光栅图形学初步

冯结青

浙江大学 CAD&CG国家重点实验室

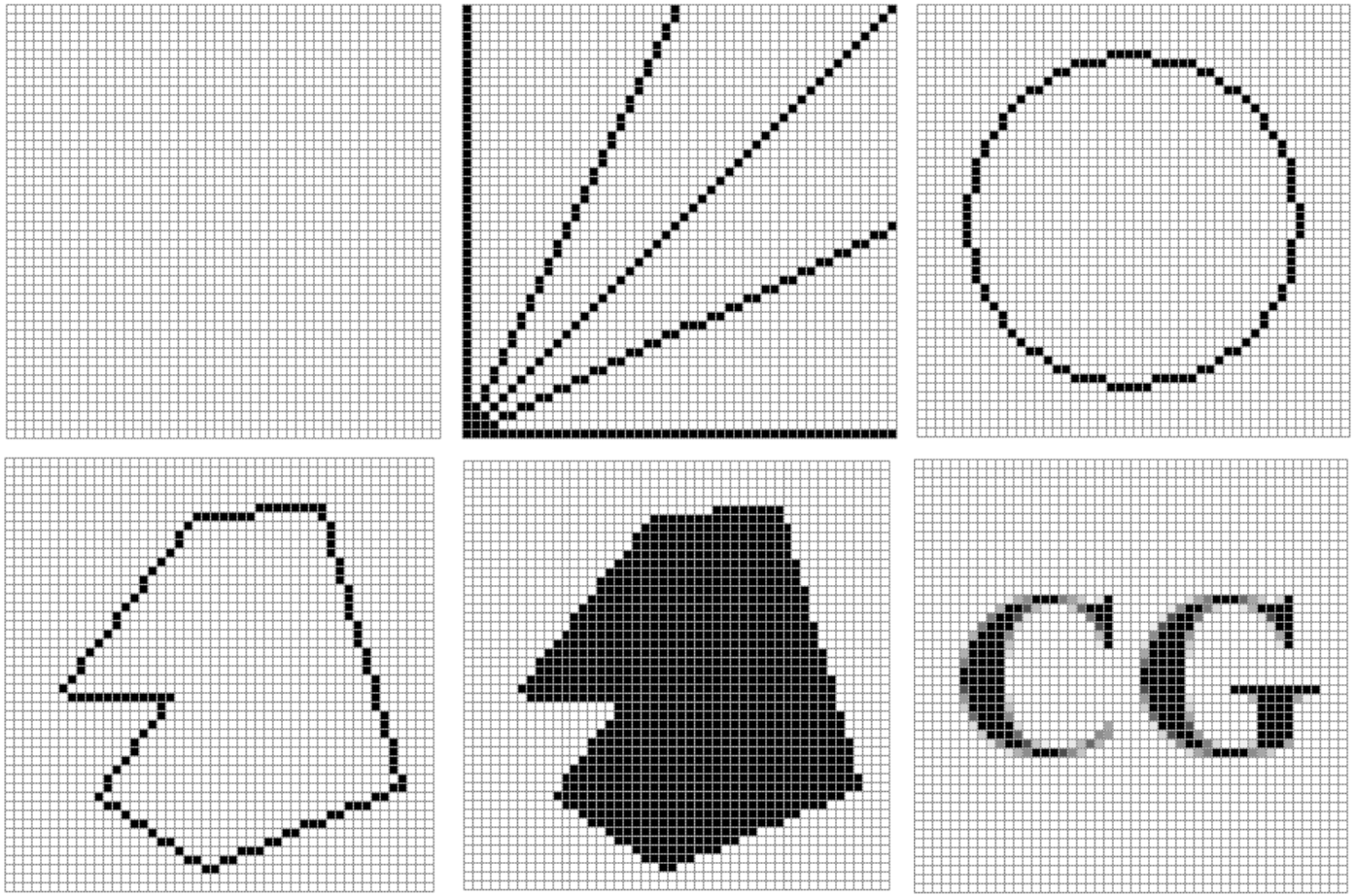
内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法

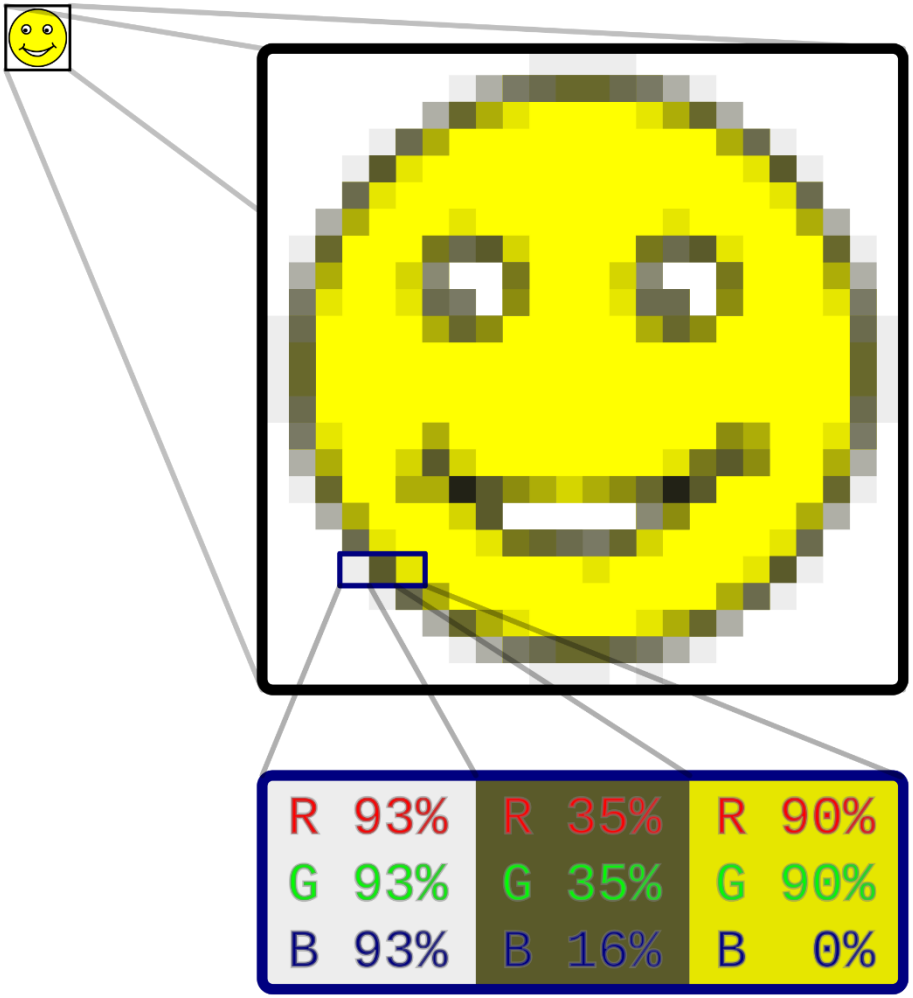
内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法

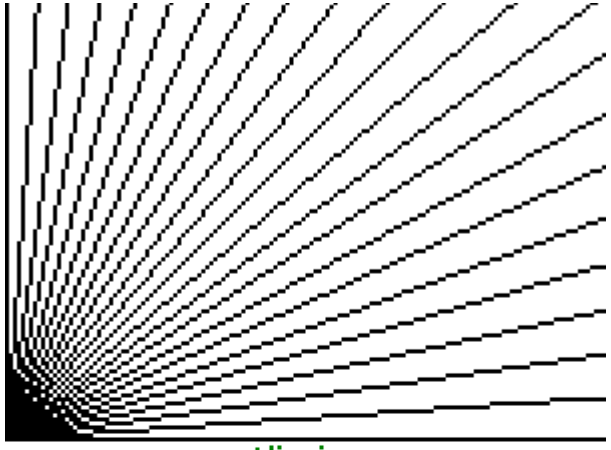
光栅与光栅图形举例

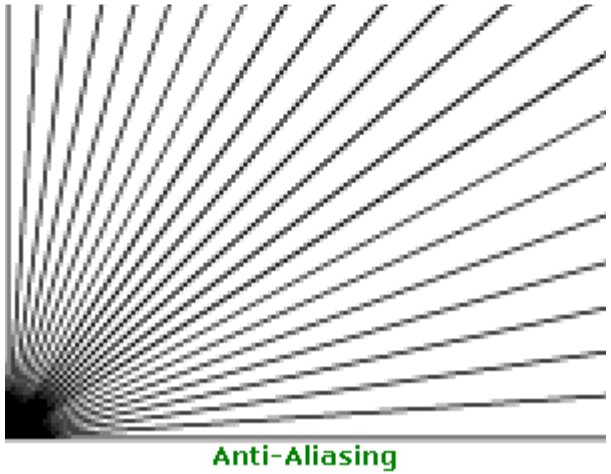


光栅图形放大

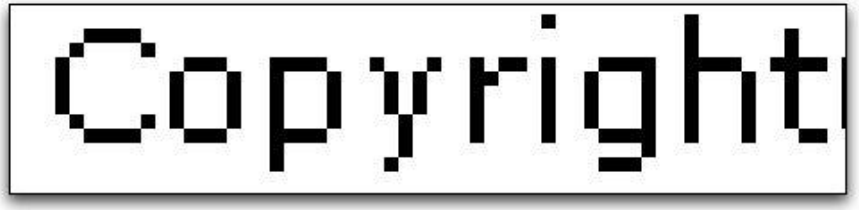


光栅走样与反走样



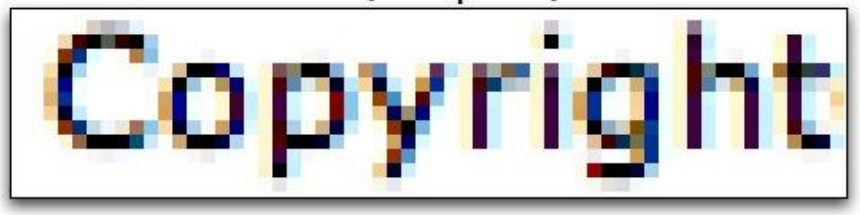


Aliased Text



Anti-aliased text (Whole pixel)

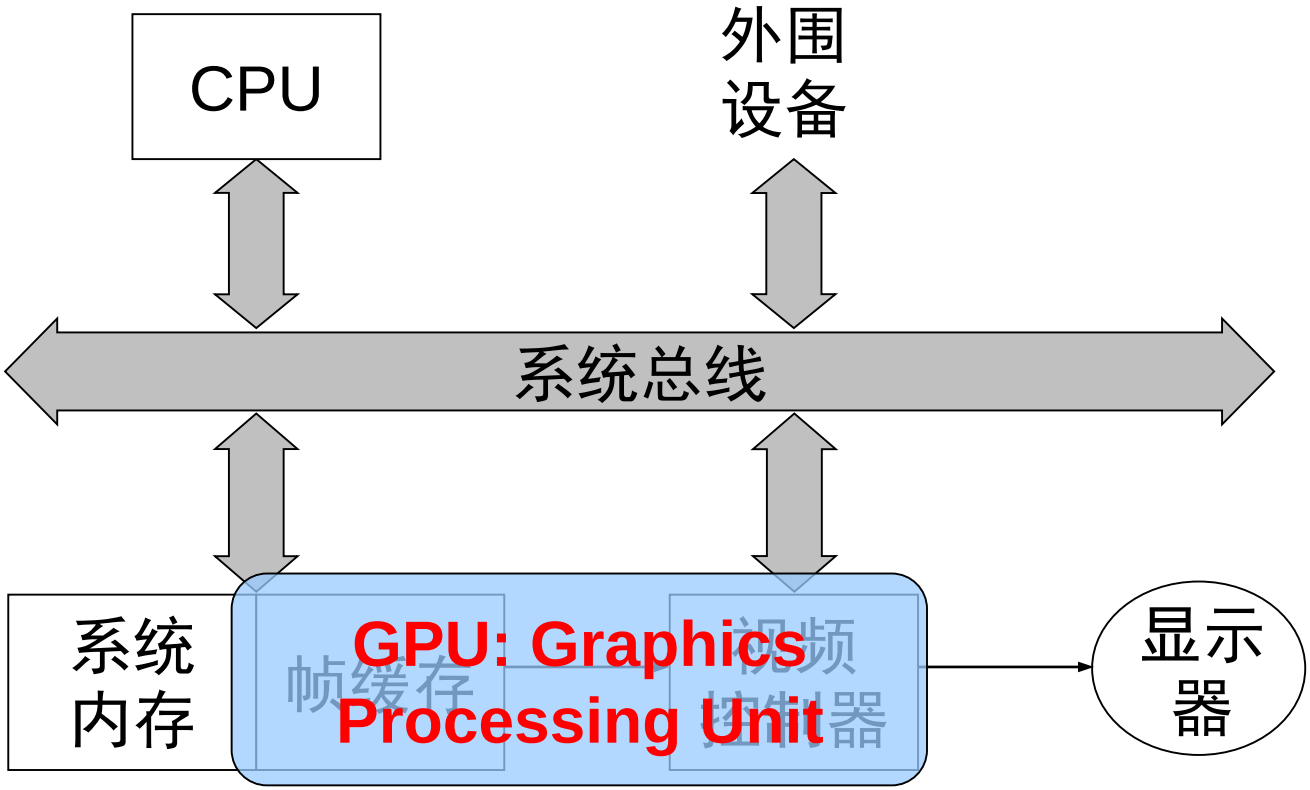
Anti-aliased text (sub pixel)



内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法

普通光栅显示的基本结构



普通光栅显示系统的结构

普通光栅显示的基本结构

- 传统扫描转换过程由软件实现
  - 当应用程序调用图形软件包中的子程序时，软件包能设置帧缓存中适当的像素
- 用于帧缓存的内存
  - 共享：内存的一部分被用来充当帧缓存
  - 独立：图形处理单元上的显存
- 视频控制器通过一个独立的访问端口访问内存，显示帧缓存中定义的图像(早期/主板显卡)
  - 固定的一部分内存永久地分配给帧缓存
  - 通过寄存器指定任意一部分内存作为帧缓存

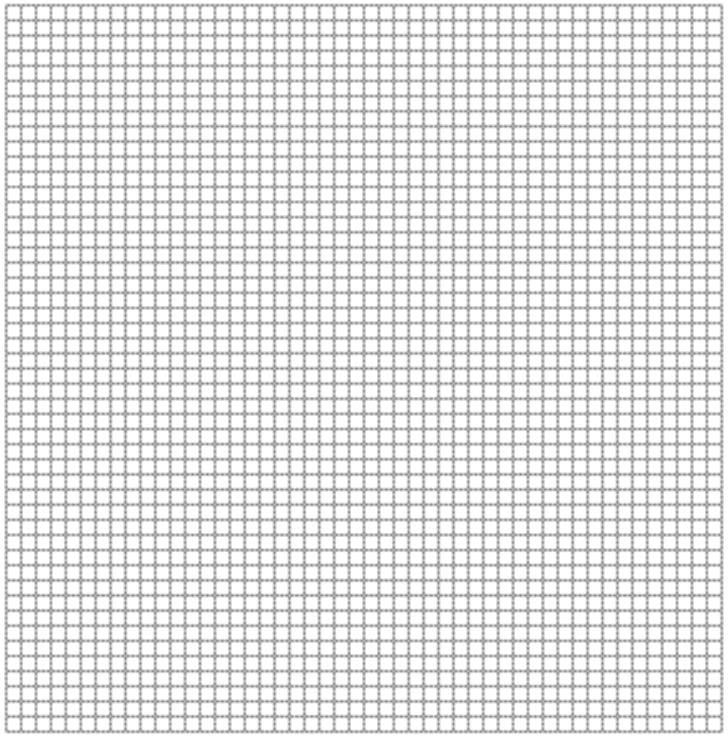
复杂光栅显示的基本结构

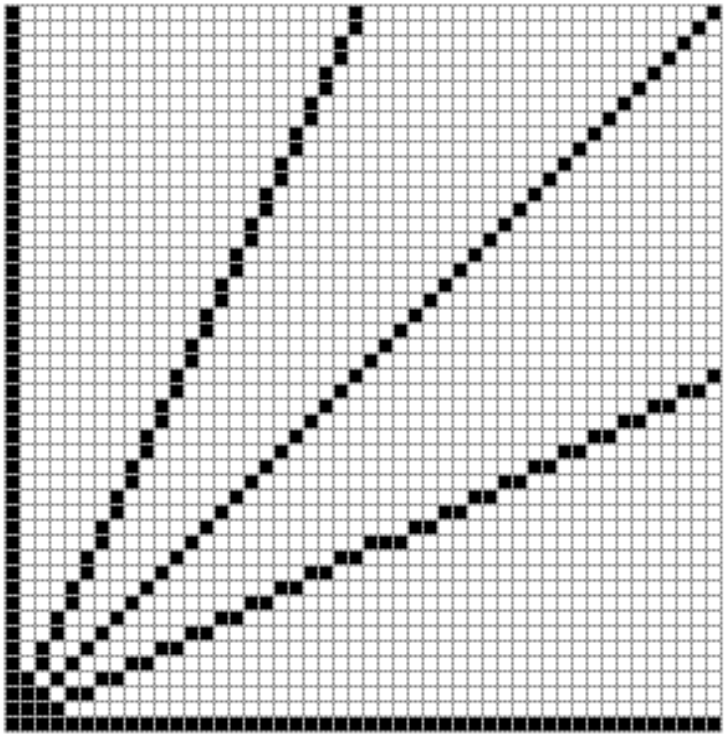
- 图形处理器(GPU，Graphics Processing Unit)
  - 执行扫描转换、将输出图元写至帧缓存、移动、拷贝、修改像素或者像素块等光栅操作
  - 集成图形卡：与CPU、视频控制器同时连接在系统总线上，共享系统内存与帧缓存
    - 单地址空间（single-address-space）显示系统体系结构
    - 允许CPU和显示处理器用统一的方式访问内存的任何部分，且编程简单
  - 独立图形卡：拥有自己的内存和帧缓存
    - 功能强大，适合于专业用户
    - 现代GPU：可编程图形处理器

内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法

光栅与光栅上的直线





内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
  - 直线生成的要求
  - 直线生成的DDA算法
  - 直线生成的Bresenham算法
- 圆的光栅化算法

直线生成的要求

- 外观笔直：只有水平、垂直，其它为锯齿状
- 精确的起点和终点：难以保证直线精确通过指定位置的起点和终点，提高分辨率有助于改善结果
- 亮度沿直线保持不变，且与直线的长度和方向无关：只有水平、垂直、45°直线
- 直线生成速度要快
- 与端点次序无关(AB=BA)

一般难以完全满足！

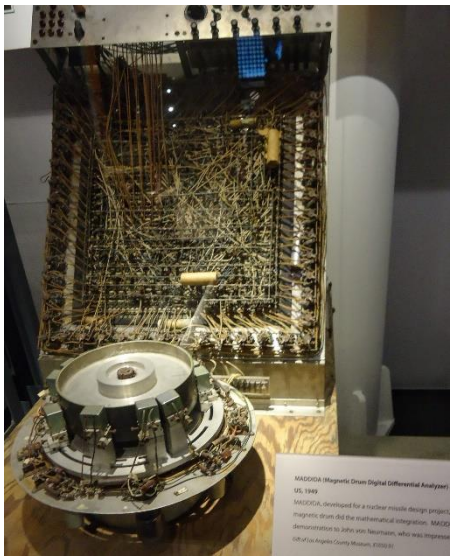
内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
  - 直线生成的要求
  - 直线生成的DDA算法
  - 直线生成的Bresenham算法
- 圆的光栅化算法

直线生成的DDA算法

- DDA (Digital Differential Analyzer)算法：

  数字微分分析器：通过同时在*x*方向和*y*方向分别增加与*dx*和*dy*成正比的小数值、来积分微分方程的“计算机”

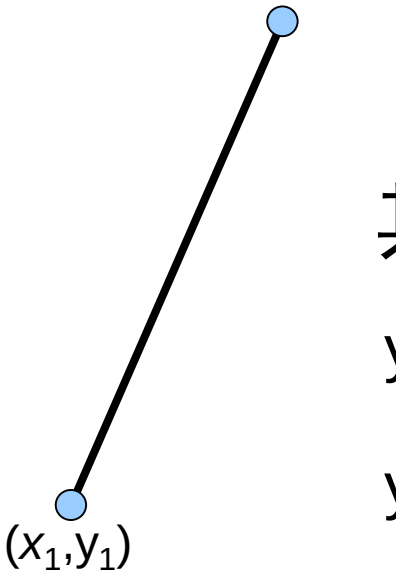


The MADDIDA (MAgnetic Drum DIgital Differential Analyzer) was a special-purpose digital computer used for solving systems of ordinary differential equations (Rleasedin1949!).

直线生成的DDA算法

直线段光栅化： 起点 (x_1,y_1)、 终点 (x_2,y_2)

原理：



$$\frac{dy}{dx}=\text{常数} \quad \text{或} \quad \frac{dy}{dx}=\frac{y_2-y_1}{x_2-x_1}$$

其解为：

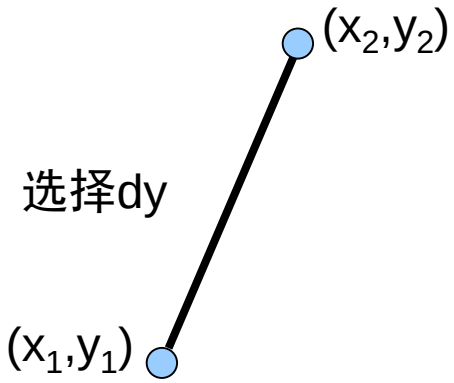
$$y_{i+1}=y_i+\Delta y \qquad x_{i+1}=x_i+\Delta x$$

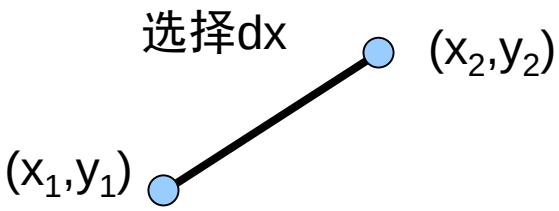
或

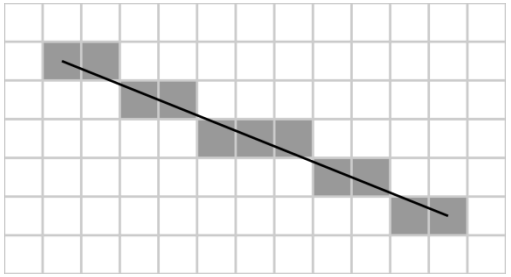
$$y_{i+1}=y_i+\frac{y_2-y_1}{x_2-x_1}\Delta x \qquad x_{i+1}=x_i+\frac{x_2-x_1}{y_2-y_1}\Delta y$$

直线生成的DDA算法

- 上述公式用于直线光栅化时的算法，称为数字微分分析器算法，即DDA算法
- 简单的DDA算法选择||dx||和||dy||中较大者为一个光栅单位







直线生成的DDA具体算法

//直线光栅化的DDA算法： (x_1,y_1)为起点， (x_2,y_2)为终点

//Integer：取整函数，即, Integer (-8.5)= -9 而不是 -8

//计算直线近似长度

if (abs(x_2-x_1) ≥ abs(y_2-y_1)) then

    Length = abs (x_2-x_1)

else

    Length = abs (y_2-y_1)

end if

//选择*dx*和*dy*较大者作为一个光栅化单位

dx = (x_2-x_1) / Length;

dy = (y_2-y_1) / Length;

直线生成的DDA算法

//将Integer函数改为四舍五入，而不是取整函数

$x = x_1 + 0.5;\quad y = y_1 + 0.5;$

//主循环

$i = 1;$

while($i \le$ Length)

    setpixel(Integer(x), Integer(y));

    $x = x + dx;$

    $y = y + dy;$

    $i = i + 1;$

end while

直线生成的DDA算法实例1

- 考虑从(0,0)到(5,5)的直线段
  - 初值计算：

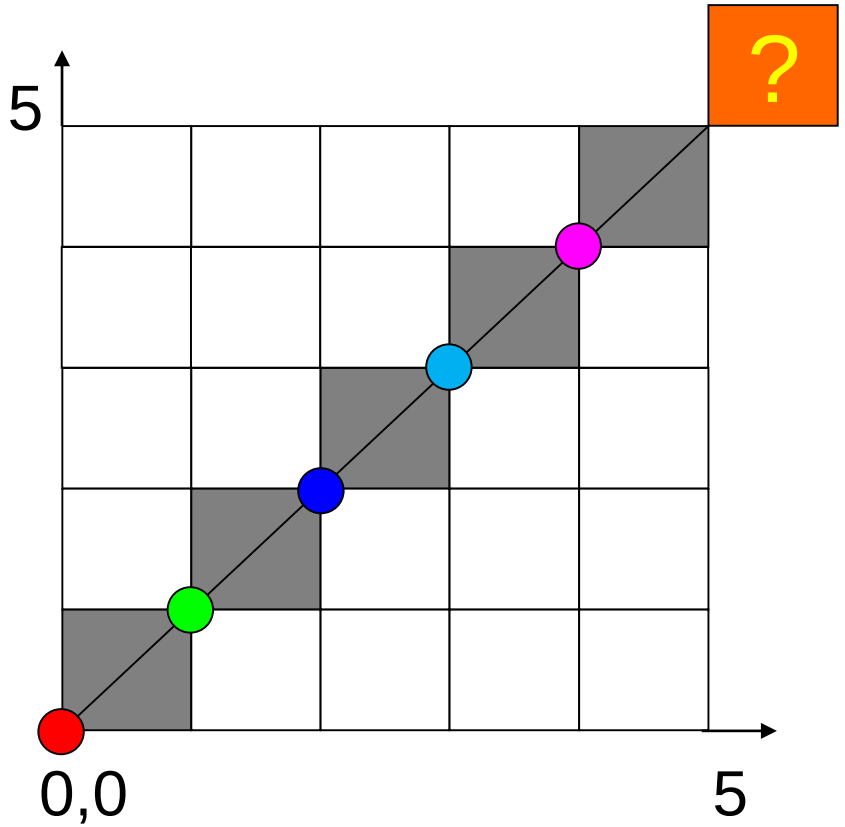
    $x_1 = y_1 = 0$； $x_2 = y_2 = 5$； Length = 5；

    $dx = dy = 1$； $x = y = 0.5$；

  - 运行结果

直线生成的DDA算法实例1

I	setpixel	x	y
		0.5	0.5
1	(0,0)		
		1.5	1.5
2	(1,1)		
		2.5	2.5
3	(2,2)		
		3.5	3.5
4	(3,3)		
		4.5	4.5
5	(4,4)		
		5.5	5.5



算法分析

- 没有精确过端点：端点(0,0)精确，但端点(5,5)没有显示，使得线段看起来稍短。
- 如果 *i* 的初值不是1而是0，那么位于(5,5)的像素会被选中：两个端点均被绘制
- *dx* 或 *dy* 的数值误差在累加过程中会导致终点的偏离：浮点误差累积

直线生成的DDA算法实例2

- 考虑从(0,0)到(-8,-4)的直线段
  - 初值计算：

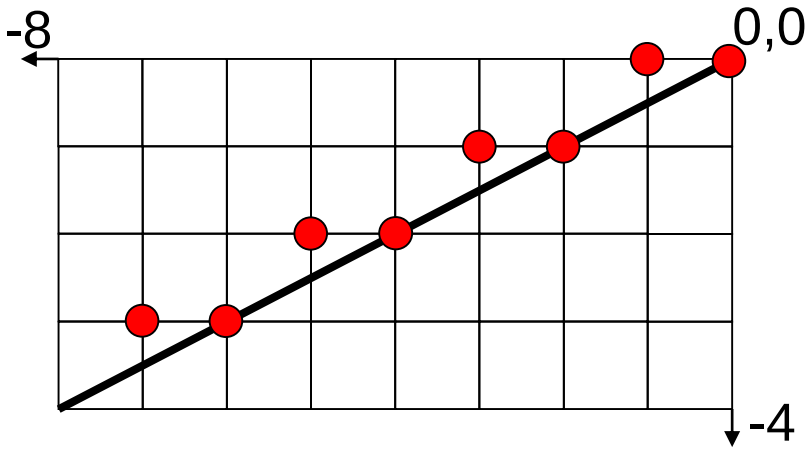
    $x_1 = 0$；$y_1 = 0$；$x_2 = -8$；$y_2 = -4$；

    Length = 8；

    $dx = -1$；$dy = -0.5$；$x = y = 0.5$；
  - 运行结果

直线生成的DDA算法实例2

i	*setpixel*	*x*	*y*
		0.5	0.5
1	(0,0)		
		–0.5	0
2	(–1,0)		
		–1.5	–0.5
3	(–2, –1)		
		–2.5	–1.0
4	(–3, –1)		
		–3.5	–1.5
5	(–4, –2)		
		–4.5	–2.0
6	(–5, –2)		
		–5.5	–2.5
7	(–6, –3)		
		–6.5	–3.0
8	(–7, –3)		
		–7.5	–3.5



算法分析

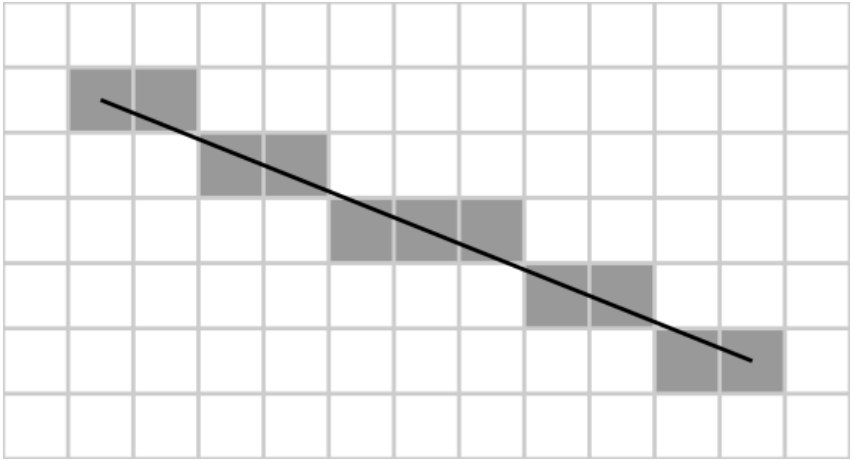
- 采用DDA算法画(0,0)到(-8,-4) 之间的线段，那么光栅化直线位于实际线段的一侧
- 如用四舍五入取整代替算法中取整函数，算法如何修改？**(课后练习)**
- **时间效率**：采用浮点运算，计算量大
- **精确性**：牺牲一定的位置精确性

内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
  - 直线生成的要求
  - 直线生成的DDA算法
  - 直线生成的Bresenham算法
- 圆的光栅化算法

DDA算法的复杂度

- 逐点计算：$(x_i,y_i)\rightarrow (x_{i+1},y_{i+1})$
- 浮点计算：计算量大
- 误差累积：偏差

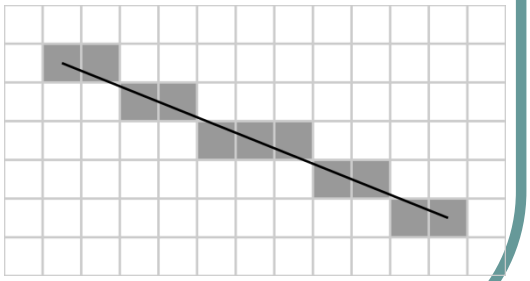


直线生成的Bresenham算法

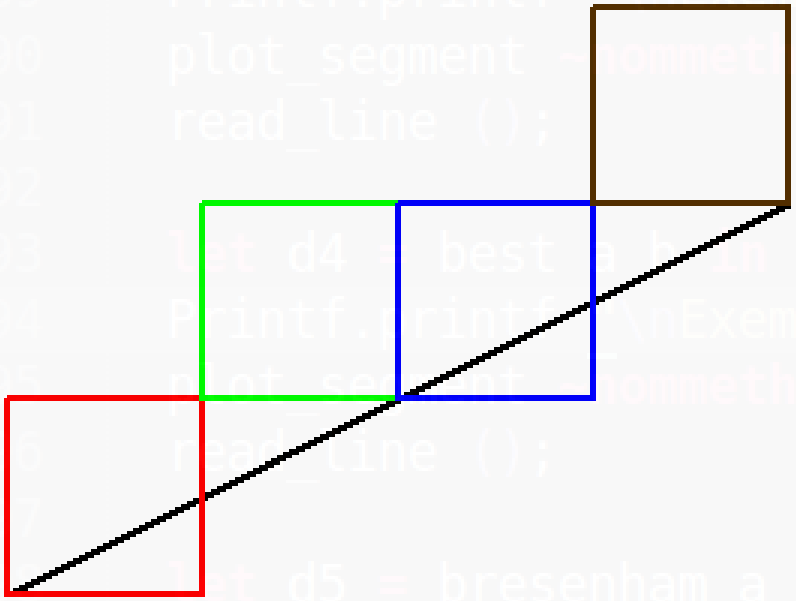
- Bresenham于1962年提出用于数字绘图仪的直线生成算法

  Bresenham, J. E. . "Algorithm for computer control of a digital plotter". *IBM Systems Journal,* 1965.1.1, **4**(1): 25–30

- 算法的核心思想：
  - 根据直线的斜率确定或者选择变量在x或y方向上每次递增(/递减)一个单位
  - 另一方向的增量为0或±1，它取决于实际直线与最近网格点位置的距离 (误差)
  - 每一步只需检查误差项的符号即可



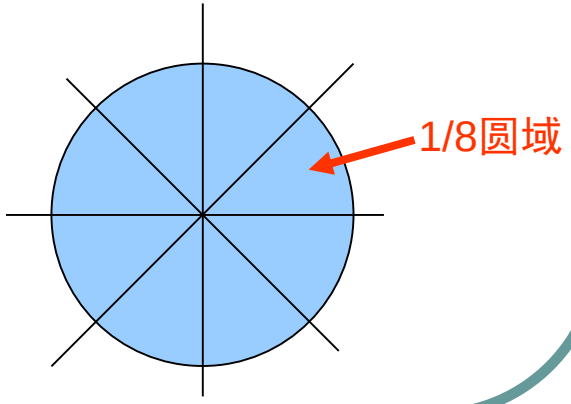
直线生成的Bresenham算法



https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

直线生成的Bresenham算法

- 算法推导(考虑1/8圆域内，即斜率$k \le 1$)
  - 直线递推公式：$y_{i+1} = y_i + m\,(x_{i+1}-x_i)$，　$m = dy / dx$
  - DDA算法的下一个像素：$x_{i+1} = x_i + 1$，$y_{i+1} = y_i + m$
  - 显示(x_i,y_i) → (x_i,y_{ir})，$y_{ir} = \text{round}(y_i)$ (见下页图)
  - 要推导y_{ir}与$y_{i+1,r}$的关系：B = (x_{i+1},y_{i+1})
    - B = (x_{i+1},y_{i+1})　A = $(x_{i+1},y_{ir}+1/2)$
    - 选择 D，　if B “>” A　(见下页图)
    - 选择 C，　if B “<” A　(见下页图)

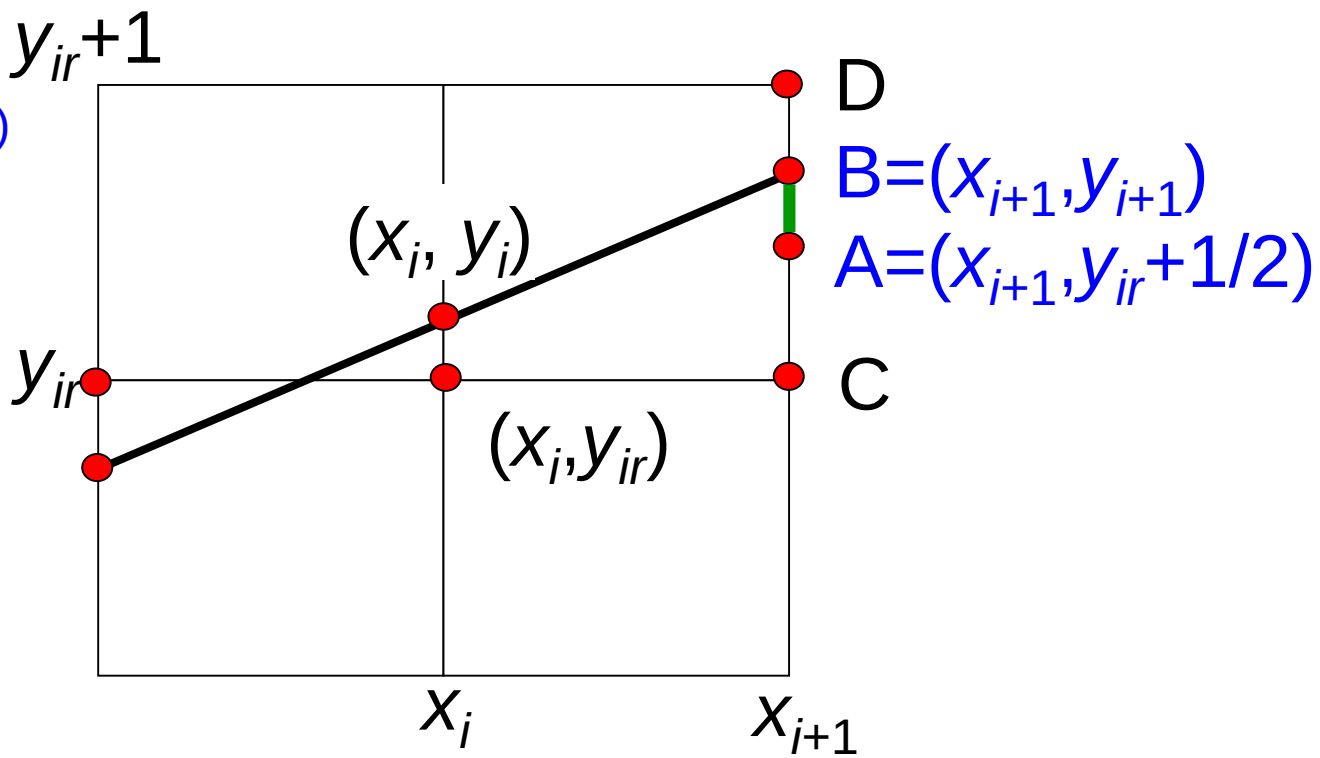


直线生成的Bresenham算法

$y_{i+1}=y_i+m(x_{i+1}-x_i)$
$m=dy/dx$

$x_{i+1}=x_i+1$
$y_{i+1}=y_i+m$

$(x_i,y_i) \rightarrow (x_i,y_{ir})$
$y_{ir}=\text{round}(y_i)$



直线生成的Bresenham算法示意图

直线生成的Bresenham算法推导

- 算法推导(考虑1/8圆域内，即斜率$k \le 1$)

  记 $e_{i+1} = y_{i+1} - (y_{ir} + 0.5)$，则：

  $y_{i+1,r} = y_{ir} + 1 \quad \text{if } e_{i+1} \ge 0$

  $y_{i+1,r} = y_{ir} \quad \text{if } e_{i+1} < 0$

$$
\begin{aligned}
e_{i+2} &= y_{i+2} - y_{i+1,r} - 0.5 \\
&= y_{i+1} + m - y_{i+1,r} - 0.5 \\
&= \begin{cases} y_{i+1} + m - y_{i,r} - 1 - 0.5 & \text{if } e_{i+1} \ge 0 \\ y_{i+1} + m - y_{i,r} - 0.5 & \text{if } e_{i+1} < 0 \end{cases} \\
&= \begin{cases} e_{i+1} + m - 1 & \text{if } e_{i+1} \ge 0 \\ e_{i+1} + m & \text{if } e_{i+1} < 0 \end{cases}
\end{aligned}
$$

直线生成的Bresenham算法描述

//1/8圆域中Bresenham算法
//$(x_1,y_1)\rightarrow(x_2,y_2)$，整数点
//变量初始化

$x=x_1$； $y=y_1$;

$dx=(x_2-x_1)$； $dy=(y_2-y_1)$;

$m=dy/dx$;

//误差补偿

$e=m-1/2$;

$y_{i+1}=y_i+m(x_{i+1}-x_i)$
$e_{i+1}=y_{i+1}-y_{ir}-0.5$

//主循环

```
for i=1 to dx
        setpixel(x,y);
        while(e>0)
                y=y+1;
                e=e-1;
        end while
        x=x+1;
        e=e+m;
        next i;
Finish
```

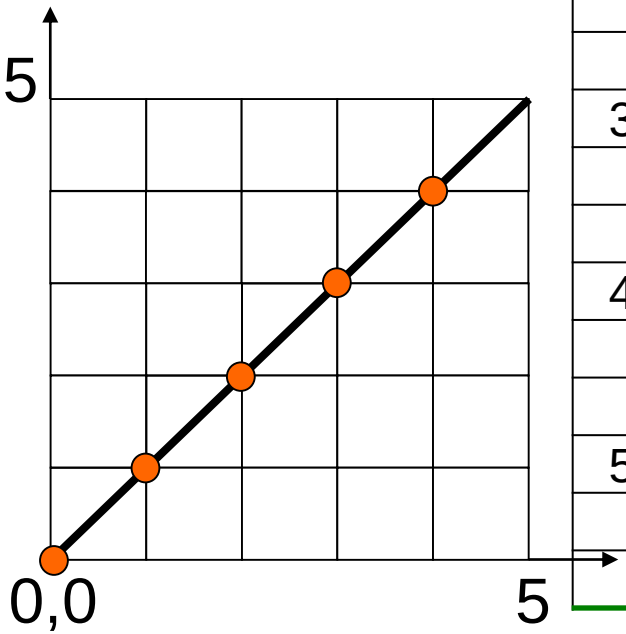
直线生成的Bresenham算法实例

```
e=m-1/2
for i=1 to dx
    setpixel(x,y);
    while(e>0)
        y=y+1;
        e=e-1;
    end while
    x=x+1;
    e=e+m;
next i;
Finish
```

- 从(0,0)到(5,5)线段
  - 初值计算

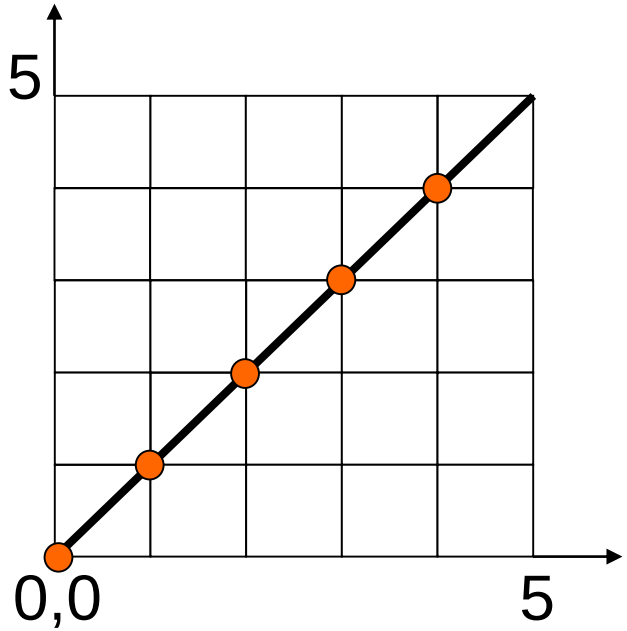
$x_1=y_1=0$; $dx=dy=5$;

$m=1$; $e=1-1/2=1/2$;



i	setpixel	*e*	*x*	*y*
		1/2	0	0
1	(0,0)			
		–1/2	0	1
		1/2	1	1
2	(1,1)			
		–1/2	1	2
		1/2	2	2
3	(2,2)			
		–1/2	2	3
		1/2	3	3
4	(3,3)			
		–1/2	3	4
		1/2	4	4
5	(4,4)			
		–1/2	4	5
		1/2	5	5

直线生成Bresenham算法分析



- 结果和预期相同
- 位于(5,5)的光栅点未被选中。如果将for-next的循环变量改为从0到*dx*，则该光栅点将被选中 (问题？)
- 若将setpixel语句移到 next *i* 前，则可消除第一个光栅点(0,0)，而绘制(5,5)。

整数Bresenham算法

- 直线斜率和误差项计算：浮点算术运算和除法
  - 整数运算和避免除法：加速算法
  - Bresenham：绘图仪连接的处理器只有整数运算!!!
- 不关心误差项数值，只关心误差项符号：
  - 将误差项计算转化为整数计算

$$e=2edx$$

- 整数算法便于硬件实现

整数Bresenham算法

$$y_{i+1}=y_i+m(x_{i+1}-x_i)$$

$$e_{i+1}=y_{i+1}-y_{ir}-0.5$$

$$m=dy/dx$$

$$2dxe_{i+1}=2dxy_i+2dy(x_{i+1}-x_i)-2dxy_{ir}-dx$$

$$=2dxy_i+2dy-2dxy_{ir}-dx$$

...

课后练习：推导整数Bresenham公式

整数Bresenham算法描述

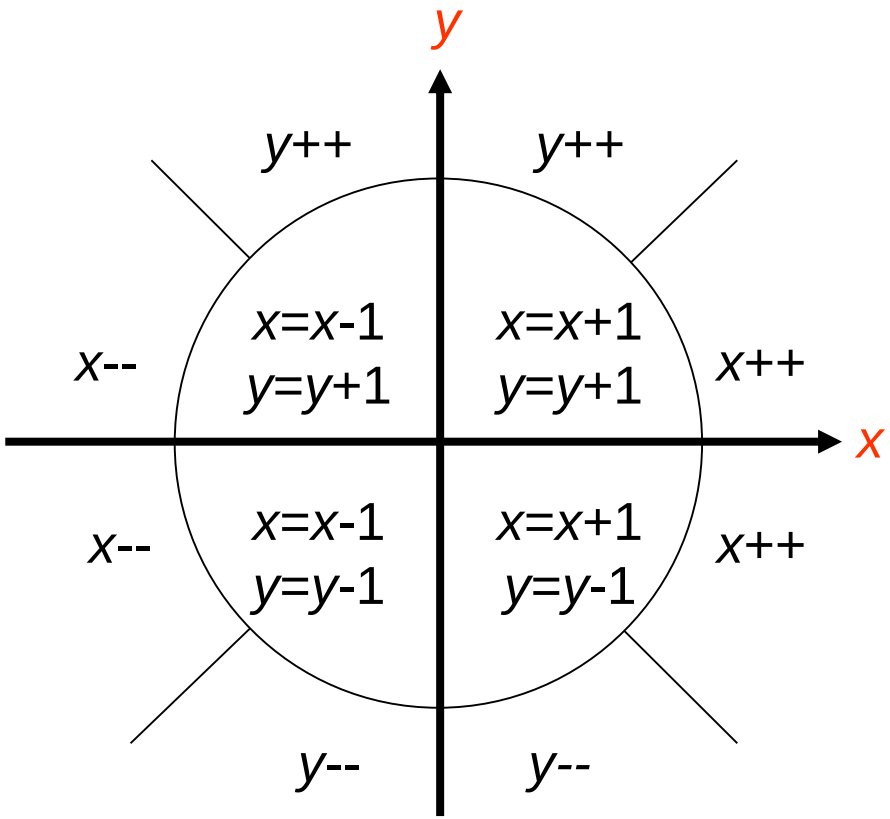
```
//1/8圆域中Bresenham算法:
//(x1,y1)→(x2,y2)，整数点
//变量初始化
x=x1;  y=y1;
dx=(x2-x1); dy=(y2-y1);

//误差补偿：将这里的e除以
//2dx即得上面非整数算法e
e=2*dy-dx;

//主循环
```

```
for  i =1  to  dx
    setpixel(x,y);
    while(e>0)
        y=y+1;
        e=e-2*dx;
    end while
    x=x+1;
    e=e+2dy;
    next i;
Finish
```

通用Bresenham算法

- 根据直线段斜率和它所在象限
  - 当直线斜率的绝对值大于1时，*y*值总是±1，再用Bresenham 误差判别式以确定*x*变量是否需要增加1。
  - *x*或*y*是增加1还是减去1取决于直线所在的象限

通用Bresenham算法



通用Bresenham算法的增量选择

通用Bresenham算法算法描述

```
//通用Bresenham算法:
// (x1,y1)→(x2,y2)，整数点
//Sign(x) = 1,0,-1 当 x>0,=0,<0
//变量初始化

x=x1;  y=y1;
dx=abs(x2-x1)；dy=abs(y2-y1)；
s1=Sign(x2-x1)；s2=Sign(y2-y1)；

//根据直线斜率的符号，交互dx和dy
if  dy > dx  then
    Temp＝dx;  dx=dy;  dy=Temp;
    Interchange=1;
else
    Interchange=0;
end if

// 误差补偿
e=2*dy-dx；
```

```
//主循环
for  i=1  to  dx
     setpixel(x,y)；
     while(e>0)
          if(Interchange==1) then
                    x=x+s1;
          else
                     y=y+s2;
          end if
          e=e-2*dx；
     end while
      if(Interchange==1) then
           y=y+s2;
     else
          x=x+s1;
     end if
          e=e+2dy；
     next i；
Finish
```

通用Bresenham算法实例

- 考虑从(0,0)到(–8, –4)画一条直线，初值计算：

  $x=y=0$;

  $dx=8$; $dy=4$;

  $s_1=s_2=-1$;

  Interchange=0;

  $e=0$

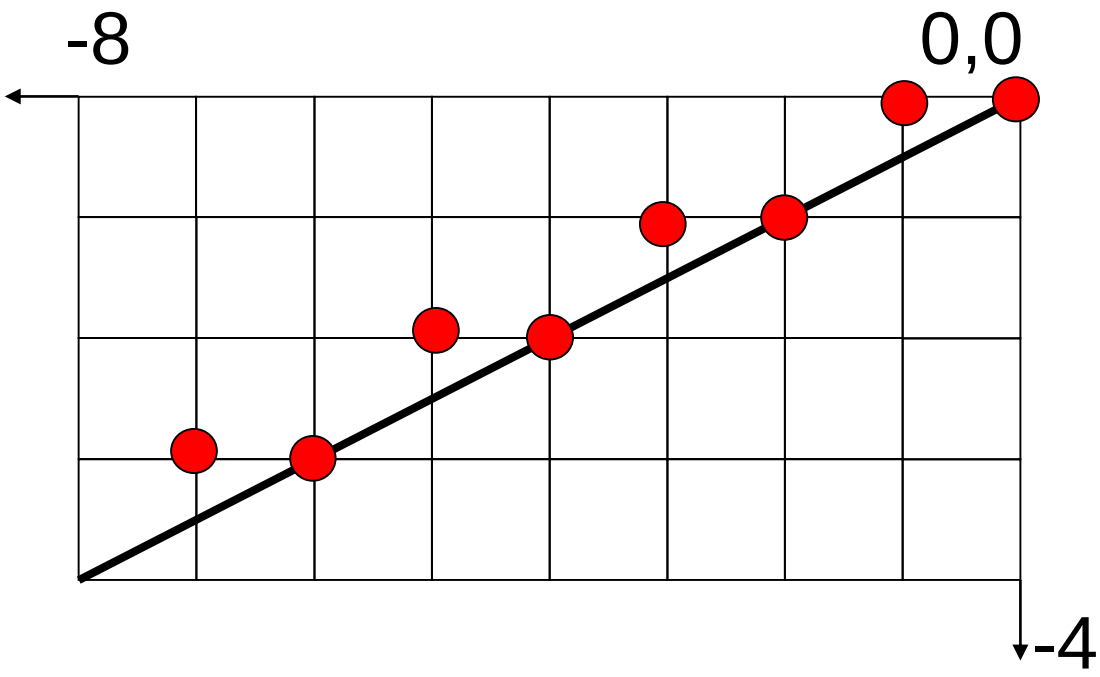
通用Bresenham算法实例

i	setpixel	*e*	*x*	*y*
		0	0	0
1	(0,0)			
		8	–1	0
2	(–1, 0)			
		–8	–1	–1
		0	–2	–1
3	(–2, –1)			
		8	–3	–1
4	(–3, –1)			
		–8	–3	–2
		0	–4	–2

i	setpixel	*e*	*x*	*y*
5	(-4,-2)			
		8	–5	-2
6	(-5,–2)			
		–8	–5	–3
		0	–6	–3
7	(-6,-3)			
		8	–7	–3
8	(-7,-3)			
		–8	–7	–4
		0	–8	–4

程序运行结果

通用Bresenham算法实例

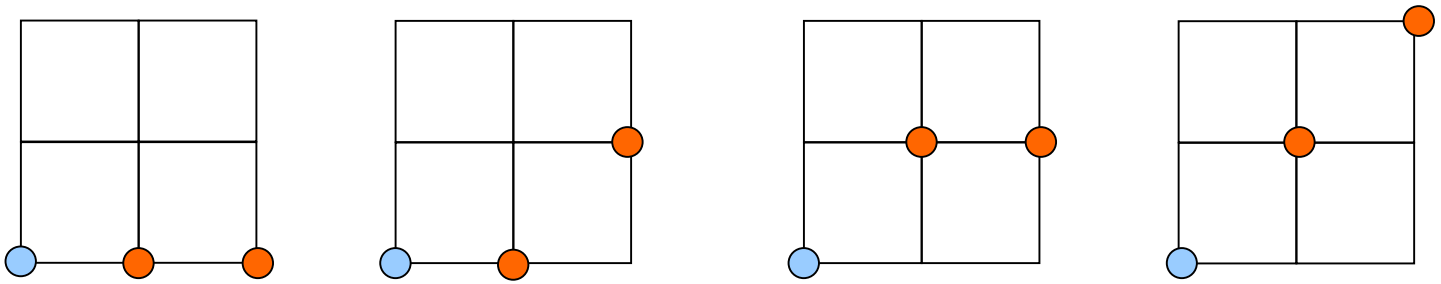


光栅化结果

Bresenham算法的改进和加速

- 对称法：从两端向中间一次生成两个点

- 两步法：沿一个方向，每判断一次就生成两个连续的点



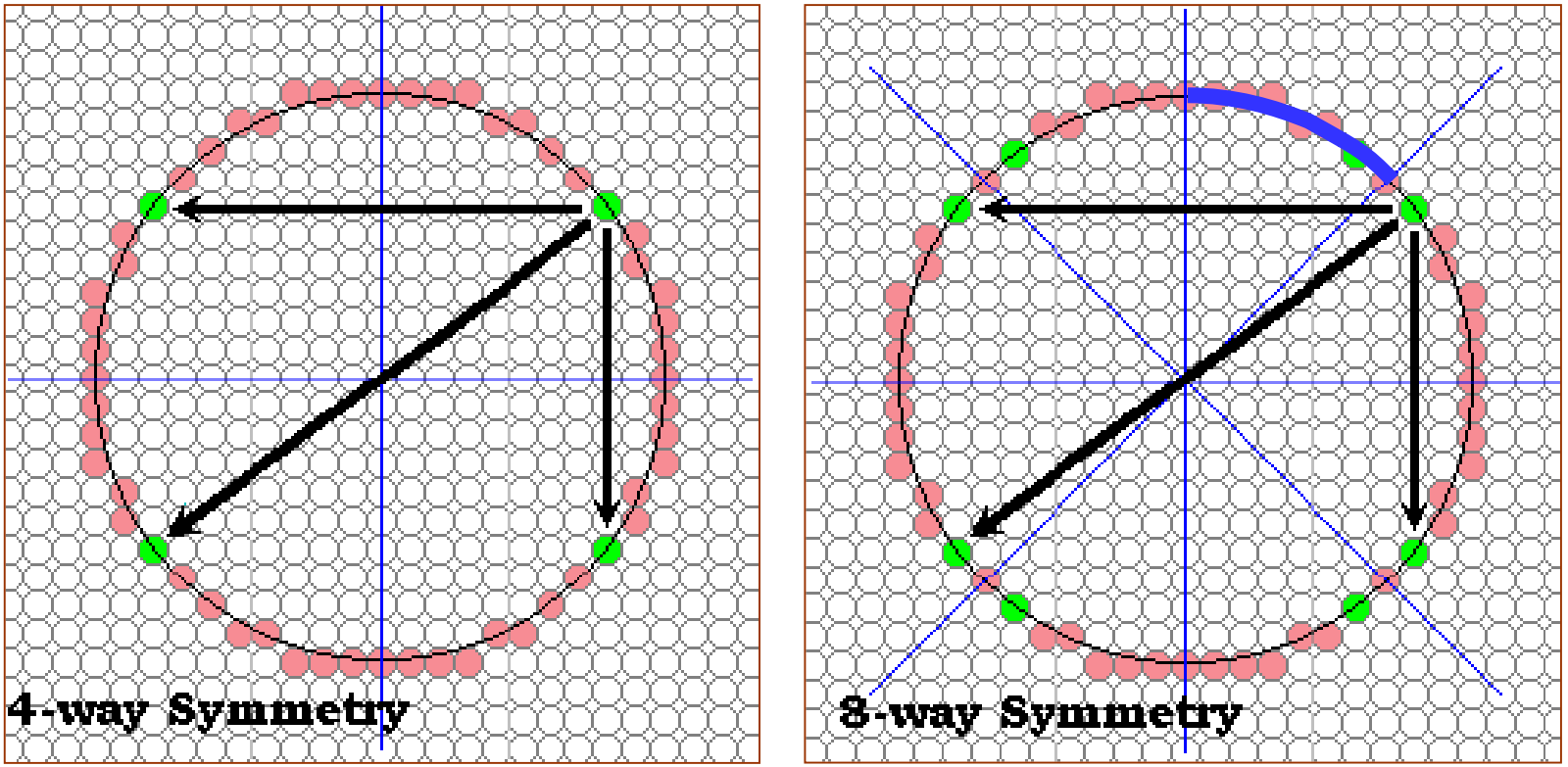
Bresenham算法的改进和加速

- 改进的两步法(对称法)：
  - 直线相对于其中点是对称的
  - 每进行一次判断，生成两个点，以及相对于直线中点对称的两个点
  - 算法速度是原Bresenham算法的四到五倍
- N步法：(N=)4步画线算法类似于两步画线算法，不同的时每判断一次，就生成四个点。
- 推广至画圆的中点法

内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法

圆的光栅化算法

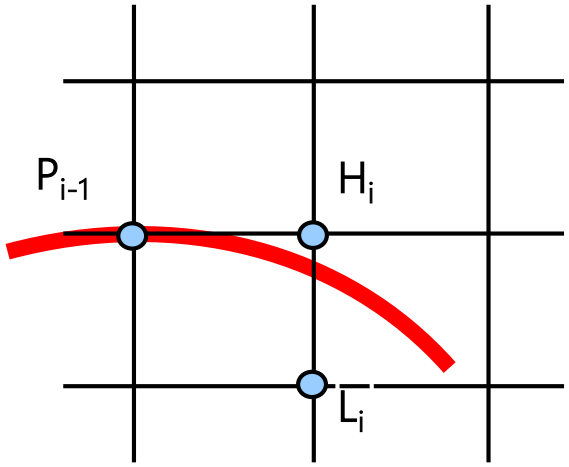


对称的圆

内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法
  - 生成圆的Bresenham算法
  - 生成圆的中点法
  - 生成圆的其它算法

生成圆的Bresenham算法



P_{i-1}是当前象素，如何确定光栅上的下一个象素是 H_i 还是 L_i？

- 判断的原则： H_i还是L_i离圆更近？

  记 P_{i-1}为(x_{i-1},y_{i-1})

  则 H_i 为(x_i,y_{i-1})、L_i 为$(x_i,y_{i-1}-1)$

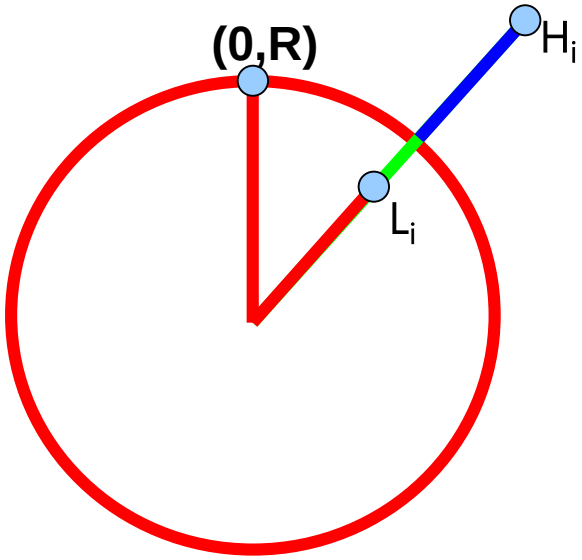
  $$x_i=x_{i-1}+1$$

- 判断函数

$$d_i = (x_i^2 + y_{i-1}^2 - R^2) + (x_i^2 + (y_{i-1}-1)^2 - R^2)$$

d_i的几何意义？

生成圆的Bresenham算法



P_{i-1}是当前象素，如何确定光栅上的下一个象素？H_i还是L_i？

d_i = 蓝色^2(+)+绿色^2(-)

= $(x_i^2 + y_{i-1}^2 - R^2)$ +

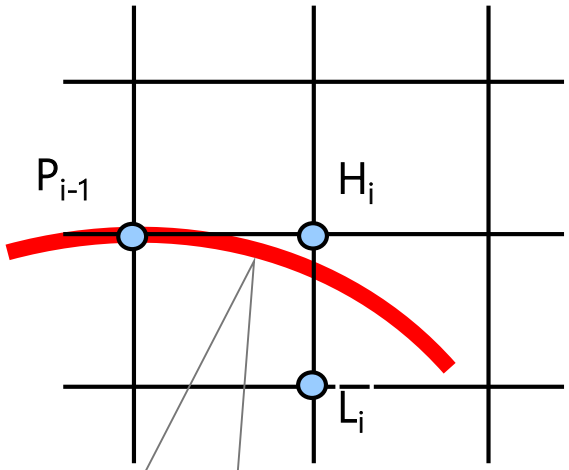
$(x_i^2 + (y_{i-1}-1)^2 - R^2)$

$d_i>0$ 取 L_i；

$d_i<0$ 取 H_i；

- d_i与d_{i+1}之间的递推关系？
- 终止条件：x<y

生成圆的Bresenham算法



P_{i-1} (x_{i-1},y_{i-1})

H_i (x_i,y_{i-1})

L_i $(x_i,y_{i-1}-1)$

$x_i=x_{i-1}+1$

- d_i与d_{i+1}之间的递推关系推导

第一个点：(0,R),两个候选点(1,R),(1,R-1)。根据前面d_i的公式

$$d_1=3-2R$$

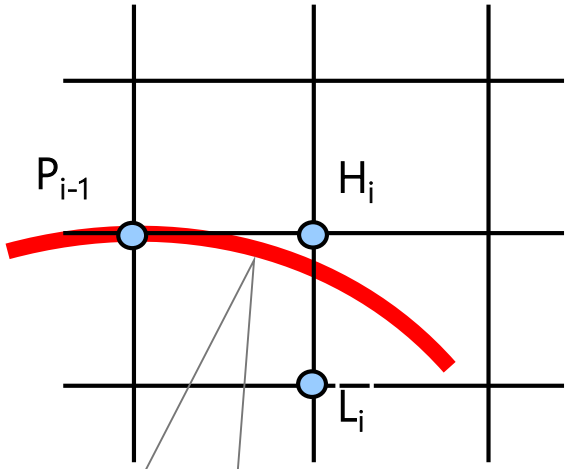
第i个点已知，可以推出

$$d_i=2x_i^2+2y_{i-1}^2-2y_{i-1}-2R^2+1$$

第i+1个点的判断函数递推公式

$$d_{i+1}= 2x_i^2 + 4x_i + 2y_i^2 - 2y_i - 2R^2 + 3$$

生成圆的Bresenham算法



P_{i-1} (x_{i-1},y_{i-1})

H_i (x_i,y_{i-1})

L_i $(x_i,y_{i-1}-1)$

$x_i=x_{i-1}+1$

由d_i的符号判断下一个点的位置，并且递推出d_{i+1}

$d_i<0$，取H_i，$y_i=y_{i-1}$，

$$d_{i+1} = d_i + 4x_{i-1} + 6$$

$d_i\geq 0$，取L_i，$y_i=y_{i-1}-1$，

$$d_{i+1} = d_i + 4(x_{i-1}-y_{i-1}) + 10$$

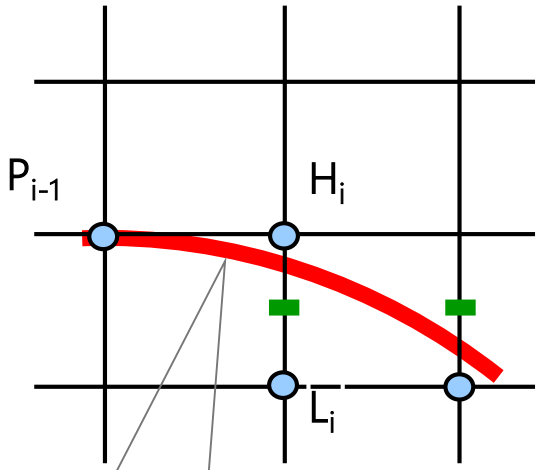
生成1/8圆弧的Bresenham算法

```
Bresenham_Arc(radius)
{
      x=0;  y=radius;  d=2-3*radius;
      While (x<y)  {
          plot(x,y);
          if (d<0)  d += 4*x+6;
          else {
            d += 4*(x-y)+10;
                y--;
          }
          x++;
          if (x==y)   plot(x,y);
      }
}
```

内容

- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法
  - 生成圆的Bresenham算法
  - 生成圆的中点法
  - 生成圆的其它算法

生成圆的中点法



P_{i-1} (x_{i-1},y_{i-1})
H_i (x_i,y_{i-1})
L_i $(x_i,y_{i-1}-1)$
$x_i=x_{i-1}+1$

对于圆 $F(x,y)=x^2+y^2-R^2$

$F(x,y)=0$ (x,y)在圆上

$F(x,y)>0$ (x,y)在圆外

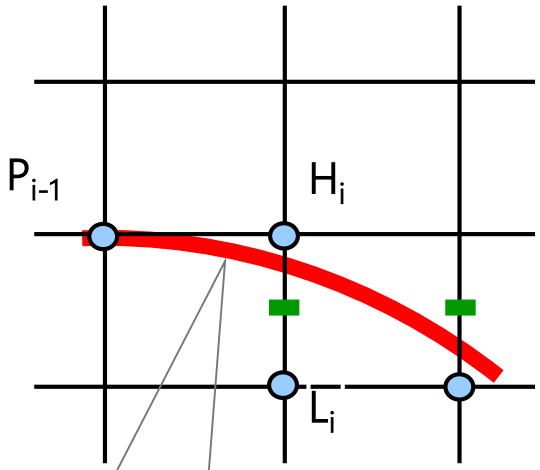
$F(x,y)<0$ (x,y)在圆内

$F(P_{i-1})=F(x_{i-1},y_{i-1})=0$，定义$d_i$：

$$d_i = F(x_i,y_{i-1}-0.5)$$

$$= F(x_{i-1}+1,y_{i-1}-0.5)$$

生成圆的中点法



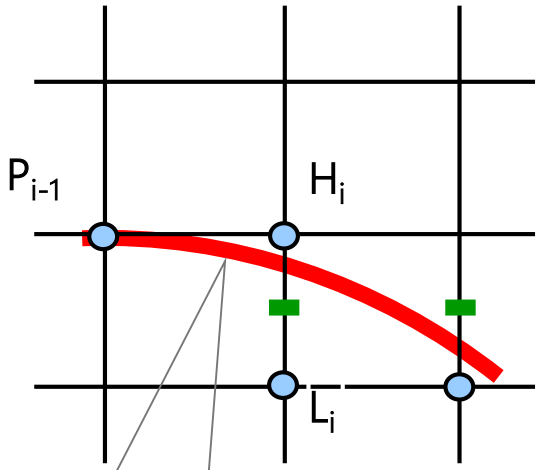
P_{i-1} (x_{i-1},y_{i-1})
H_i (x_i,y_{i-1})
L_i $(x_i,y_{i-1}-1)$
$x_i=x_{i-1}+1$

由d_i的符号判断下一个点的位置，并且递推出d_{i+1}

$d_i<0$, 取H_i；

$$d_{i+1} = F(x_{i+1},y_i-0.5)$$
$$= F(x_{i-1}+2,y_{i-1}-0.5)$$
$$= d_i+2x_{i-1}+3$$

生成圆的中点法



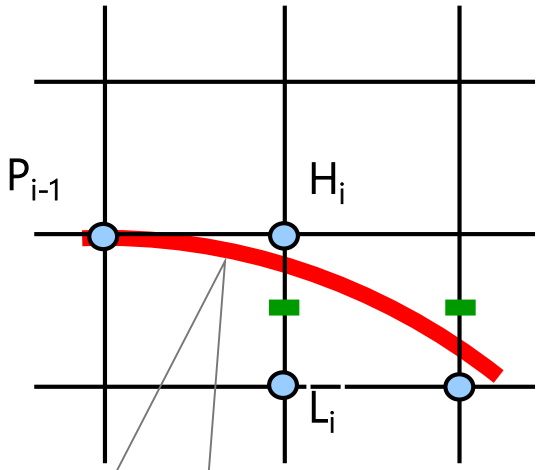
P_{i-1} (x_{i-1},y_{i-1})
H_i (x_i,y_{i-1})
L_i $(x_i,y_{i-1}-1)$
$x_i=x_{i-1}+1$

由d_i的符号判断下一个点的位置，并且递推出d_{i+1}

$d_i \geq 0$，　取L_i；

$$d_{i+1} = F(x_{i+1},y_i-0.5)$$
$$= F(x_{i-1}+2, y_{i-1}-1.5)$$
$$= d_i+2(x_{i-1}-y_{i-1}) +5$$

生成圆的中点法



P_{i-1} (x_{i-1},y_{i-1})

H_i (x_i,y_{i-1})

L_i $(x_i,y_{i-1}-1)$

$x_i=x_{i-1}+1$

第一个点 (0,R)

$d_0=F(1,R-0.5)=1.25-R$

生成圆的中点法

```
MiddlePoint_Arc(radius)
{
    x=0;  y=radius;  d=1.25-radius;
    while (x<y)  {
        if (d<0)  d += 2*x+3;
       else {
        d +=  2*(x-y)+5;
            y--;
       }
       x++; plot(x,y);
    }
}
```

内容

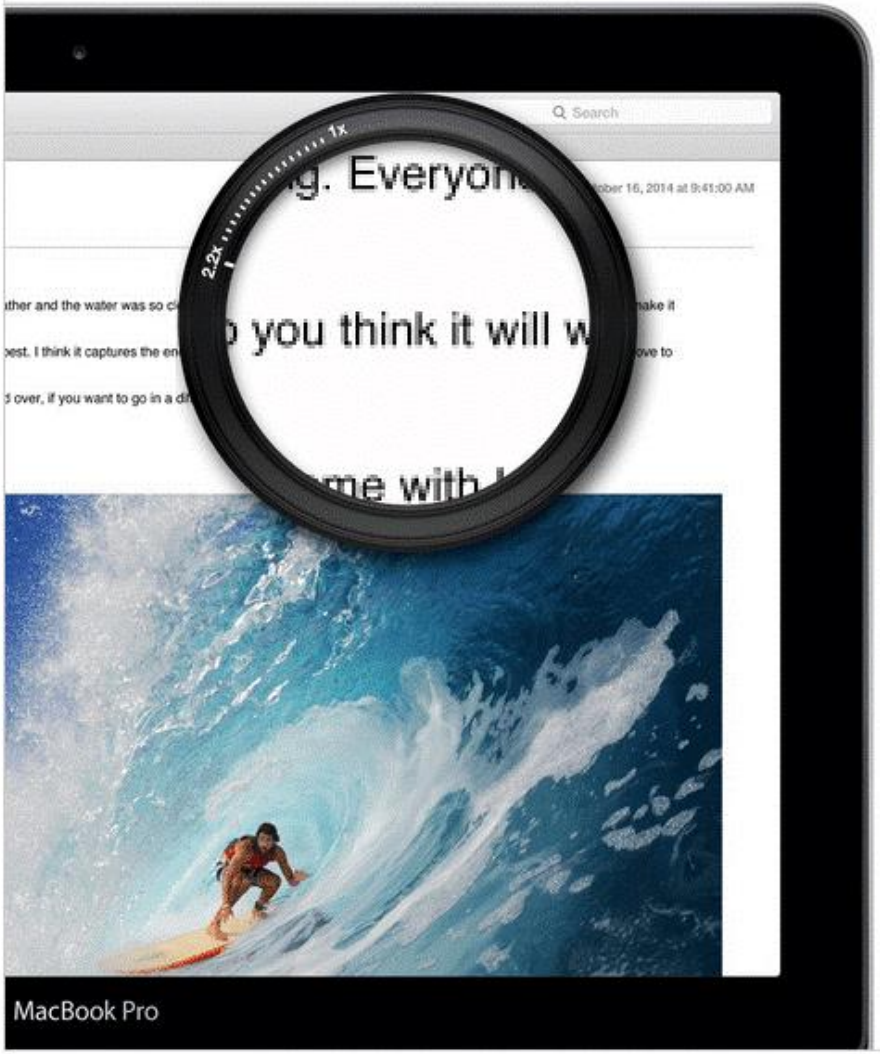
- 一些光栅图形的例子
- 光栅显示的体系结构
- 直线的光栅化算法
- 圆的光栅化算法
  - 生成圆的Bresenham算法
  - 生成圆的中点法
  - 生成圆的其它算法

生成圆的其它方法

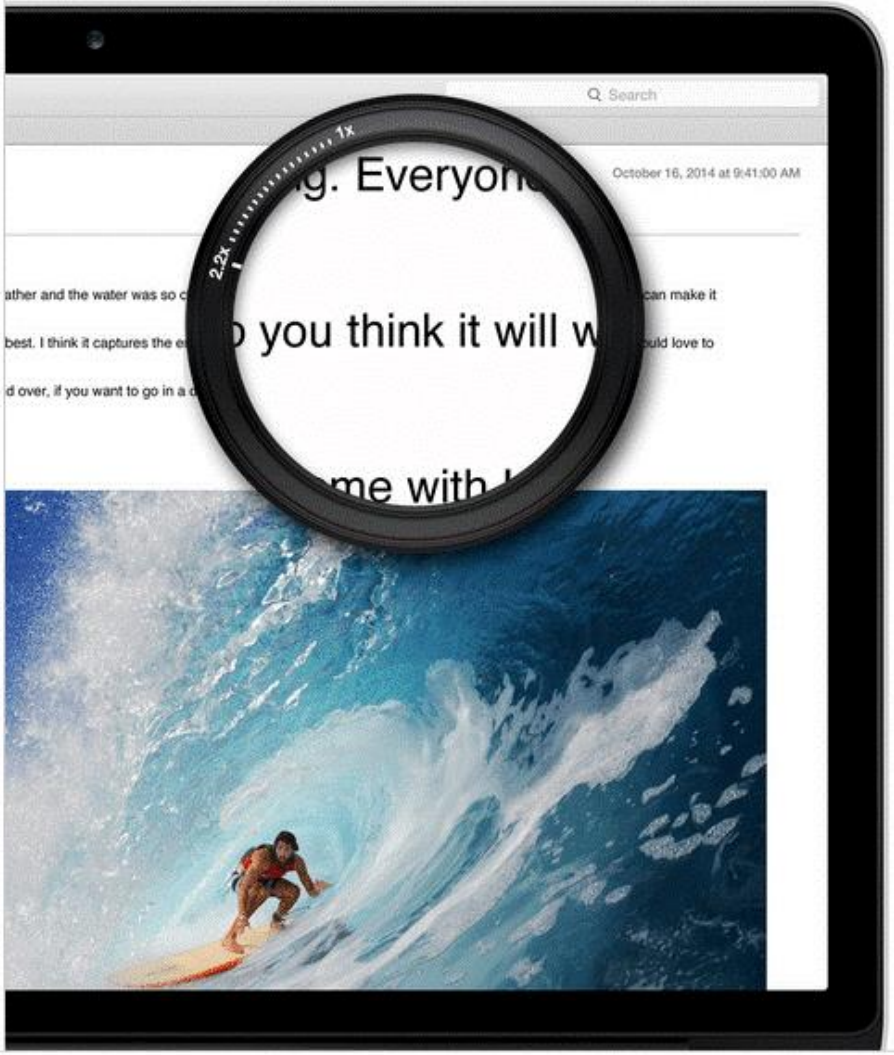
- 多边形逼近法
- 正负法
  - 蔡耀志：正负法数控绘图 (工程图学学报1984)
  - 金通洸：TN法数控绘图(Tangent-Normal)
- 圆弧的生成算法
- 任意位置的圆及圆弧生成算法
- 生成园的Bresenham算法与中点法的不同？

Retina Display: 300ppi@10~12in

13-inch MacBook Pro



13-inch MacBook Pro with Retina display



课外阅读(1)

- Malcon Sabin (1999). **Explorations in 3D integer-based linear geometry**. Technical Report Technical Report DAMTP/1999/NA05, University of Cambridge, Department of Applied Mathematics and Theoretical Physics

- 是否可以用整数定义线性几何？

课外阅读(2)

- Johannes Kopf and Dani Lischinski (2012). **Depixelizing Pixel Art**. ACM Transactions on Graphics (SIGGRAPH 2011), 2011, 30(4):99:1 -- 99:8
- 光栅化的逆过程？

Nearest-neighbor result
(Original: 40 x 16 pixels)

Depixelizing Pixel Art result

https://johanneskopf.de/publications/pixelart/

课后阅读

- 石教英、彭群生等译，《计算机图形学的算法基础》，机械工业出版社，2002.1. pp 47-70