Computer Graphics

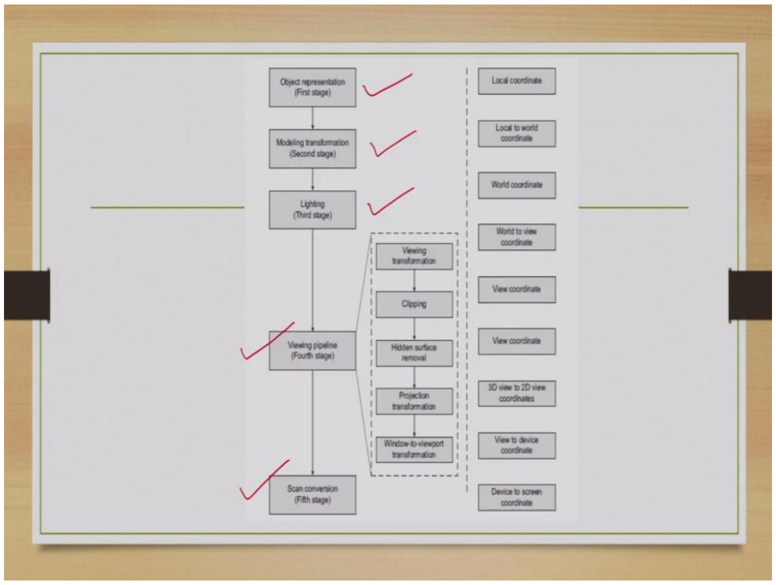
Professor Dr. Samit Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Lecture - 26

Scan Conversion of Basic Shapes - 2

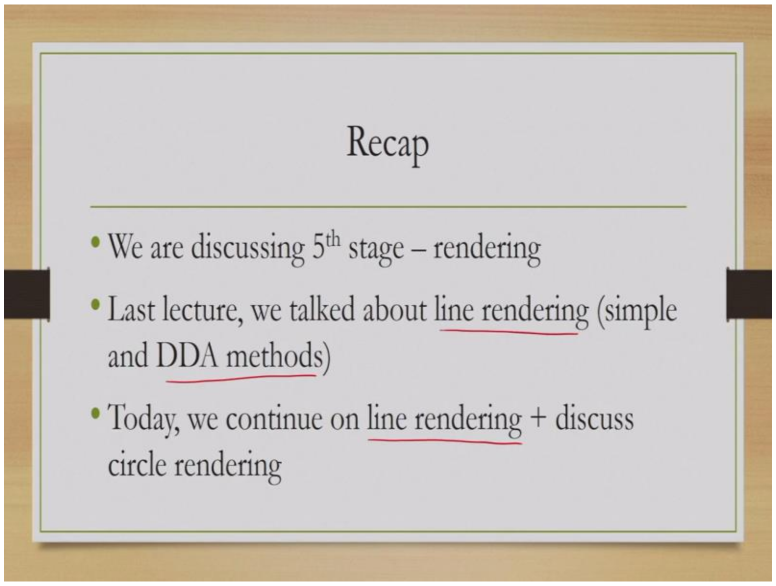
Hello and welcome to lecture number 26 in the course Computer Graphics, we will continue our discussion on the graphics pipeline. For a quick recap, let us just go through the stages.

(Refer Slide Time: 00:47)


Object representation (First stage)
Modeling transformation (Second stage)
Lighting (Third stage)
Viewing transformation
Clipping
Hidden surface removal
Projection transformation
Window-to-viewport transformation
Viewing pipeline (Fourth stage)
Scan conversion (Fifth stage)
Local coordinate
Local to world coordinate
World coordinate
World to view coordinate
View coordinate
View coordinate
3D view to 2D view coordinates
View to device coordinate
Device to screen coordinate

So, we have already discussed the first stage that is object representation, second stage modeling transformation, third stage lighting, fourth stage viewing pipeline, and the only stage that is remaining is the fifth stage scan conversion. We are currently discussing the fifth stage.

(Refer Slide Time: 01:14)



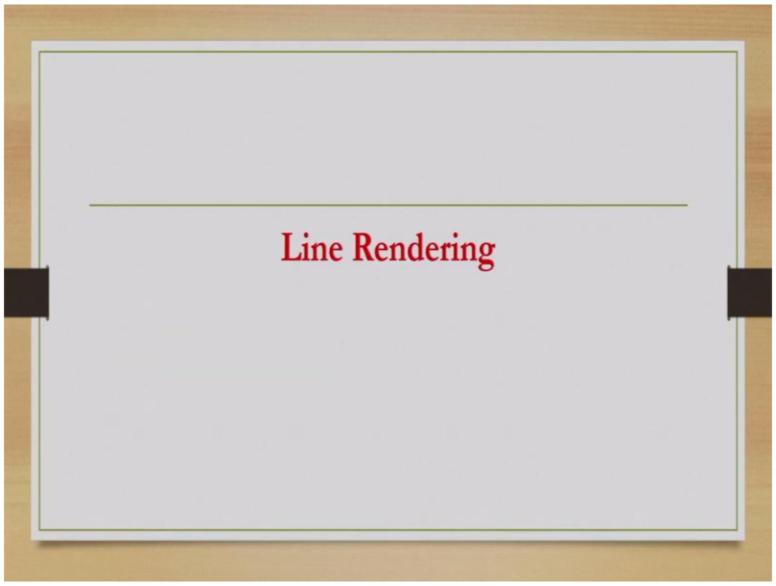
Recap

- We are discussing 5th stage – rendering
- Last lecture, we talked about line rendering (simple and DDA methods)
- Today, we continue on line rendering + discuss circle rendering

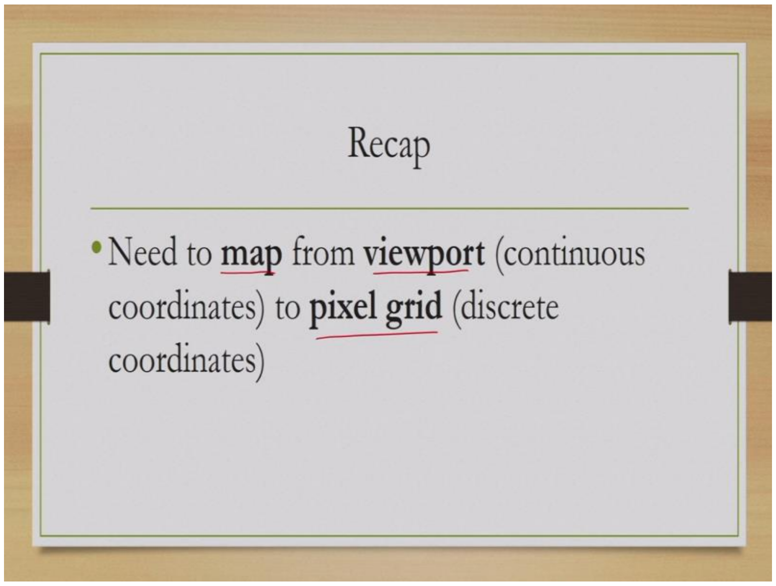
In the last lecture, we talked about rendering of lines, which is part of the fifth stage. And there we talked about a very intuitive approach as well as a slightly improved approach that is the DDA methods. Today, we will continue our discussion on line rendering, where we will talk about even better approach. And also we will discuss rendering of circles.

Now, before we go into the discussion on a better line rendering approach, let us quickly recap what we have seen in the previous lecture on line rendering.

(Refer Slide Time: 02:02)



Line Rendering

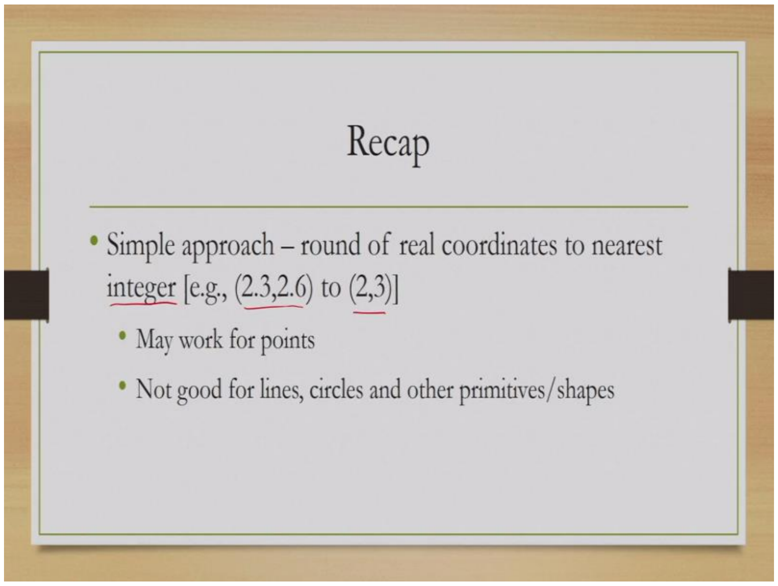


Recap

- Need to **map** from **viewport** (continuous coordinates) to **pixel grid** (discrete coordinates)

So the idea was to map a description from viewport to a pixel grid. That is, of course, the objective of the fifth stage.

(Refer Slide Time: 02:20)

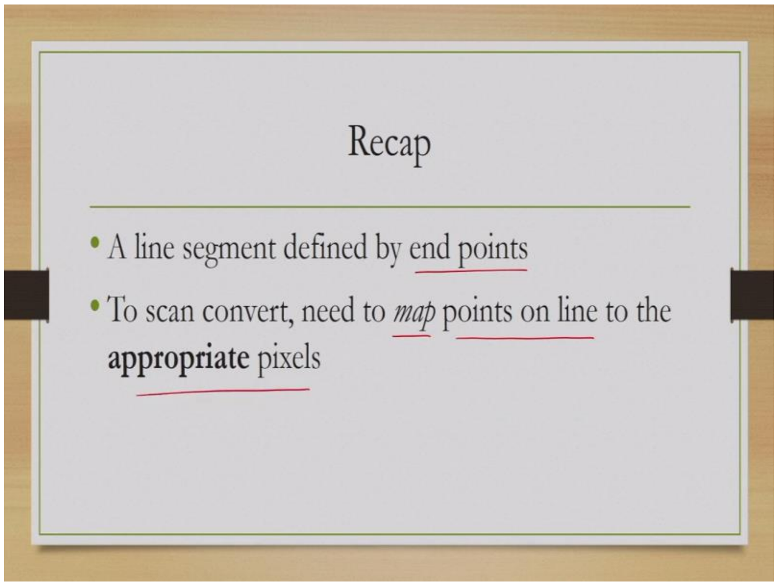


Recap

- Simple approach – round of real coordinates to nearest integer [e.g., (2.3,2.6) to (2,3)]
  - May work for points
  - Not good for lines, circles and other primitives/shapes

In order to do that, simplest approach is just to round off the real coordinates to the nearest integers, which are pixels, for example, from (2.3, 2.6) to (2, 3). Now, this is good for points but for mapping lines or circles or other primitive shapes, this may not be good.

(Refer Slide Time: 02:49)

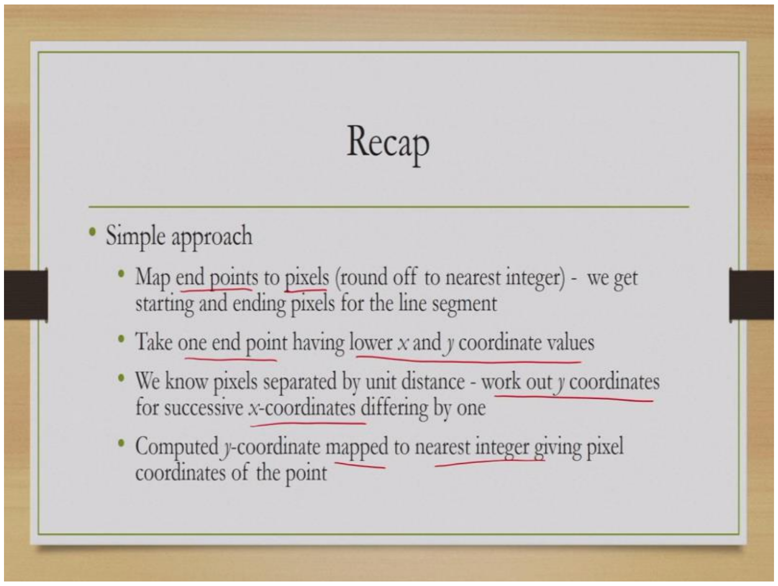


Recap

- A line segment defined by end points
- To scan convert, need to *map* points on line to the **appropriate** pixels

And for line what we did? So we first assume that a line segment is defined by the endpoints. And our objective is to map all the points that are on the line to the appropriate pixels.

(Refer Slide Time: 03:08)

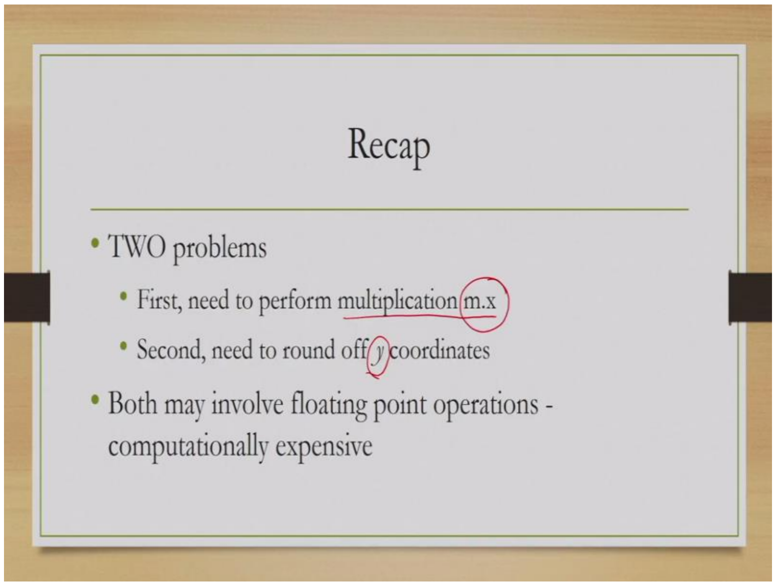


Recap

- Simple approach
  - Map end points to pixels (round off to nearest integer) - we get starting and ending pixels for the line segment
  - Take one end point having lower *x* and *y* coordinate values
  - We know pixels separated by unit distance - work out *y* coordinates for successive *x*-coordinates differing by one
  - Computed *y*-coordinate mapped to nearest integer giving pixel coordinates of the point

The straightforward approach that we discussed is that first we map the end points to pixels, then we start with one endpoint which is having the lower x and y coordinate values, then we work out y-coordinate values for successive x-coordinates, where the x-coordinates differ by 1 because we are talking pixel grids. And then, this y values that we computed are mapped to the nearest integer thereby getting the pixels.

(Refer Slide Time: 03:55)

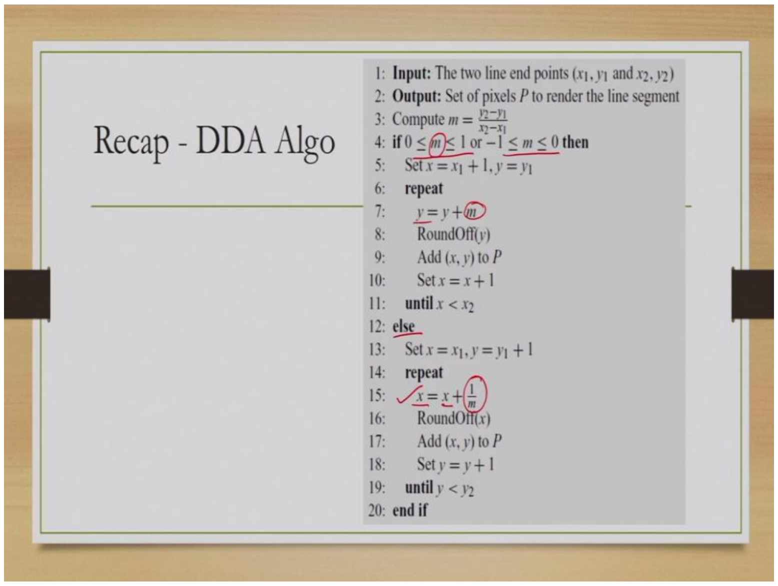


Recap

- TWO problems
  - First, need to perform multiplication m.x
  - Second, need to round off y coordinates
- Both may involve floating point operations - computationally expensive

Now, this approach has two problems. First, we require multiplication which is a floating-point operation. And secondly, we require rounding off which is also a floating-point operation. Now, these floating-point operations are computationally expensive and may result in slower rendering of lines.

(Refer Slide Time: 04:20)

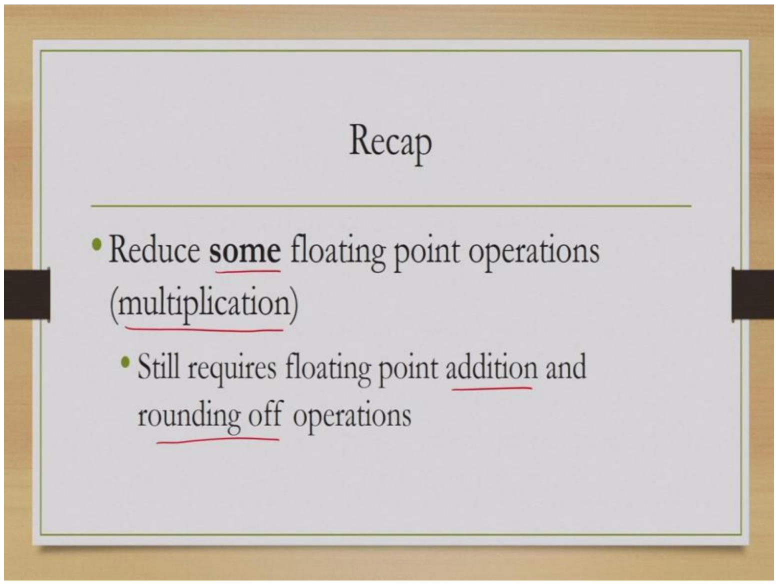


Recap - DDA Algo

```
Input: The two line end points ($x_1, y_1$ and $x_2, y_2$)
Output: Set of pixels $P$ to render the line segment
Compute $m = \frac{y_2-y_1}{x_2-x_1}$
if $0 \le m \le 1$ or $-1 \le m \le 0$ then
    Set $x = x_1 + 1, y = y_1$
    repeat
        $y = y + m$
        RoundOff($y$)
        Add $(x, y)$ to $P$
        Set $x = x + 1$
    until $x < x_2$
else
    Set $x = x_1, y = y_1 + 1$
    repeat
        $x = x + \frac{1}{m}$
        RoundOff($x$)
        Add $(x, y)$ to $P$
        Set $y = y + 1$
    until $y < y_2$
end if
```

To improve, we discussed one incremental approach. There we did not go for multiplication, instead we used addition. So to compute y, we simply added this m value to the current value, or to compute x, new x, we simply added this 1 by m value to the current x value. Now when to choose whether to compute x or y, that depends on the slope. So if the m value is within this range, then we compute y given x, otherwise, we compute x given y using the line equation.

(Refer Slide Time: 05:18)

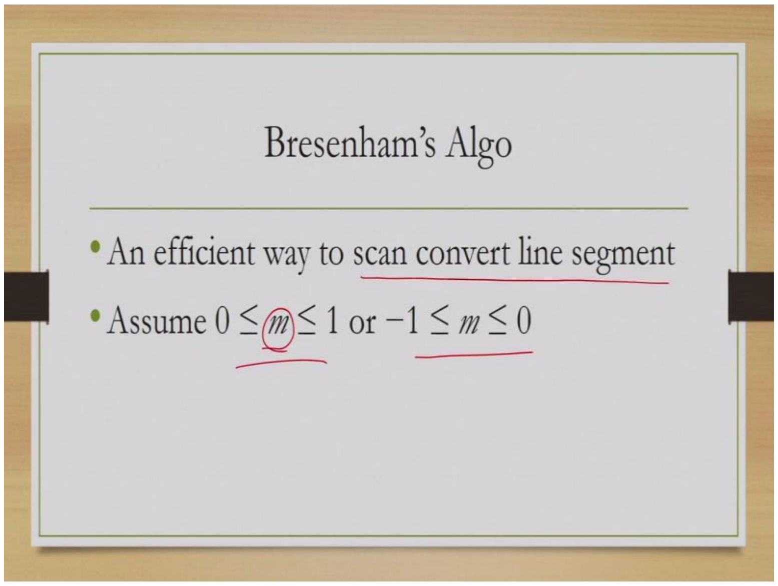


Recap

- Reduce **some** floating point operations (multiplication)
  - Still requires floating point addition and rounding off operations

Now, the DDA can reduce some floating-point operations as we have discussed, particularly multiplications. However, it still requires other floating-point operations, namely additions and rounding off. So it is still not completely efficient so to speak, and we require a better approach. One such approach is given by Bresenham's algorithm.

(Refer Slide Time: 06:03)

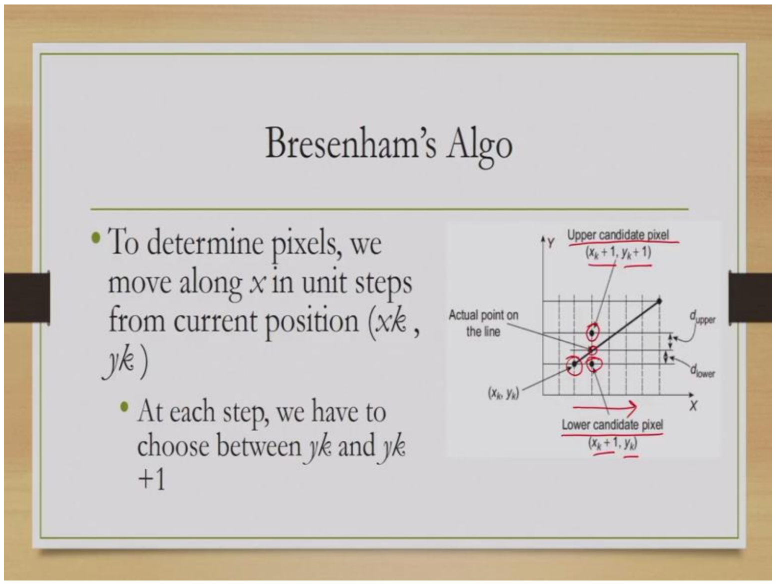


Bresenham's Algo

- An efficient way to scan convert line segment
- Assume $0 \le m \le 1$ or $-1 \le m \le 0$

Now, this is an efficient way to scan convert line segments and we will discuss the algorithm assuming m to be within these ranges. That means we will concentrate on computing y value given the x value.

(Refer Slide Time: 06:26)



Bresenham's Algo

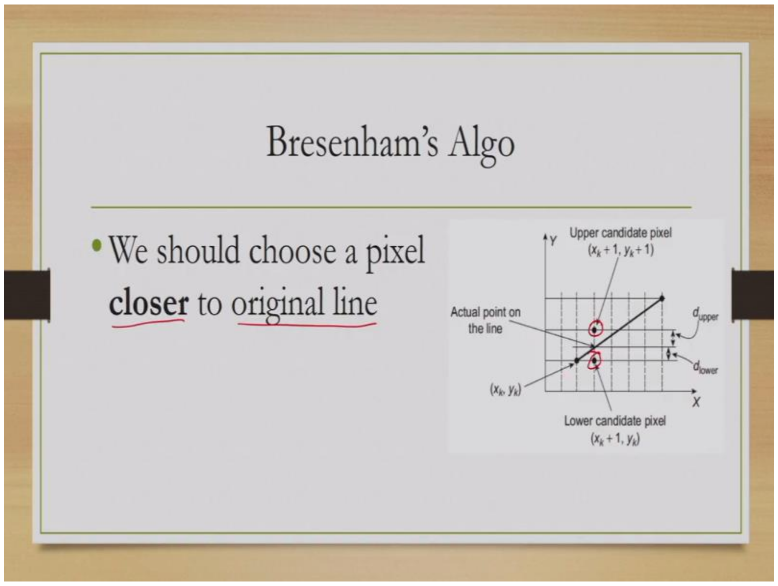
- To determine pixels, we move along x in unit steps from current position (xk , yk)
  - At each step, we have to choose between yk and yk +1

Y
Upper candidate pixel
$(x_k + 1, y_k + 1)$
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
X
Lower candidate pixel
$(x_k + 1, y_k)$

Now, let us try to understand the situation. Suppose, this is the actual point on the line and we are moving along the x-direction, the current position is given by this point (x_k, y_k). Now, the actual point on the line is a floating-point number real number, so we need to map it to the nearest pixel grid point.

Now, there are two potential candidates for that, one is this pixel or the upper candidate pixel that is (x_k+1, y_k+1), and the other one is the lower candidate pixel that is (x_k+1, y_k), and we have to choose 1 of those. How to choose that?

(Refer Slide Time: 07:24)



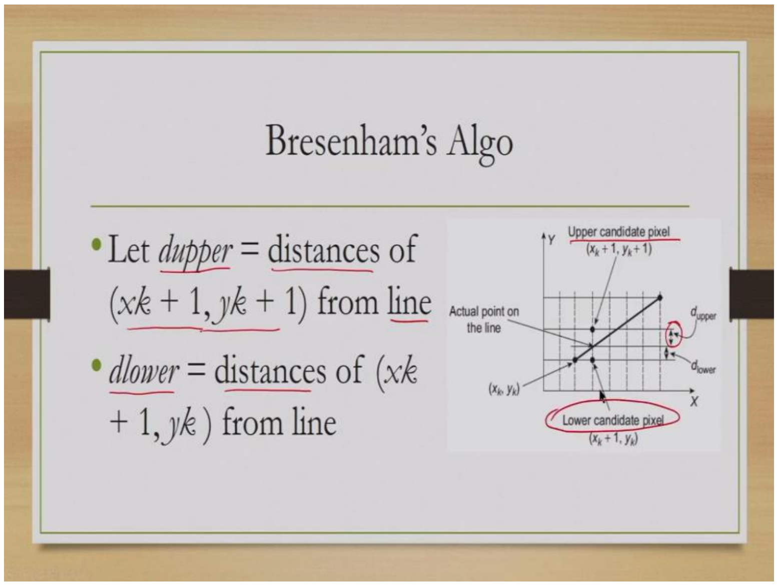
Bresenham's Algo

- We should choose a pixel **closer** to original line

Upper candidate pixel
(x_k+1, y_k+1)
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
Y
X
Lower candidate pixel
(x_k+1, y_k)

Our objective is to choose a pixel that is closer with respect to the other pixel to the original line. So between these two pixels, we have to decide which one is closer to the original line and choose that pixel.

(Refer Slide Time: 07:51)



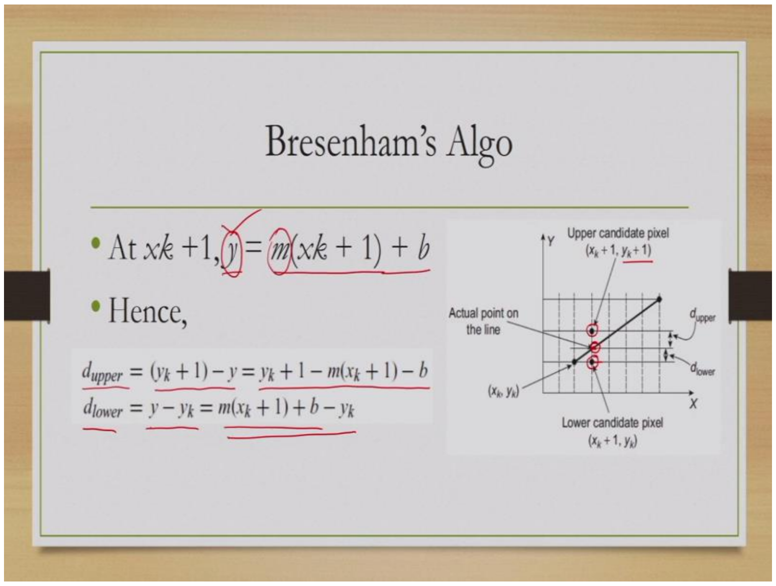
Bresenham's Algo

- Let *dupper* = distances of (*xk* + 1, *yk* + 1) from line
- *dlower* = distances of (*xk* + 1, *yk*) from line

Upper candidate pixel
(x_k+1, y_k+1)
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
Y
X
Lower candidate pixel
(x_k+1, y_k)

Let us denote by d_{upper}, the distance of the pixel (x_k+1), (y_k+1) from the line that is the upper candidate pixel from the line as shown here. Similarly, d lower indicates the distance of the lower candidate pixel from the line.

(Refer Slide Time: 08:28)



Bresenham's Algo

- At $xk+1$, $y = m(xk+1) + b$
- Hence,

$d_{upper} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$

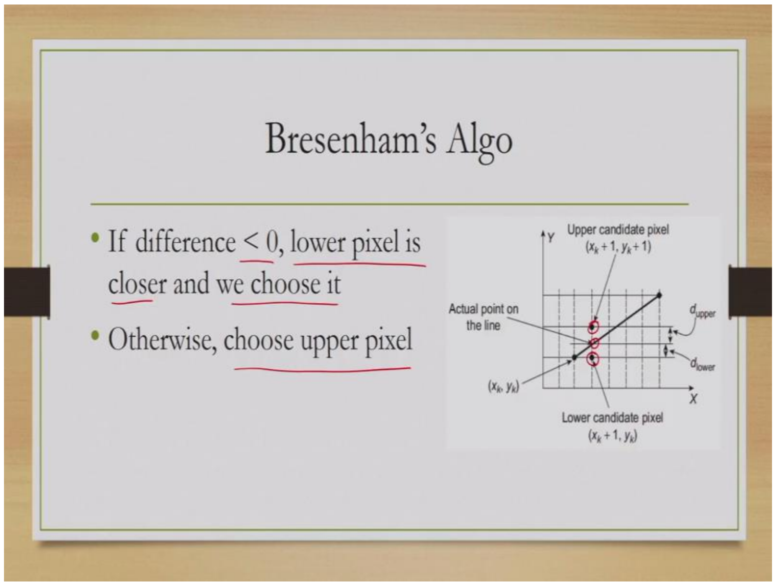
$d_{lower} = y - y_k = m(x_k + 1) + b - y_k$

Upper candidate pixel
(x_k+1, y_k+1)
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
Y
X
Lower candidate pixel
(x_k+1, y_k)

Now, at (x_k+1), that is at these points y is given by this expression using the line equation, where m is the slope. Then we can say that d upper can be given by ((y_k+1) – y), that is this value minus this y value, which is given here or this expression.

Similarly, d_{lower} can also be given as y minus y_k. As you can see here, this is the y value and this is the y_k value. So replacing the y from this equation, you can get this expression. Now, let us do some mathematical trick on these expressions.

(Refer Slide Time: 09:49)



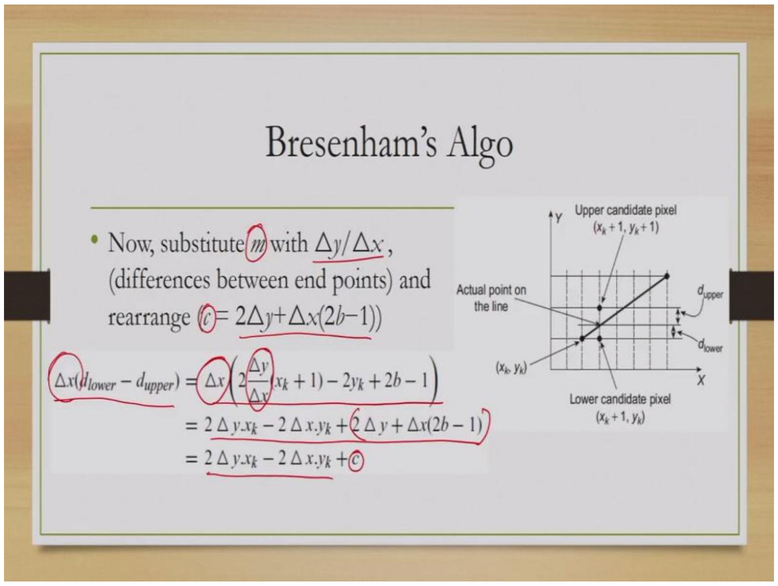
Bresenham's Algo

- If difference < 0, lower pixel is closer and we choose it
- Otherwise, choose upper pixel

Upper candidate pixel
(x_k+1, y_k+1)
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
Y
X
Lower candidate pixel
(x_k+1, y_k)

But before that, we should note that if the difference is less than 0 then the lower pixel is closer and we choose it, otherwise, we choose the upper pixel. Distance between the y values here, here, and here; the two distances that we have used in expressing d_{upper} and d_{lower}. If the difference is less than 0, then we choose the lower pixel because that point is closer to the line, otherwise, we choose the upper pixel.

(Refer Slide Time: 10:41)



Y
X
Upper candidate pixel
(x_k+1, y_k+1)
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
Lower candidate pixel
(x_k+1, y_k)

Bresenham's Algo

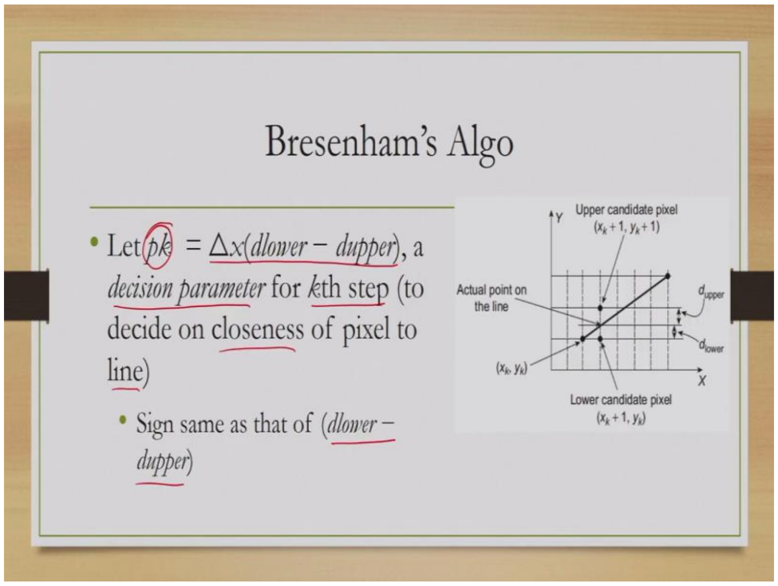
- Now, substitute m with $\Delta y/\Delta x$, (differences between end points) and rearrange ($c = 2\Delta y+\Delta x(2b-1)$)

$$\Delta x(d_{lower} - d_{upper}) = \Delta x\left(2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1\right)$$
$$= 2\,\Delta\, y.x_k - 2\,\Delta\, x.y_k + 2\,\Delta\, y + \Delta x(2b - 1)$$
$$= 2\,\Delta\, y.x_k - 2\,\Delta\, x.y_k + c$$

Now, let us substitute m with this ratio, Δ_y/Δ_x, where Δ_y is the y coordinate difference between the endpoints and Δ_x is the x coordinate difference between the endpoints. And then we rearrange and replace this expression with c, which is a constant term. As you can see here, all are constants.

Then what do we get? This term to be equal to this term, both sides we have multiplied by Δ_x and replace m with these expressions. Rearranging and expanding, we get this, then we replace this constant term here with c to get this expression. This is a simple manipulation of the terms.

(Refer Slide Time: 11:55)



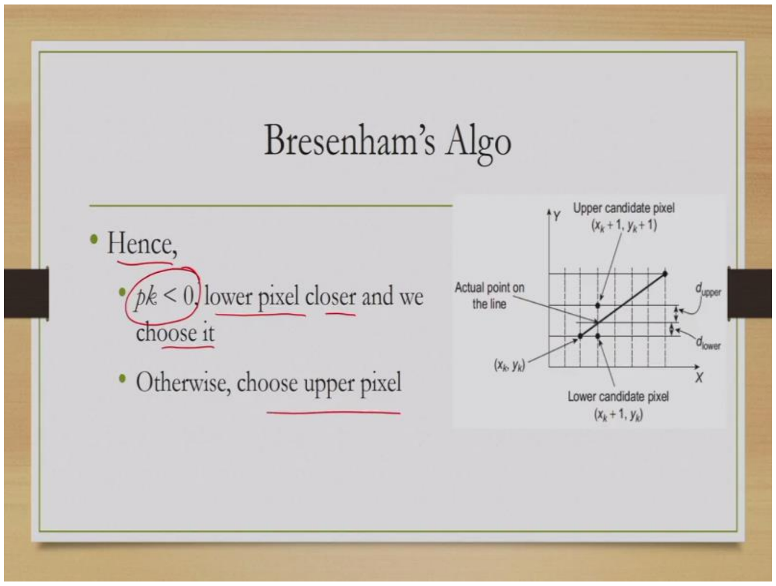
Bresenham's Algo

- Let $pk = \Delta x(dlower - dupper)$, a *decision parameter* for *k*th step (to decide on closeness of pixel to line)
  - Sign same as that of $(dlower - dupper)$

Upper candidate pixel
$(x_k + 1, y_k + 1)$
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
Lower candidate pixel
$(x_k + 1, y_k)$
Y
X

Now, let us denote the left-hand side by p_k, we call it a decision parameter for the k^{th} step. Now, this parameter is used to decide the closeness of a pixel to the line. Now, its sign will be same as that of the sign of the difference d_{lower} - d_{upper}.

(Refer Slide Time: 12:28)



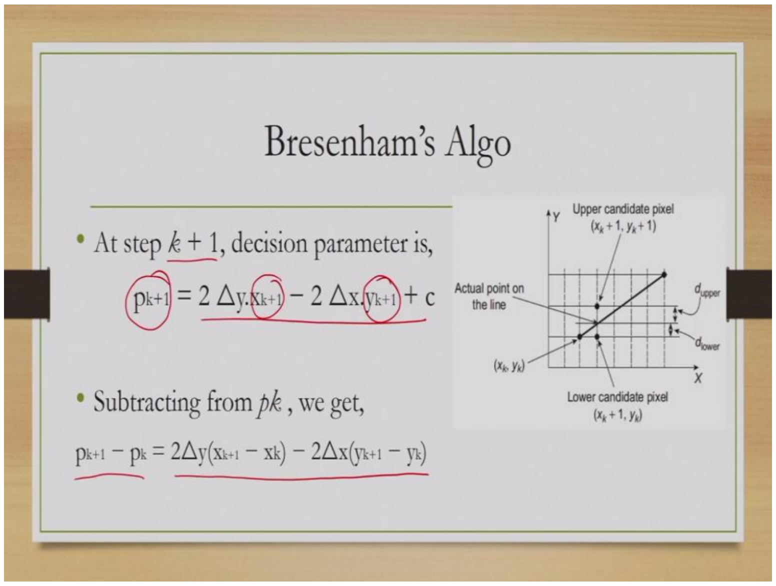
Bresenham's Algo

- Hence,
  - $pk < 0$, lower pixel closer and we choose it
  - Otherwise, choose upper pixel

Upper candidate pixel
$(x_k + 1, y_k + 1)$
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
Lower candidate pixel
$(x_k + 1, y_k)$
Y
X

Thus, if p_k<0, then this lower pixel is closer to the line and we choose it, otherwise, we choose the upper pixel.

(Refer Slide Time: 12:47)



Bresenham's Algo

- At step $k + 1$, decision parameter is,

$$p_{k+1} = 2\,\Delta y . x_{k+1} - 2\,\Delta x . y_{k+1} + c$$

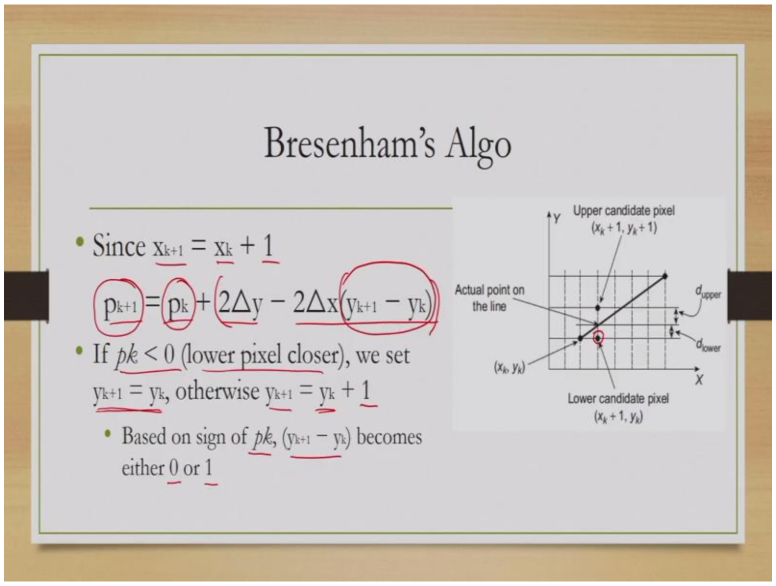
- Subtracting from pk, we get,

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

Y
Upper candidate pixel
$(x_k + 1, y_k + 1)$
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
X
Lower candidate pixel
$(x_k + 1, y_k)$

So that is at step k. Now, at step k+1 that is the next step, we get p_{k+1}. Now, that is essentially given by this expression where we replaced x_k with x_{k+1} and y_k with y_{k+1}. These two terms we replaced with the next term. Then we take a difference between the two, p_{k+1} - p_k, which gives us this expression.

(Refer Slide Time: 13:26)



Bresenham's Algo

- Since $x_{k+1} = x_k + 1$

$$p_{k+1} = p_k + (2\Delta y - 2\Delta x(y_{k+1} - y_k))$$

- If $pk < 0$ (lower pixel closer), we set $y_{k+1} = y_k$, otherwise $y_{k+1} = y_k + 1$
    - Based on sign of pk, $(y_{k+1} - y_k)$ becomes either 0 or 1

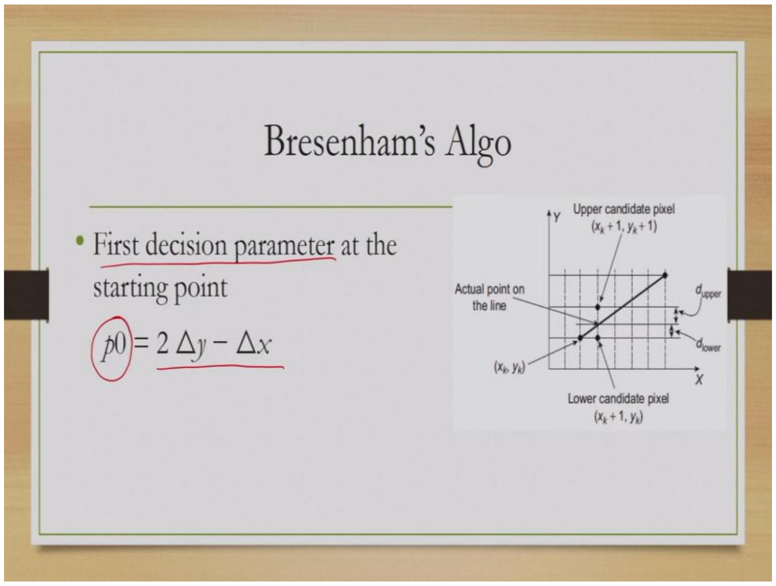
Y
Upper candidate pixel
$(x_k + 1, y_k + 1)$
Actual point on the line
d_{upper}
d_{lower}
(x_k, y_k)
X
Lower candidate pixel
$(x_k + 1, y_k)$

Now, we know because we are dealing with a pixel grid that x_{k+1} is essentially x_k + 1. So we can rearrange and rewrite the expression as p_{k+1} = p_k + $2\Delta_y$ – {this term}. That means the decision variable at k+1th step is given by the decision variable at kth step plus a term; this and that term.

Now, if $p_k<0$, that is the lower pixel is closer, then we set $y_{kp+1}=y_k$, otherwise, we set $y_{k+1}=y_k+1$. Thus based on the sign of p_k, this term becomes either 0 or 1. So you can see the physical significance from this figure. So if $p_k<0$ that means in the current stage, lower pixel is closer. That means, we have chosen this one.

Then in the next stage, we have to choose $y_{k+1} = y_k$ that is the lower pixel. If that is not the case, then we have to choose $y_{k+1} = y_k +1$ that is the upper pixel. So depending on the sign of p_k, this term $y_{k+1} - y_k$ turns out to be either 0 or 1. If $p_k<0$ then this is 0; if $p_k \nless 0$ then it is 1.

(Refer Slide Time: 15:50)



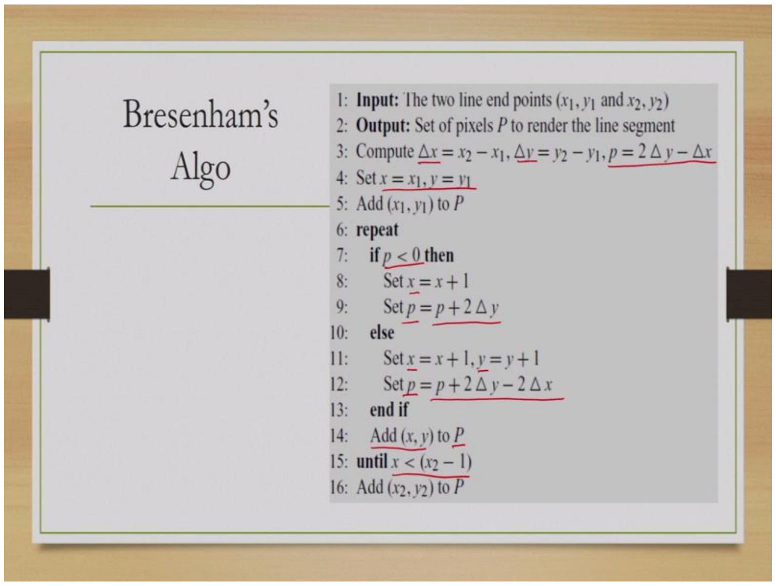
Bresenham's Algo

- First decision parameter at the starting point

  $p0 = 2\,\Delta y - \Delta x$

So, where we start, then we have to decide on the first decision parameter and that we call p0, which is given by twice delta y minus delta x. This value we calculate and then we continue.

(Refer Slide Time: 16:13)



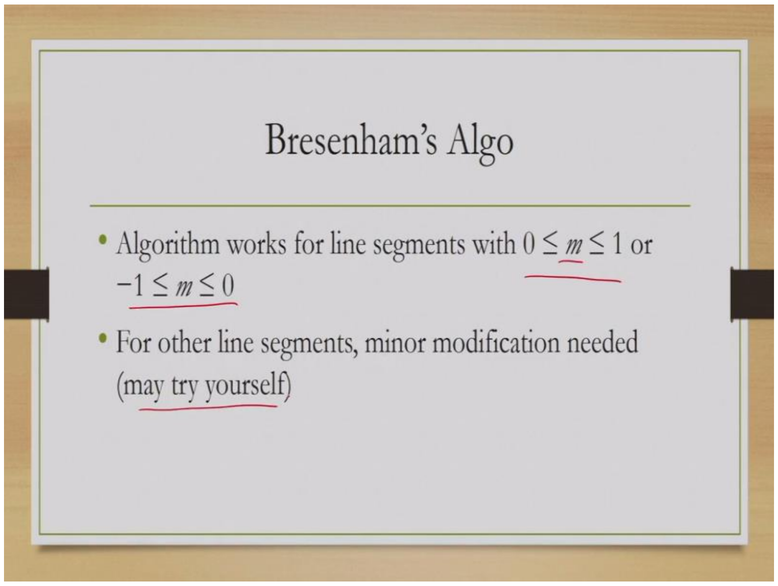
Bresenham's Algo

```
1: Input: The two line end points ($x_1, y_1$ and $x_2, y_2$)
2: Output: Set of pixels $P$ to render the line segment
3: Compute $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$, $p = 2\Delta y - \Delta x$
4: Set $x = x_1$, $y = y_1$
5: Add $(x_1, y_1)$ to $P$
6: repeat
7:   if $p < 0$ then
8:     Set $x = x + 1$
9:     Set $p = p + 2\Delta y$
10:  else
11:    Set $x = x + 1$, $y = y + 1$
12:    Set $p = p + 2\Delta y - 2\Delta x$
13:  end if
14:  Add $(x, y)$ to $P$
15: until $x < (x_2 - 1)$
16: Add $(x_2, y_2)$ to $P$
```

So the overall algorithm is given here. We first compute these differences between the endpoints and the first decision parameter. Then we go inside a loop till we reach the end point. We start with one end point and till we reach the other end point, we continue in the loop.

Now, if $p<0$, we set that difference to be 0 and then update p as $p+2\Delta_y$. If $p\geq0$, then we update p as given in this expression and then add the corresponding x-y value into the set of pixels that is the output of the algorithm. So, depending on the decision value, we choose a pixel and add it to the set of pixels.

(Refer Slide Time: 17:31)

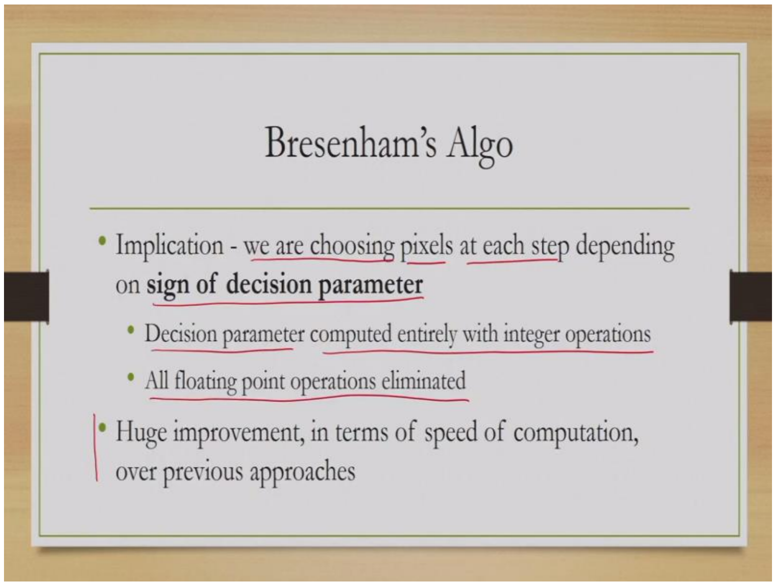


Bresenham's Algo

- Algorithm works for line segments with $0 \leq m \leq 1$ or $-1 \leq m \leq 0$
- For other line segments, minor modification needed (may try yourself)

Now, here we assume that m is within this range. When m is outside this range, we have to modify this algorithm but that is a minor modification. So you may try it yourself.

(Refer Slide Time: 17:50)



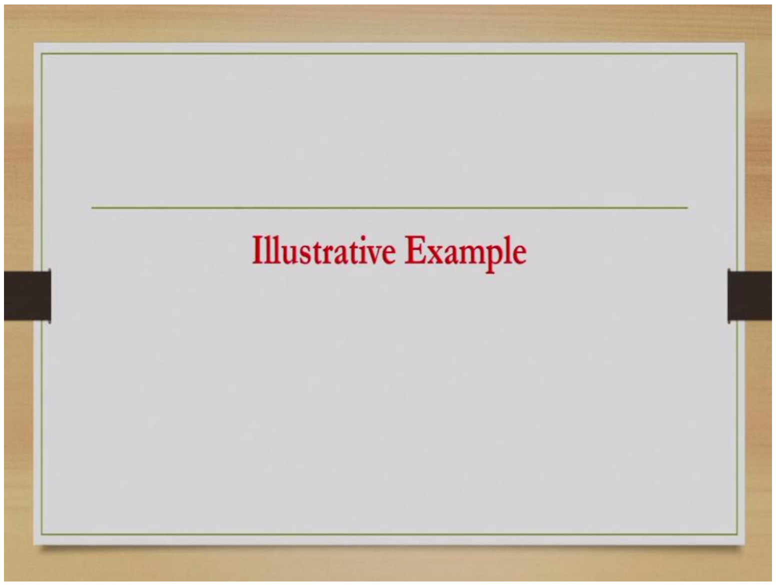
Bresenham's Algo

- Implication - we are choosing pixels at each step depending on **sign of decision parameter**
  - Decision parameter computed entirely with integer operations
  - All floating point operations eliminated
- Huge improvement, in terms of speed of computation, over previous approaches

So what is the advantage of this algorithm? Here if you note, we are choosing the pixels at each step depending on the sign of decision parameter, and the decision parameter is computed entirely with integer operations so there is no floating-point operation.

Thus we have eliminated all floating-point operations; additions, rounding off, as well as multiplications which is a huge improvement because, in reality, we need to render a large number of lines in a short span of time, a very short span of time. So there this saving is substantial. There are even better approaches but we will not discuss those any further.

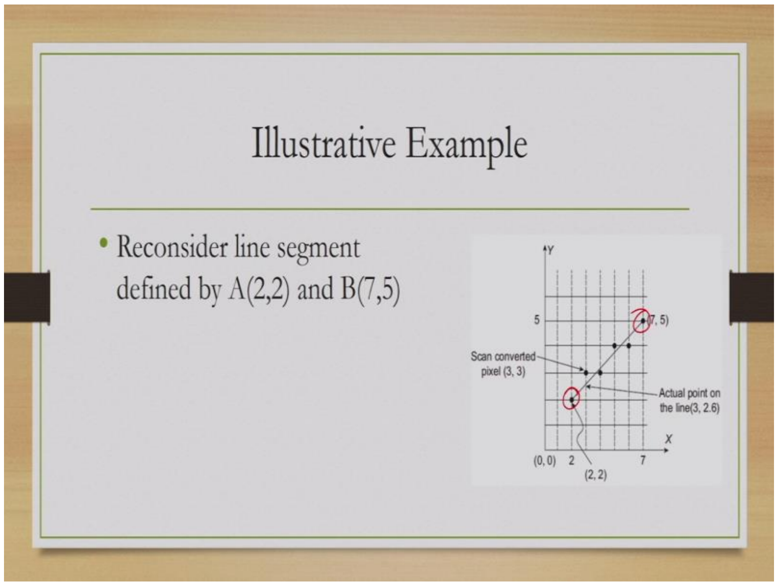
(Refer Slide Time: 18:48)



Illustrative Example

Now, let us try to understand the algorithm in terms of one example.

(Refer Slide Time: 18:54)

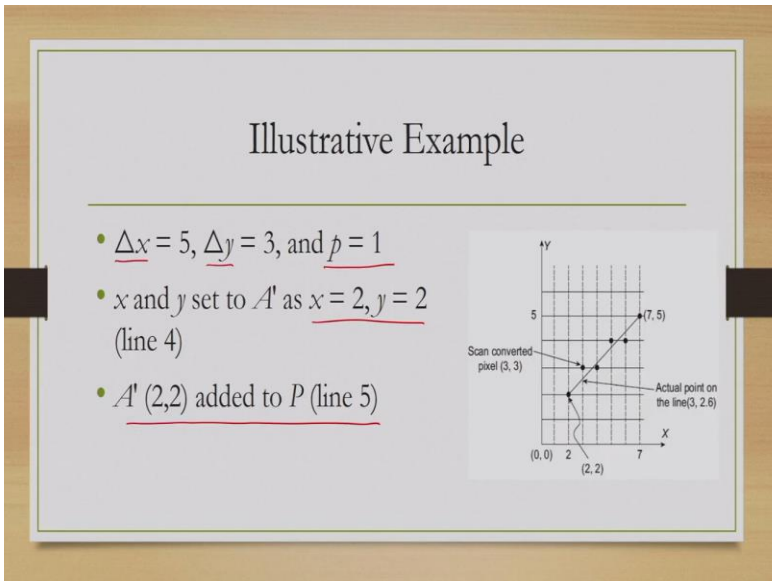


Illustrative Example

- Reconsider line segment defined by A(2,2) and B(7,5)

We will continue with our example that we have introduced in the previous lecture. So this is the line segment given, these are the endpoints already mapped and our job is to find out the intermediate pixels that correspond to the points on the line.

(Refer Slide Time: 19:17)



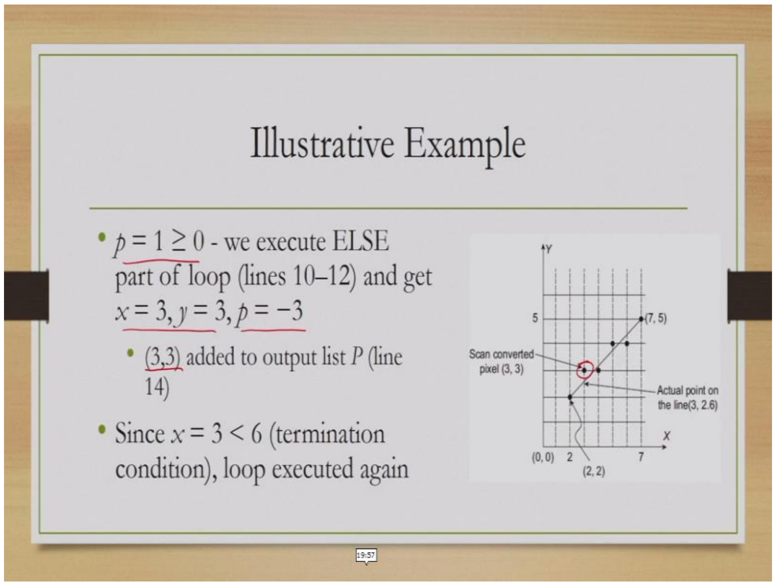
Illustrative Example

- $\Delta x = 5$, $\Delta y = 3$, and $p = 1$
- x and y set to A' as $x = 2, y = 2$ (line 4)
- A' (2,2) added to P (line 5)

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)

So we will start with computing Δ_x, Δ_y, and initial p. Then we start with one endpoint and add it to the list of pixels, the endpoint.

(Refer Slide Time: 19:38)



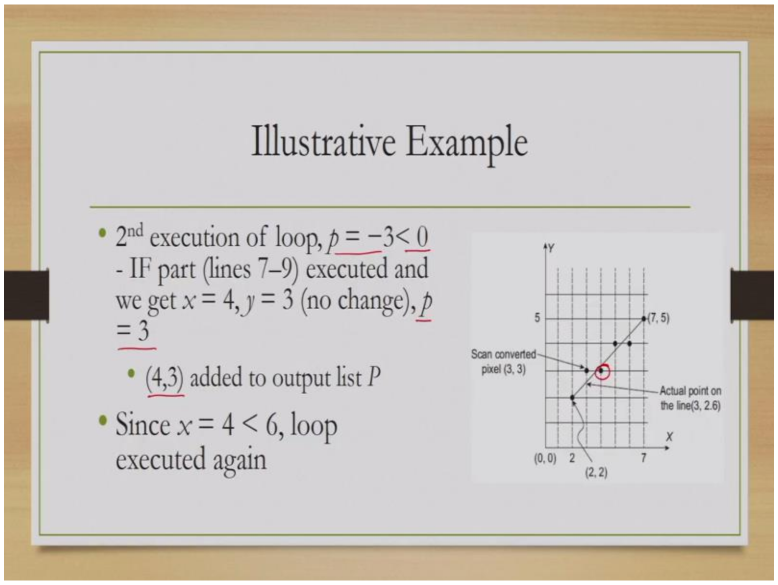
Illustrative Example

- $p = 1 \geq 0$ - we execute ELSE part of loop (lines 10–12) and get $x = 3, y = 3, p = -3$
  - (3,3) added to output list P (line 14)
- Since $x = 3 < 6$ (termination condition), loop executed again

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)

Now, we have computed p to be 1, which is ≥0. So here, the upper pixel is closer that means we choose this one. We add this and update p with the expression to be -3, and (3, 3) is added to the grid. Now, this is not the end point, we have not yet reached the end point so we will continue.

(Refer Slide Time: 20:13)



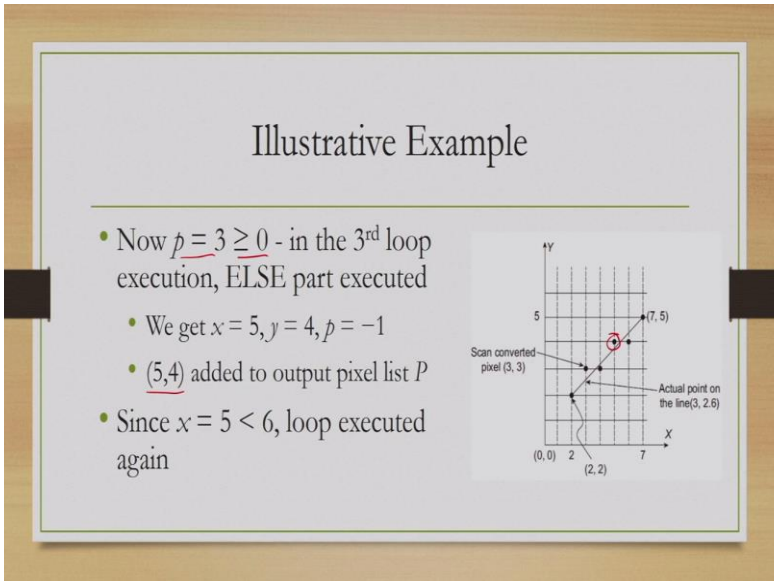
Illustrative Example

- 2nd execution of loop, $p = -3 < 0$ - IF part (lines 7–9) executed and we get $x = 4$, $y = 3$ (no change), $p = 3$
  - (4,3) added to output list P
- Since $x = 4 < 6$, loop executed again

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)

In the second execution, we check that p is -3, which ≤0. So in the second case, the lower pixel is chosen and we update p again, to be 3, add this lower pixel to the output list and check whether we have reached the end point. Since we have not yet reached we continue the loop. And in this way, we continue to get other points.

(Refer Slide Time: 20:43)



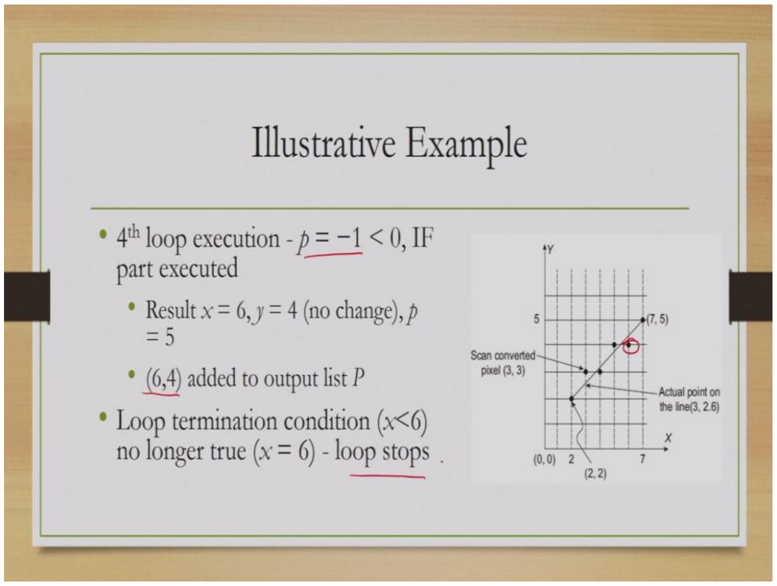
Illustrative Example

- Now $p = 3 \geq 0$ - in the 3rd loop execution, ELSE part executed
  - We get $x = 5, y = 4, p = -1$
  - (5,4) added to output pixel list P
- Since $x = 5 < 6$, loop executed again

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)

So in the next stage, p=3 > 0. So we choose the upper pixel, add this one to the output pixel list, continue the loop since we are yet to reach the end point.

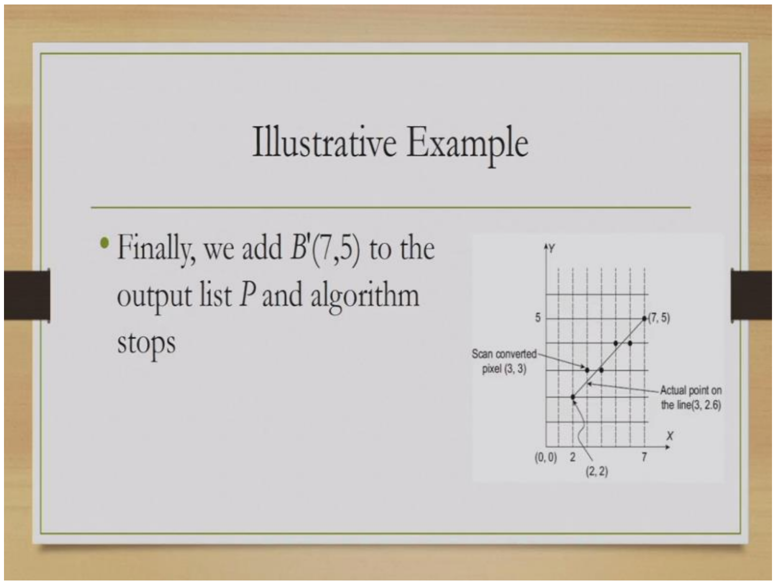
(Refer Slide Time: 21:05)



Illustrative Example

- 4th loop execution - $p = -1 < 0$, IF part executed
  - Result $x = 6, y = 4$ (no change), $p = 5$
  - (6,4) added to output list P
- Loop termination condition ($x<6$) no longer true ($x = 6$) - loop stops

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)



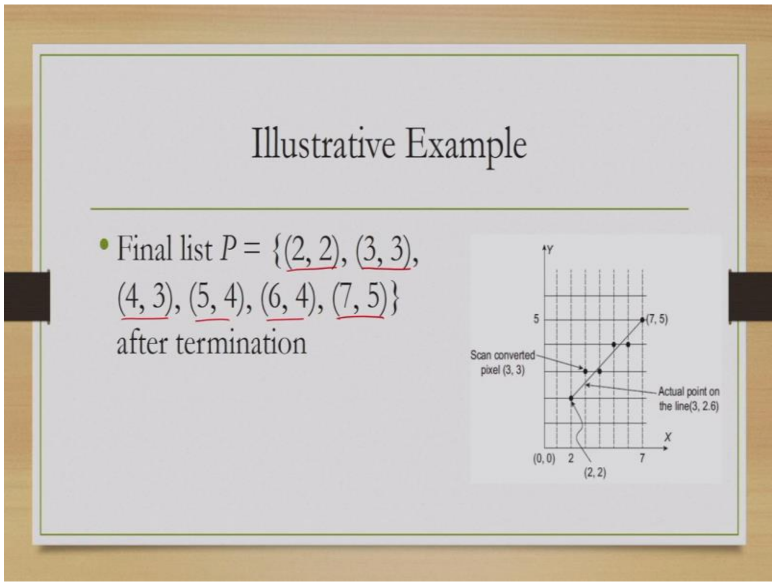
Illustrative Example

- Finally, we add $B'(7,5)$ to the output list P and algorithm stops

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)

Then we find p to be -1, less than 0. So we choose the lower pixel, add the pixel to the output list, and now, we see that we have reached the other end point. So we stopped the loop and add the other endpoint into the list. That is our last step.

(Refer Slide Time: 21:46)



Illustrative Example

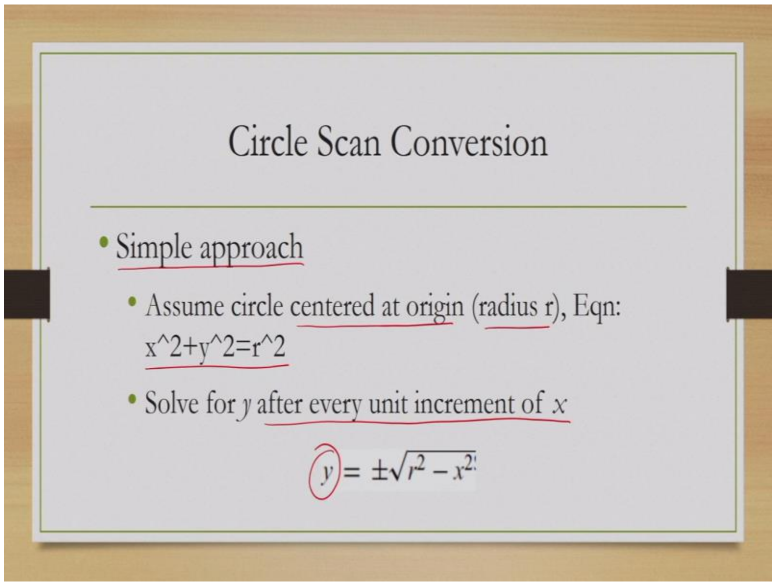
- Final list $P = \{(2, 2), (3, 3), (4, 3), (5, 4), (6, 4), (7, 5)\}$ after termination

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)

So finally, what are the points that we get? These are the pixels that we get following the steps of the Bresenham's algorithm. Now, you can compare it with the previous methods that we used. However, while comparing, you should keep in mind the number of floating-point operations that we avoided because that is the advantage. So if you find that both the sets or all the sets that we have found earlier are same that is not a problem because we saved in terms of computation.

So, with that, we end our discussion on lines scan conversion. So we learned three things; first, we started with a simple approach, found its problems, then we discussed one improved approach that is the DDA approach. And then, we finally discussed even better approach, the Bresenham's line drawing algorithm, which eliminates all the floating-point operations. Now we will move to scan conversion of another primitive shape that is circle.

(Refer Slide Time: 23:19)



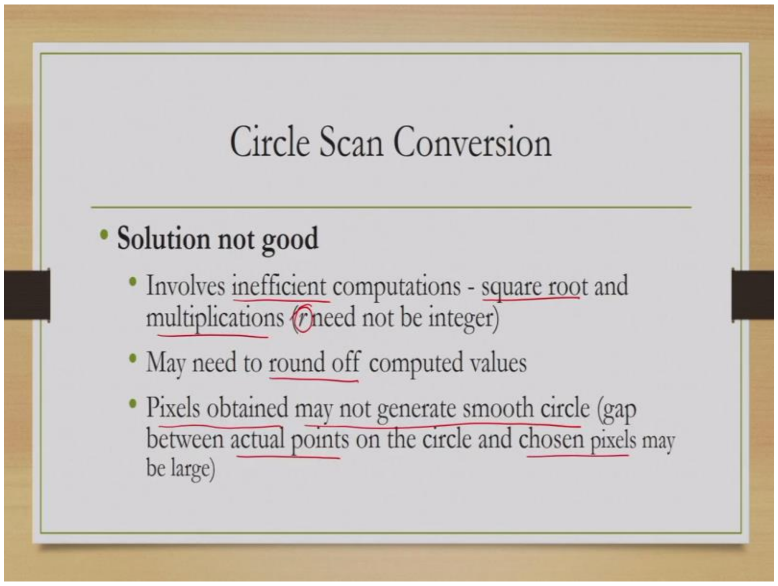
Circle Scan Conversion

- Simple approach
  - Assume circle centered at origin (radius r), Eqn: x^2+y^2=r^2
  - Solve for y after every unit increment of x

$$y = \pm\sqrt{r^2 - x^2}$$

Initially, we will assume that the circle is centered at origin with radius r and its equation is given by x2 + y2 = r2. We all know this equation. Now, in the simple approach, the most intuitive and straightforward approach what we do? We solve for y after every unit increment of x in the pixel grid by using the equation.

(Refer Slide Time: 23:54)

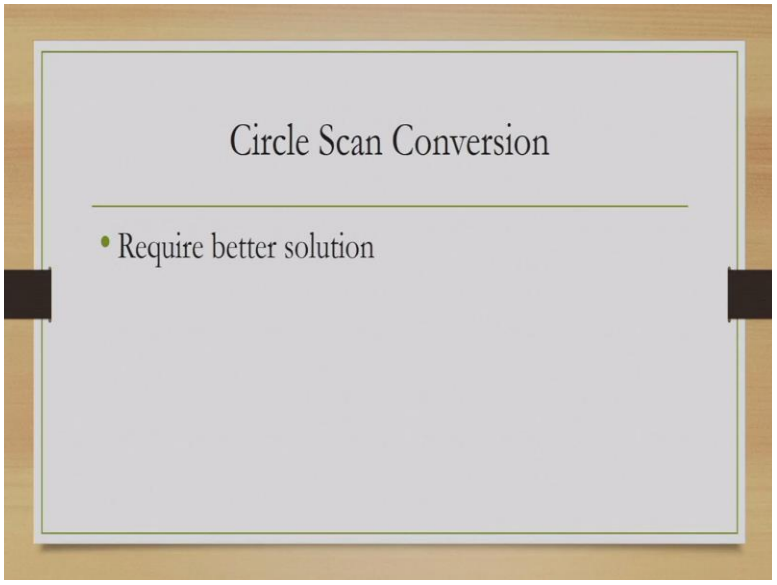


Circle Scan Conversion

- **Solution not good**
  - Involves inefficient computations - square root and multiplications (r need not be integer)
  - May need to round off computed values
  - Pixels obtained may not generate smooth circle (gap between actual points on the circle and chosen pixels may be large)

Clearly, here we have lots of computations, floating-point computations, which involve square root and multiplications because r need not be integer. So this is inefficient. We may also need to round off the computed values, which is addition of other floating-point operations and the pixels that we obtain may not generate smooth circle because there may be gap between actual points and the chosen pixels after rounding off.

(Refer Slide Time: 24:44)



Circle Scan Conversion

- Require better solution

So it suffers from many problems and we require a better solution.

(Refer Slide Time: 24:52)

Midpoint Algorithm

Let us try to go through one such solution, which is called the Midpoint algorithm.

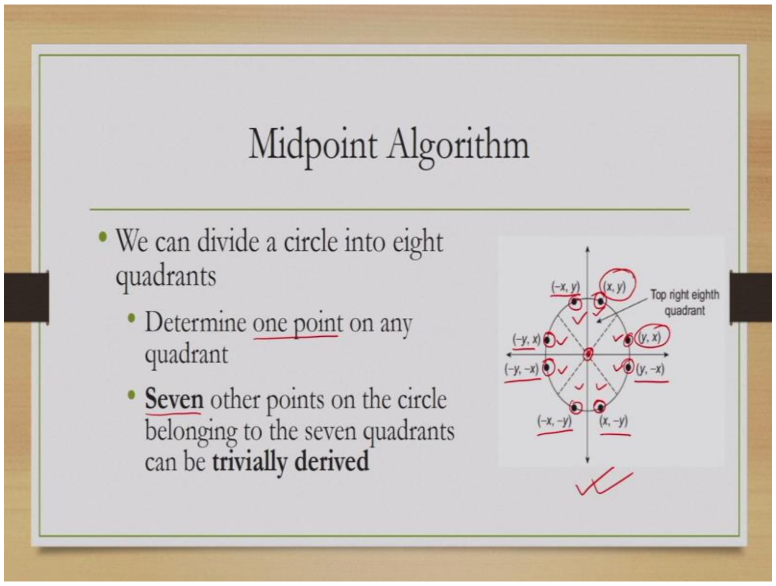
(Refer Slide Time: 25:05)

Midpoint Algorithm

- A circle about origin has an important property — **eight-way symmetry**

Now, this algorithm exploits an interesting property of circle that is called eight-way symmetry. Now, what is this property?

(Refer Slide Time: 25:19)



Midpoint Algorithm

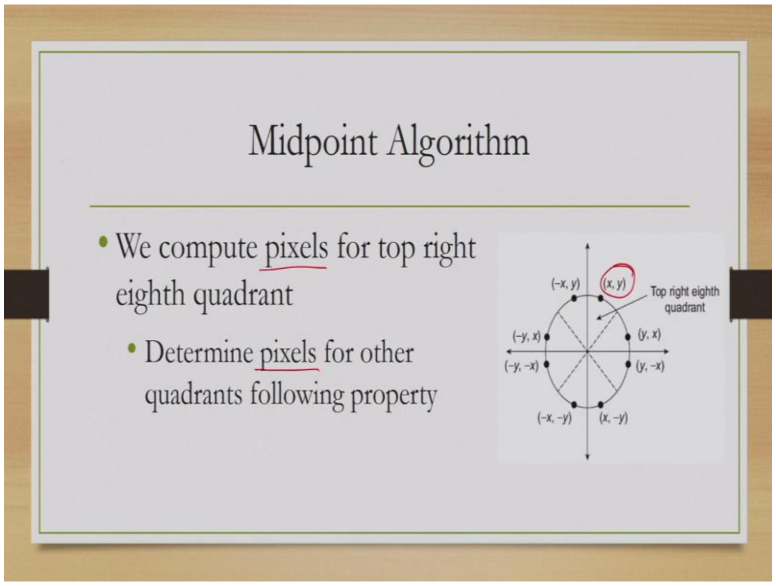
- We can divide a circle into eight quadrants
  - Determine one point on any quadrant
  - **Seven** other points on the circle belonging to the seven quadrants can be **trivially derived**

(−x, y)
(x, y)
Top right eighth quadrant
(−y, x)
(y, x)
(−y, −x)
(y, −x)
(−x, −y)
(x, −y)

If we look at this figure, we will see that this is the origin and the circle is around the origin, we can divide the circle into 8 quadrants, these are the quadrants. And if we determine one point on any quadrants say this point, then we can determine seven other points on the circle belonging to the seven quadrants without much computations.

So if this point is (x, y), then we can say this point will be (y, x), will be (y, -x). This one will be (x, -y), this one will be (-x, -y), this one will be (-y, -x), this one will be (-y, x), and this one will be (-x, y). So this we can straight away determined without any further computation.

(Refer Slide Time: 26:28)



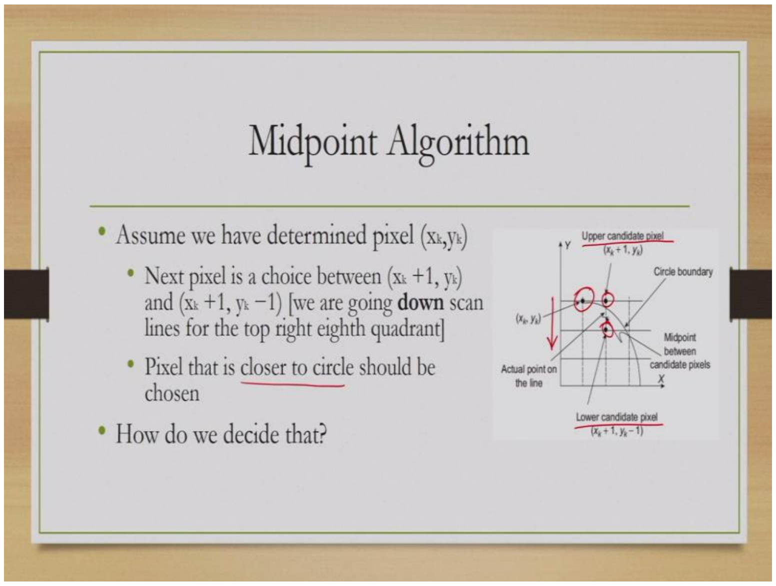
Midpoint Algorithm

- We compute pixels for top right eighth quadrant
  - Determine pixels for other quadrants following property

(−x, y)
(x, y)
Top right eighth quadrant
(−y, x)
(y, x)
(−y, −x)
(y, −x)
(−x, −y)
(x, −y)

We can exploit this property in circle scan conversion and by computing one point on a quadrant and then use that point to derive the other seven points on the circle. That means we determine one pixel, and from there we determine the other seven pixels. So instead of determining the pixels through computation eight times, we do it once and the other seven time computations we save.

(Refer Slide Time: 27:07)



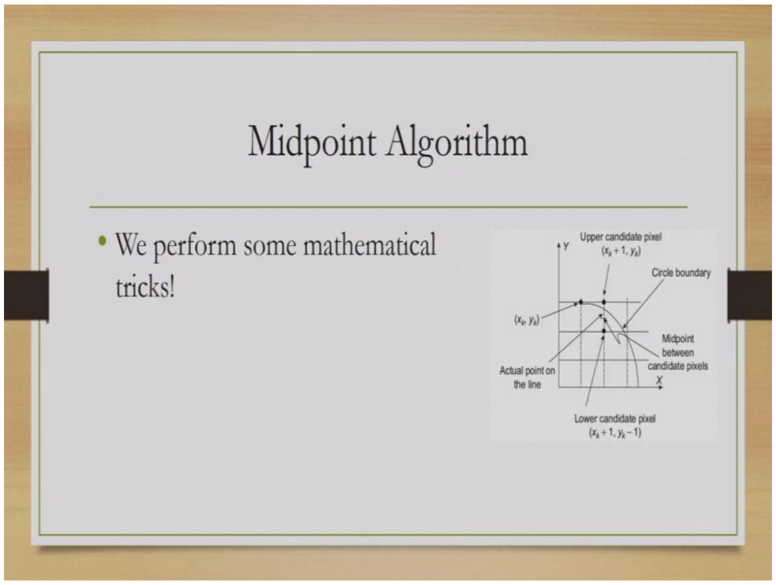
Midpoint Algorithm

- Assume we have determined pixel (x_k,y_k)
  - Next pixel is a choice between $(x_k +1, y_k)$ and $(x_k +1, y_k -1)$ [we are going **down** scan lines for the top right eighth quadrant]
  - Pixel that is closer to circle should be chosen
- How do we decide that?

Y
Upper candidate pixel
(x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel
(x_k+1, y_k-1)

Now, let us see how to determine a pixel for a given quadrant. Suppose, we have determined a pixel (x_k, y_k); this is one quadrant of the circle given. Now, next pixel should be either this one or this one. Again, we can call them upper candidate pixel and lower candidate pixel. And in this case, note that we are going down along this direction, down the scan lines, and our objective is to choose a pixel that is closer to the circle. Now, how do we decide which of these two candidate pixels is closer to the circle?

(Refer Slide Time: 28:11)



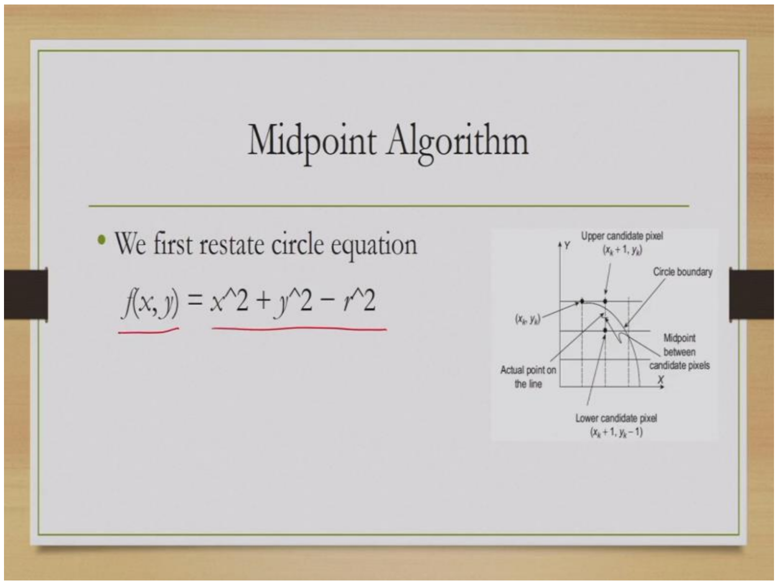
Midpoint Algorithm

- We perform some mathematical tricks!

Y
Upper candidate pixel (x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel (x_k+1, y_k-1)

Again, we go for some mathematical trick.

(Refer Slide Time: 28:18)



Midpoint Algorithm

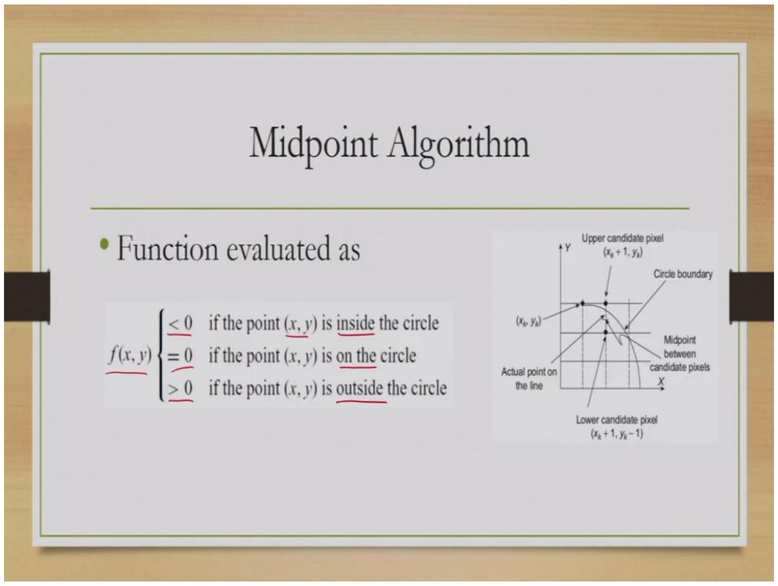
- We first restate circle equation

$\underline{f(x, y)} = \underline{x\char`^2 + y\char`^2 - r\char`^2}$

Y
Upper candidate pixel (x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel (x_k+1, y_k-1)

Now, the circle equation, we can reorganize in this way, $f(x, y) = x^2 + y^2 - r^2$, this is the circle equation we can restate in this way.

(Refer Slide Time: 28:33)



Midpoint Algorithm

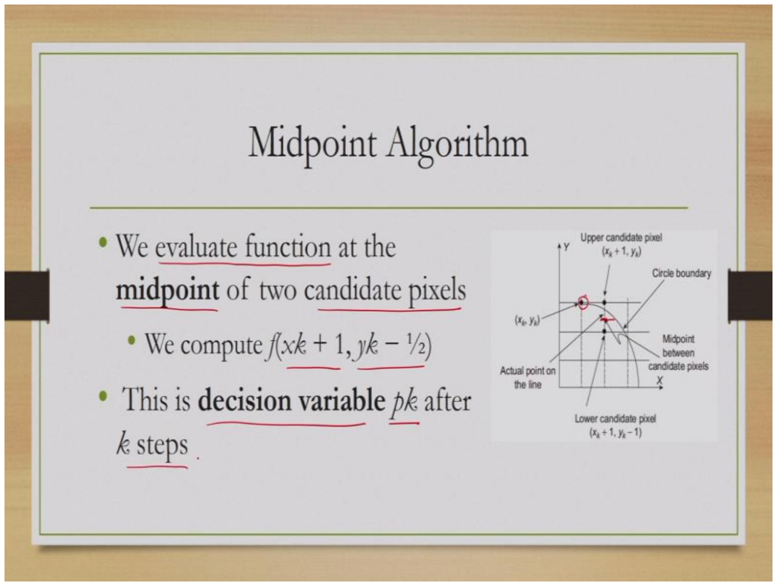
- Function evaluated as

$$f(x,y)\begin{cases} <0 & \text{if the point } (x,y) \text{ is inside the circle} \\ =0 & \text{if the point } (x,y) \text{ is on the circle} \\ >0 & \text{if the point } (x,y) \text{ is outside the circle} \end{cases}$$

Y
Upper candidate pixel
(x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel
(x_k+1, y_k-1)

Now, this function we can evaluate as shown here. That is if (x, y) < 0, if the point (x, y) is inside the circle; it is 0 if it is on the circle, and it will be greater than 0 if the point is outside the circle. This we know from geometry.

(Refer Slide Time: 29:05)



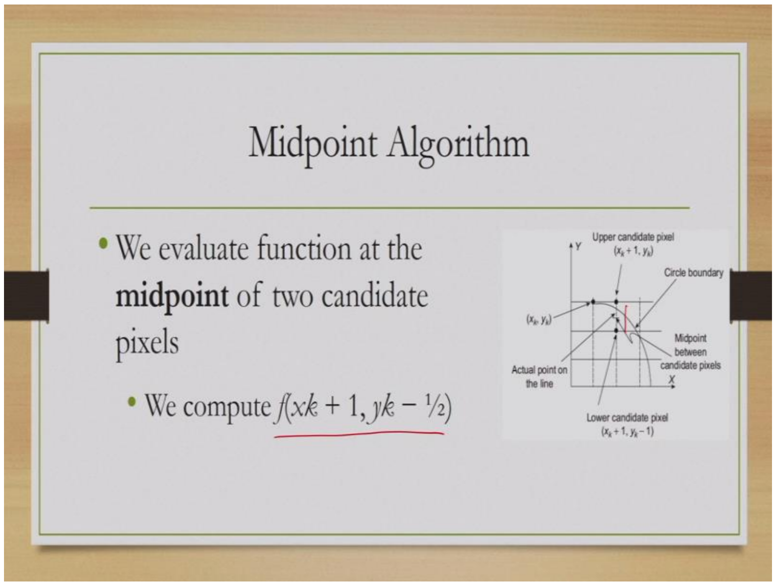
Midpoint Algorithm

- We evaluate function at the **midpoint** of two candidate pixels
    - We compute $f(xk + 1, yk - ½)$
- This is **decision variable** *pk* after *k* steps

Y
Upper candidate pixel
(x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel
(x_k+1, y_k-1)

Then, we can evaluate the function at the midpoint of the two candidate pixels. That means at $(x_k+1, y_k – ½)$. Note that this is y_k, so midpoint will be this point that is $y_k – ½$ and it will be x_k+1 that will be the new x coordinate. Now, this will be our decision variable p_k after k steps. So let us try to see what this variable looks like.

(Refer Slide Time: 29:48)



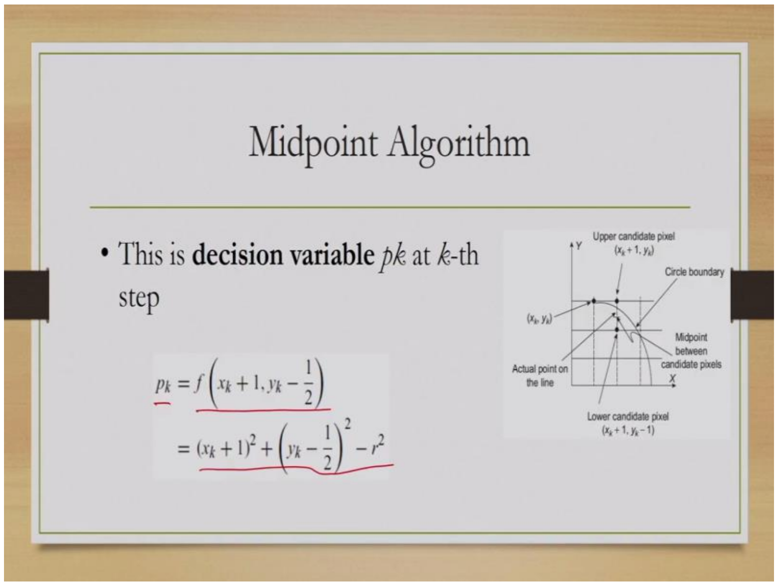
Midpoint Algorithm

- We evaluate function at the **midpoint** of two candidate pixels
  - We compute $f(xk + 1, yk – ½)$

Upper candidate pixel $(x_k + 1, y_k)$
Y
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel $(x_k + 1, y_k - 1)$

So essentially, we compute this function at the point $(x_k+1, y_k–½)$, which is the midpoint between the two candidate pixels, and y half because this is the unit distance, so half or the midpoint will be half of this unit distance.

(Refer Slide Time: 30:16)



Midpoint Algorithm

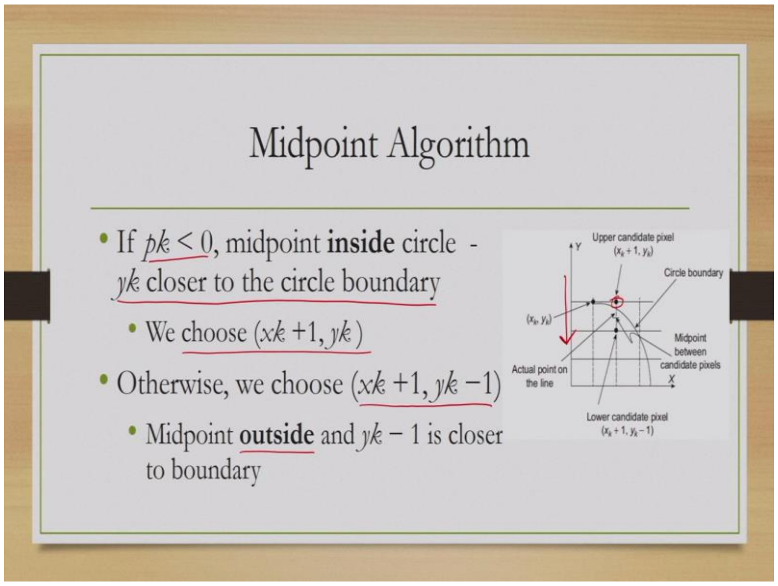
- This is **decision variable** pk at k-th step

$$p_k = f\left(x_k + 1, y_k - \frac{1}{2}\right)$$
$$= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2$$

Upper candidate pixel (x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
Lower candidate pixel (x_k+1, y_k-1)
Y
X

So, p_k will be the function value at this point. Now, if we expand the function with these coordinates, then we get this expression which is the expression for p_k.

(Refer Slide Time: 30:42)



Midpoint Algorithm

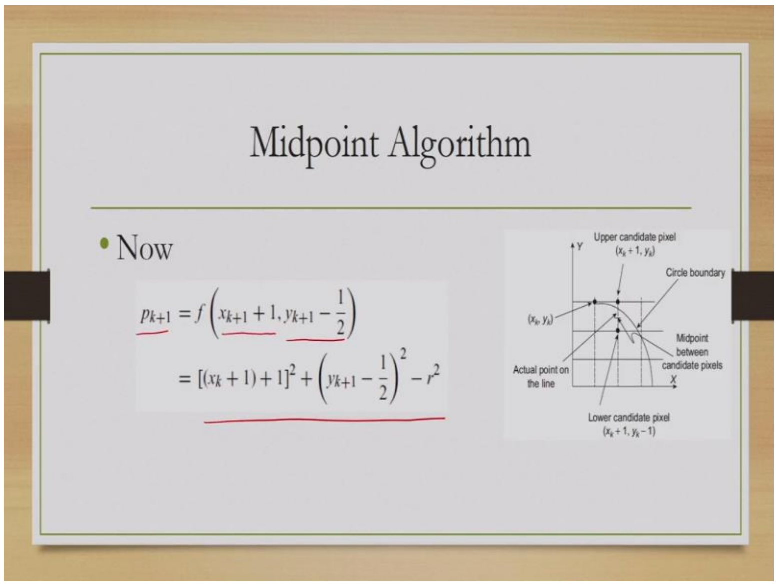
- If $pk < 0$, midpoint **inside** circle - yk closer to the circle boundary
  - We choose $(xk + 1, yk)$
- Otherwise, we choose $(xk + 1, yk - 1)$
  - Midpoint **outside** and $yk - 1$ is closer to boundary

Upper candidate pixel (x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
Lower candidate pixel (x_k+1, y_k-1)
Y
X

Now, if p_k<0 that means, the function evaluates to be less than 0. Then we know from the geometric properties that midpoint is inside the circle. That means the upper candidate pixel will be closer to the circle boundary. So we choose (x_k+1, y_k). If that is not the case then we choose the other candidate pixel that is (x_k+1, y_k-1).

Note that we are going down the scan lines, so next y coordinate will be y_{k-1}. Because in that case, midpoint is outside and this lower candidate pixel is closer to the boundary.

(Refer Slide Time: 31:33)



Midpoint Algorithm

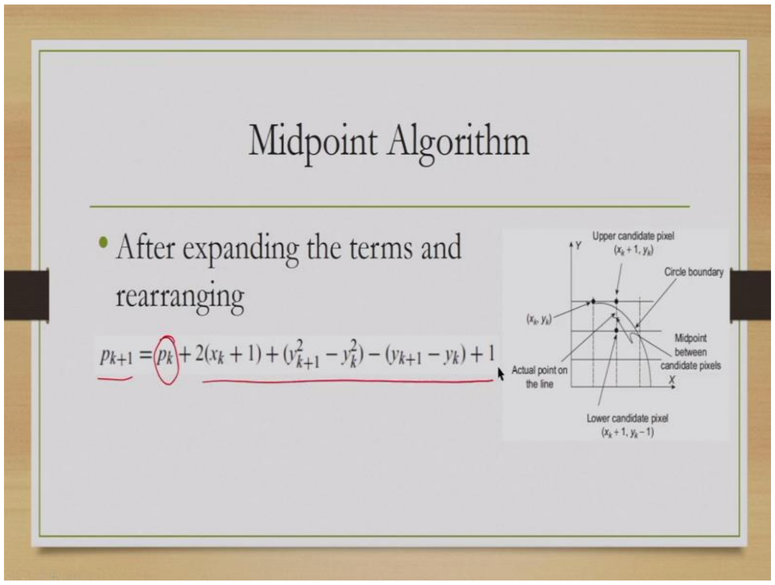
- Now

$$\underline{p_{k+1}} = f\left(\underline{x_{k+1}} + 1, \underline{y_{k+1}} - \frac{1}{2}\right)$$
$$\underline{= [(x_k + 1) + 1]^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2}$$

Upper candidate pixel
(x_k+1, y_k)
Y
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel
(x_k+1, y_k-1)

Now, let us see the expression for the decision variable at the next step that is pk plus 1. So here, our new point will be $x_{k+1}+1$, increment by 1, and $y_{k+1}-½$, which after expansion will look like this.

(Refer Slide Time: 31:55)



Midpoint Algorithm

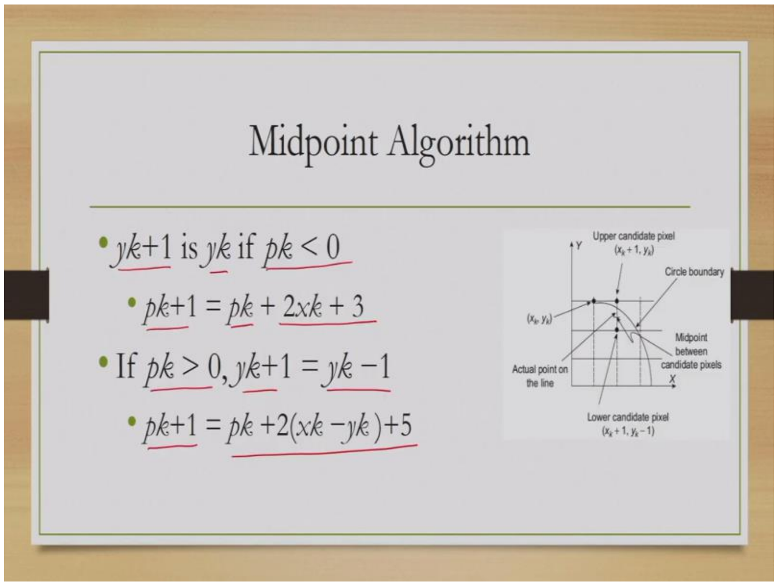
- After expanding the terms and rearranging

$$\underline{p_{k+1}} = p_k + \underline{2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1}$$

Upper candidate pixel
(x_k+1, y_k)
Y
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel
(x_k+1, y_k-1)

Now, we may expand it further and then rearrange to get a simplified expression that is p_{k+1} is the current decision value plus this term.

(Refer Slide Time: 32:14)



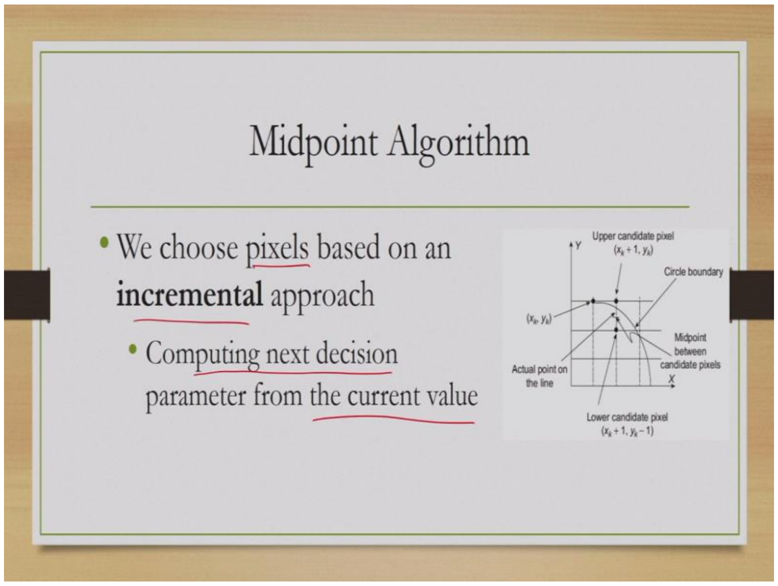
Midpoint Algorithm

- $yk+1$ is yk if $pk < 0$
  - $pk+1 = pk + 2xk + 3$
- If $pk > 0$, $yk+1 = yk - 1$
  - $pk+1 = pk + 2(xk - yk) + 5$

Upper candidate pixel
(x_k+1, y_k)
Y
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel
(x_k+1, y_k-1)

Now, y_{k+1} is y_k if p_k is less than 0 that we have already seen. So in that case, p_{k+1} will be p_{k+2}, x_{k+3}. Now, if p_k greater than 0 then y_{k+1} will be y_{k-1} that also we have seen. Then the p_{k+1} term will become something like this.

(Refer Slide Time: 33:01)



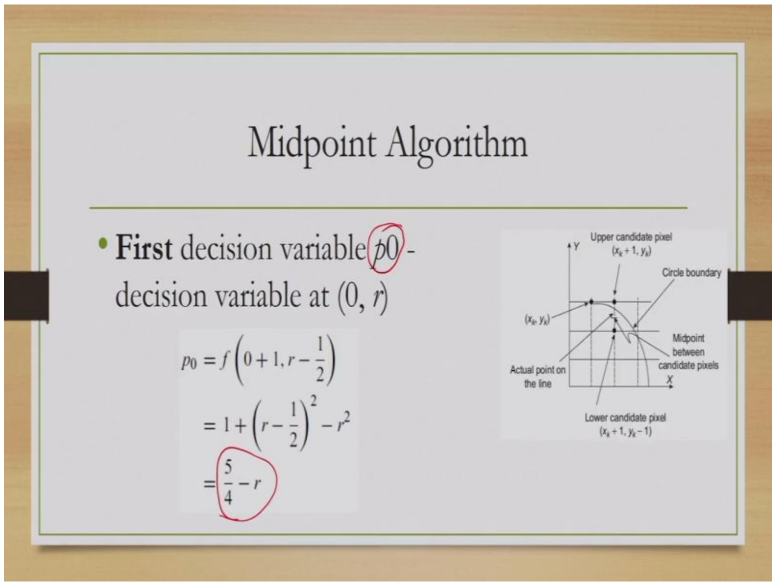
Midpoint Algorithm

- We choose pixels based on an **incremental** approach
  - Computing next decision parameter from the current value

Upper candidate pixel
(x_k+1, y_k)
Y
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
X
Lower candidate pixel
(x_k+1, y_k-1)

As you can see these are all integer operations and we choose the pixels based on an incremental approach that is computing the next decision parameter from the current value and that too by avoiding floating-point operations.

(Refer Slide Time: 33:22)


Upper candidate pixel
(x_k+1, y_k)
Circle boundary
(x_k, y_k)
Midpoint between candidate pixels
Actual point on the line
Lower candidate pixel
(x_k+1, y_k-1)
Y
X

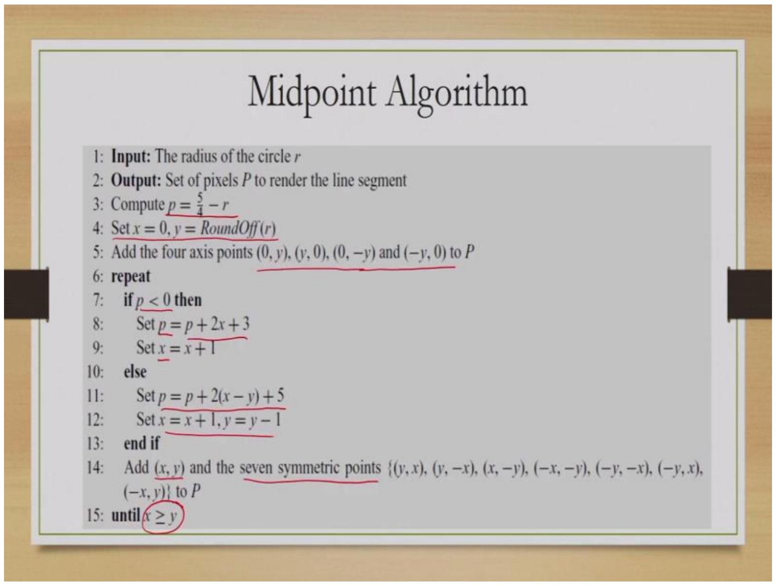
Midpoint Algorithm

- **First** decision variable $p0$ - decision variable at $(0, r)$

$$p_0 = f\left(0+1, r-\frac{1}{2}\right)$$
$$= 1+\left(r-\frac{1}{2}\right)^2 - r^2$$
$$= \frac{5}{4} - r$$

However, that need not be the case because here, the initial decision parameter or decision variable involves floating-point operation. And we have to keep that in mind that unlike Bresenham’s algorithm, although, the expression for computing the next decision variable does not involve any floating-point operation apparently, but when we start with maybe a floating-point value and then that will remain. So here we are not completely eliminating floating-point operations but we are reducing them significantly.

(Refer Slide Time: 34:11)



Midpoint Algorithm

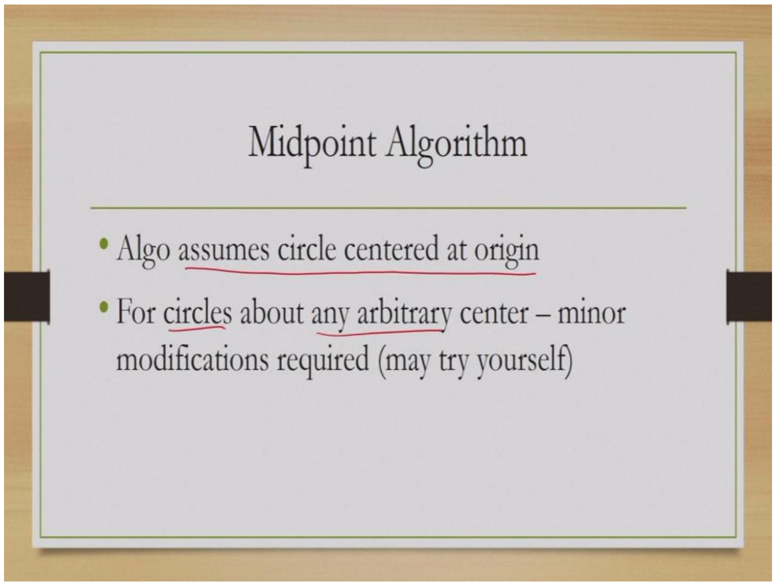
```
1: Input: The radius of the circle $r$
2: Output: Set of pixels $P$ to render the line segment
3: Compute $p = \frac{5}{4} - r$
4: Set $x = 0, y = RoundOff(r)$
5: Add the four axis points $(0, y), (y, 0), (0, -y)$ and $(-y, 0)$ to $P$
6: repeat
7:   if $p < 0$ then
8:     Set $p = p + 2x + 3$
9:     Set $x = x + 1$
10:  else
11:    Set $p = p + 2(x - y) + 5$
12:    Set $x = x + 1, y = y - 1$
13:  end if
14:  Add $(x, y)$ and the seven symmetric points $\{(y, x), (y, -x), (x, -y), (-x, -y), (-y, -x), (-y, x),$
     $(-x, y)\}$ to $P$
15: until $x \geq y$
```

So what is the complete algorithm? We first compute the first decision variable and choose the first or the starting point. Now, one point we have chosen, then using symmetry we can add four other points or pixels.

Now, when the decision parameter is less than 0, we update the parameter in this way and get the pixel to add to the set of pixels. When the decision parameter is greater than 0, then we update the decision parameter in this way and get this point as the new pixel. And then we add the new pixel to the list of pixels plus we add the seven symmetric points using the symmetric property and we continue it until we reach the end of the quadrant.

So that is how midpoint algorithm works. As you have noted, if we go for simple approach, we require a lot of floating-point operations, multiplications, square root, which we avoided by this midpoint algorithm.

(Refer Slide Time: 35:58)

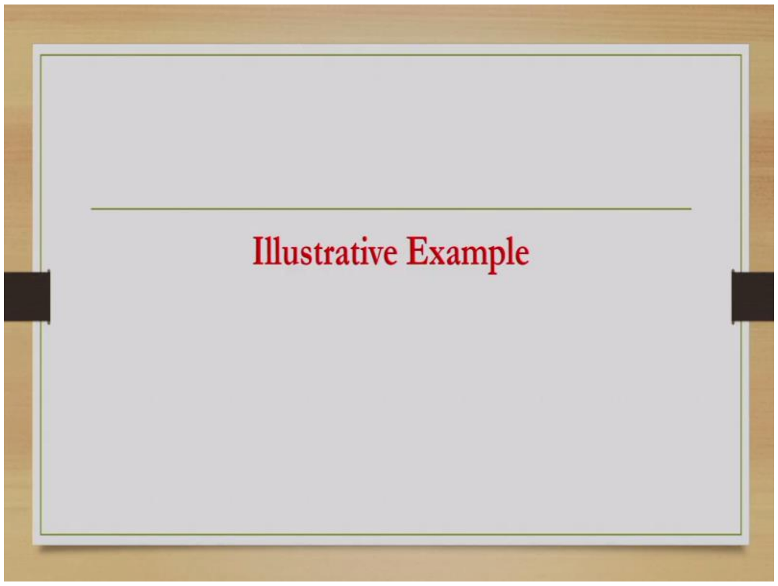


Midpoint Algorithm

- Algo assumes circle centered at origin
- For circles about any arbitrary center – minor modifications required (may try yourself)

Now here, of course, it may be noted that the algorithm assumes circle is centered at origin and we require some modification when we are assuming circles which has its center at any arbitrary location. But that minor modification we will not go into the details, you may try it yourself.

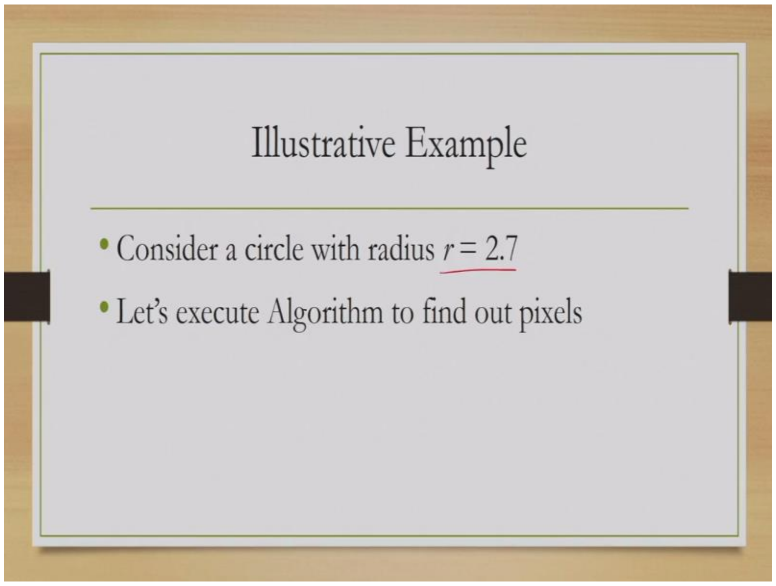
(Refer Slide Time: 36:23)



Illustrative Example

Now, let us try to understand this algorithm better in terms of one illustrative example.

(Refer Slide Time: 36:34)

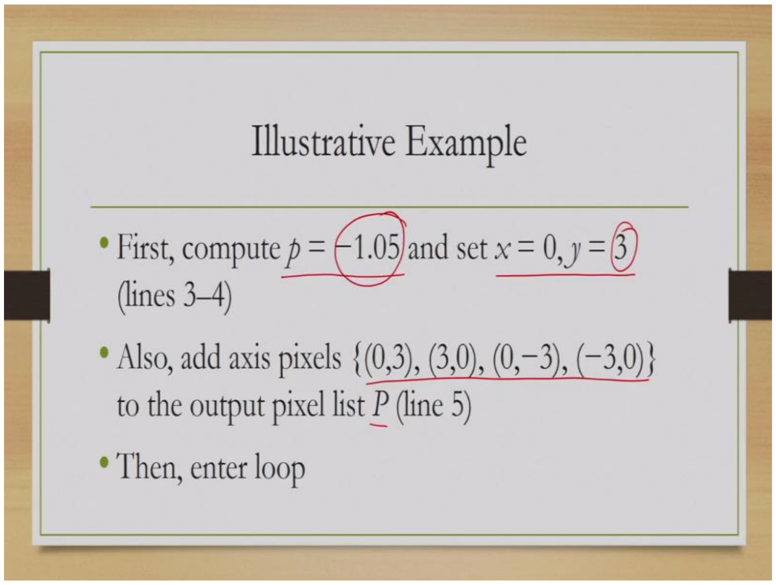


Illustrative Example

- Consider a circle with radius $r = 2.7$
- Let's execute Algorithm to find out pixels

Let us start with the assumption that we have a circle with radius r to be 2.7 that means a real number. Now, let us execute the algorithm to find out the pixels that we should choose to represent the circle.

(Refer Slide Time: 36:59)

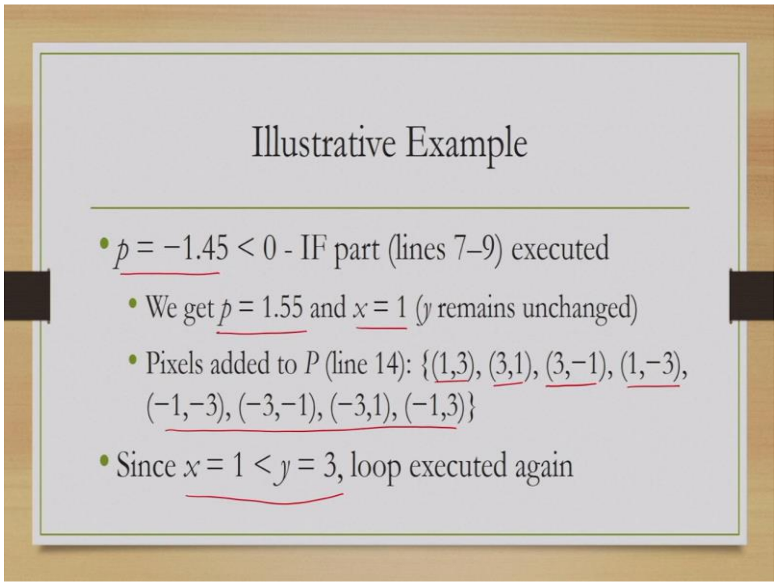


Illustrative Example

- First, compute $p = -1.05$ and set $x = 0, y = 3$ (lines 3–4)
- Also, add axis pixels $\{(0,3), (3,0), (0,-3), (-3,0)\}$ to the output pixel list P (line 5)
- Then, enter loop

First stage is compute p, which is 5/4 - r, which gives us this value. And we start with this point by rounding off r to 3 and we get this point as the first point in our list, first pixel. And based on this first pixel, we add other four pixels in the output list. Then we enter the main loop.

(Refer Slide Time: 37:34)

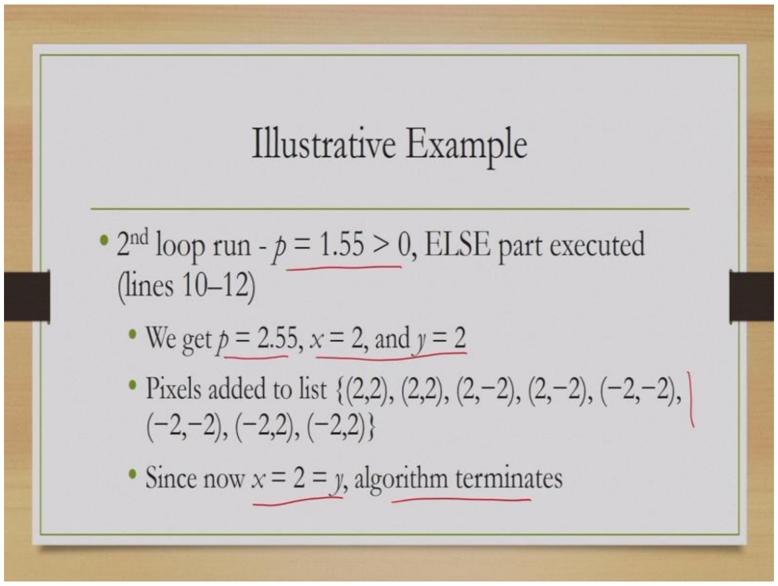


Illustrative Example

- $p = -1.45 < 0$ - IF part (lines 7–9) executed
  - We get $p = 1.55$ and $x = 1$ (y remains unchanged)
  - Pixels added to P (line 14): {(1,3), (3,1), (3,−1), (1,−3), (−1,−3), (−3,−1), (−3,1), (−1,3)}
- Since $x = 1 < y = 3$, loop executed again

So we have p<0, then we update p as per the expression for p<0 and then, get this new pixel value. With that, we add eight pixels to the output list (1, 3), (3, 1) (3, -1) (1, -3), and so on. Since we have not yet reached the end of the loop, we continue with the loop.

(Refer Slide Time: 38:11)

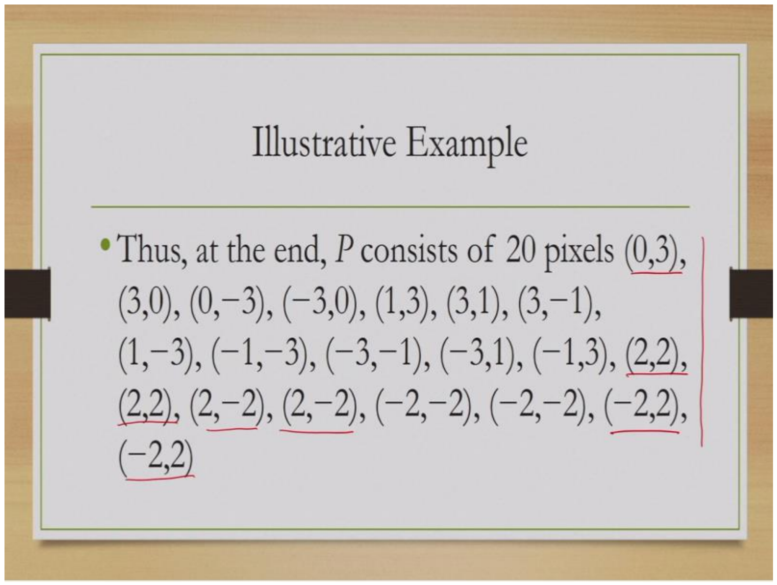


Illustrative Example

- 2nd loop run - $p = 1.55 > 0$, ELSE part executed (lines 10–12)
  - We get $p = 2.55$, $x = 2$, and $y = 2$
  - Pixels added to list {(2,2), (2,2), (2,−2), (2,−2), (−2,−2), (−2,−2), (−2,2), (−2,2)}
  - Since now $x = 2 = y$, algorithm terminates

In the second loop run, we have p>0. So we use the expression to update p and we get the new value and we decide on this new pixel, based on that we choose the eight pixels as shown here. Now, we have arrived at the end of the loop, the termination condition is reached so we stop. So then at the end what we get.

(Refer Slide Time: 38:44)

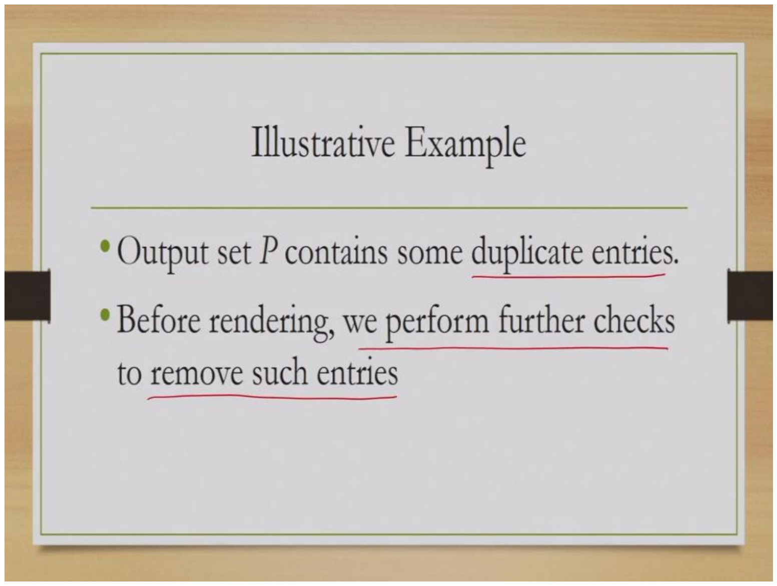


Illustrative Example

- Thus, at the end, P consists of 20 pixels (0,3), (3,0), (0,−3), (−3,0), (1,3), (3,1), (3,−1), (1,−3), (−1,−3), (−3,−1), (−3,1), (−1,3), (2,2), (2,2), (2,−2), (2,−2), (−2,−2), (−2,−2), (−2,2), (−2,2)

We get 20 pixels shown here. One thing you may note is that there are some pixels that are repeated. For example, (2, 2) occur twice; (-2, -2) occurred twice; (2, -2) occurred twice. So this repetition is there at the end of the execution of the algorithm.

(Refer Slide Time: 39:27)



Illustrative Example

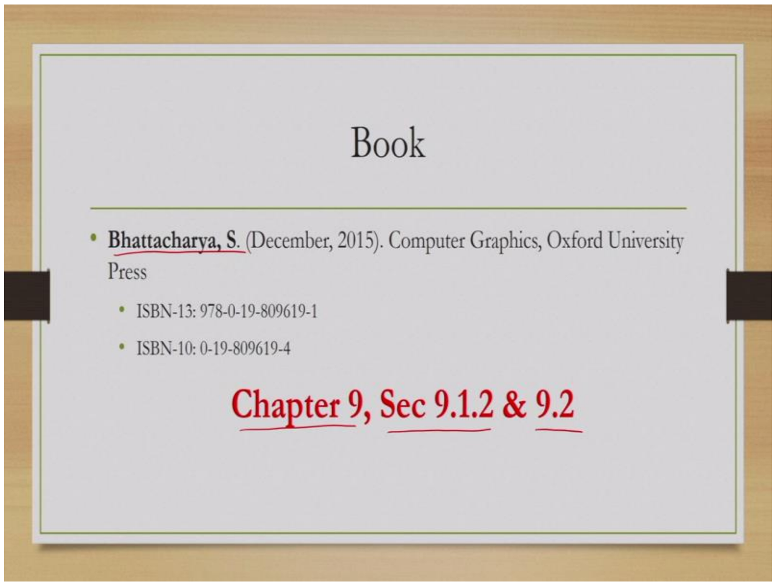
- Output set P contains some duplicate entries.
- Before rendering, we perform further checks to remove such entries

These duplicate entries, we need to remove. So before rendering, we perform further checks and processing on the output list to remove such duplicate entries. So the algorithm may give us a list having duplicate entries, we need to perform some checks before we use those pixels to render the circle to avoid duplications. So that is what we do to render circle.

So we have learned how to render line, we have learned how to render circle. In both cases, our objective was to map from real number values to pixel grids. And our objective was to do so without involving floating-point operations to the extent possible because, in practical applications, we need to render these shapes very frequently. And there, if too many floating-point operations are involved, then the speed at which we can render may slow down giving us the perception of a distorted image or flickers which are unwelcome.

In the next class, we will discuss more on rendering other things. Whatever we have discussed today, can be found in this book.

(Refer Slide Time: 41:15)



Book

- Bhattacharya, S. (December, 2015). Computer Graphics, Oxford University Press
  - ISBN-13: 978-0-19-809619-1
  - ISBN-10: 0-19-809619-4

Chapter 9, Sec 9.1.2 & 9.2

You may go through chapter 9, section 9.1.2 and 9.2 to get the details on the topics that we covered today. So we will meet in the next lecture. Till then, thank you and goodbye.