Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT205 Computer Graphics

Geometric Primitives

Lecture 03
2024-25

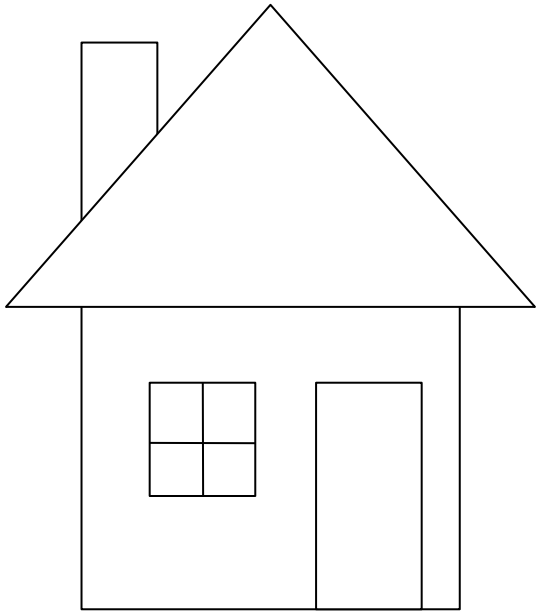
Yong Yue and Nan Xiang

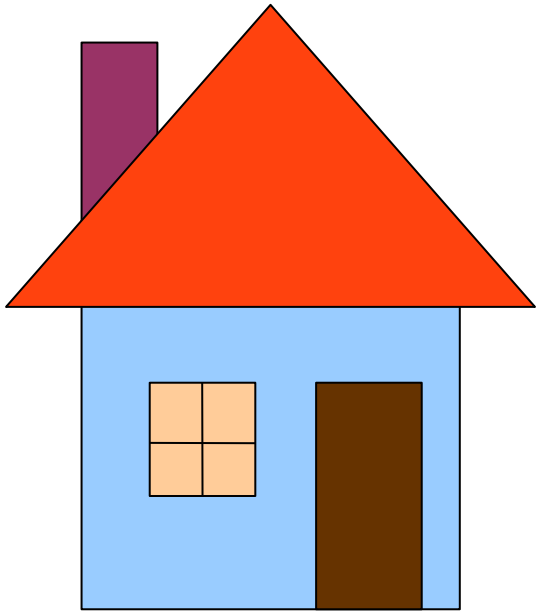
Topics for today

- **Graphics Primitives**
  - Points
  - Lines
  - Polygons 多边形
- **Line Algorithms**
  - Digital Differential Analyser (DDA) 直线插补算法
  - Bresenham Algorithm 布雷森汉姆直线算法
  - Circles 圆
  - Antialiasing 抗锯齿
- **Polygon Fill** 多边形填充
- **Graphics Primitives with OpenGL** OPENGL 几何单元.
  - glBegin(GL_POINTS); glBegin(GL_LINES)
  - glBegin(GL_POLYGON); glBEgin(GL_QUAD)
  - …

Question

How would you draw / model this house?





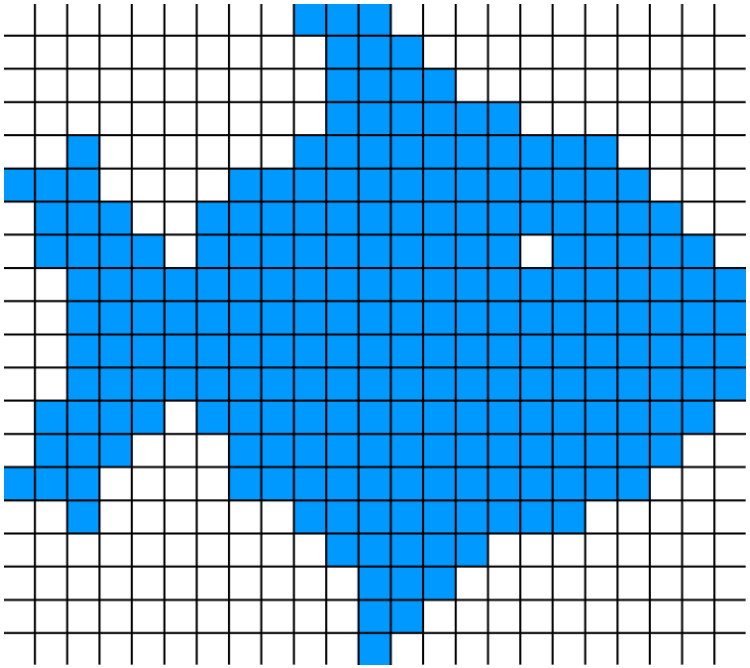
Quick answer

- **Applications**
  Packages
  Tools
  - Powerpoint
  - Word
  - Paint
  - 3DS Max
  - Maya
- **API**
  Library
  - OpenGL
  - DirectX
  - Java2D
  - Metal
  - Vulkan
  - WebGL
  - WebGPU
- **Algorithms**
  Techniques
  - DDA
  - Midpoint
  - Bresenham
  - Floodfill
  - Antialiasing

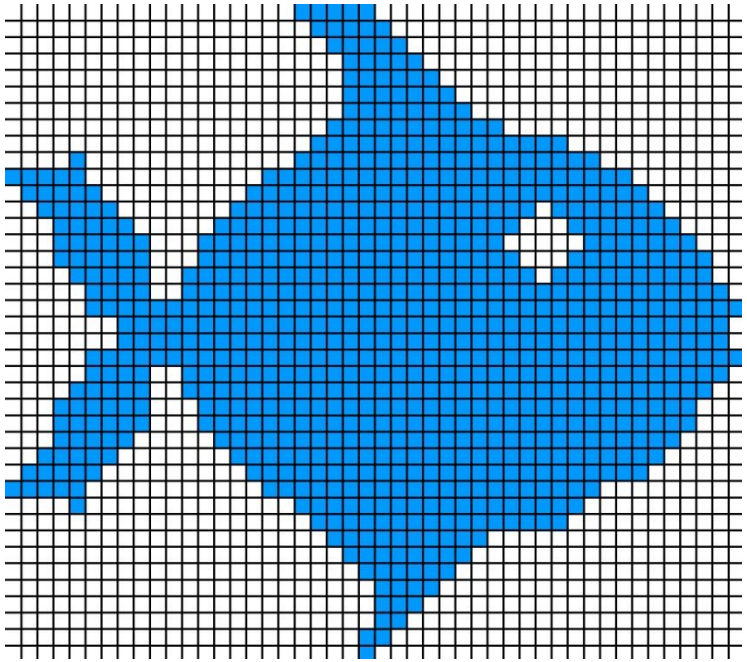
Quick answer

- **Applications**
  Packages
  Tools
  - Powerpoint
  - Word
  - Paint
  - 3DS Max
  - Maya
- **API**
  Library
  - OpenGL
  - DirectX
  - Java2D
  - Metal
  - Vulkan
  - WebGL
  - WebGPU
- **Algorithms**
  Techniques
  - DDA
  - Midpoint
  - Bresenham
  - Floodfill
  - Antialiasing

Image with 2D primitives

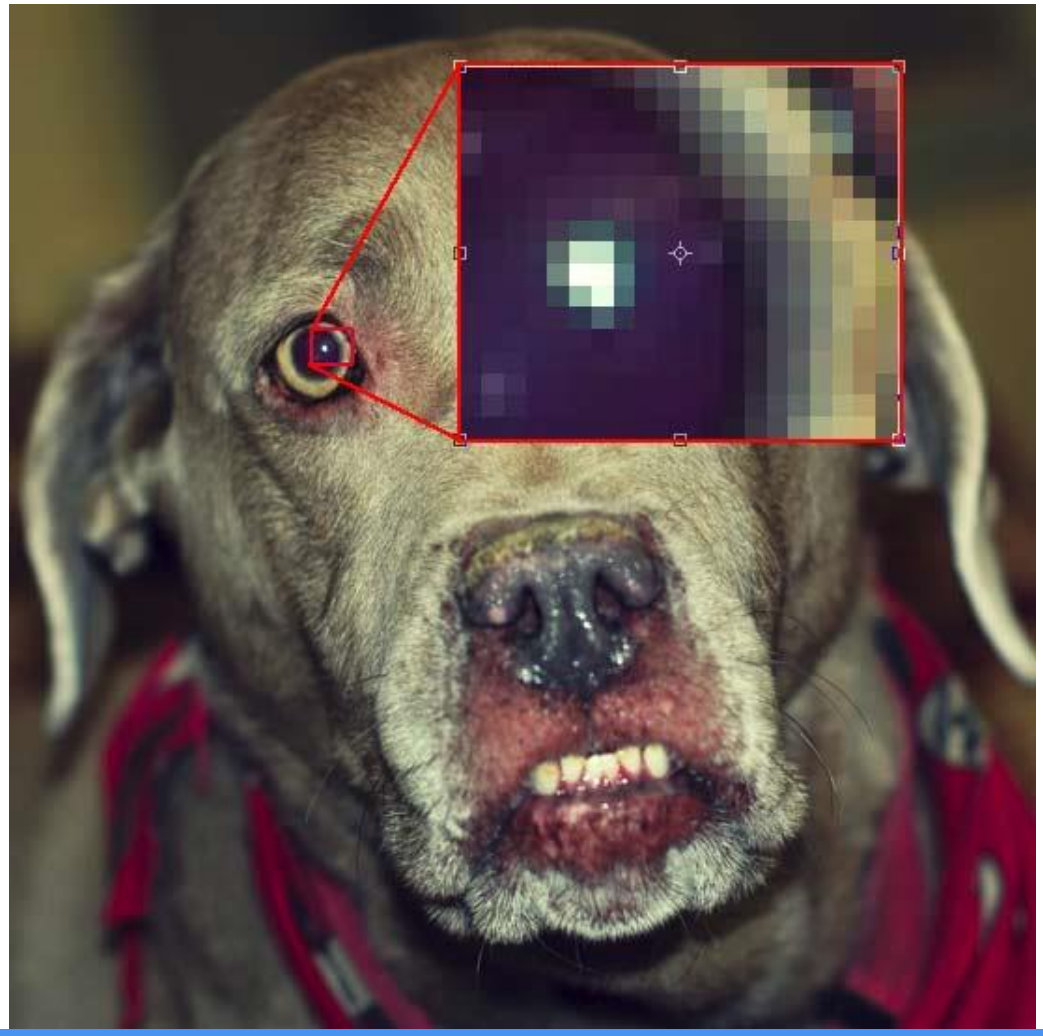


Low resolution



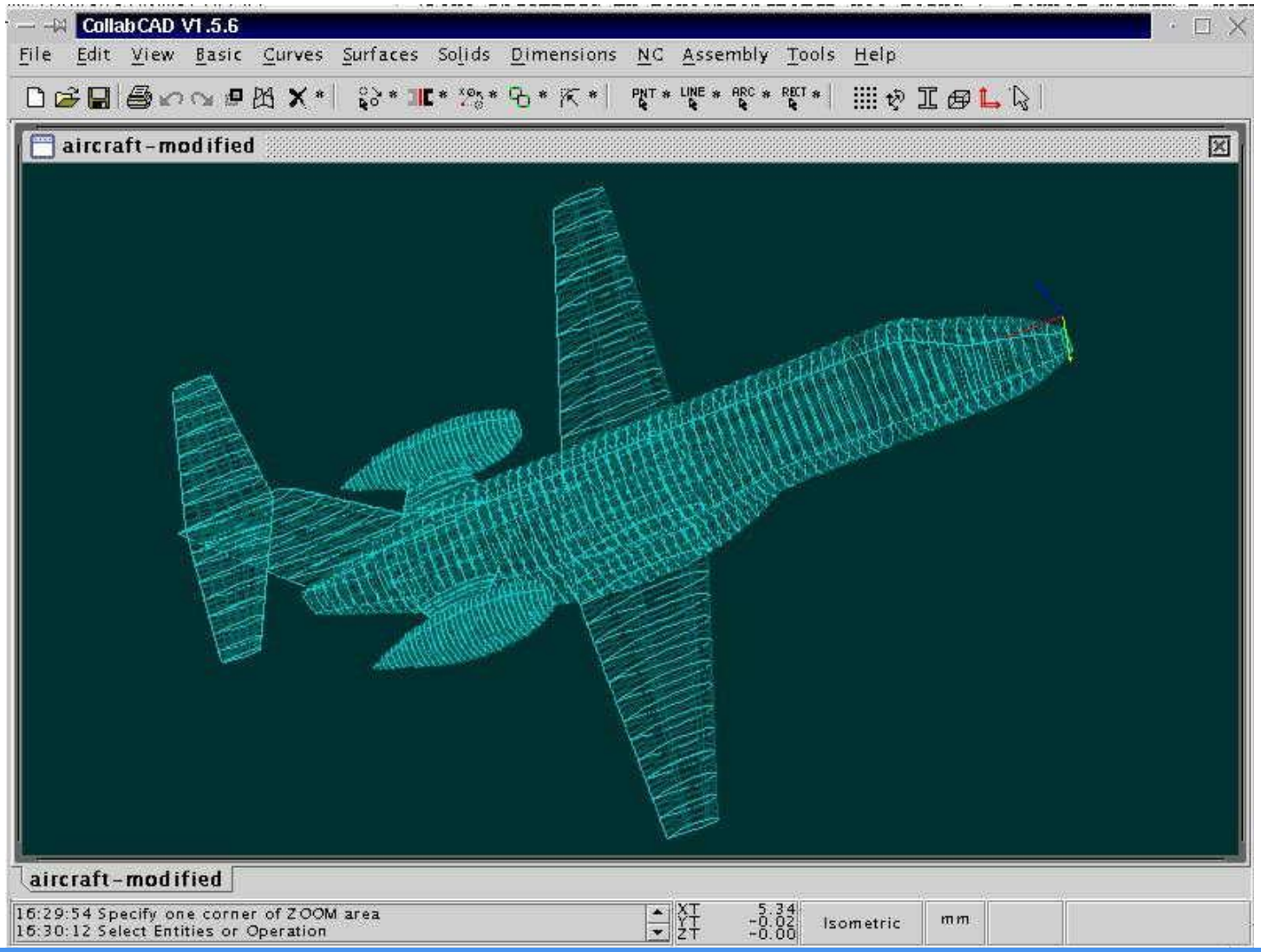
High resolution

Example of points – photograph

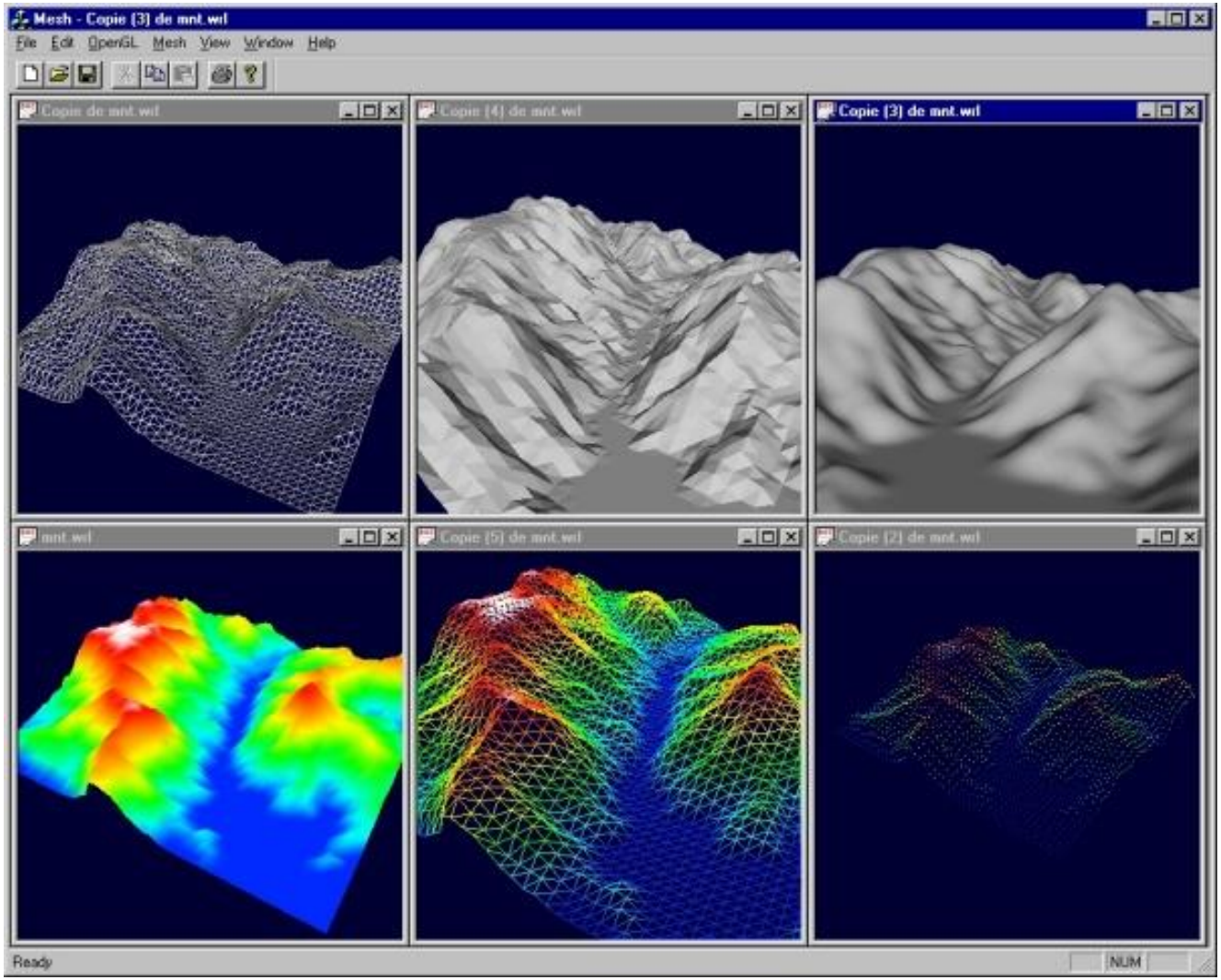


High-resolution points can be seen when we zoom in.

Example of lines – wireframe

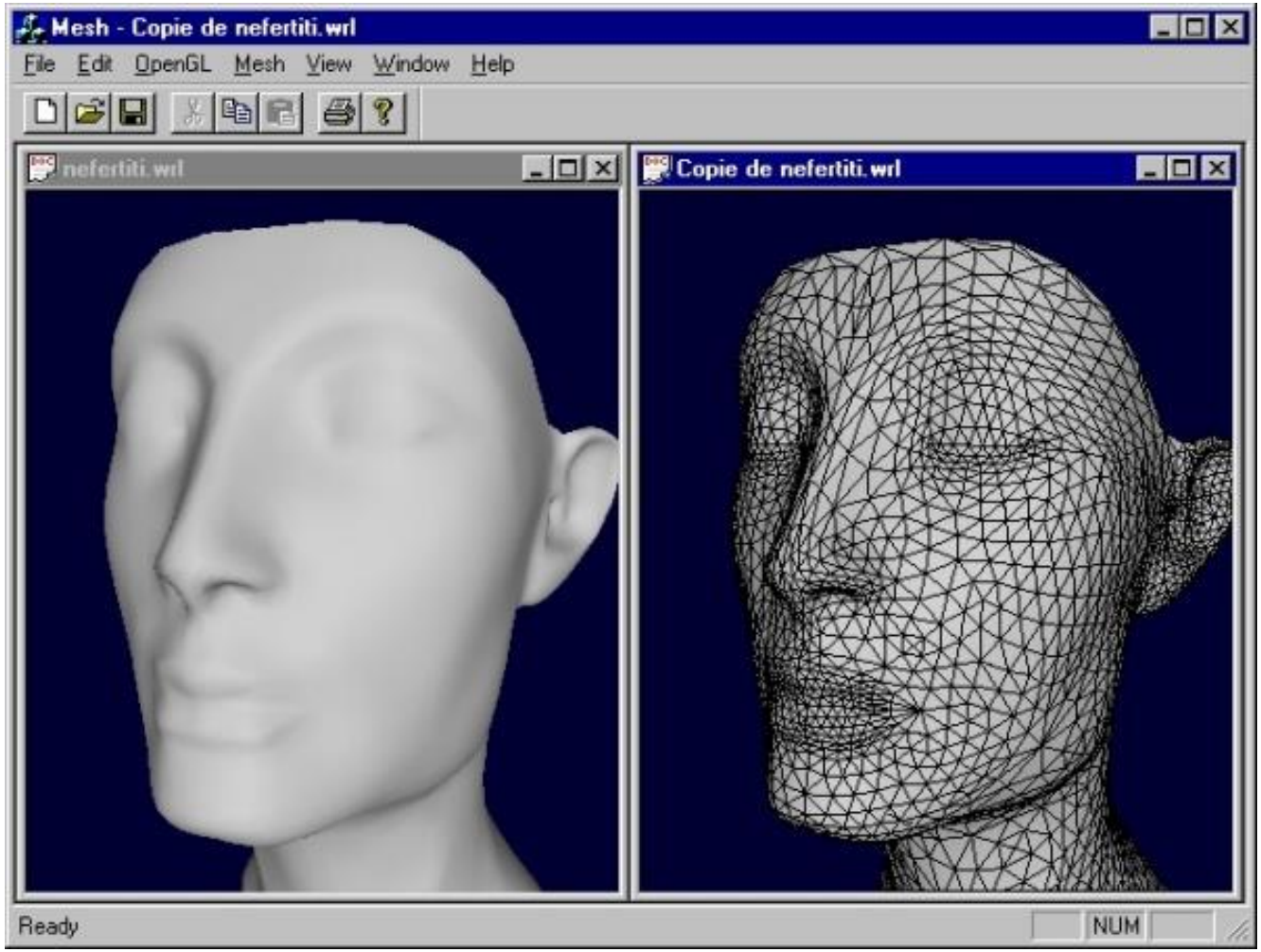


Example of lines – sculptured mesh



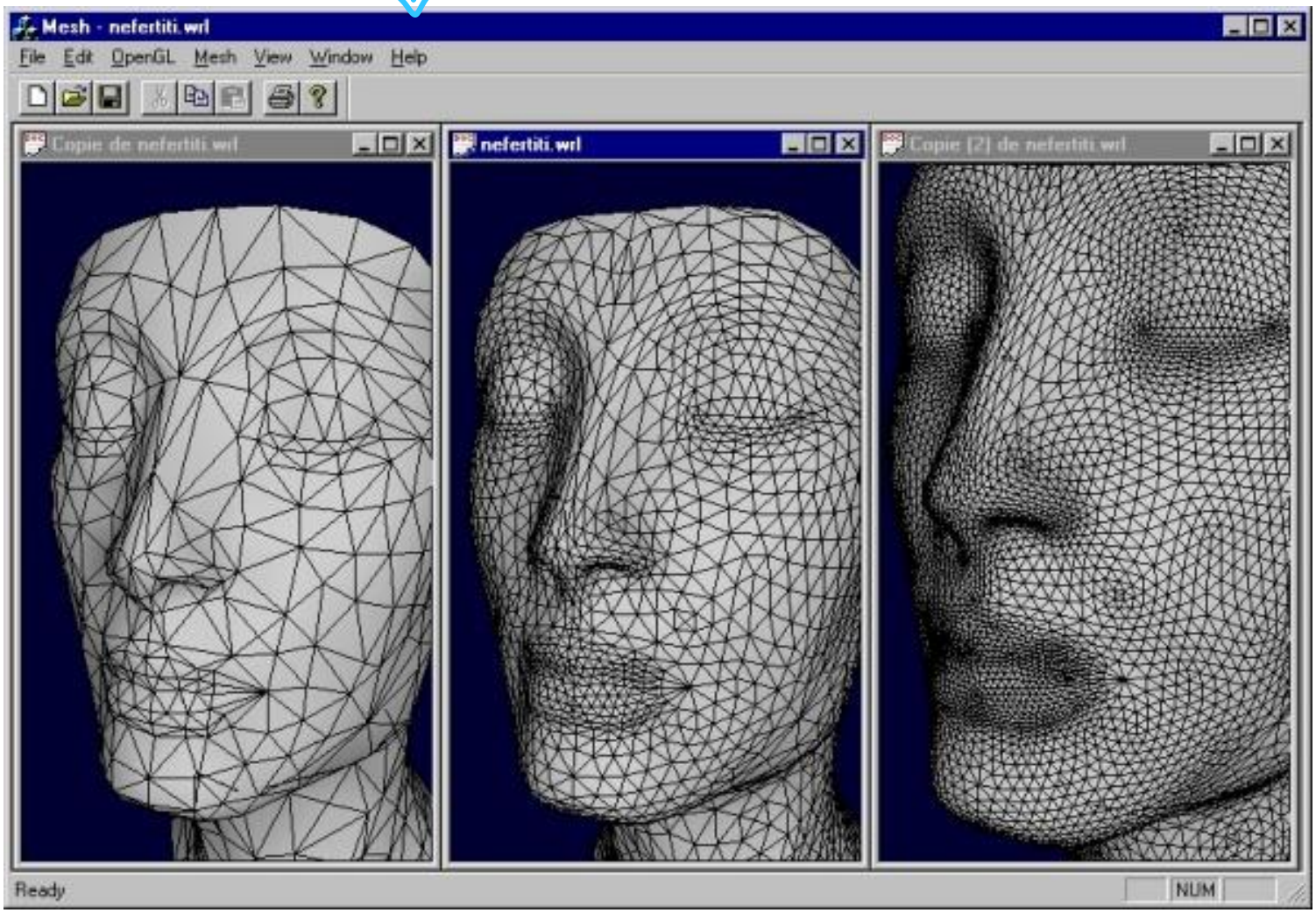
Example of lines – face mesh

使用Mesh模型预览Mesh模型



Mesh: level of detail (low, medium and high)

曲面细分



Line characterisations

- Implicit

$$y = mx + b$$

or

$$F(x,y) = ax + by + c = 0$$

Line characterisations

- Parametric (explicit) $P(t) = (1-t)P_0 + tP_1$ P_0到P_1连线

  where $P(0) = P_0$; $P(1) = P_1$

也可表示为

- Intersection of 2 planes 两个平面交集
- Shortest path between 2 points 2点间最短路径
- *Convex hull* of 2 discrete points 2个离散点的凸包

“Good” discrete lines

- No gaps in adjacent pixels
- Pixels close to ideal line
- Consistent choices; same pixels in same situations
- Smooth looking
- Even brightness in all orientations
- Same line for $P_0 P_1$ as for $P_1 P_0$
- Double pixels stacked up?

Line algorithms

- Drawing a horizontal line from (x1,y) to (x2,y) are easy ... just increment along x

"while" loop

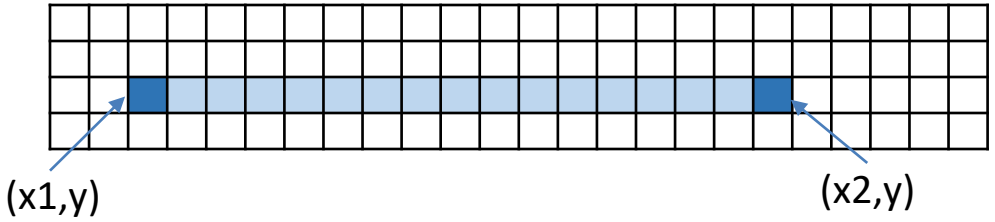
```
x = x1 while x <= x2
do {
  DrawPoint(x,y)
  x = x + 1
}
```

"for"-loop

```
for x = x1 to x2
Do {
  DrawPoint(x,y)
}
```

generator

```
DrawLine(x1 to x2, y)
```



- How do we draw lines that are not aligned to the X or Y axis?

DDA – Digital Differential Algorithm

$$y = mx + b$$

$$m = (y_2 - y_1) / (x_2 - x_1) = \Delta y / \Delta x$$

$$\Delta y = m\Delta x$$

As we move along x by incrementing x, $\Delta x = 1$, so $\Delta y = m$

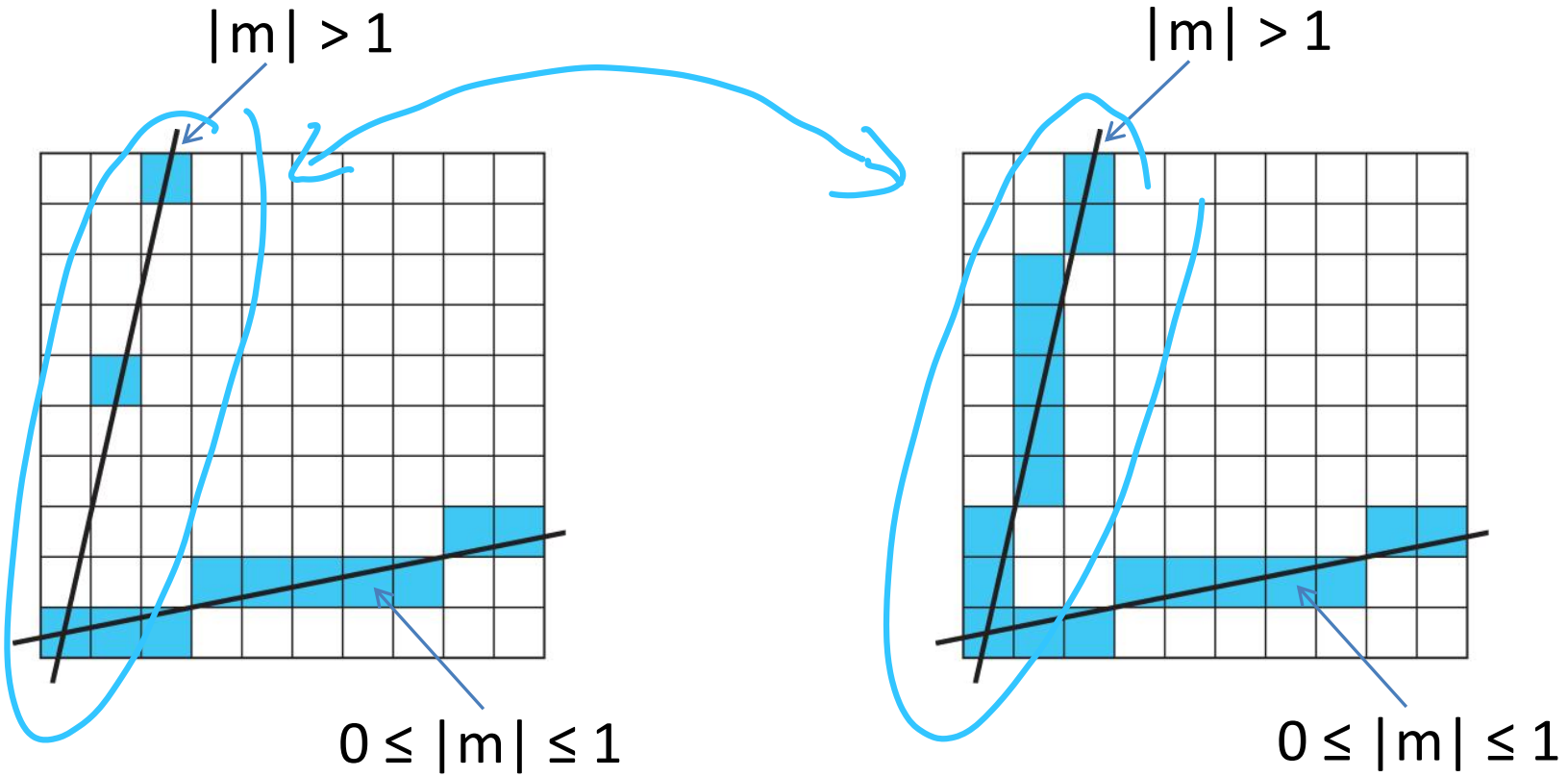
When $0 \le |m| \le 1$

```
int x;
float y=y1;
for(x=x1; x<=x2; x++){
        write_pixel(x, round(y), line_color);
        y+=m;
}
```

四舍五入 画点

DDA – Digital Differential Algorithm

上述算法在 |m|>1 欠采样！



Sampling along x-axis for both lines

Set { Δx = 1 → Sampling along x-axis (0≤|m|≤1)
Δy = 1 → Sampling along y-axis (|m|>1)

DDA – Digital Differential Algorithm

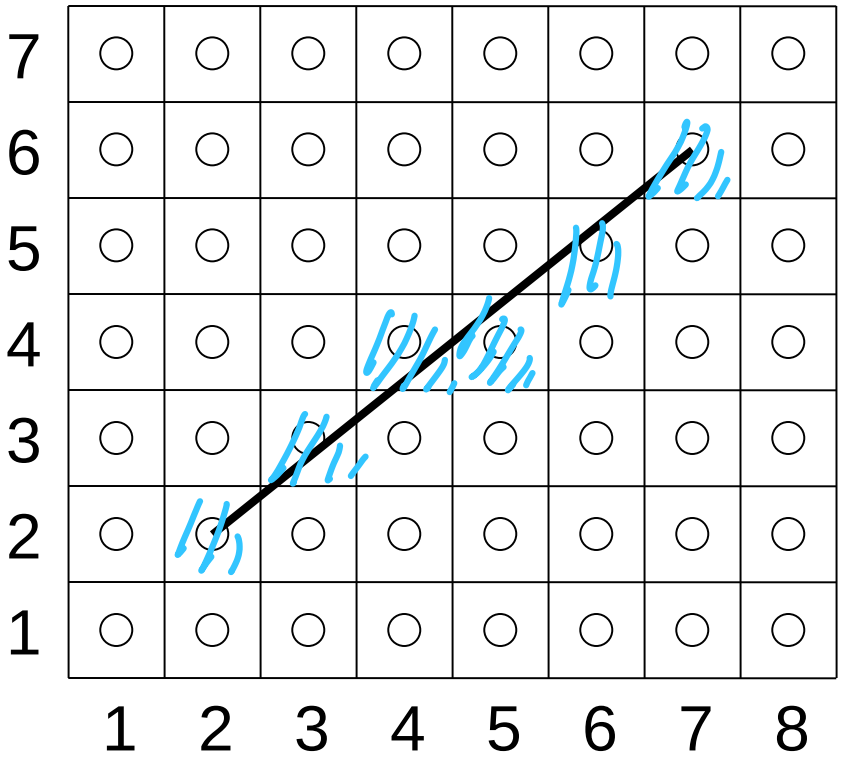
When |m| > 1, we swap the roles of x and y ($\Delta y = 1$)
($\Delta y = m\Delta x$ so $\Delta x = \Delta y/m = 1/m$).

```
int y;
float x=x1;
for(y=y1; y<=y2; y++){
        write_pixel(y, round(x), line_color);
        x+=1/m;
}
```

Questions:

1) Can you combine the two in one pseudo-code?
2) Can you derive the parts of the algorithm for negative slopes?

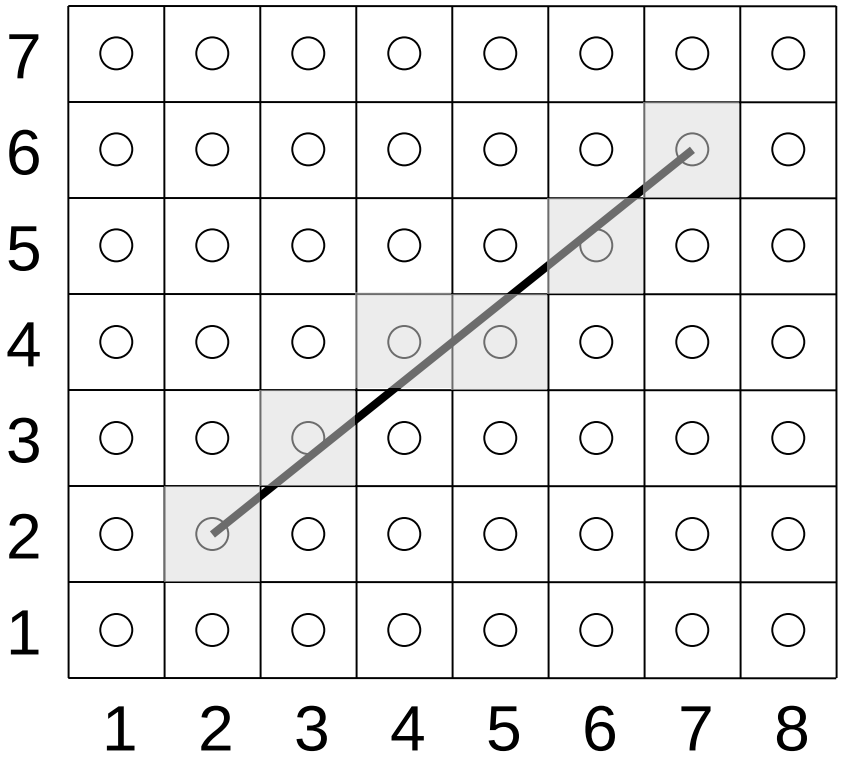
Example: line from (2,2) to (7,6)



$$m = \Delta y / \Delta x$$
$$= (6-2) / (7-2)$$
$$= 0.8$$

x_{int}	y_{flot}	y_{int}
2	2.0	2
3	2.8	3
4	3.6	4
5	4.4	4
6	5.2	5
7	6	6

Example: line from (2,2) to (7,6)



$$m = \Delta y / \Delta x$$
$$= (6-2) / (7-2)$$
$$= 0.8$$

$$y_{k+1} = y_k + m$$

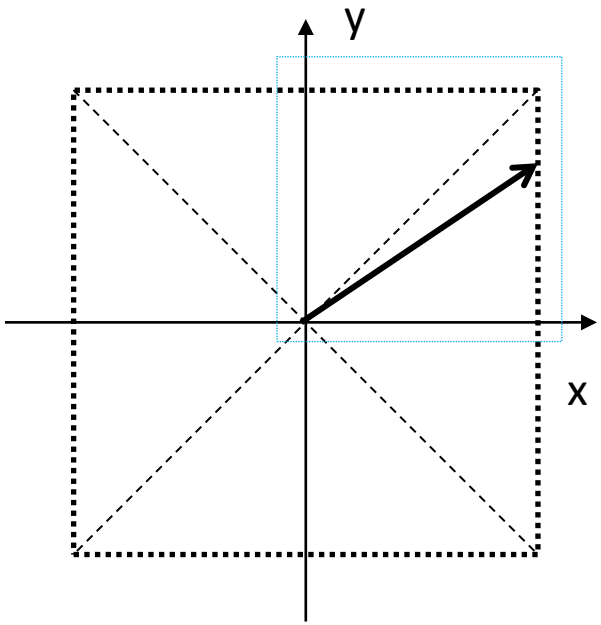
x_{int}	y_{flot}	y_{int}
2	2.0	2
3	2.8	3
4	3.6	4
5	4.4	4
6	5.2	5
7	6.0	6

The Bresenham line algorithm

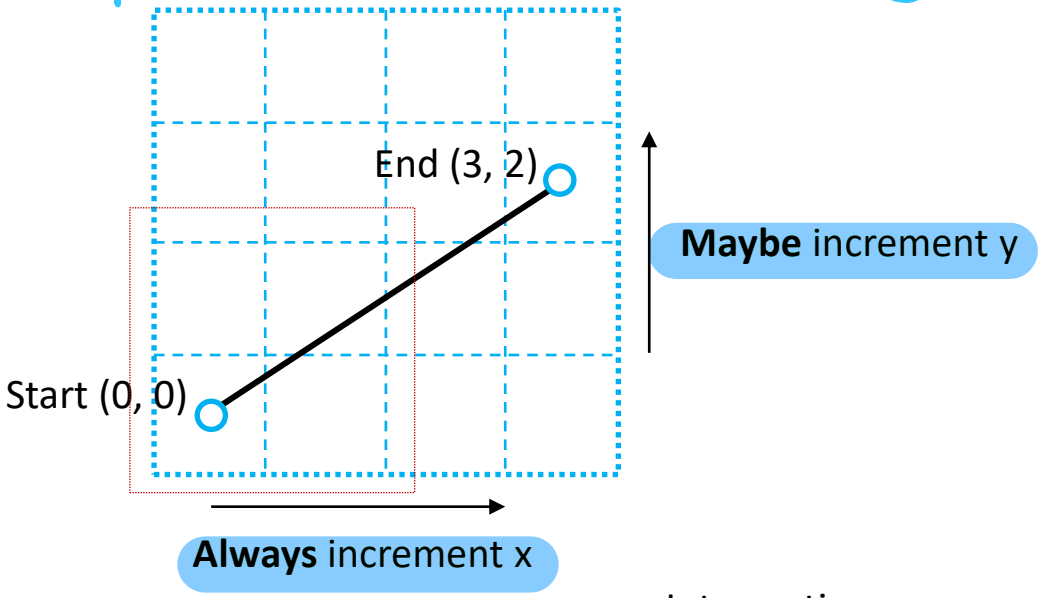
- The Bresenham algorithm is another incremental scan conversion algorithm.
- It is accurate and efficient.
- Its big advantage is that it uses only integer calculations (unlike DDA which requires float-point additions).
- The calculation of each successive pixel requires only an addition and a sign test.
- It is so efficient that it has been incorporated as a single instruction on graphics chips.

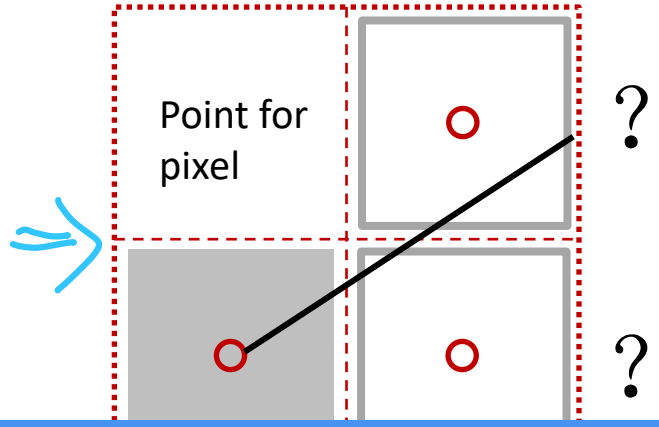
Jack Elton Bresenham worked for 27 years at IBM before entering academia. Bresenham developed his famous algorithms at IBM in the early 1960s.

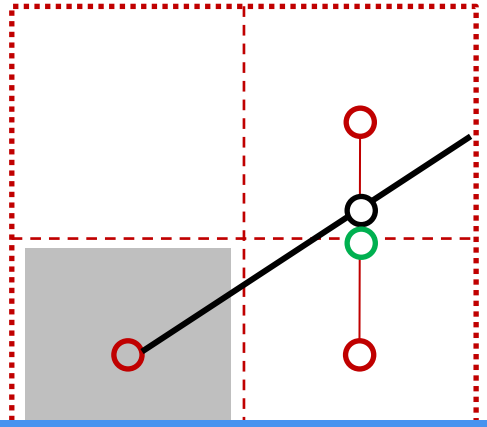
The Bresenham line algorithm



核心：总是增加x 可能增加y







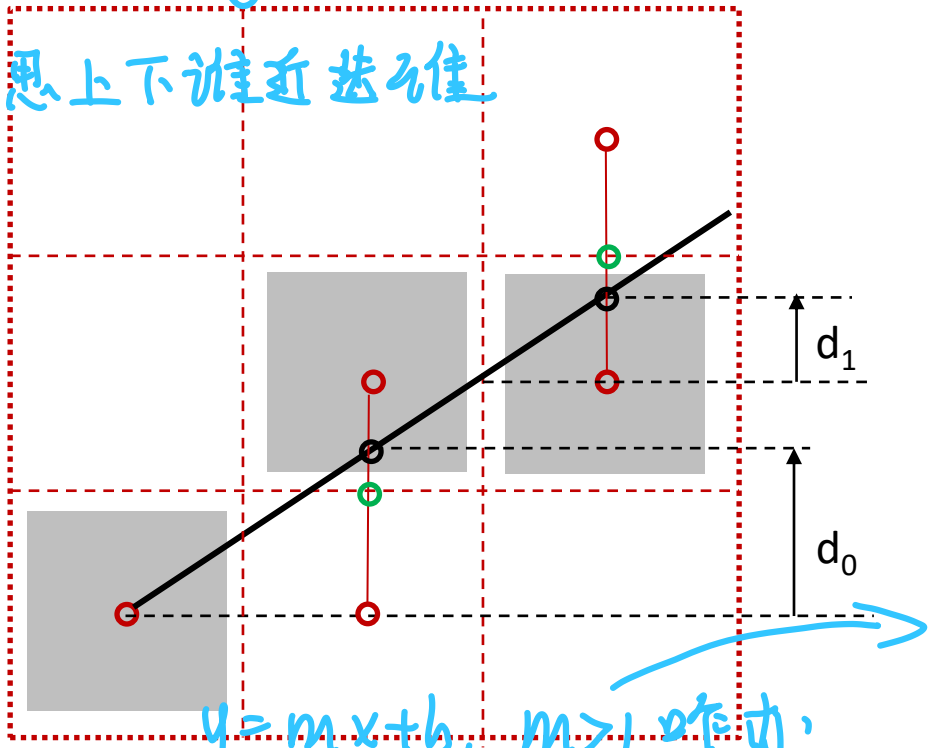
Intersection point between the two pixels

Midpoint

The Bresenham line algorithm

x+1 下看 y 的 中心 和斜线的交点 距离 d

意思上下谁近选谁佳

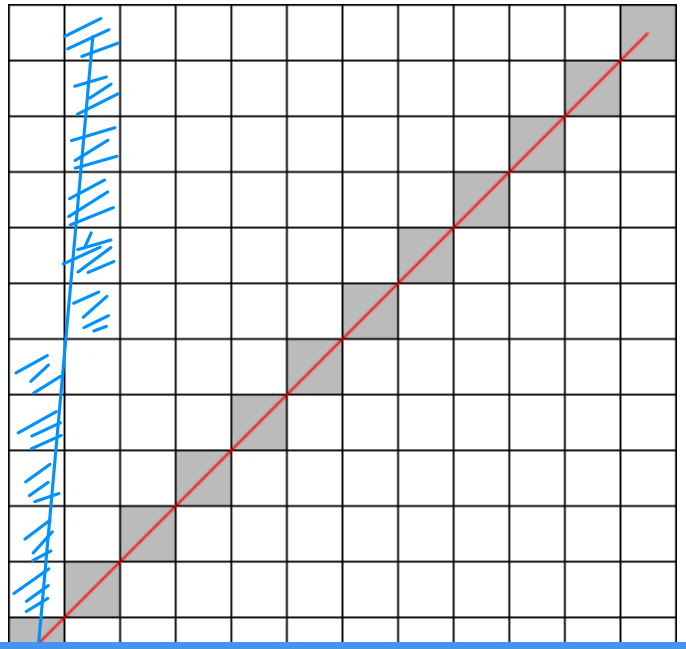


$$y_{i+1} = \begin{cases} y_i + 1, & d > 0.5 \\ y_i, & d \leqslant 0.5 \end{cases}$$

When d >= 1, Do d -= 1

y=mx+b, m>1 的时，

变成总是增加 y，x 可能增加



Questions:

1) How to deal with different slope conditions?
2) How to convert it into integer calculations?

Generation of circles

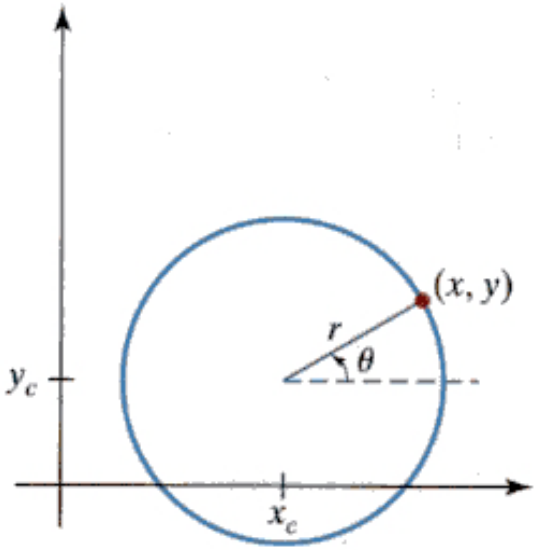
In Cartesian co-ordinates

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

笛卡尔坐标系

The position of points on the circle circumference can be calculated by stepping along the x axis in unit steps from x_c - r to x_c + r and calculating the corresponding y value at each position as

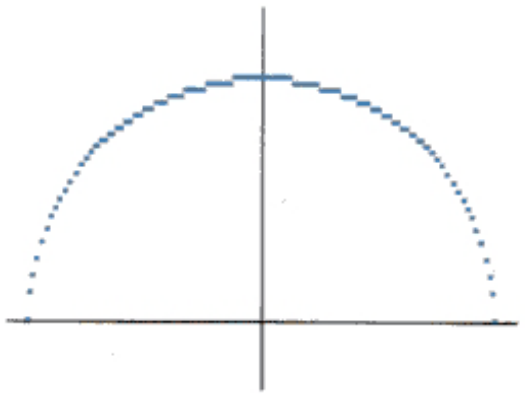
$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$



Generation of circles - problems

用上述坐标系画圆问题

- Considerable amount of computation;
- The spacing between plotted pixel positions is not uniform; 绘制像素间不均匀
  - This could be adjusted by interchanging *x* and *y* (stepping through *y* values and calculating the *x* values) whenever the absolute value of the slope of the circle is greater than 1.
  - However this simply increases the computation and processing required by the algorithm.



Generation of circles – polar co-ordinates

⇒使用极坐标系

In polar co-ordinates

$$\begin{cases} x = x_c + r\cos\theta \\ y = y_c + r\sin\theta \end{cases}$$

以知 $\theta = 0^\circ$ 画一个点 (x_0, y_0)

$= \frac{1}{r}^\circ$ 画一个点 (x_1, y_1)

$= \frac{2}{r}^\circ$

$= 360^\circ$ (x_n, y_n)

- When a display is generated with these equations using a fixed angular step size, a circle is plotted with equally spaced points along the circumference.
- To reduce calculations, a large angular separation can be used between points along the circumference and connect the points with straight-line segments to approximate the circle path.
- For a more continuous boundary on a raster display, the angular step size can be set at $1/r$. This plots pixel positions that are approximately one unit apart.

Generation of circles - symmetry

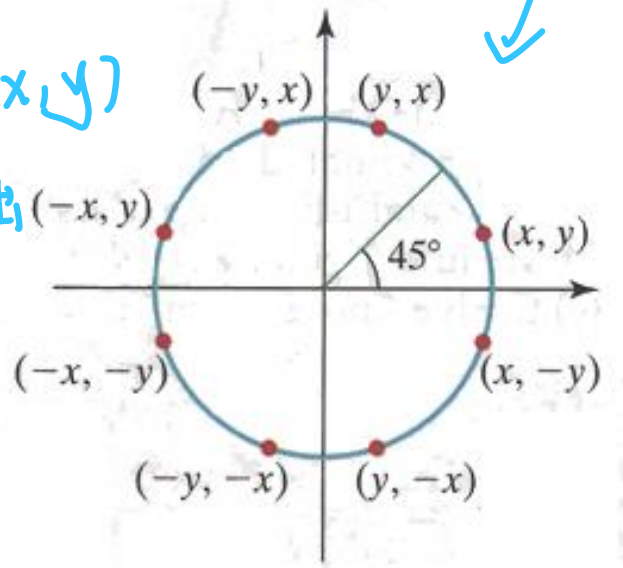
- Computations can be reduced by considering the symmetry of the circle.
- If the curve positions in the first quadrant are determined, the circle section in the second quadrant can be generated by noting that the two circle sections are symmetric with respect to the y axis.
- Circle sections in the third and fourth quadrants can be obtained from the sections in the first and second quadrants by considering the symmetry about the x axis.
- Taking this one step further, it can be noted that there is also symmetry between **octants**. Circle sections in adjacent octants within one quadrant are symmetric with respect to the 45° line dividing the two octants.

八分体

可以只画一个45°，其他基于对称减少计算量。

Generation of circles - symmetry

- Considering symmetry conditions between octants, a point at position (x, y) on a one-eighth circle sector is mapped into the seven circle points in the other octants of the x-y plane.
- Taking the advantage of the circle symmetry in this way, all pixels around a circle can be generated by calculating only the points within the sector from $x = r$ to $x = y$.
- The slope of the curve in this octant has an absolute magnitude equal to or larger than 1.

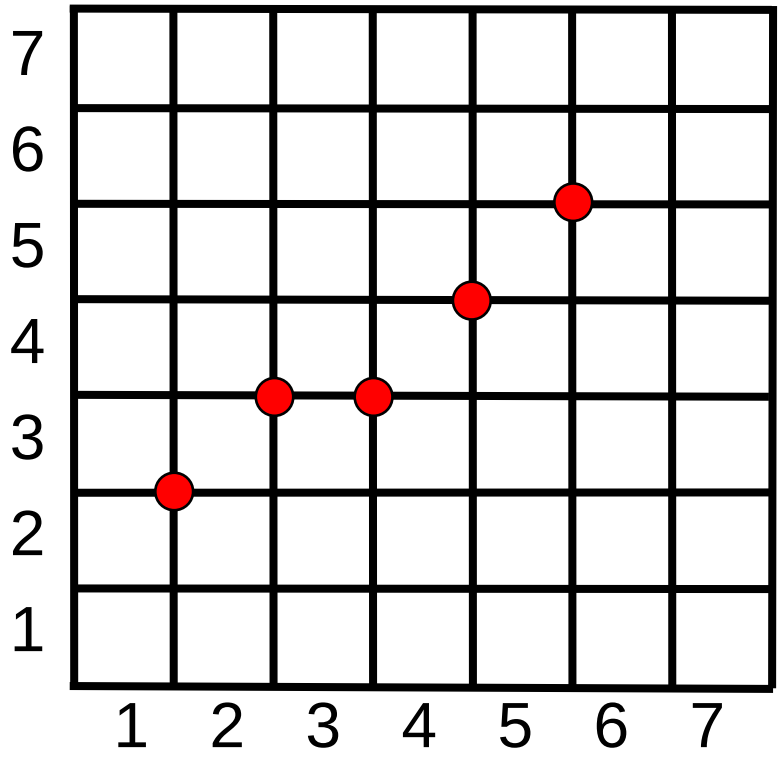


Generation of circles - Efficiency

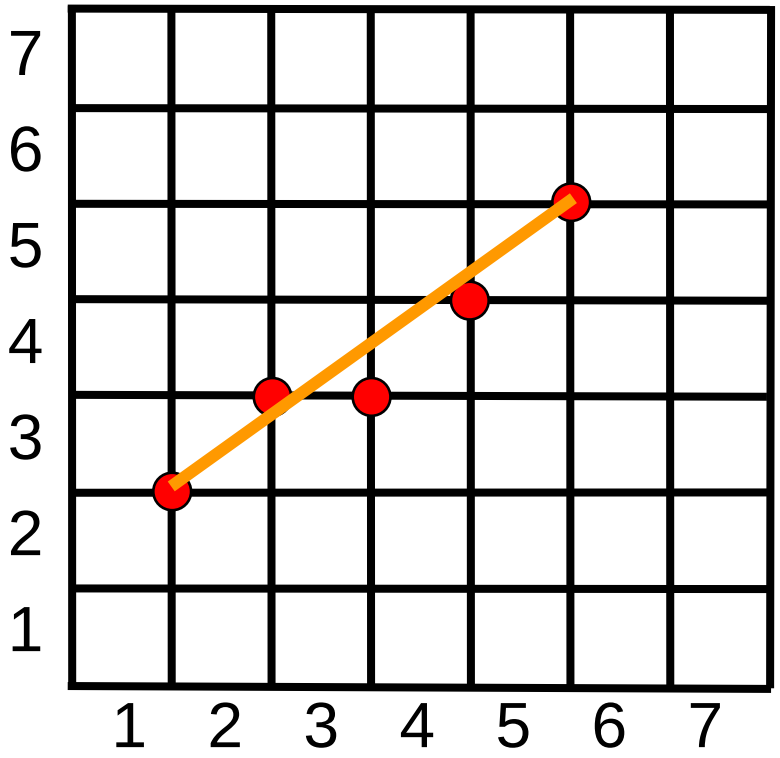
- Determining pixel positions along a circle circumference using symmetry and the equation either in Cartesian or polar co-ordinates, still requires a good deal of computation.
- The Cartesian equation involves multiplication and square root calculations.
- The parametric equations contain multiplications and trigonometric calculations.
- More efficient circle algorithms are based on incremental calculation of decision parameters, which involves only simple integer operations.

Line: raster points

线实际上光栅点(=离散像素点)

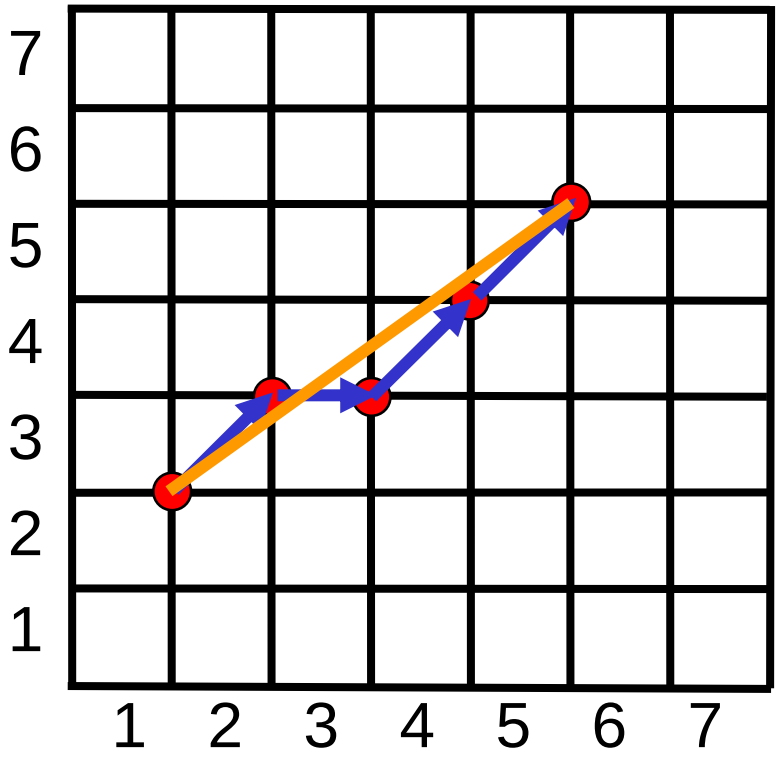


Lines vs points

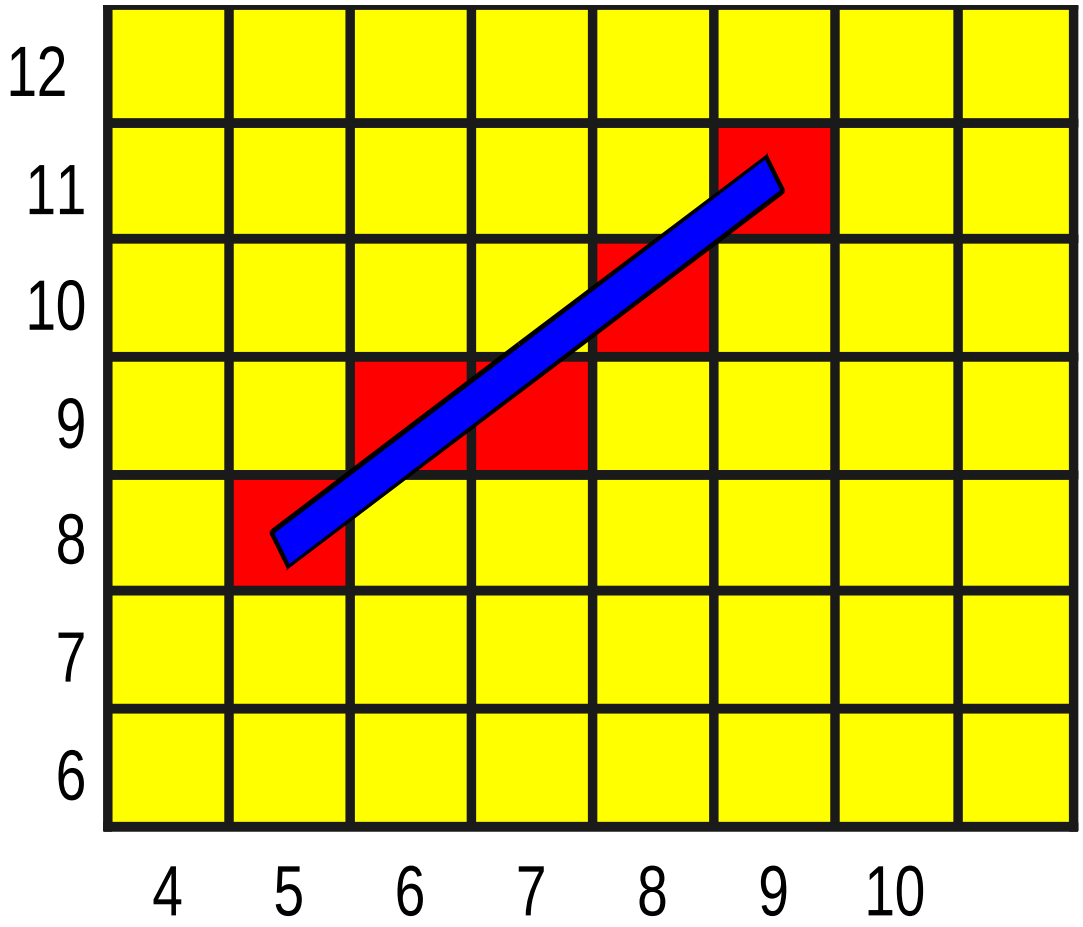


锯齿 → Jaggies

由于像素离散化导致图形边缘出现锯齿状现象.



Pixel space

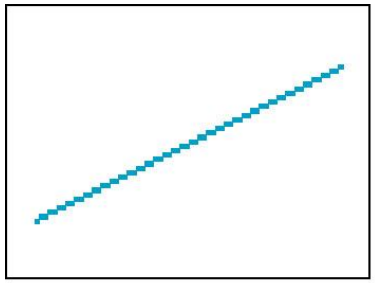


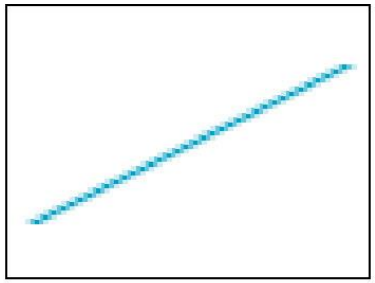
抗锯齿

Antialiasing by area averaging

Colour multiple pixels for each x depending on coverage by ideal line

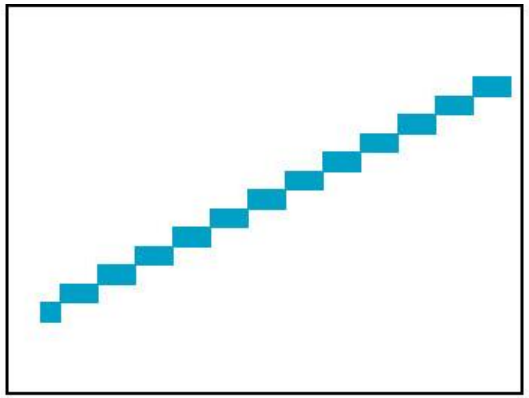
Original

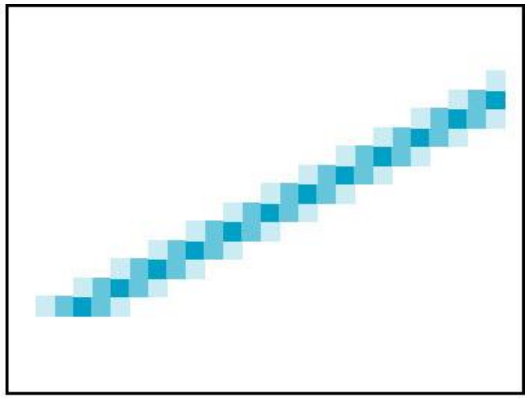




Antialiased

看上去更加平滑





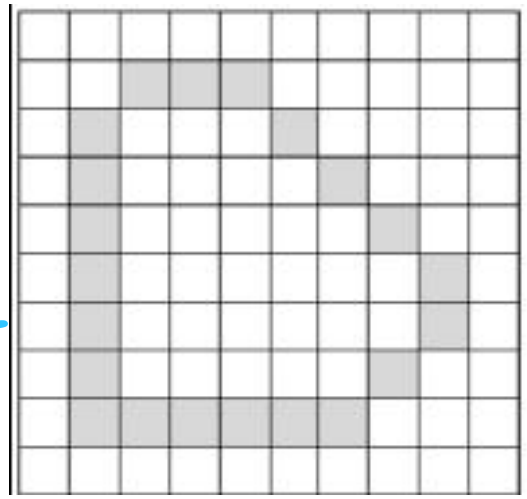
Magnified (放大)

Geometric primitives – polygons and triangles

- The basic graphics primitives are points, lines and polygons
  - A polygon can be defined by an **ordered** set of vertices
- Graphics hardware is optimised for processing points and flat polygons
- Complex objects are eventually divided into triangular polygons (a process called tessellation)
  - Because triangular polygons are always flat

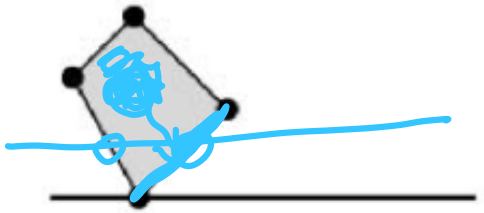
Scan conversion

- Also called rasterization.
- The 3D to 2D projection gives us 2D vertices (points) to define 2D graphic primitives.
- We need to fill in the interior
  - the rasterizer must determine which pixels in the framebuffer are inside the polygon.

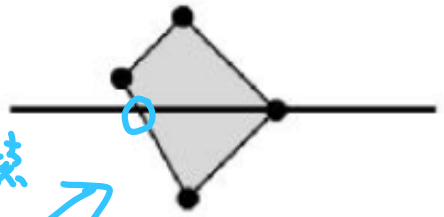


Polygon fill

- Rasterize edges into framebuffer.
- Find a seed pixel inside the polygon.
- Visit neighbours recursively and colour if they are not edge pixels.
- When vertices lie on the scanlines, cases (a) and (b) must be treated differently when using odd-even fill definition
  - Case (a): zero or two crossings
  - Case (b): one edge crossing



(a)



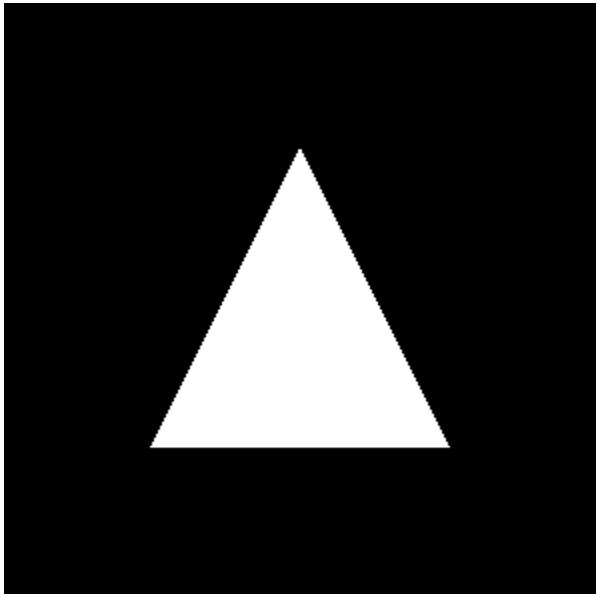
(b)

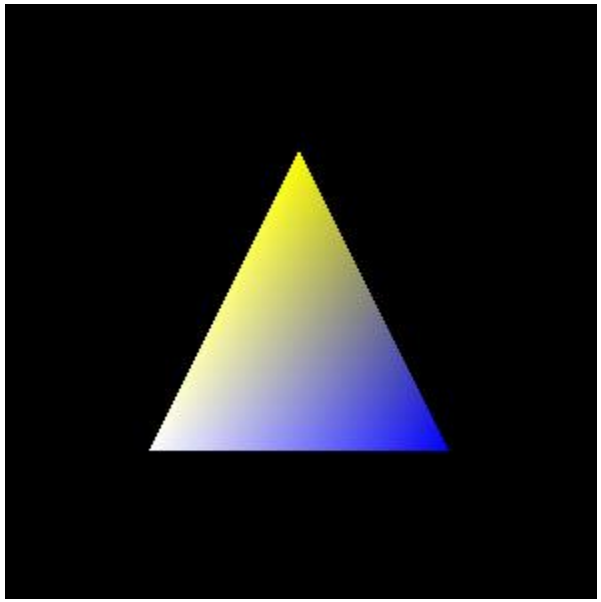
Polygon fill

填充单一颜色

使用每顶点颜色进行变换模拟光照效果

Flat shading vs Smooth shading

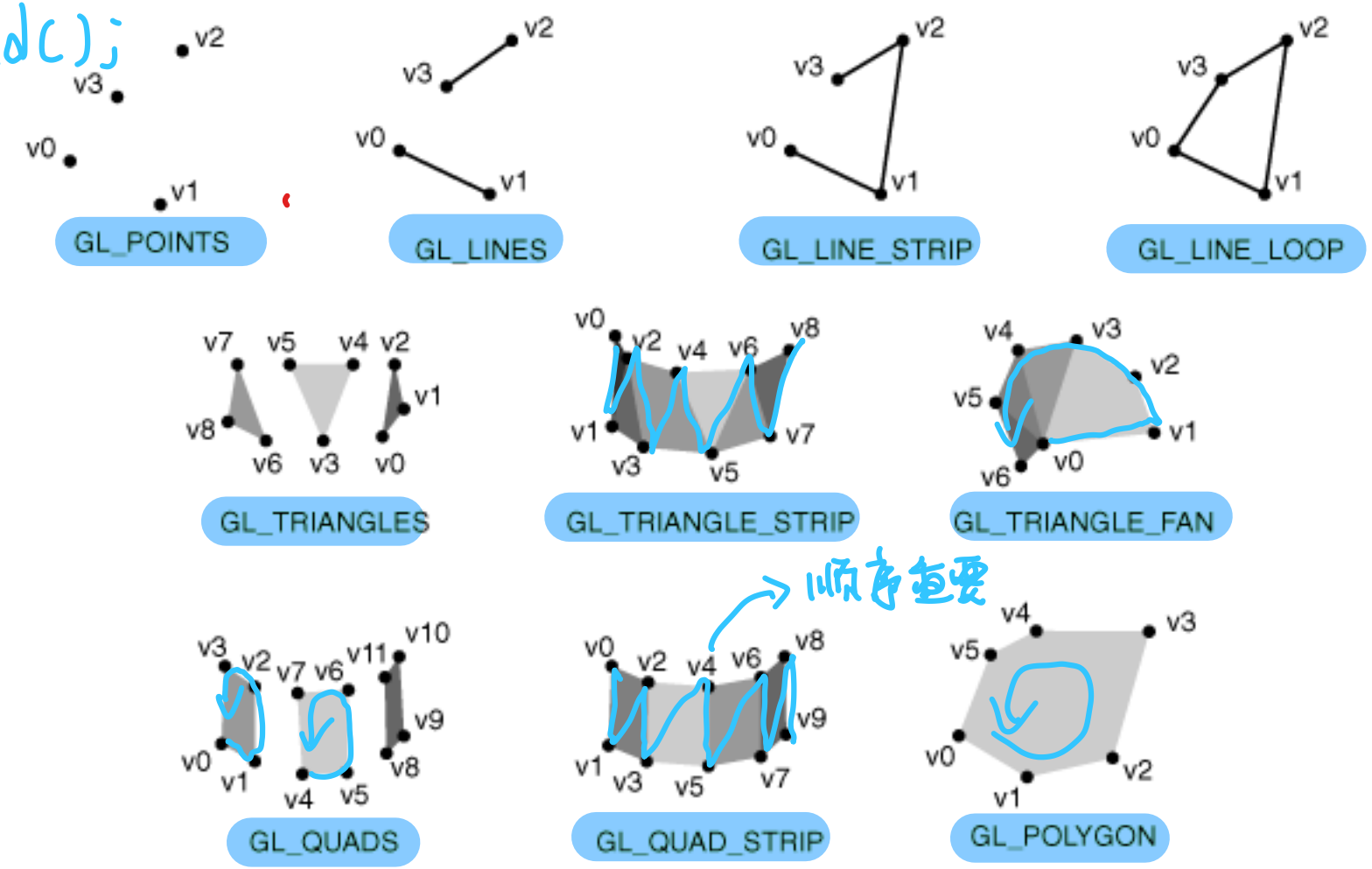




glBegin(GL_POINTS);
glVertex2f(---);
⋮
glEnd();

替换下面内容

Geometric primitives in OpenGL



glBegin(parameters)

- GL_POINTS: individual points
- GL_LINES: pairs of vertices interpreted as individual line segments
- GL_LINE_STRIP: series of connected line segments
- GL_LINE_LOOP: same as above, with a segment added between last and first vertices
- GL_TRIANGLES: triples of vertices interpreted as triangles
- GL_TRIANGLE_STRIP: linked strip of triangles
- GL_TRIANGLE_FAN: linked fan of triangles
- GL_QUADS: quadruples of vertices interpreted as four-sided polygons
- GL_QUAD_STRIP: linked strip of quadrilaterals
- GL_POLYGON: boundary of a simple, convex polygon

GL_POINTS

```
// This code will draw a point located at (100, 100).

glBegin(GL_POINTS);
    glVertex2f(100.0f, 100.0f);
        ...
        // add more points if required
glEnd( );
```

GL_LINES

```
// This code will draw a line at starting and ending
// coordinates specified with glVertex2f().

glBegin(GL_LINES);
        glVertex2f(100.0f, 100.0f); // origin of line
        glVertex2f(200.0f, 140.0f); // end point of line
glEnd( );
```

How to make lines efficient?

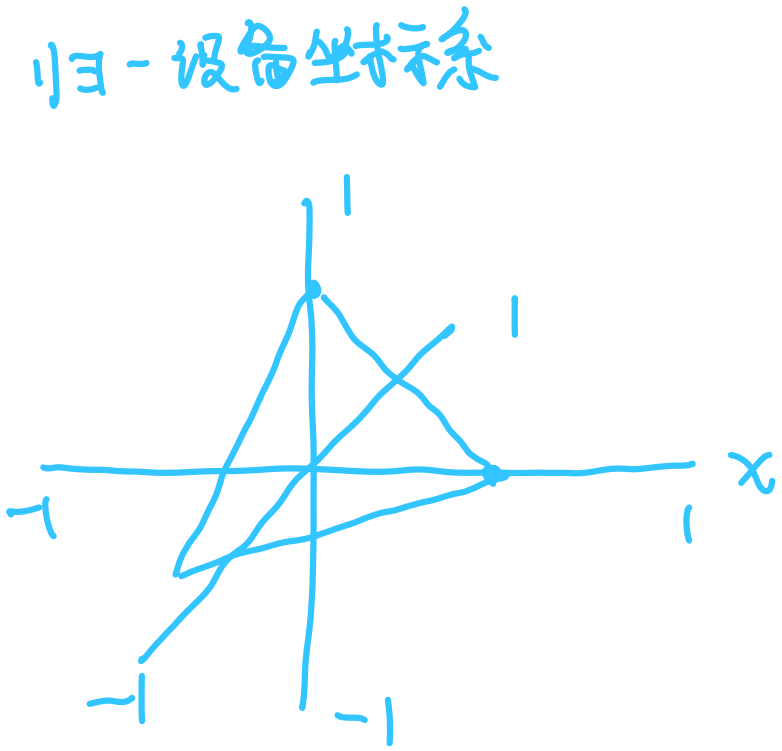
```
// This code will draw two lines "at a time" to save
// the time it takes to call glBegin() and glEnd().

glBegin(GL_LINES);
    glVertex2f(100.0f, 100.0f); // origin of the FIRST line
    glVertex2f(200.0f, 140.0f); // end point of the FIRST line

    glVertex2f(120.0f, 170.0f); // origin of the SECOND line
    glVertex2f(240.0f, 120.0f); // end point of the SECOND line
glEnd( );
```

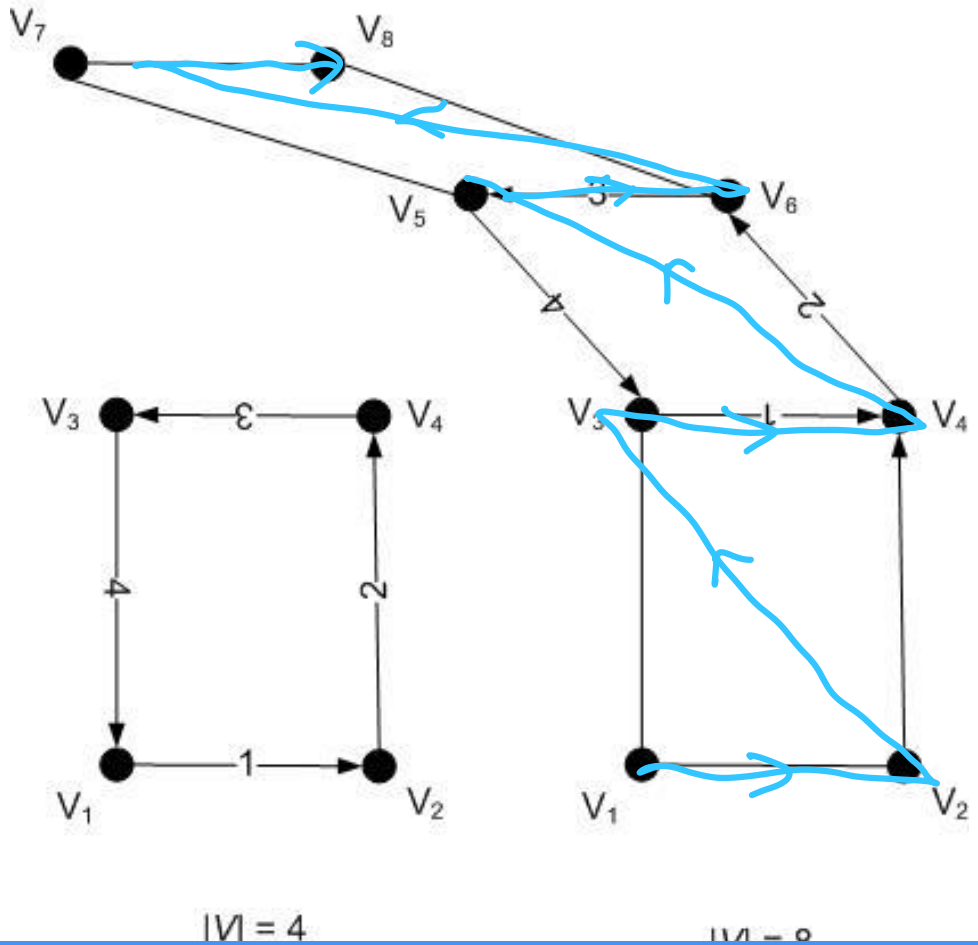
Triangle in OpenGL

```
glBegin(GL_TRIANGLES);
        glVertex2f(-0.5,-0.5);
        glVertex2f(0.5,0.0);
        glVertex2f(0.0,0.5);
glEnd();
```



glQuad_STRIP

Ordering of coordinates very important



|V| = 4

Summary

- **Graphics Primitives**
  - Points
  - Lines
  - Polygons
- **Line Algorithms**
  - Digital Differential Analyser (DDA)
  - Bresenham Algorithm
  - Circles
  - Antialiasing
- **Polygon Fill**
- **Graphics Primitives with OpenGL**
  - glBegin(GL_POINTS); glBegin(GL_LINES)
  - glBegin(GL_POLYGON); glBEgin(GL_QUAD)
  - …