Computer Graphics

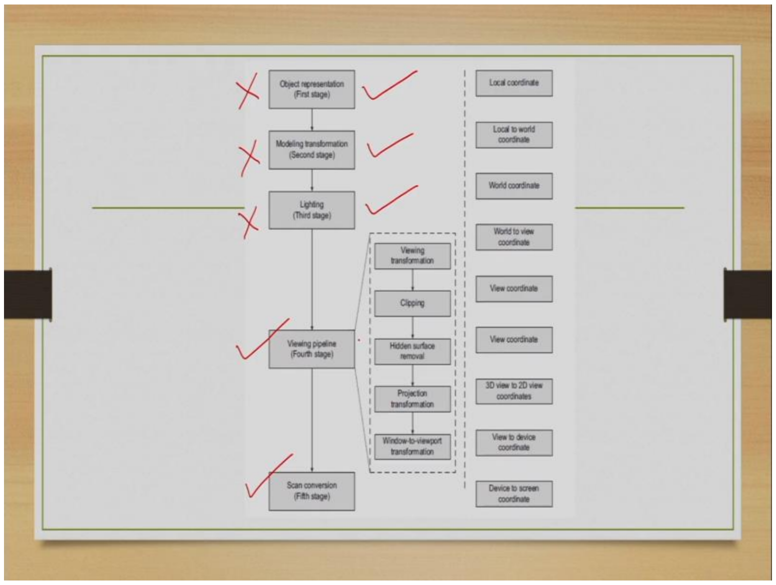
Professor Dr. Samit Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Lecture - 25

Scan Conversion of Basic Shapes - 1

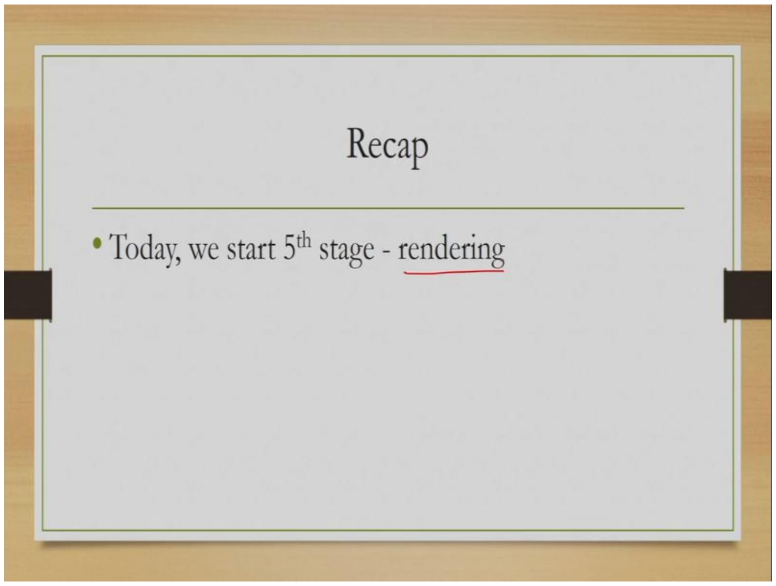
Hello and welcome to lecture number 25 in the course Computer Graphics. We are currently discussing the 3D graphics pipeline which consists of five stages. Let us quickly recap the stages.

(Refer Slide Time: 00:52)


Object representation (First stage)
Modeling transformation (Second stage)
Lighting (Third stage)
Viewing pipeline (Fourth stage)
Scan conversion (Fifth stage)
Viewing transformation
Clipping
Hidden surface removal
Projection transformation
Window-to-viewport transformation
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

As you can see in this figure, first stage is object representation, then we have modeling transformation, then lighting or the coloring of objects, then viewing pipeline, and the fifth stage is scan conversion. Among them, we have already discussed the first four stages namely, object representation, modeling transformation, lighting, and the viewing pipeline.

(Refer Slide Time: 01:23)

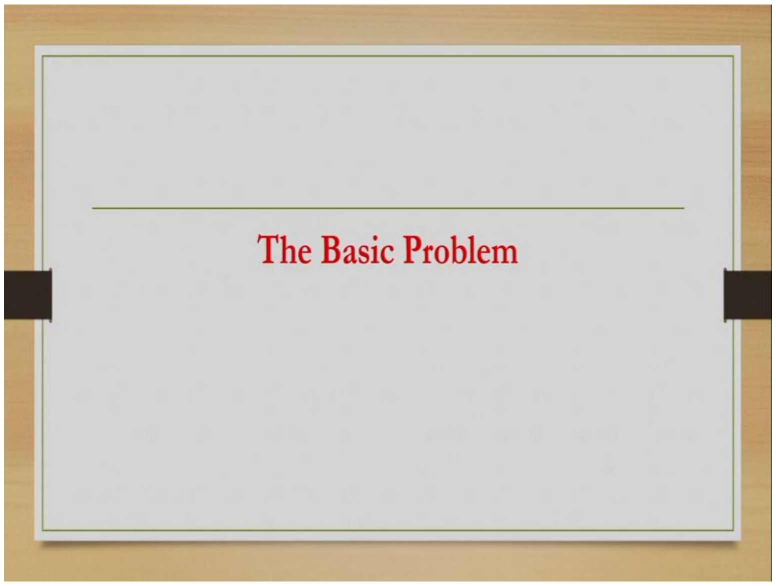


Recap

- Today, we start 5th stage - rendering

Now we are going to discuss the fifth stage that is rendering or also known as scan conversion. So what is this stage is all about?

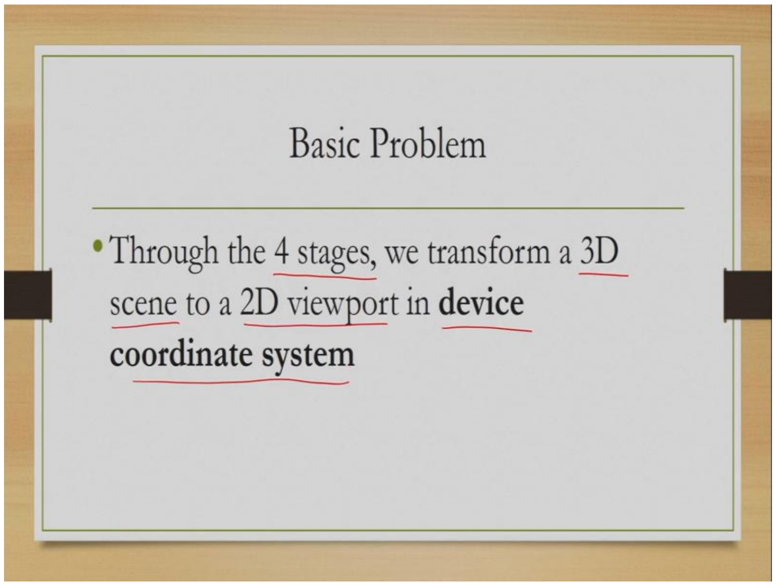
(Refer Slide Time: 01:40)



The Basic Problem

Let us have a look at the very basic problem that we try to address in this fifth stage.

(Refer Slide Time: 01:48)

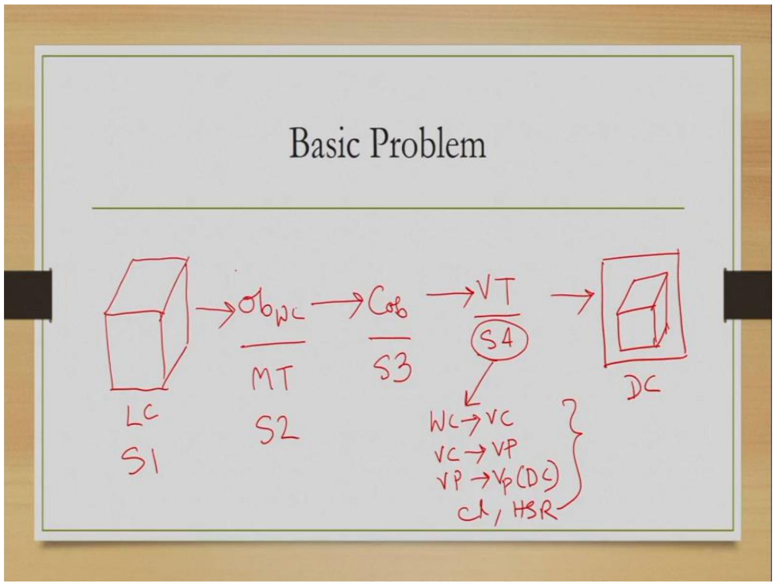


Basic Problem

- Through the 4 stages, we transform a 3D scene to a 2D viewport in **device coordinate system**

So far, whatever we have learned that gives us some idea of one particular thing. Through these four stages that we have discussed so far, we can transform a 3D scene to a 2D viewport description, which is in the device coordinate system. Just quickly have a relook at how it is done as we have learned in our previous discussions.

(Refer Slide Time: 02:19)



Basic Problem

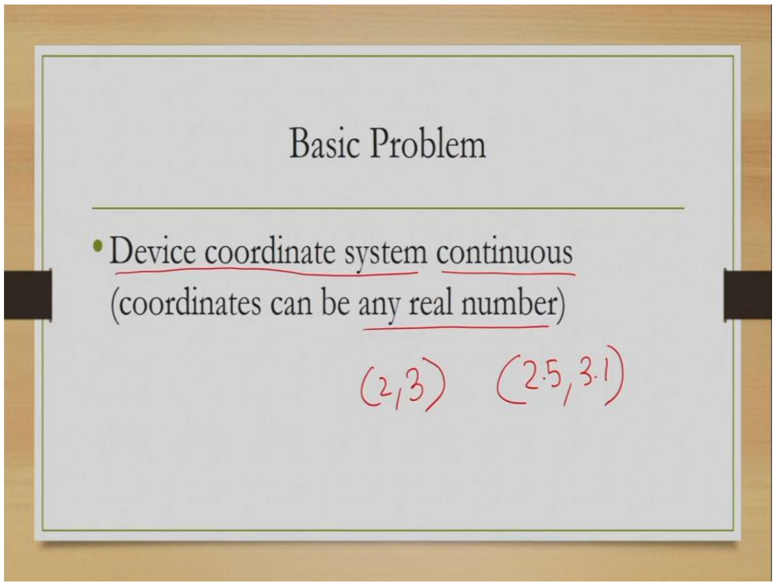
So first, we have a 3D scene say, for example, this cube. Now, this cube is defined in its own coordinate system or local coordinate, which we do in the first stage that is object representation. Then what we do, we transfer it to a world coordinate system through modeling transformation.

So the object is now in world coordinate. So this is stage one. And in second stage through modeling transformation, we transfer it to world coordinate description which is stage two.

Then we assign colors by using the lighting, shading models. So we color the object in stage-3 and after that, we perform viewing transformation, which is stage four in which we transfer it to a viewport description. So it involves first, transferring the world coordinates into a view coordinate system, then from view coordinate, we perform projects and transformation and transfer it to view plane, then from view plane, we perform a window to viewport mapping to transfer it to this viewport, which is in the device coordinate system.

So these three transformations take place in stage four along with, of course, clipping and hidden surface removal. And after that what we get is a 2D representation of the object or scene on a viewport, which is in the device coordinate system. This is how things get transformed from object definition to viewport description.

(Refer Slide Time: 05:10)



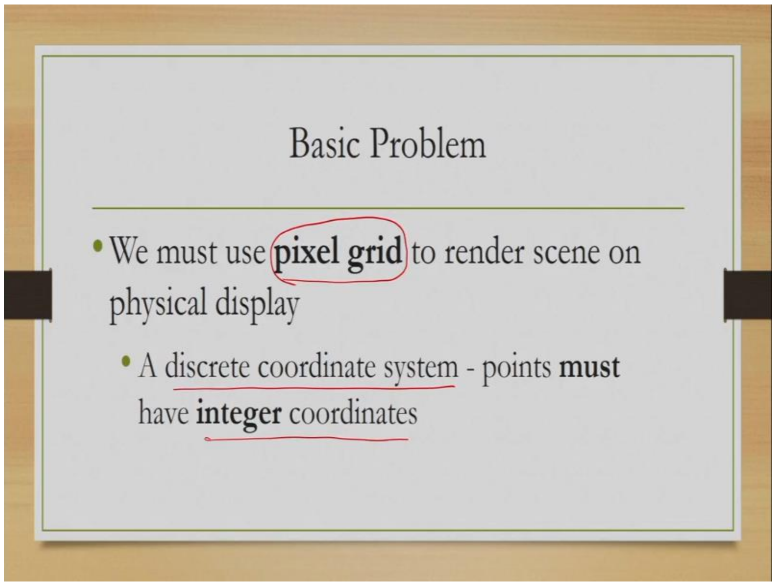
Basic Problem

- Device coordinate system continuous (coordinates can be any real number)

(2, 3)　(2.5, 3.1)

However, the device coordinate system that we are talking about is a continuous system that means the coordinate values of any point can be any real number. So we can have coordinate like 2, 3, which are integers, whereas, we can also have a coordinate like 2.5, 3.1, which are real numbers. So all sorts of coordinates are allowed in device coordinate system.

(Refer Slide Time: 05:47)

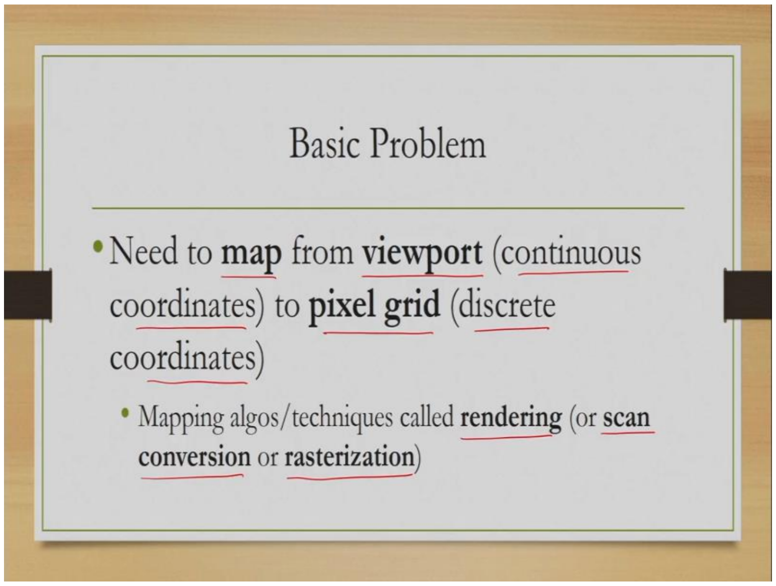


Basic Problem

- We must use **pixel grid** to render scene on physical display
  - A discrete coordinate system - points **must** have **integer** coordinates

In contrast, when we are actually trying to display it on a screen, we have a pixel grid that means it is a discrete coordinate system. So all possible coordinates are not allowed, instead, we must have something where only integer coordinates are defined. So whatever we want to display on the pixel grid must be displayed in terms of integer coordinates, we cannot have real coordinate values.

(Refer Slide Time: 06:35)

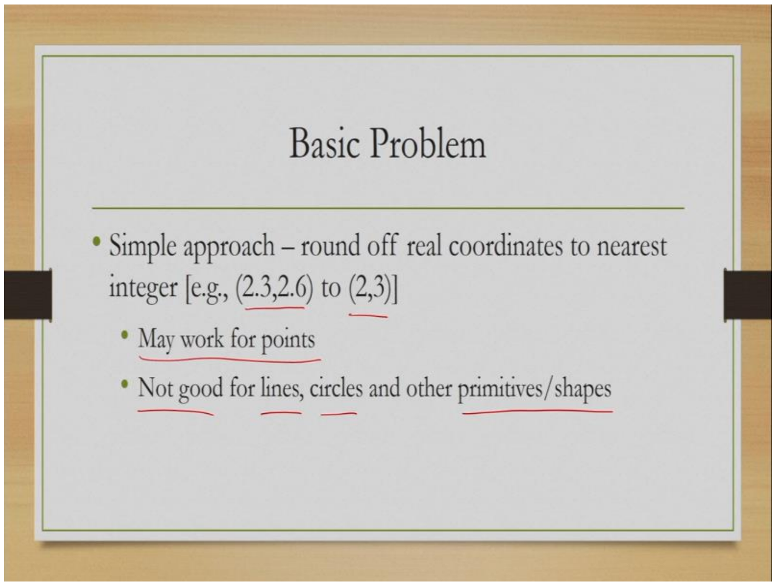


Basic Problem

- Need to **map** from **viewport** (continuous coordinates) to **pixel grid** (discrete coordinates)
  - Mapping algos/techniques called **rendering** (or **scan conversion** or **rasterization**)

Thus what we need? We need to map from the viewport description, which is a continuous coordinate space to a pixel grid, which is a discrete coordinate space. So this is the final mapping that we need to do before a scene is rendered on a physical display screen. Now, these mapping algorithms or the techniques that we use for mapping are collectively known as rendering or more popularly, they are called scan conversion or sometime rasterization as we are mostly dealing with raster scan devices. So all these three terms are used, rendering, scan conversion, or rasterization.

(Refer Slide Time: 07:34)



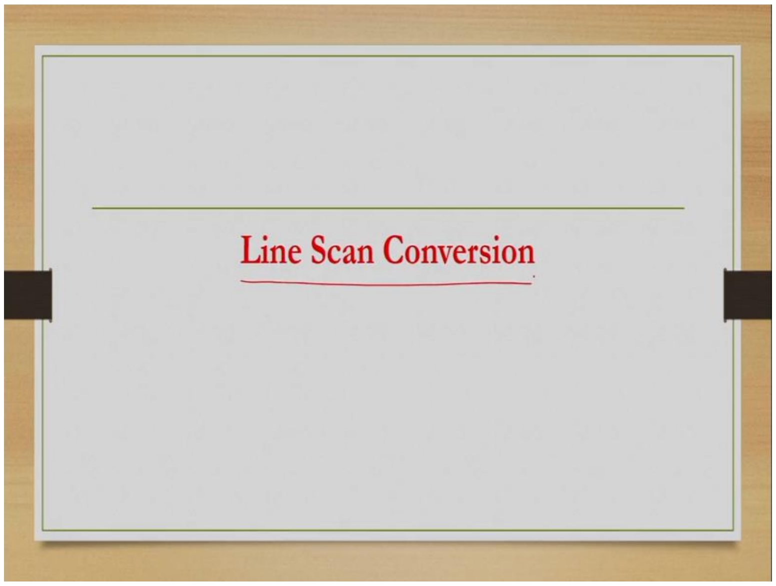
Basic Problem

- Simple approach – round off 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

So what can be the straightforward approach to do this? You may think it is pretty simple. What we can do is simply round off the real coordinates to the nearest integer coordinates. For example, if we have a coordinate value like (2.3, 2.6), we can round it up to (2, 3) that is the nearest integer coordinate values.

However, this may be good for converting points or mapping points from continuous coordinate space to discrete coordinate space, however, same scheme may not be good for lines, circles, or other primitive shapes that are required for rendering a scene. Now, let us try to understand how then we can take care of scan conversion of lines, circles, or other primitive shapes.

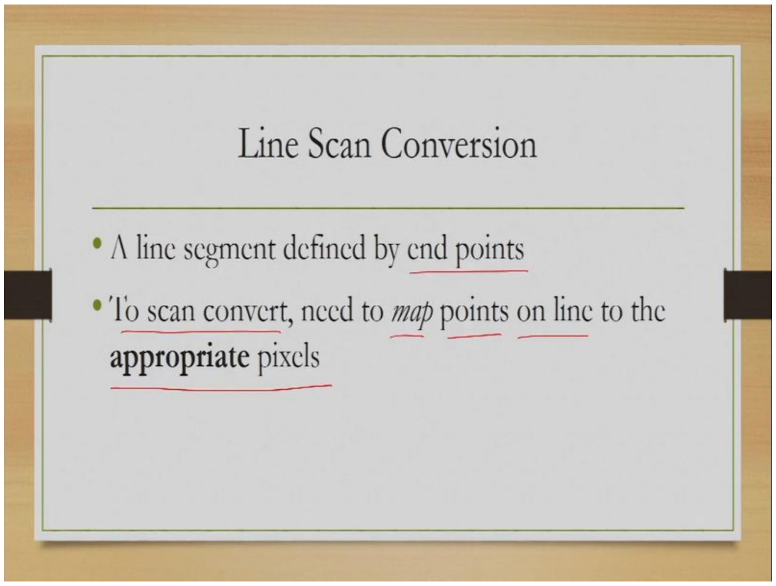
(Refer Slide Time: 08:53)



Line Scan Conversion

Let us start with line scan conversion, how we can basically map a line defined in a continuous coordinate space to a line defined in a discrete coordinate space.

(Refer Slide Time: 09:13)

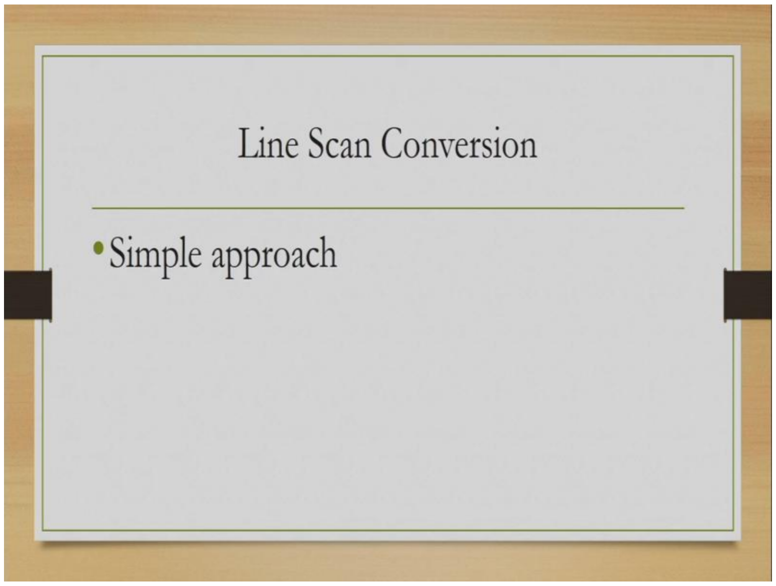


Line Scan Conversion

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

We will start with a very simple and intuitive approach and then, we will try to understand the problem with the intuitive approach, and then, we will introduce better and better approaches. Now, we all know that we can define a lightened segment in terms of its endpoints. So to scan convert, what we need? We need to first map the end points on the line to the appropriate pixels and also the other points that are on the line to the appropriate pixels.

(Refer Slide Time: 10:06)

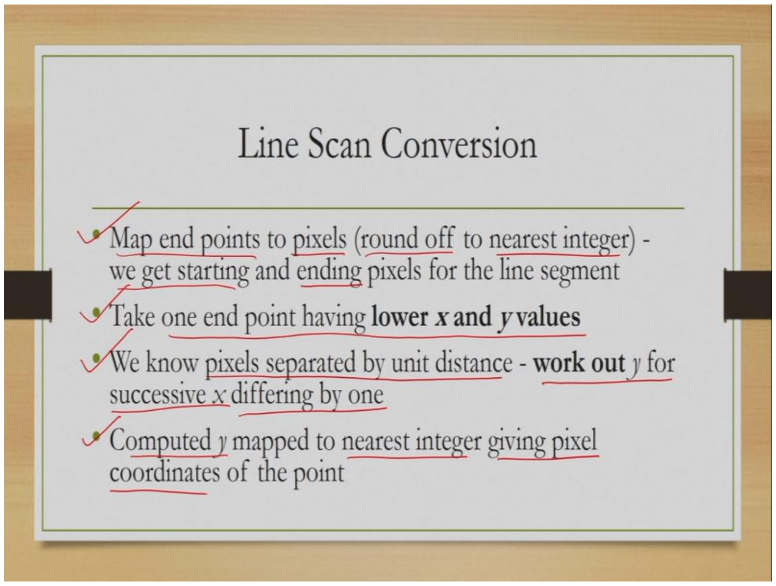


Line Scan Conversion

- Simple approach

Now, let us go through a very simple approach, how we can map the points that are on the line to the nearest pixels in the pixel grid.

(Refer Slide Time: 10:24)



Line Scan Conversion

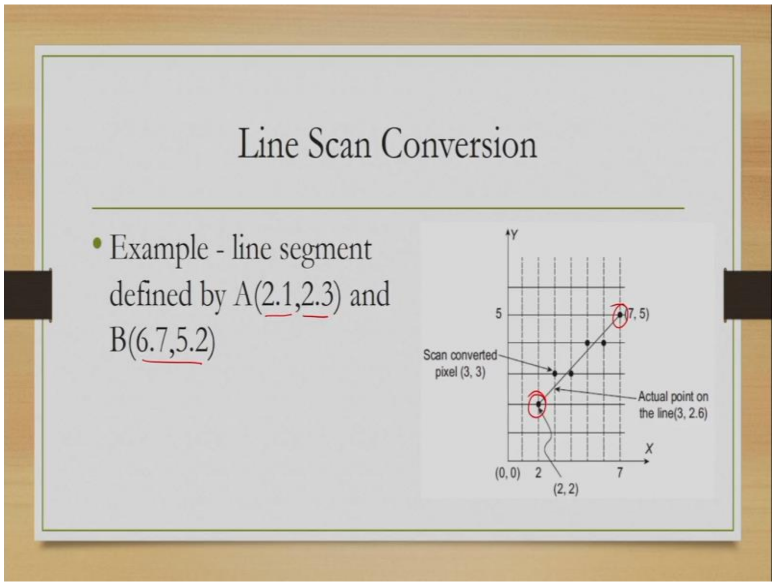
- Map end points to pixels (round off to nearest integer) - we get starting and ending pixels for the line segment
- Take one end point having **lower *x* and *y* values**
- We know pixels separated by unit distance - **work out** *y* for successive *x* differing by one
- Computed *y* mapped to nearest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

So we can follow a four-step approach. In the first step, we map the end points to pixels simply by rounding off to the nearest integer. In that way, we get the starting and ending pixels for the line segment. So we now know which pixels are defining the line. Then in the second step, we take one endpoint having the lower x and y values as the starting point. In the third step, we work out the y value for successive x values.

Now, since we are dealing with a pixel grid, we know that pixels are separated by unit distances. So the successive x values will have a value difference of 1, so these successive values will differ by 1. In the fourth step, this computed y values are mapped to the nearest integer giving us the pixel coordinates of that particular point.

So we first convert the end points to the nearest pixels, then we choose the end point having lower x and y values as the starting point, and starting with that point, we compute y value taking as input the x value, where the successive x values differ by 1 and we continue this till the other endpoint. So we compute all the y values between the starting and ending pixels. And these computed y values are then mapped to the nearest integer values, giving us the pixel coordinates for those points. Let us try to understand this in terms of one example.

(Refer Slide Time: 13:07)

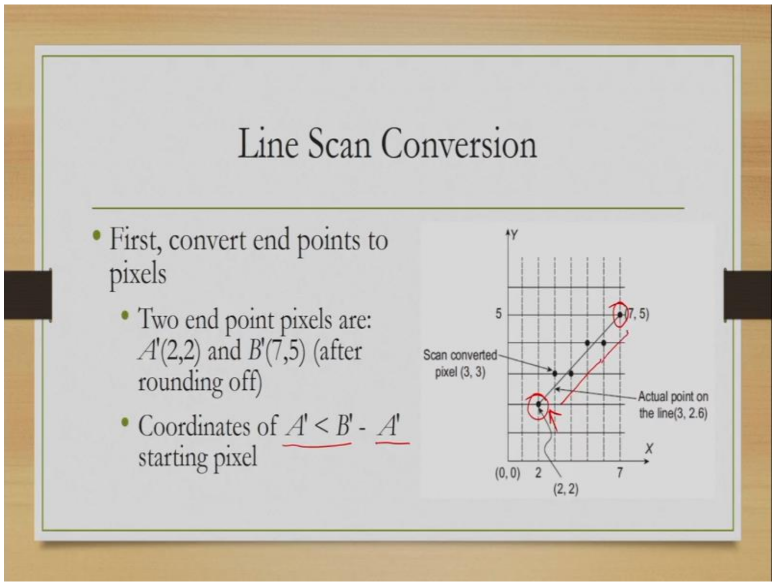


Line Scan Conversion

- Example - line segment defined by A(2.1,2.3) and B(6.7,5.2)

Suppose this is our line segment, as shown in this figure. So this is one endpoint, this one is another endpoint. Initially, the line was defined by these two endpoints, A and B. As you can see, both are real numbers. So we have to map it to the nearest pixel.

(Refer Slide Time: 13:35)



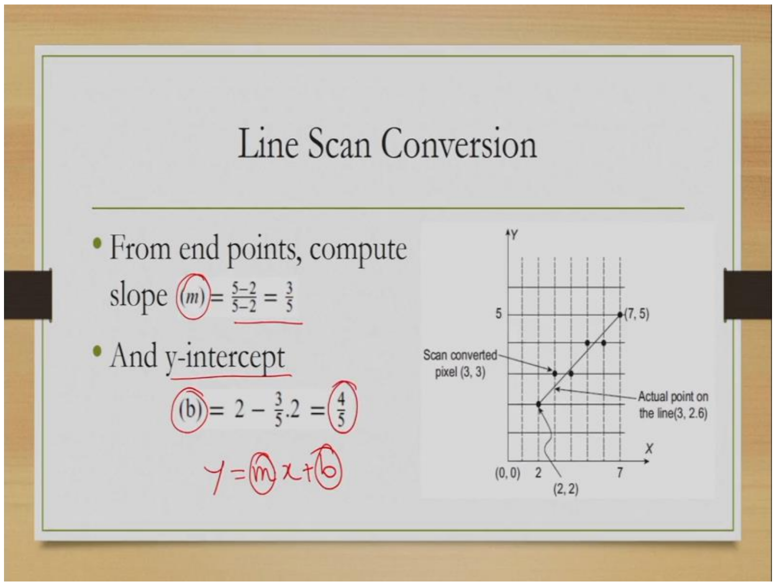
Line Scan Conversion

- First, convert end points to pixels
  - Two end point pixels are: $A'(2,2)$ and $B'(7,5)$ (after rounding off)
  - Coordinates of $\underline{A' < B'}$ - $\underline{A'}$ starting pixel

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

For A, if we do the rounding off, we will get this pixel as the nearest pixel, and for B will get this one as the nearest pixel if we perform the rounding off. Now, we can see that coordinates of A′ is less than B′. So we start with A′, we choose it as our starting pixel. So we start with this pixel and continue finding out the y values till we reach this other pixel, other endpoint.

(Refer Slide Time: 14:21)



Line Scan Conversion

- From end points, compute slope $(m) = \underline{\frac{5-2}{5-2} = \frac{3}{5}}$
- And $\underline{\text{y-intercept}}$

$$(b) = 2 - \frac{3}{5}.2 = \frac{4}{5}$$

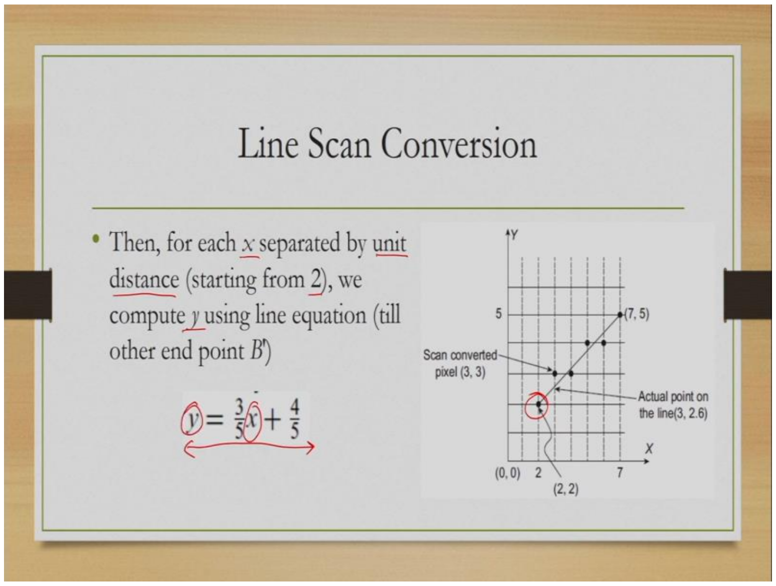
$$y = m x + b$$

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, our objective is to compute the y values for successive x values. For that, we require the line equation which involves computation of the slope m, which in our case turns out to be this.

Because we know the two endpoints we can compute the slope and also, the y-intercept value. We are assuming here that the line is expressed in terms of this equation, y = mx + b, where m is the slope, and b is the y-intercept. Given the two endpoints, we can solve for m and b and find that m is 3 by 5 and b is 4 by 5.

(Refer Slide Time: 15:16)



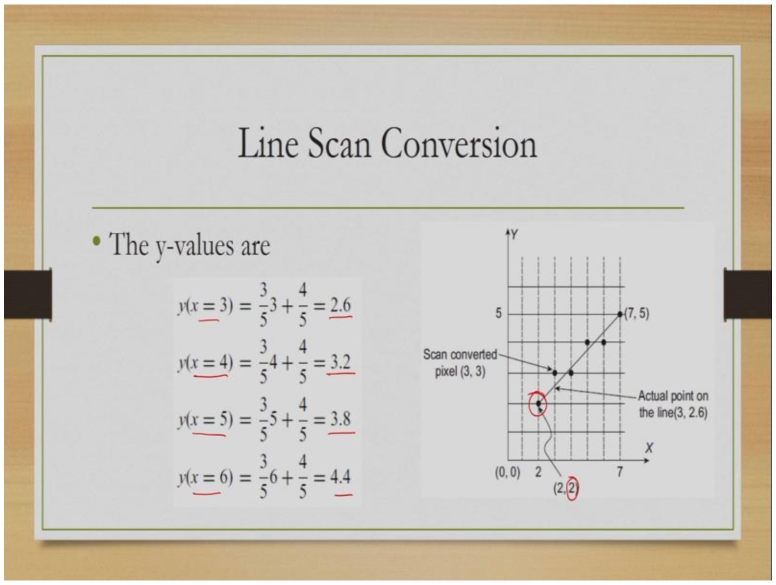
Line Scan Conversion

- Then, for each *x* separated by unit distance (starting from 2), we compute *y* using line equation (till other end point *B*')

$$y = \frac{3}{5}x + \frac{4}{5}$$

Then what we do? For each x separated by unit distance, starting from the lower end pixel that is x value is 2, we compute the y values using the line equation till we reach the other endpoint. And the line equation is given here. So in this equation, we use the x values to get the y values.

(Refer Slide Time: 15:58)



Line Scan Conversion

- The y-values are

$$y(x=3) = \frac{3}{5}3 + \frac{4}{5} = 2.6$$

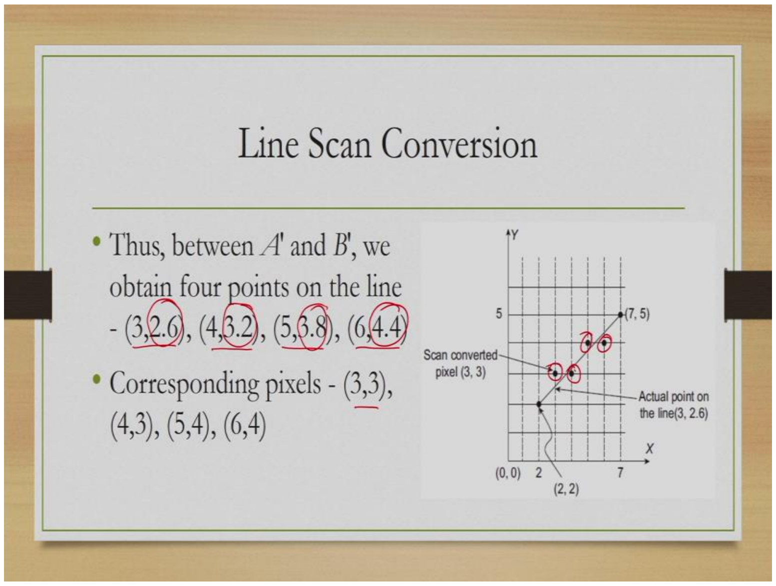
$$y(x=4) = \frac{3}{5}4 + \frac{4}{5} = 3.2$$

$$y(x=5) = \frac{3}{5}5 + \frac{4}{5} = 3.8$$

$$y(x=6) = \frac{3}{5}6 + \frac{4}{5} = 4.4$$

If we do so, what we will find? So for x =2, we have y=2; when x is 3, so the next pixel, x coordinate, we get y is 2.6; when x is 4, then we get y to be 3.2; x=5, y=3.8; x=6, y=4.4. So we get the four intermediate pixels and corresponding y values computed using the line equations.

(Refer Slide Time: 16:45)

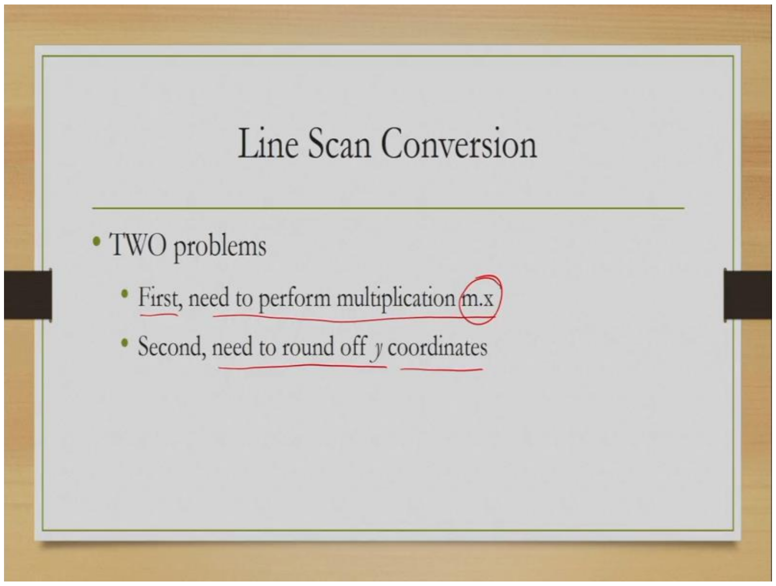


Line Scan Conversion

- Thus, between A' and B', we obtain four points on the line - (3,2.6), (4,3.2), (5,3.8), (6,4.4)
- Corresponding pixels - (3,3), (4,3), (5,4), (6,4)

So the four points are (3, 2.6), (4, 3.2), (5, 3.8), and (6, 4.4). These are the four values that we compute using the line equation. Now, we map this y values as shown here to the nearest integer to get the pixel locations. So if we do the rounding off, we will get the four pixels to be (3, 3) which is here; then (4, 3), which is here; (5, 4) here, and (6, 4) here. So these are our four intermediate pixels corresponding to the points on the line.

(Refer Slide Time: 17:45)

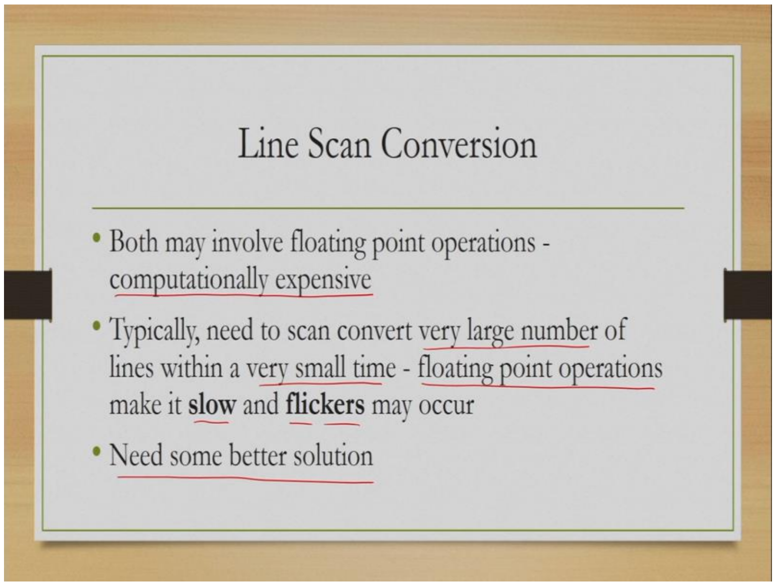


Line Scan Conversion

- TWO problems
  - First, need to perform multiplication m.x
  - Second, need to round off *y* coordinates

Now, with this approach, as you can see, the way we have computed the values and ultimately found out the pixels, there are two problems broadly. First problem is we need to perform multiplication of m and x. Now, m is likely to be a real value, so that is a floating-point operation. Secondly, we need to round off y coordinate values that is also floating-point operation. Together these floating-point operations are computation intensive.

(Refer Slide Time: 18:40)

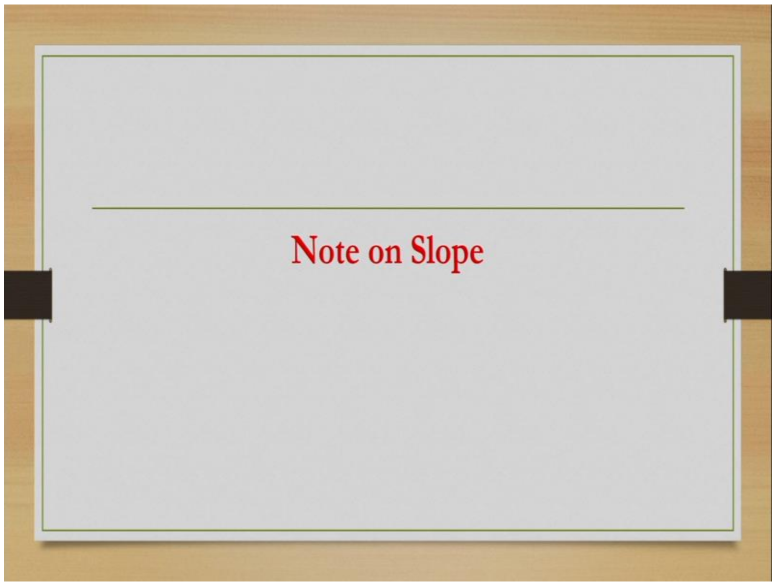


Line Scan Conversion

- Both may involve floating point operations - computationally expensive
- Typically, need to scan convert very large number of lines within a very small time - floating point operations make it **slow** and **flickers** may occur
- Need some better solution

So we have a computationally expensive approach to convert a line to corresponding pixels. In reality, we need to scan convert very large number of lines within a very small time. Now, if we have floating-point operations involved, then this process will become slow and we will perceive flickers, which is of course something which we do not want. So what do we need, we need some better solutions. Let us have a look at it slightly better approach.

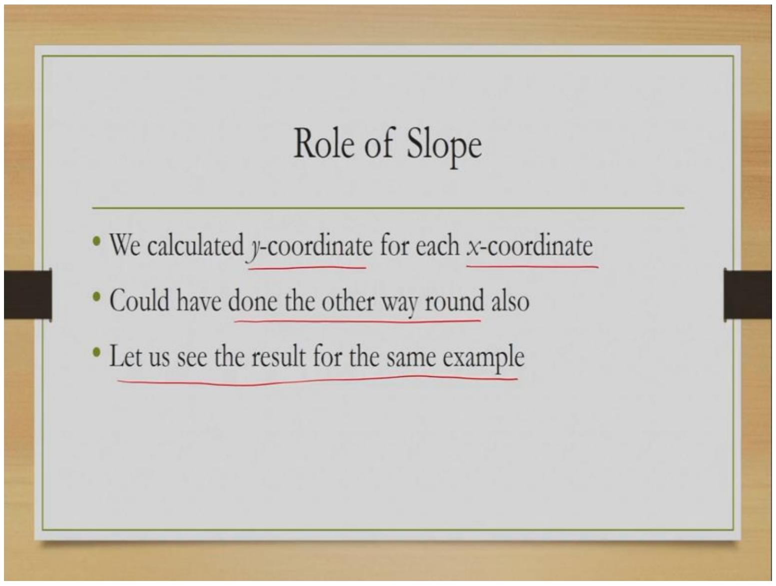
(Refer Slide Time: 19:29)



Note on Slope

But before that, I would like to point your attention towards another important topic that is consideration of slope of a line when we are performing this scan conversion.

(Refer Slide Time: 19:47)

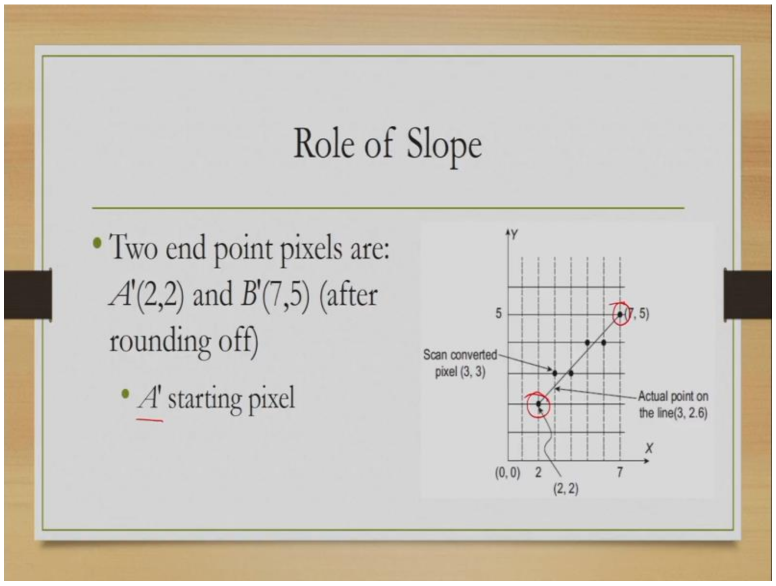


Role of Slope

- We calculated *y*-coordinate for each *x*-coordinate
- Could have done the other way round also
- Let us see the result for the same example

In our example, what we did, we calculated the y coordinate values for each x coordinate value. So we increased x by 1 and corresponding y values, we calculated using the line equation. You may wonder why we did so, we could have similarly done it the other way around. We could have increased y and calculated the x values. Let us try to see what happens if we do so if we calculate x values by increasing y values.

(Refer Slide Time: 20:26)

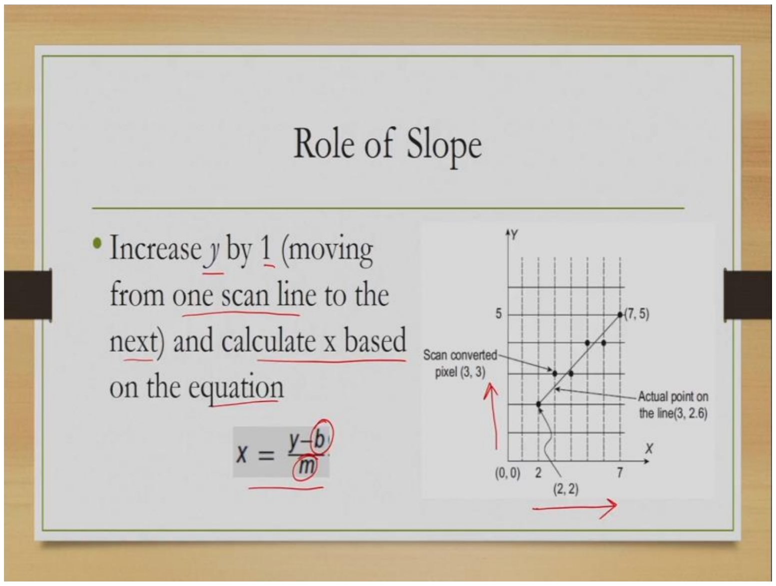


Role of Slope

- Two end point pixels are: *A*'(2,2) and *B*'(7,5) (after rounding off)
  - *A*' starting pixel

Now, we have these two endpoint pixels and the lower pixel is the starting point denoted by A′.

(Refer Slide Time: 20:41)



Role of Slope

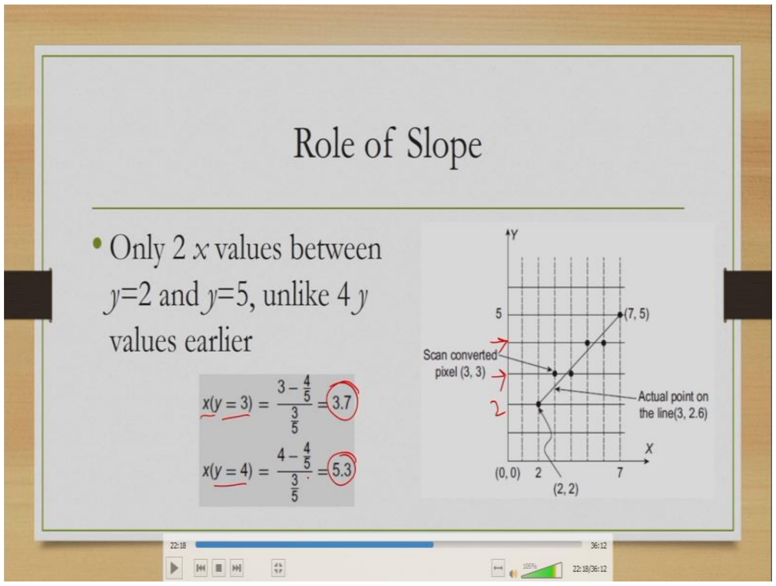
- Increase *y* by 1 (moving from one scan line to the next) and calculate x based on the equation

$$x = \frac{y-b}{m}$$

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

This time we are increasing y by 1. That means we are moving this way. Earlier, we were moving along this direction, now we are moving this way from one scan line to the next. And then, we calculate the x based on the equation. Now, we need a modified equation which is given here and we already know b and m, the y-intercept and slope respectively. So we simply replace the y value to get the x value.

(Refer Slide Time: 21:24)



Role of Slope

- Only 2 *x* values between *y*=2 and *y*=5, unlike 4 *y* values earlier

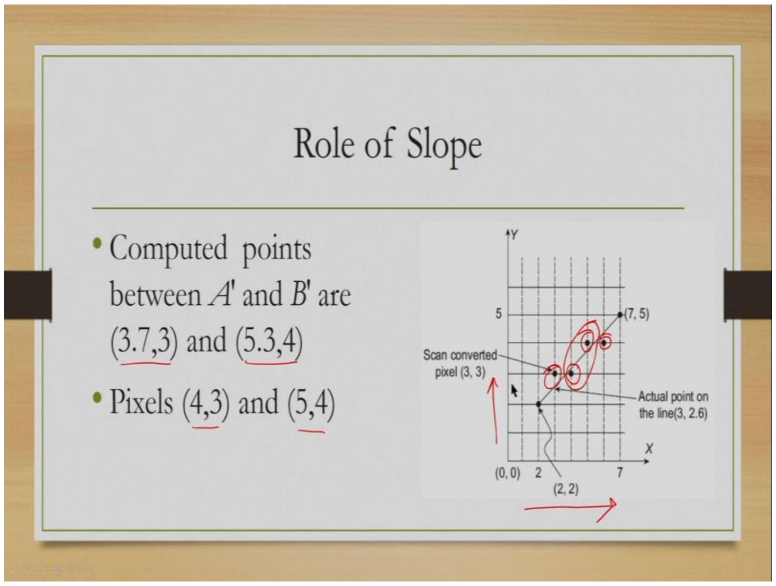
$$x(y=3) = \frac{3-\frac{4}{5}}{\frac{3}{5}} = 3.7$$

$$x(y=4) = \frac{4-\frac{4}{5}}{\frac{3}{5}} = 5.3$$

Y
5
(7, 5)
Scan converted pixel (3, 3)
Actual point on the line(3, 2.6)
X
(0, 0)
2
7
(2, 2)
2
22:18
36:12

Now, if we do so, we will see that we have only two successive y values, this one and this one between y=2 and y=5. So earlier, we computed four x values, this time we required to compute only two values because we have only two increases of y between the endpoints. So we are required to calculate two x values. So when y=3, then the x turns out to be 3.7 using the equation, and when y=4, x is 5.3.

(Refer Slide Time: 22:19)



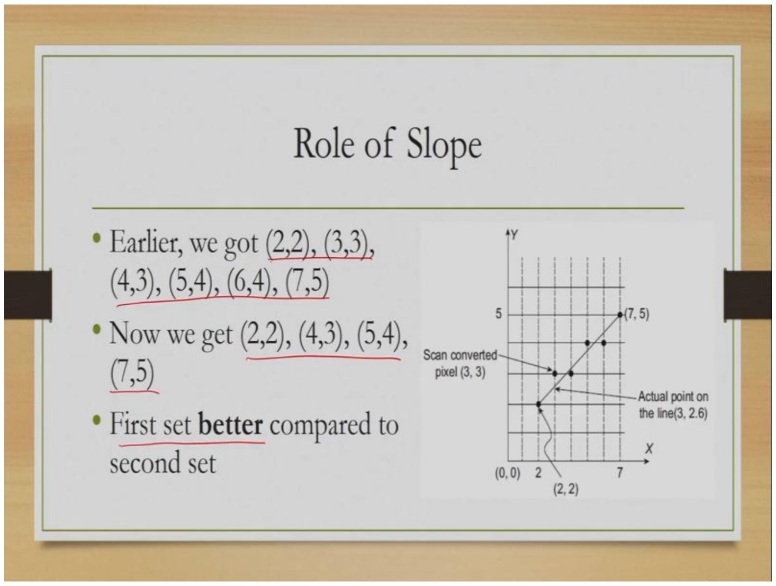
Role of Slope

- Computed points between A' and B' are (3.7,3) and (5.3,4)
- Pixels (4,3) and (5,4)

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 what we have computed between the two end points? Two new points, (3.7, 3) and (5.3, 4). If we round it off to the nearest integers, then we get the pixels (4, 3) and (5, 4); let us place it here. So (4, 3) is this point and (5, 4) is this point. Note that earlier, we got two additional points when we moved along x-direction, now we are getting only two points, these two, when we are moving along y-direction and computing x.

(Refer Slide Time: 23:11)

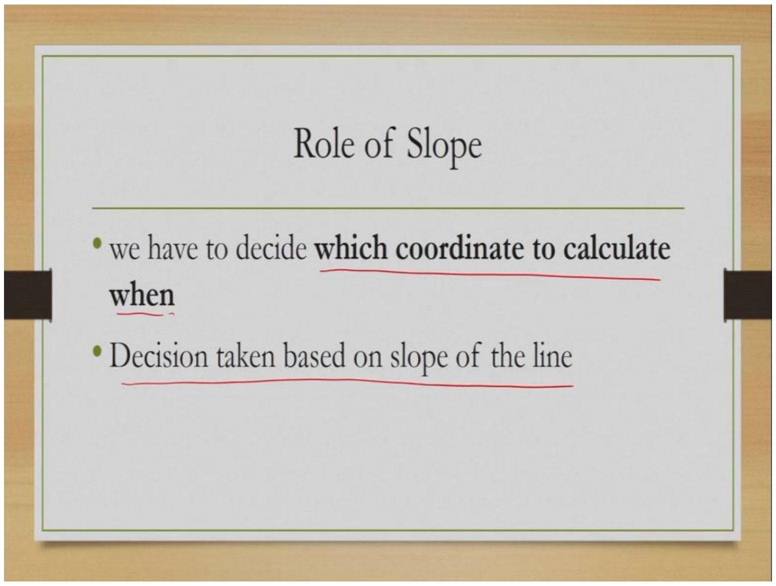


Role of Slope

- Earlier, we got (2,2), (3,3), (4,3), (5,4), (6,4), (7,5)
- Now we get (2,2), (4,3), (5,4), (7,5)
- First set **better** compared to second set

Clearly, the first set that is these 4 plus the 2 endpoints, total 6, pixels will give us a better approximation to the line compared to the second set consisting of total 4 pixels, the two endpoints, and the two newly computed pixels. So you have the first set, which is better than second set because the approximation is better due to the larger number of pixels.

(Refer Slide Time: 23:55)

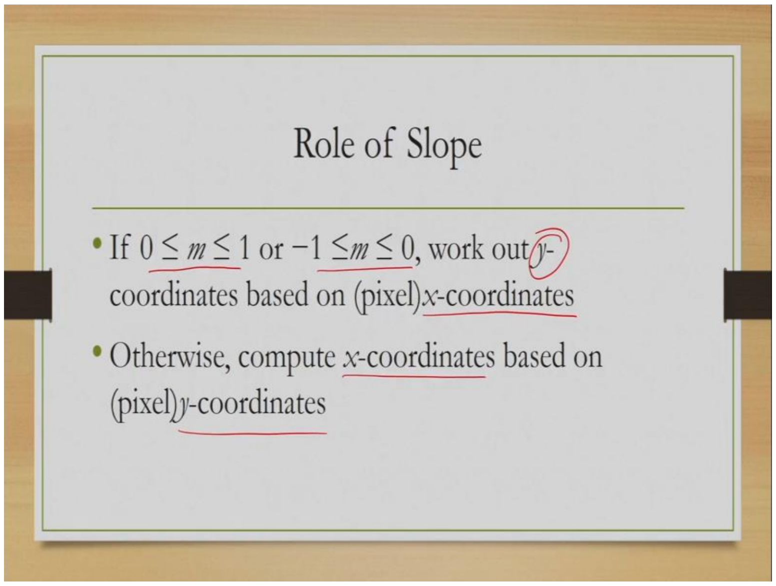


Role of Slope

- we have to decide **which coordinate to calculate when**
- Decision taken based on slope of the line

Now, that is the issue here. How do we decide which coordinate to calculate and when? Should we start with x and calculate y, or should we increase y and calculate x? Now, this decision is taken based on the slope of the line, depending on the slope we take a call.

(Refer Slide Time: 24:30)

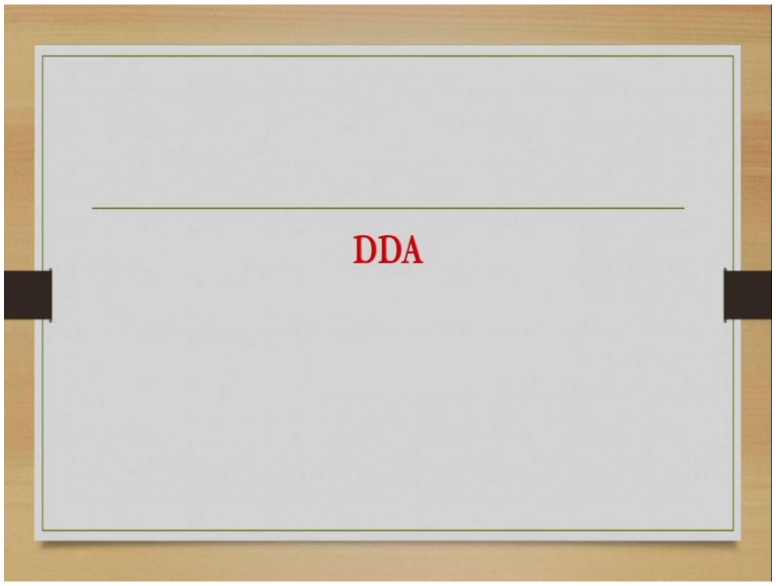


Role of Slope

- If $0 \leq m \leq 1$ or $-1 \leq m \leq 0$, work out y-coordinates based on (pixel) x-coordinates
- Otherwise, compute x-coordinates based on (pixel) y-coordinates

If the slope is within these ranges, then we work out or calculate y values based on x coordinates of the pixels. So you increase x by 1 and compute the y values when m is within this range. If m is not within this range, then we compute x by increasing y coordinates. So that is our rule. So when m is within the ranges given here, then we compute y based on x, where x indicates the pixel coordinates that means integers. Otherwise, that means when m is not within this range, we compute x based on y where y indicates the pixel coordinates in integers. So that is how we make the decision.

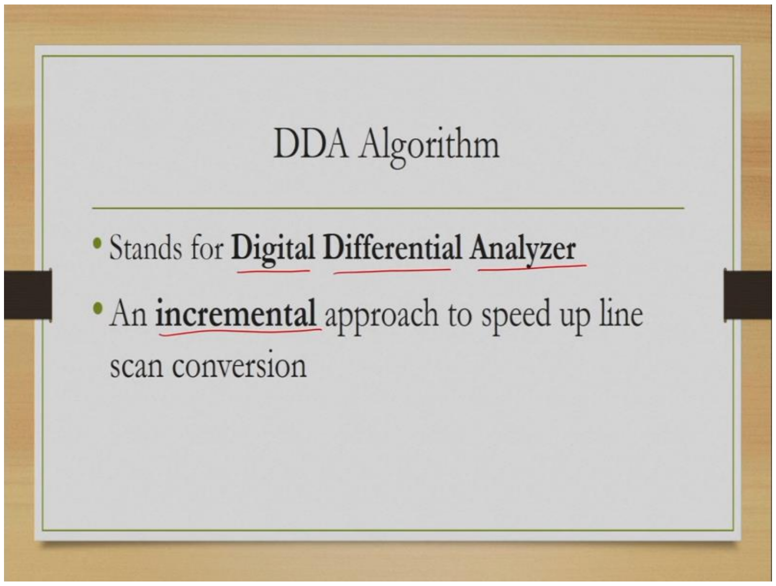
(Refer Slide Time: 25:48)



DDA

Now, let us go back to a better line scan conversion algorithm compared to the simple approach that we have learned earlier. So this approach is called DDA or digital differential analyzer.

(Refer Slide Time: 26:06)

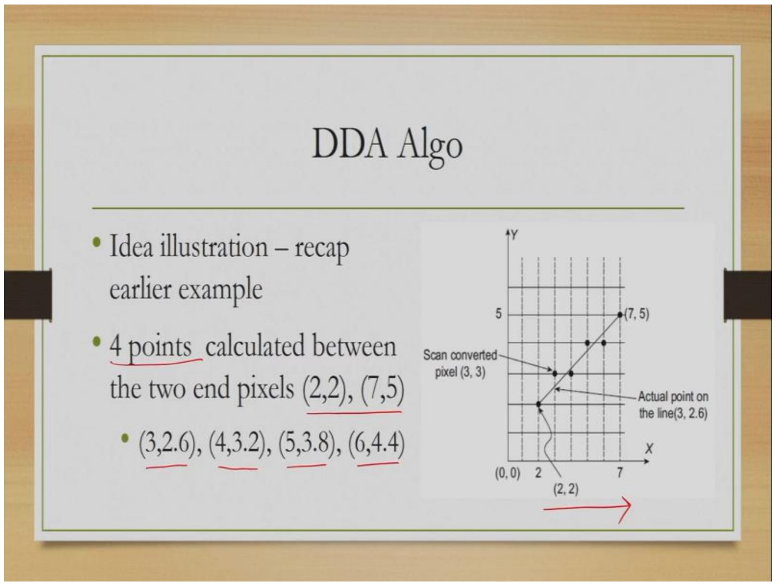


DDA Algorithm

- Stands for **Digital Differential Analyzer**
- An **incremental** approach to speed up line scan conversion

DDA stands for digital differential analyzer, and this is an incremental approach which is developed to reduce floating-point operations. That means to increase the computation speed, speed up the scan conversion process.

(Refer Slide Time: 26:32)

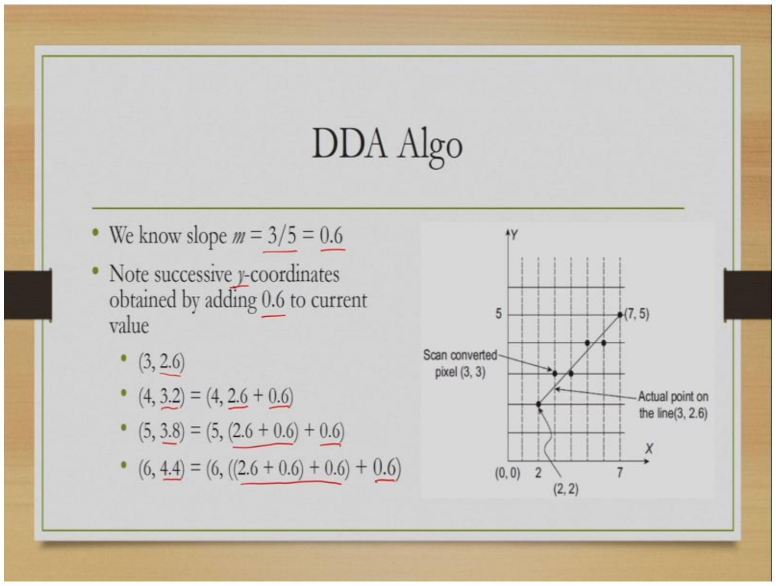


DDA Algo

- Idea illustration – recap earlier example
- 4 points calculated between the two end pixels (2,2), (7,5)
  - (3,2.6), (4,3.2), (5,3.8), (6,4.4)

Let us try to first understand the idea. We will use the same example that we have seen earlier but this time, we will note a few more points. So earlier, we computed 4 points between the two end points (2, 2) and (7, 5) by increasing x and computing y values. Now, these 4 points are (3, 2.6), (4, 3.2), (5, 3.8) and (6, 4.4). Now, we will have a closer look at these points, what they tell us.

(Refer Slide Time: 27:17)



DDA Algo

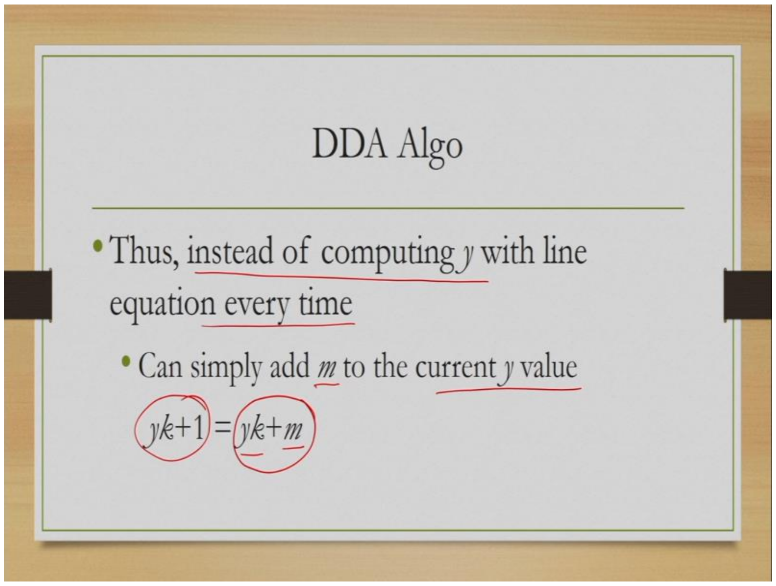
- We know slope $m = 3/5 = 0.6$
- Note successive y-coordinates obtained by adding 0.6 to current value
  - (3, 2.6)
  - (4, 3.2) = (4, 2.6 + 0.6)
  - (5, 3.8) = (5, (2.6 + 0.6) + 0.6)
  - (6, 4.4) = (6, ((2.6 + 0.6) + 0.6) + 0.6)

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

We computed that slope is 3/5 or 0.6. Now, the successive y values are actually addition of this slope value to the current value. The first value that we got is 2.6. Second value that we got is 3.2, which we can get by adding 0.6 that is the slope value to the earlier value that is 2.6.

Next value we got is 3.8, which is again the earlier value plus the slope. Finally, we got 4.4, which is again the earlier value plus slope. So there is a pattern. We add the slope value to the earlier value to get the new value. This idea is exploited in this DDA algorithm.

(Refer Slide Time: 28:31)



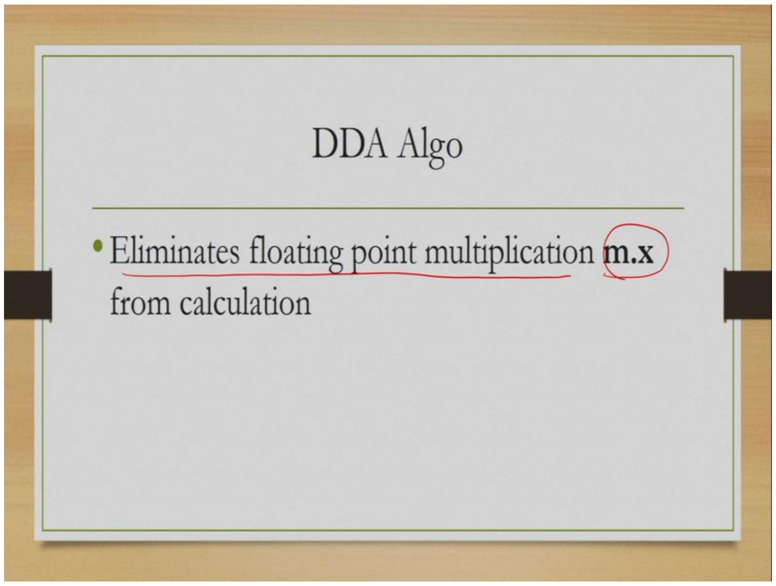
DDA Algo

- Thus, instead of computing y with line equation every time
  - Can simply add m to the current y value

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

So instead of computing y with the line equation every time, we can simply add m to the current y value. That means the new y we can get by adding m to the current value. So we do not need to go for solving the line equation every time we want to compute the y value.

(Refer Slide Time: 28:59)

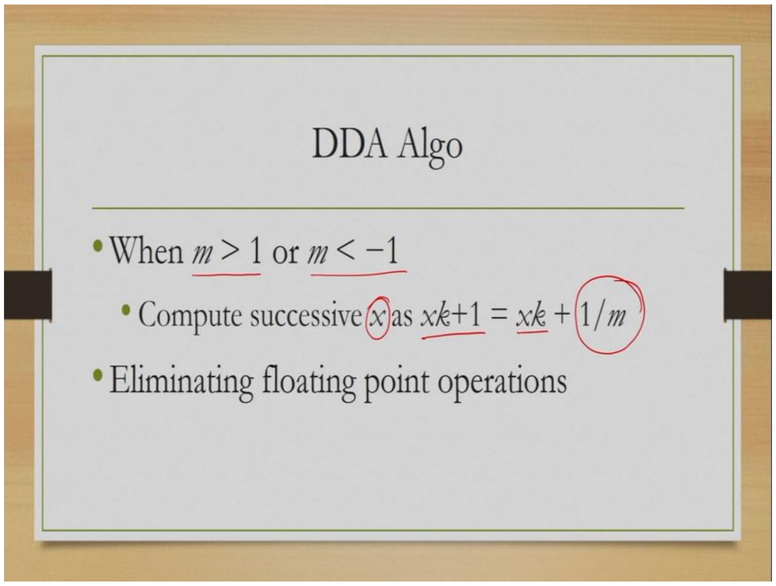


DDA Algo

- Eliminates floating point multiplication **m.x** from calculation

What is the advantage of that? It eliminates floating-point multiplication which is involved in this computation that is m into x. So we can eliminate these calculations which in turn is going to reduce the computational complexities.

(Refer Slide Time: 29:28)

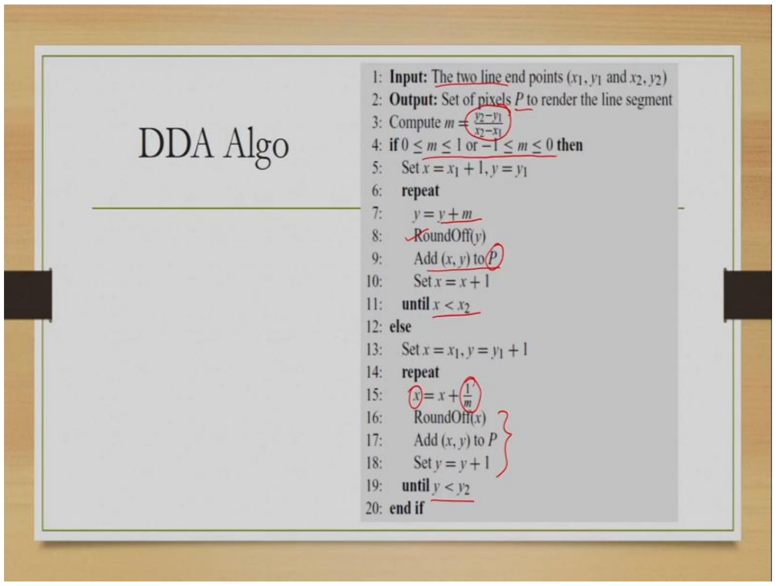


DDA Algo

- When $m > 1$ or $m < -1$
  - Compute successive x as $xk+1 = xk + 1/m$
- Eliminating floating point operations

Now, as I said earlier, slope is an important consideration here. So when the slope is not within the range that means the slope is greater than 1 or less than minus 1, then we do not compute successive y values, instead we compute x values. Again, in a similar way that is new value is the old value plus a constant term, which in this case is 1/m, earlier it was only m. So we can obtain the new x value by adding this constant 1/m to the current value. And here also, by this, we are eliminating floating-point operations.

(Refer Slide Time: 30:19)



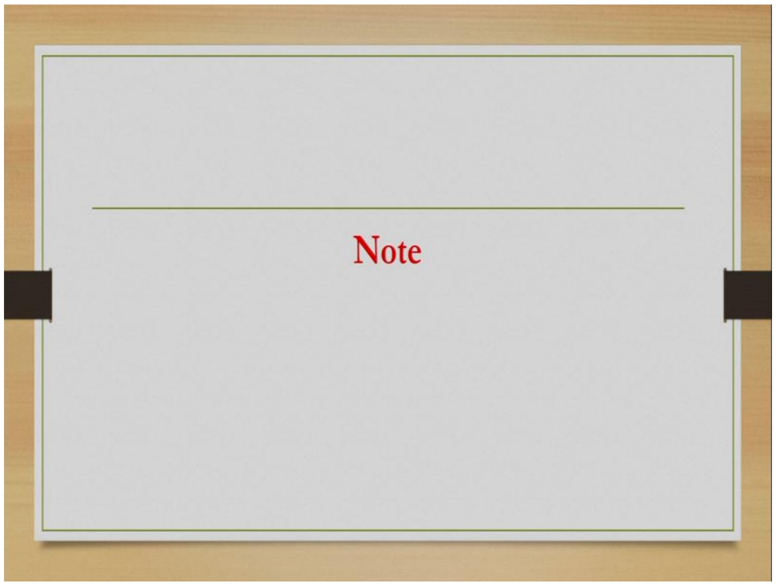
DDA Algo

```
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 $0 \leq m \leq 1$ or $-1 \leq m \leq 0$ 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
```

So the complete algorithm is shown here. The input is the endpoint, the two endpoints are the input. And the output is the set of all pixels that are part of the line. So we compute m. Now, when m is within this range, we compute successive y values as shown here, by adding m to the current y value, round it off to get the pixel, and add the pixel to the set. And when m is not within this range, we compute successive x values by adding 1/m to the current value and perform the same steps again.

So we continue in both the cases till the other end point as you can see in this loop termination conditions. So that is how we improve on the simple lines scan conversion approach by exploiting one particular property that is, we can compute the successive x or y values by simply adding a constant term.

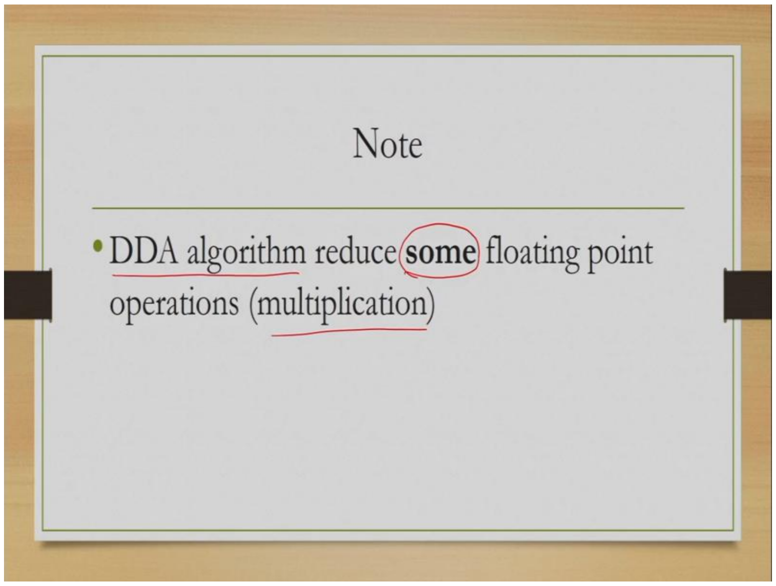
(Refer Slide Time: 31:58)



Note

This is clearly some improvement over the simple approach. However, there are still issues.

(Refer Slide Time: 32:07)

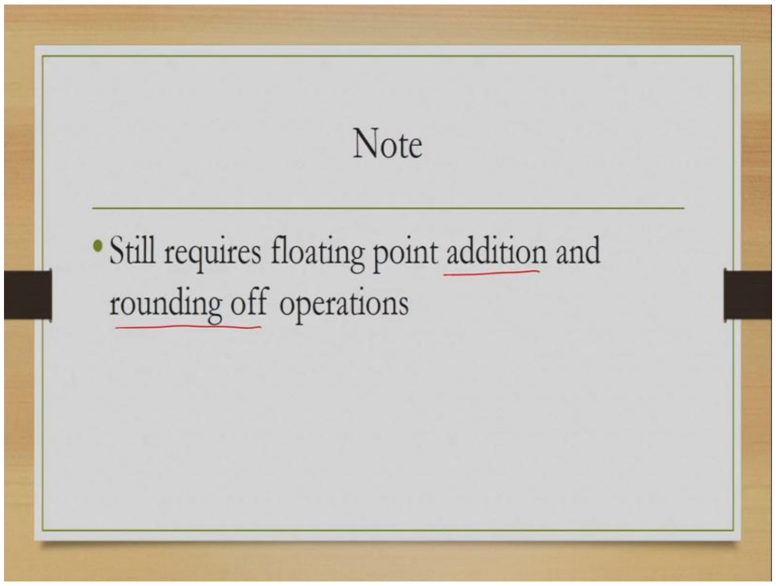


Note

- DDA algorithm reduce **some** floating point operations (multiplication)

With the DDA algorithm as we have noted, we can reduce floating-point operations, but only some of those floating-point operations. We cannot remove all, we can only reduce multiplications.

(Refer Slide Time: 32:30)

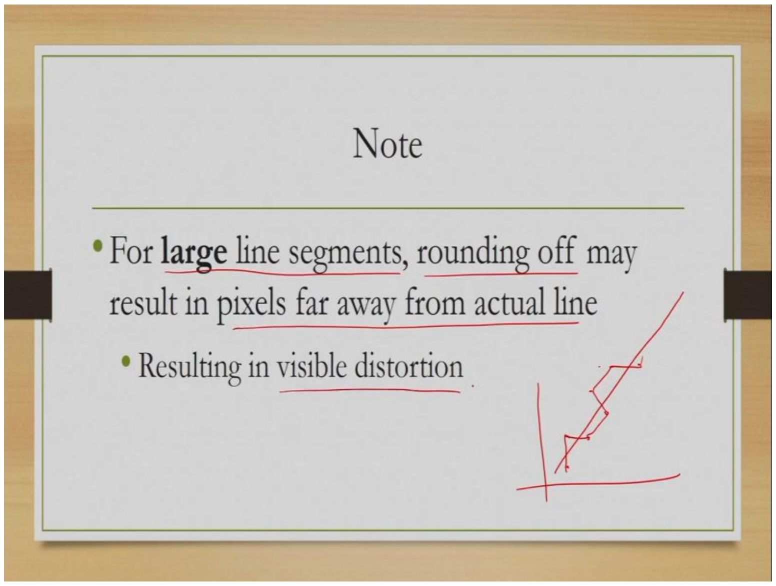


Note

- Still requires floating point addition and rounding off operations

That still leaves us with other floating-point operations, which are addition and rounding off. Now, any floating-point operation is computationally expensive and it involves additional resources. So when we, in reality, require to generate large number of line segments in a very short span of time, our ideal objective should be to eliminate all floating-point operations altogether, rather than eliminating few. Eliminating few, of course, improves the overall rendering rate, but eliminating all should be our ultimate objective.

(Refer Slide Time: 33:22)

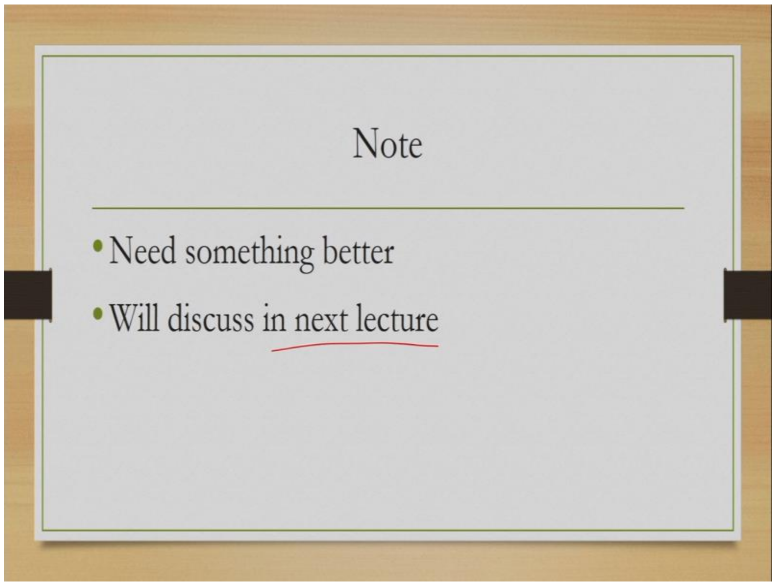


Note

- For **large** line segments, rounding off may result in pixels far away from actual line
  - Resulting in visible distortion

That is so, since, for large line segments or large number of line segments, this floating-point operations may create problem. Particularly, when we are dealing with a very large line segment, the rounding off may result in pixels far away from actual line, for example, consider a very big line like this. Now, if we perform rounding off then we make keep on getting pixel, like, something like this. This actually looks distorted line for large line segments. For small segments, this may not be the case, but for large line segments, there is a possibility of visible distortion, which of course, we do not want.

(Refer Slide Time: 34:28)

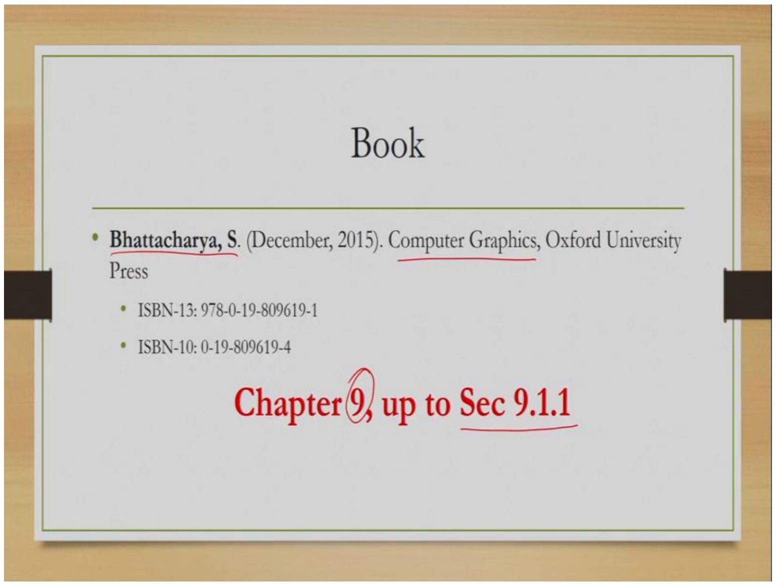


Note

- Need something better
- Will discuss in next lecture

That is one problem of course, plus our ultimate objective is to remove all floating-point operations because along with this distortion, they also increases the time to render a line segment, and also requires resources. So we need a better solution than what is provided by DDA. One such approach we will discuss in the next lecture.

(Refer Slide Time: 35:06)



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, up to Sec 9.1.1

So whatever we have discussed today can be found in this book, Computer Graphics. You are advised to go through Chapter 9 up to Section 9.1.1 to know in more details whatever we have discussed. So the improved line drawing algorithm will be taken up in the next lecture. Till then, thank you and goodbye.