بيانات الحاسوب / المحاضرة (2) الخوارزميات الأساسية في الرسوم ثنائية البعد (2D Basics Algorithms)

1- المقدمة

تشكل خوازميات الرسوم ثنائية البعد 2D الأساس لفهم كيفية إنشاء ومعالجة الصور الرقمية على شاشة الحاسوب، تهتم هذه الخوارزميات برسم العناصر الأساسية مثل (الخطوط، الدوائر، والمضلعات)، مع الأخذ بعين الإعتبار كفاءة ودقة الخوارزميات خاصة في الرسوم النقطية (Raster Graphics) التي يتم تمثيلها بشبكة من البكسلات.

في هذه المحاضرة سنغطي الخوارزميات الرئيسية:

1- رسم الخطوط مثل خوارزمية (DDA و Bresenham)

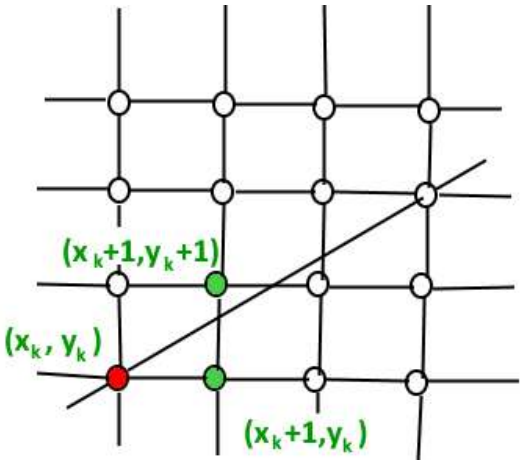
2- رسم الدوائر مثل خوارزمية (Midpoint Circle)

3- ملء المضلعات مثل خوارزمية (Scanline Fill)

تستخدم هذه الخوارزميات في الأجهزة ذات الموارد المحدودة، وكذلك الأمر في برامج مثل OpenGL ، سنشرح كل خوارزمية بالتفصيل، مع المعادلات الرياضية، الخطوات، وأمثلة عملية.

2- خوارزميات رسم الخطوط

خوارزميات رسم الخطوط المستقيمة بين نقطتين، هي من أهم الخوارزميات في الرسوم ثنائية البعد ولدينا خوارزميتان أساسيتان:



2.1. خوارزمية المحلل التفاضلي الرقمي DDA Digital Differential Analyzer

تستخدم طريقة بسيطة لرسم الخطوط، تعتمد على حساب الفرق في الإحداثيات.

بفرض أن لدينا نقطتين (x1, y1) و (x2, y2)

1- نحسب الفرق dx = x2 − x1 ، dy = y2 − y1

2- نحسب عدد الخطوات step = max(|dx|, |dy|)

3- نحسب الزيادة في كل محور: x_inc = dx / step ، y_inc = dy / step

```
def DDA(x1, y1, x2, y2):
    # نحسب الفرق في الإحداثيات
    dx = x2 - x1
    dy = y2 - y1
    # نحسب عدد الخطوات بناءً على أكبر فرق
    steps = max(abs(dx), abs(dy))
    # نحسب الزيادة في كل خطوة على المحورين
    x_inc = dx / steps
    y_inc = dy / steps
    x, y = x1, y1
    # حلقة لرسم كل بكسل
    for _ in range (steps + 1):
      # دالة لرسم البكسل
       plot(round(x), round(y))
       x += x_inc
       y += y_inc
```

هذه الخوارزمية تستخدم النقطة العائمة، مما يجعلها بطيئة في بعض الأجهزة، لكنها سلسة وسهلة الفهم.

2.2. خوارزمية بريزنهام Bresenham

تهدف هذه الخوارزمية إلى رسم خط مستقيم بين نقطتين (x0,y0) و (x1,y1) على شاشة مكونة من نقاط (Pixels) بحيث يكون الخط أقرب ما يمكن إلى الخط الرياضي الحقيقي ، بطريقة سريعة ودقيقة وبدون استخدام العمليات العشرية أو الكسور (الضرب والقسمة)، وتستخدم بدلاً من ذلك، عمليات الجمع والطرح والمقارنة، مما يجعلها مثالية للأجهزة محدودة الموارد أو الرسومات في الزمن الحقيقي.

كما نعلم فإن المعادلة الرياضية للخط المستقيم هي : $y = m.x + b$ حيث:

m هو الميل ويساوي $m = (y1-y0) / (x1-x0)$

و b هو التقاطع مع محور y

لكن هذه الصيغة تتضمن عمليات ضرب وقسمة، بدلاً من ذلك، تقوم خوارزمية بريزنهام بتقريب أقرب بكسل إلى الخط الحقيقي في كل خطوة على طول المحور الأساسي (المحور x عادةً).

<u>خطوات الخوارزمية</u>

1. نحسب القيم الابتدائية:

$$dx = |x1 - x0| \quad , \quad dy = |y1 - y0|$$

2. نحسب المتغير الأولي للقرار (decision parameter):

$$p = 2dy - dx$$

3. نبدأ من النقطة الأولى (x0, y0)

4. لكل x من x_0 إلى x_1:

نرسم النقطة الحالية (x, y)

إذا كان ($p < 0$):

تكون النقطة التالية هي (x+1, y)

نحدّث $p = p + 2dy$

أما إذا كان (p >= 0):

تكون النقطة التالية هي (x+1, y+1)

نحدّث p = p + 2(dy - dx)

الفكرة الأساسية للقرار p هو أن قيمة p تمثل مدى قرب الخط الحقيقي من البكسل الأعلى أو الأسفل، فتختار الخوارزمية البكسل الأقرب إلى المسار الرياضي الحقيقي بناءً على إشارة p.

مثال تطبيقي:

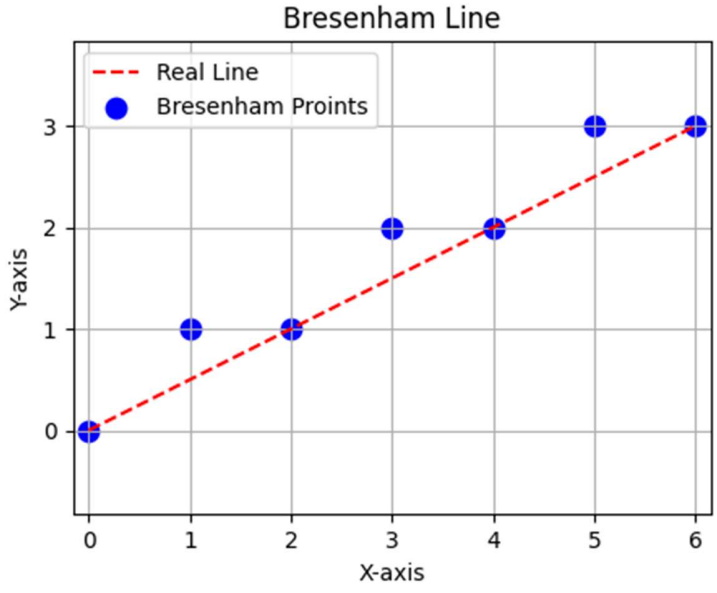
لنفترض أننا نريد رسم خط من (0، 0) إلى (3، 6):

dy = 3 ، dx = 6

p0 = 2dy - dx = 2×3 - 6 = 0

ثم نحسب باقي النقاط تدريجياً كما هو موضح بالجدول التالي:

الخطوة	x	y	p (قبل القرار)	القرار المتخذ	p (بعد التحديث)	النقطة المرسومة
0	0	0	0	$p \geq 0 \Rightarrow y+1$	$p = 0 + 2(3-6) = -6$	**(0, 0)**
1	1	1	−6	$p < 0 \Rightarrow$ لا يتغير y	$p = -6 + 2(3) = 0$	**(1, 1)**
2	2	1	0	$p \geq 0 \Rightarrow y+1$	$p = 0 + 2(3-6) = -6$	**(2, 1)**
3	3	2	−6	$p < 0 \Rightarrow$ لا يتغير y	$p = -6 + 2(3) = 0$	**(3, 2)**
4	4	2	0	$p \geq 0 \Rightarrow y+1$	$p = 0 + 2(3-6) = -6$	**(4, 2)**
5	5	3	−6	$p < 0 \Rightarrow$ لا يتغير y	$p = -6 + 2(3) = 0$	**(5, 3)**
6	6	3	—	—	—	**(6, 3)**



دالة لتنفيذ المثال السابق بلغة بايثون:

```
import matplotlib.pyplot as plt
import numpy as np

def bresenham (x0, y0, x1, y1):
    # قائمة لتخزين النقاط الناتجة
    points = []
    # حساب الفروقات بين إحداثيات النقطتين
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)
    # تحديد اتجاه الحركة على كل محور
    sx = 1 if x0 < x1 else -1
    sy = 1 if y0 < y1 else -1
    p = 2 * dy - dx  # قيمة القرار الابتدائية

    x, y = x0, y0

    while x != x1 + sx:
        points.append((x, y))
        x += sx
        if p >= 0:
            y += sy
            p += 2 * (dy - dx)
        else:
            p += 2 * dy
    return points
# ------------------ تنفيذ الدالة ------------------
x0, y0, x1, y1 = 0, 0, 6, 3
points = bresenham (x0, y0, x1, y1)
# الخط الرياضي الحقيقي
x_line = np.linspace(x0, x1, 100)
y_line = (y1 - y0)/(x1 - x0) * (x_line - x0) + y0
# رسم الخط والنقاط
plt.figure(figsize=(6,3))
plt.plot(x_line, y_line, 'r--', label='Real Line')
plt.scatter([p[0] for p in points], [p[1] for p in points],
            color='blue', s=80, label='Bresenham Proints')
plt.grid(True)
plt.axis('equal')  # توحيد مقياس المحورين
plt.title("Bresenham Line")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()
plt.show()
```

3- خوارزمية رسم الدائرة Midpoint Circle Algorithm

تهدف هذه الخوارزمية إلى رسم دائرة باستخدام نقاط البكسل فقط، دون الحاجة لاستخدام دوال رياضية بهدف تقليل عدد العمليات الحسابية، تعتمد الخوارزمية على تماثل الدائرة، وبالتالي يمكن رسم 8 نقاط متماثلة من خلال حساب نقطة واحدة فقط.

معادلة الدائرة: $x^2 + y^2 = r^2$

تُستخدم دالة القرار لتحديد البكسل الذي سيتم رسمه في كل خطوة

خطوات الخوارزمية

1. التهيئة

- نبدأ من النقطة (0، r)
- القيمة الأولية لدالة القرار: $d = 1 - r$

2. التكرار، في كل خطوة،

- إذا كانت $d < 0$: نختار البكسل E (الشرق)
  - نعدل القيمة $d = d + 2x + 3$
- إذا كانت $d \geq 0$: نختار البكسل SE (الجنوب الشرقي)
  - نعدل الفيمة $d = d + 2(x - y) + 5$
  - $y = y - 1$
- $x = x + 1$

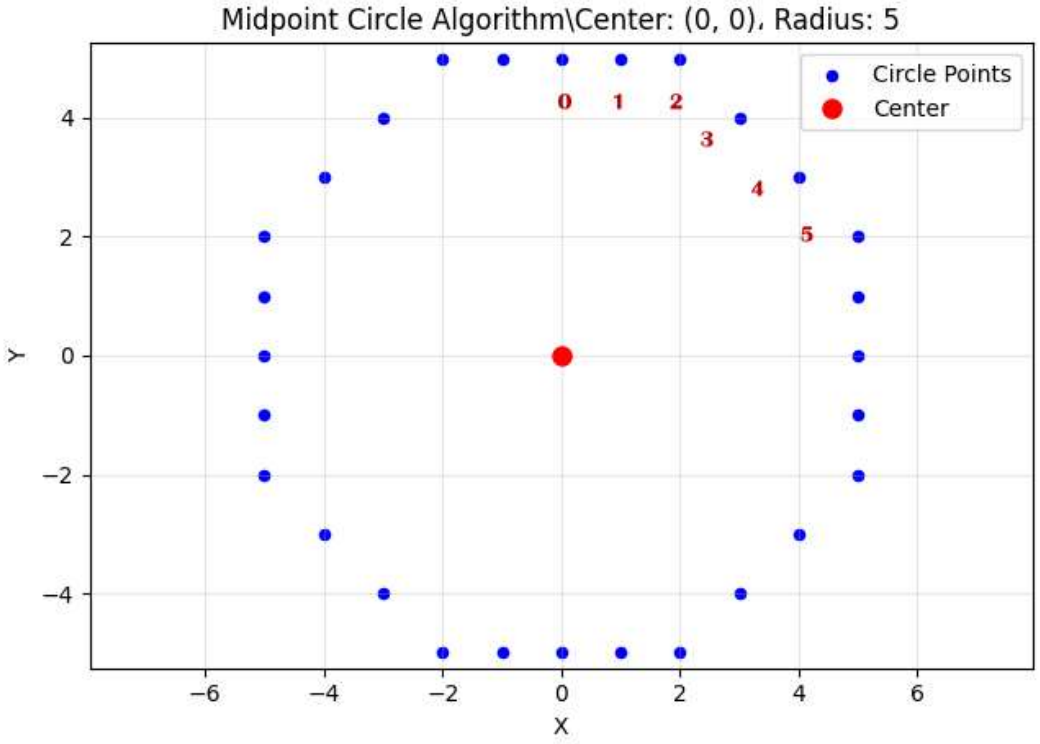
3. التماثل

- لكل نقطة (x, y) محسوبة، نرسم 8 نقاط متماثلة:
  - (x, y), (−x, y), (x, −y), (−x, −y)
  - (y, x), (−y, x), (y, −x), (−y, −x)

عندها نكون قد أكملنا الربع الأول، يكون قد تم توليد جميع النقاط الأخرى بالتماثل.

مثال تطبيقي (r = 5):

الخطوة	x	y	d	الشرط	الإجراء	القرار الجديد d	النقاط المضافة (x,y)
0	0	5	-4	-	البداية	-4	(0,5), (0,-5), (5,0), (-5,0)
1	1	5	-4	d < 0	اختيار E	-4 + 2×1 + 3 = 1	(1,5), (-1,5), (1,-5), (-1,-5), (5,1), (-5,1), (5,-1), (-5,-1)
2	2	5	1	d ≥ 0	اختيار SE	1 + 2×(2-5) + 5 = 0	(2,5), (-2,5), (2,-5), (-2,-5), (5,2), (-5,2), (5,-2), (-5,-2)
3	3	4	0	d ≥ 0	اختيار SE	0 + 2×(3-4) + 5 = 3	(3,4), (-3,4), (3,-4), (-3,-4), (4,3), (-4,3), (4,-3), (-4,-3)
4	4	3	3	d ≥ 0	اختيار SE	3 + 2×(4-3) + 5 = 10	(4,3), (-4,3), (4,-3), (-4,-3), (3,4), (-3,4), (3,-4), (-3,-4)
5	5	2	10	d ≥ 0	اختيار SE	10 + 2×(5-2) + 5 = 21	(5,2), (-5,2), (5,-2), (-5,-2), (2,5), (-2,5), (2,-5), (-2,-5)



دالة لتنفيذ المثال السابق بلغة بايثون:

```
import matplotlib.pyplot as plt

def midpoint_circle(xc, yc, r):
    points = []
    x = 0
    y = r
    d = 1 - r  # معلمة القرار الأولية

    # دالة مساعدة لإضافة النقاط الثمانية المتماثلة
    def plot_symmetric_points(x, y):
        points.extend([
            (xc + x, yc + y), (xc - x, yc + y),
            (xc + x, yc - y), (xc - x, yc - y),
            (xc + y, yc + x), (xc - y, yc + x),
            (xc + y, yc - x), (xc - y, yc - x)
        ])

    # إضافة النقاط الأولية
    plot_symmetric_points(x, y)

    # معالجة الربع الأول فقط (من 0° إلى 45°)
    while x < y:
        if d < 0:
            # تحريك أفقيًا (شرقًا)
            d += 2 * x + 3
            x += 1
        else:
            # تحريك قطريًا (جنوب شرق)
            d += 2 * (x - y) + 5
            x += 1
            y -= 1
        plot_symmetric_points(x, y)

    # إزالة التكرارات وترتيب النقاط
    unique_points = sorted(set(points))
    return unique_points
# ============== تشغيل المثال ==============
# إعدادات الدائرة
center_x, center_y = 0, 0
radius = 15

# توليد نقاط الدائرة
circle_points = midpoint_circle(center_x, center_y, radius)
```

```
# فصل الإحداثيات
x_coords, y_coords = zip(*circle_points)
# matplotlib رسم الدائرة باستخدام
plt.figure(figsize=(8, 8))
plt.scatter(x_coords, y_coords, color='blue', s=20, label='Circle Points')
plt.plot(center_x, center_y, 'ro', markersize=8, label='Center')
plt.title(f'Midpoint Circle Algorithm\Center: ({center_x}, {center_y}) ،Radius:
{radius}')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.legend()
plt.show()
```

4- خوارزميات ملء المضلعات Scanline Fill Algorithm

خوارزمية Scanline Fill (الملء بمسح الخطوط) تستخدم هذه الخوارزمية لملء المضلعات في رسومات الحاسوب، تعتمد على فكرة مسح الشكل خطاً تلو الآخر، وتحديد نقاط التقاطع مع حواف المضلع.

تعتمد الخوارزمية على: مسح الخطوط الأفقية من الأعلى إلى الأسفل، تحديد نقاط التقاطع مع حواف المضلع، تعبئة الأجزاء الداخلية :بين نقاط التقاطع.

خطوات الخوارزمية

1. إيجاد حواف المضلع
   - تخزين جميع حواف المضلع في جدول الحواف (Edge Table – ET) .
2. إنشاء جدول الحواف النشطة (Active Edge Table – AET)
   - يحتوي على الحواف التي تتقاطع مع خط المسح الحالي
3. عملية المسح
   - من أعلى نقطة (أصغر y) إلى أسفل نقطة (أكبر y).
   - لكل خط مسح:
     - إضافة الحواف التي تبدأ عند هذا الخط إلىAET
     - إزالة الحواف التي تنتهي عند هذا الخط منAET
     - ترتيب الحواف في AET حسب إحداثيات x
     - تعبئة البكسلات بين أزواج نقاط التقاطع

مثال تطبيقي:

مطلوب تعبئة المضلع المحدد بالإحداثيات التالية:

A(2,1), B(6,1), C(8,5), D(6,8), E(2,8), F(4,5)

الخطوة 1: بناء جدول الحواف ET ، نرتّب الحواف تصاعدياً حسب قيمة ymin

Edge	from → to	ymin	ymax	x @ ymin	inv_slope (dx/dy)
BC	(6,1)→(8,5)	1	5	6	+0.5
FA	(2,1)→(4,5)	1	5	2	+0.5
CD	(8,5)→(6,8)	5	8	8	−0.6666667
EF	(4,5)→(2,8)	5	8	4	−0.6666667

الخطوة 2: حساب نقاط التقاطع مع كل عملية مسح

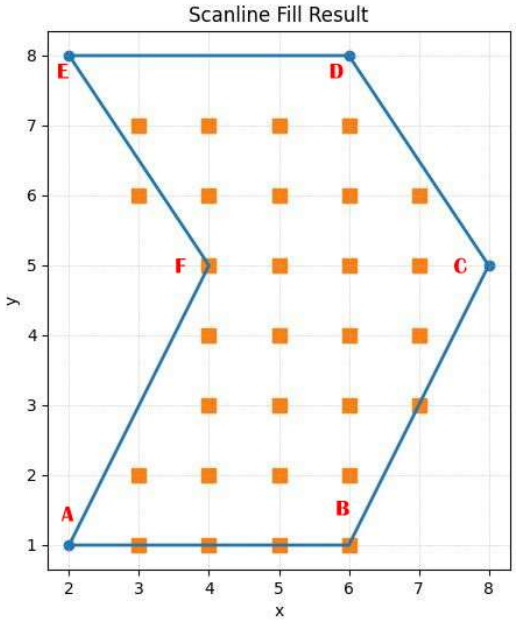
تحسب نقاط التقاطع وفق العلاقة التالية: Cross x = **x @ ymin** + **inv_slope** × (y_scan − **ymin**)

عند خط المسح y_scan = 1.5 تكون نقاط التقاطع:

FA x = 2 + 0.5 ×(1.5 - 1) = 2.25
BC x = 6 + 0.5 ×(1.5 - 1) = 6.25

وبالتالي نقاط التعبئة هي : 3، 4، 5، 6

نحسب باقي النقاط بنفس الطريقة، لنحصل على جدول الحواف النشطة AET:



y	y_scan	تقاطع X عند	بكسلات مملوءة y على
1	1.5	2.25 , 6.25	3,4,5,6
2	2.5	2.75 , 6.75	3,4,5,6
3	3.5	3.25 , 7.25	4,5,6,7
4	4.5	3.75 , 7.75	4,5,6,7
5	5.5	3.67 , 7.67	4,5,6,7
6	6.5	3.0 , 7.0	3,4,5,6,7
7	7.5	2.33 , 6.33	3,4,5,6

دالة لتنفيذ المثال السابق بلغة بايثون:

```
import math
import matplotlib.pyplot as plt

def scanline_fill(polygon):

    #حساب أعلى وأسفل y للمضلّع
    ys = [p[1] for p in polygon]
    y_min = math.floor(min(ys))
    y_max = math.ceil(max(ys))

    filled_pixels = []

    #نعمل مسحًا لكل صف y (نأخذ نقطة المسح عند y + 0.5)
    for y in range(y_min, y_max):
        y_scan = y + 0.5

        # جمع نقاط تقاطع الخط مع حواف المضلّع
        x_intersections = []
        n = len(polygon)
        for i in range(n):
            x1, y1 = polygon[i]
            x2, y2 = polygon[(i + 1) % n]

            # تجاهل الحواف الأفقية
            if y1 == y2:
                continue

            # نتحقق إن كانت نقطة المسح تقع داخل مدى y
            y_min_edge = min(y1, y2)
            y_max_edge = max(y1, y2)
            if (y_scan >= y_min_edge) and (y_scan < y_max_edge):
                # نحسب إحداثيات x للتقاطع
                t = (y_scan - y1) / (y2 - y1)
                x_int = x1 + t * (x2 - x1)
                x_intersections.append(x_int)

        # إذا لم يوجد تقاطعات، نتخطّى هذا الصف
        if not x_intersections:
            continue

        # نرتّب نقاط التقاطع ونأخذها زوجًا زوجًا
        x_intersections.sort()
```

```
        # يجب أن تكون التقطعات زوجية
        for i in range(0, len(x_intersections), 2):
            x_left = x_intersections[i]
            # قد يحدث، لأسباب رقمية، أن لا يوجد زوج كامل؛ نتفادى ذلك
            if i+1 >= len(x_intersections):
                break
            x_right = x_intersections[i+1]

            # نحول النطاق العائم إلى بكسلات صحيحة
            x_start = math.ceil(x_left)
            x_end = math.floor(x_right)

            # y نضيف كل بكسيل داخل النطاق [x_start, x_end] عند الصف
            for x in range(x_start, x_end + 1):
                filled_pixels.append((x, y))

    return filled_pixels
```