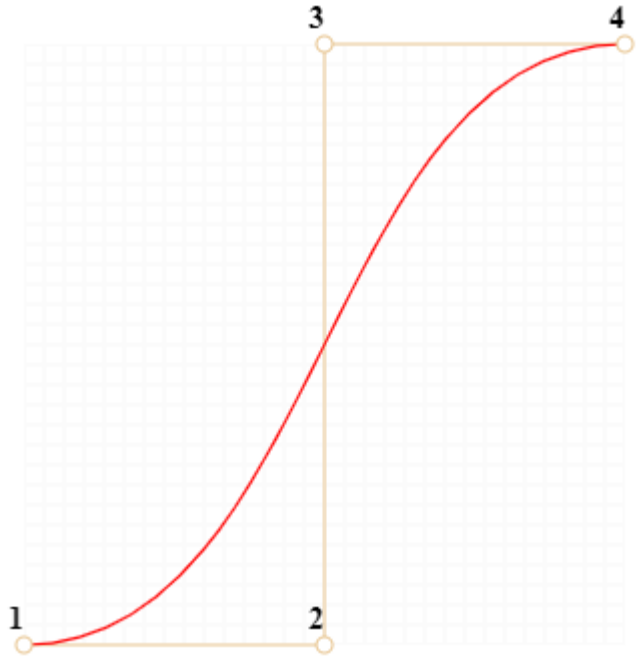
Gaussian Blur

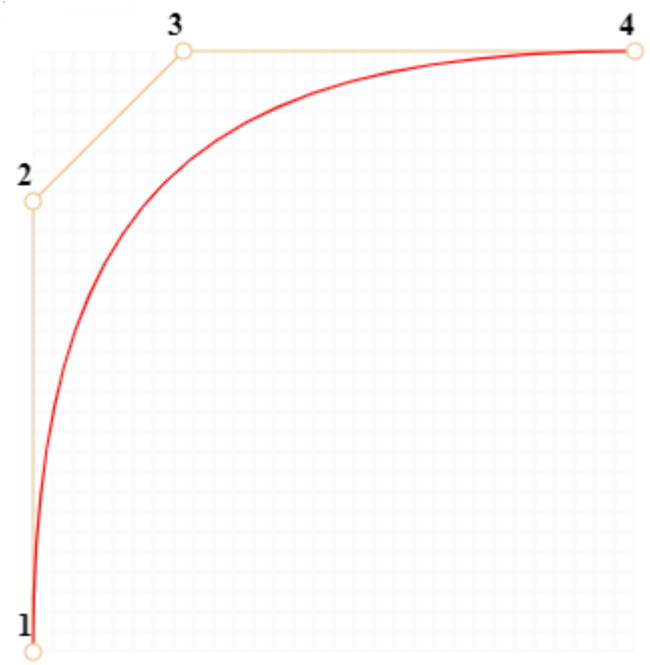
- 高斯模糊是可以叠加的，模糊的效果随着卷积核的大小也各不相同，越大的卷积核模糊的效果越好，高斯模糊叠加的越多最后的结果看起来将更加模糊
- 如图，分别是原始，1次高斯模糊和10高斯模糊的结果

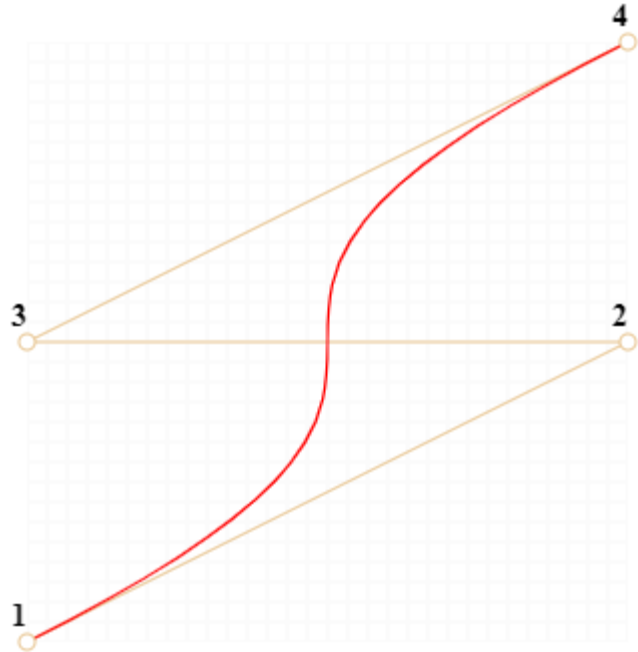


Bezier Curve

- 贝塞尔曲线是另一个非常常见的图形学算法，人们通常希望绘制一些圆滑的线条或者图形而不是有棱有角的线段，于是贝塞尔曲线被提出用于绘制圆滑的线条，在这里不会详细展开数学公式和代码，有兴趣的可以自行了解https://zh.javascript.info/bezier-curve
- 简单描述：用四个点确定一个圆滑曲线的走向，如下图

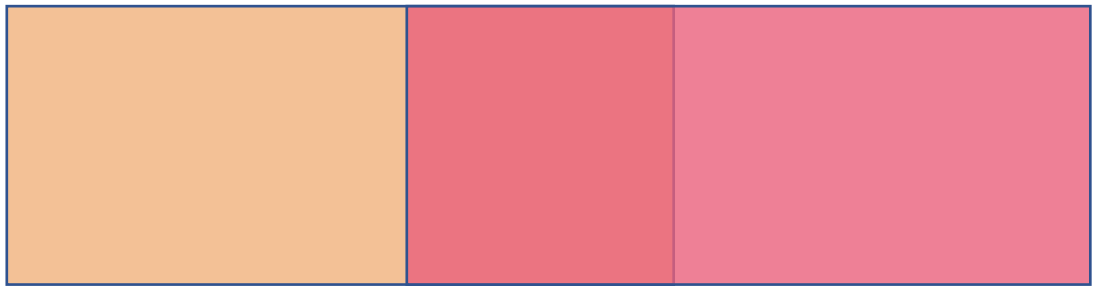






Alpha Compositing

- 与之前介绍SDF Alpha Blending不同的是，Alpha Compositing维护了一个4通道的图片（RGBA），而Alpha Blending是在两个三通道图片基础上进行混合
- 换句话说，也就是每张图片都带有一个对应的遮罩mask，记录了这个图象对应的Alpha值
- 那么对于两个带Alpha通道的图像进行叠加，我们可以想象是两块彩色的半透明玻璃，透明度分别为a1,a2，那么光线通过率分别为1-a1和1-a2



rgba(235, 152, 80, 0.6)　　rgba(234, 97, 124, 0.8)

Alpha Compositing

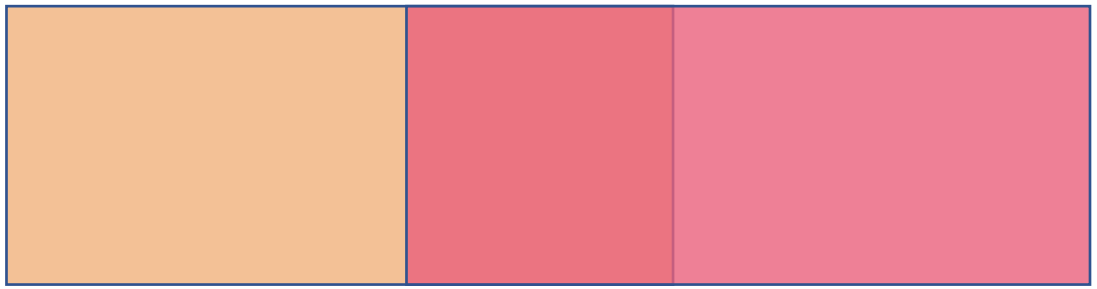
- 那么可以认为，通过第一块玻璃后，光线还剩下1-a1，再通过第二块玻璃后，还剩下(1-a1)(1-a2)，所以两块玻璃混合后最终的透明度

  $alpha = 1 - (1 - a_1)(1 - a_2) = a_1 + a_2 - a_1 a_2$

- 再考虑最终色彩，色彩的结果应该是原始色*Alpha，即

  $Color_{final} = Color_{origin} * alpha$

- 两色彩叠加的最终色彩 $C_{final} = C_1 a_1 (1 - a_2) + C_2 * a_2 = C_1 a_1 + C_2 a_2 - C_1 a_1 a_2$
- 那么原始颜色应该是 $Color_{origin} = \frac{C_{final}}{alpha} = \frac{C_1 a_1 + C_2 a_2 - C_1 a_1 a_2}{a_1 + a_2 - a_1 a_2}$



rgba(235, 152, 80, 0.6)

rgba(234, 97, 124, 0.8)

rgba(234, 104, 118, 0.92)

Performance Optimization

- 前面讲的很多算法都有一定程度上的效率问题，加上本身EasyX的效率极低，所以这一板块提出一些高阶的性能优化方法
- 底层像素操作
- 离屏渲染
- 向量优化

Buffer Operation

- 底层像素操作，我们可以使用下列方式将绘图板的底层像素提取出来，然后逐像素按自己的希望进行赋值

```
// 获取指向显示缓冲区的指针
DWORD* buffer = GetImageBuffer();

// 赋值方法1
for (int y = 0; y < src_height; y++) {
  for (int x = 0; x < src_width; x++) {
    buffer[y * src_width + x] = BGR(RGB(r, g,
b));
}

// 赋值方法2
BYTE* buffer_byte = (BYTE*)buffer;
for (int y = 0; y < src_height; y++) {
  for (int x = 0; x < src_width; x++) {
    const int index = y * src_width * 4 + x * 4;
    buffer_byte[index + 2] = r;
    buffer_byte[index + 1] = g;
    buffer_byte[index] = b;
  }
}

// 使显示缓冲区生效
FlushBatchDraw();
```

Off-screen/Offline Rendering

- 对于复杂度较高的渲染计算，我们可以执行离屏渲染，即在程序启动的时候进行预处理
- 如高斯模糊这类比较耗时的操作，可以预处理好储存在IMAGE对象中然后使用时进行放置
- 另外重复的图元也可以进行预处理

```
IMAGE* particle = new IMAGE(200, 200);
// 对particle_buffer进行处理
// 方法1：操作底层像素
BYTE* particle_buffer = (BYTE*)GetImageBuffer(particle);
// do something...

// 方法2：替换工作设备
SetWorkingImage(particle);
// do something...
SetWorkingImage(NULL);

// 将图片渲染至屏幕
// 方法1 函数
putimage(xx, yy, particle);
// 方法2 逐像素复制
for (int sy = yy, y = 0; sy < yy + 200; sy++, y++) {
  for (int sx = xx, x = 0; sx < xx + 200; sx++, x++) {
    buffer[sy * src_height + sx] = particle_buffer[y * 200 +
x]}
}
```

Vectorization optimization

- 现代处理器有极强的浮点运算性能，以及拥有强大的AVX向量化计算指令，编译器在调整至release进行编译时会对代码进行重新调整和优化以应用向量优化
- 在代码中尽可能书写循环和简单的语句，尽可能避免复杂的计算，最好每次计算只涉及一次乘法和加法
- 可以自己维护一个float变量类型的buffer，然后每次渲染时将其逐元素赋值到buffer中，这不仅会比计算整数类型精度高，而且直接保存float类型避免了每次都要转换为整数和移位