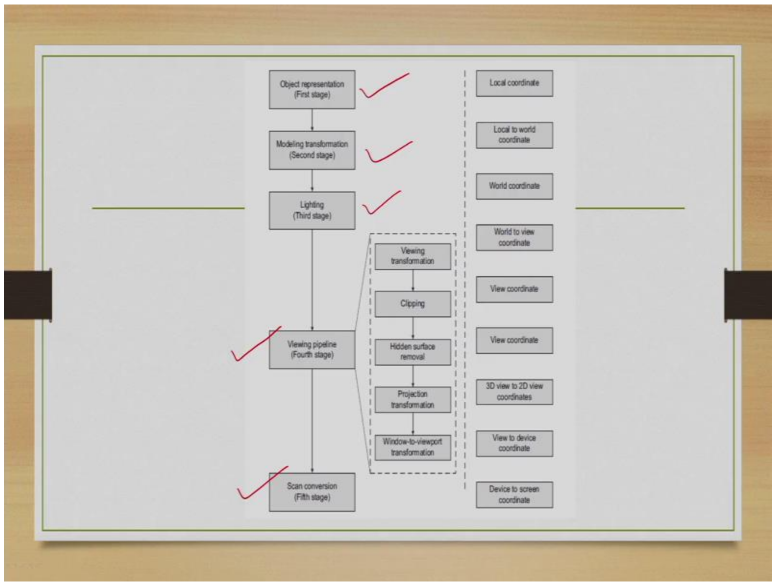
Computer Graphics
Professor Doctor Samit Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Lecture 27
Fill Area and Character Scan Conversion

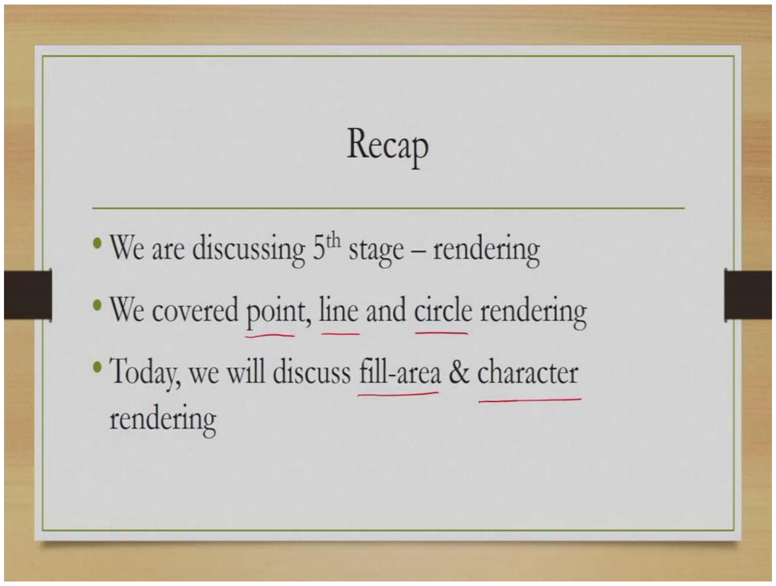
Hello and welcome to lecture number 27 in the course computer graphics. We are discussing the 3D graphics pipeline, as you may recollect it has 5 stages and we have already discussed 4 stages in details and currently we are in the fifth stage. So, let us just have a quick relook at the 5 stages.

(Refer Slide Time: 00:55)


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. We have already discussed first stage in details, object representation, then second stage modelling transformation, third stage lighting or assigning colour, fourth stage viewing pipeline and currently we are at the fifth stage scan conversion or rendering.

(Refer Slide Time: 01:23)



Recap

- We are discussing 5th stage – rendering
- We covered point, line and circle rendering
- Today, we will discuss fill-area & character rendering

Now, in scan conversion what we do, we essentially try to map description of an image given in the device coordinate system to a description on the pixel grid that means, set up pixels. So, in the earlier lectures we have covered the methods that are followed for such mapping, for point, line and circle.

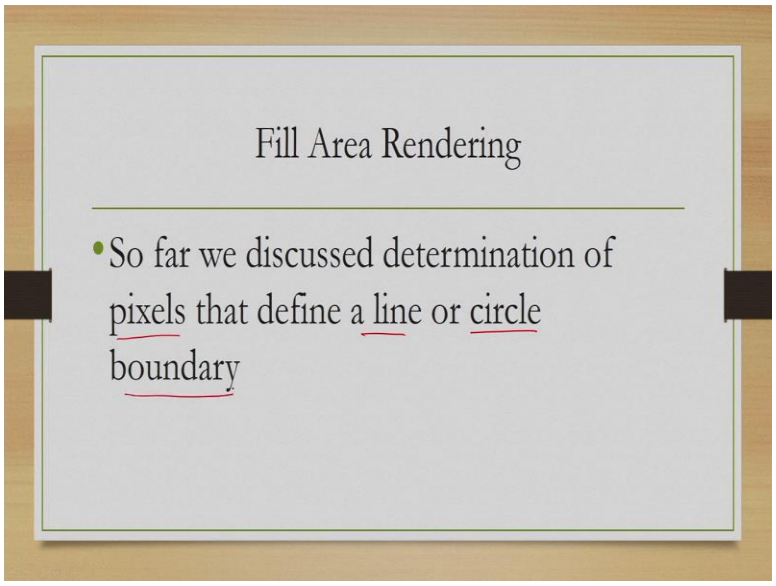
And we have seen how we can improve efficiency of these methods by introducing better approaches such as the Bresenham's line drawing algorithm used for line scan conversion, midpoint algorithm for circle scan conversion and so on. Today we are going to discuss another scan conversion technique related to fill areas, along with that we will also discuss how we display characters that means the letters, numbers etcetera on the screen. We will try to get a broad idea on character rendering.

(Refer Slide Time: 02:56)

Fill Area Rendering

Let us, start with fill area rendering. So, first let us try to understand what is a fill area.

(Refer Slide Time: 03:05)

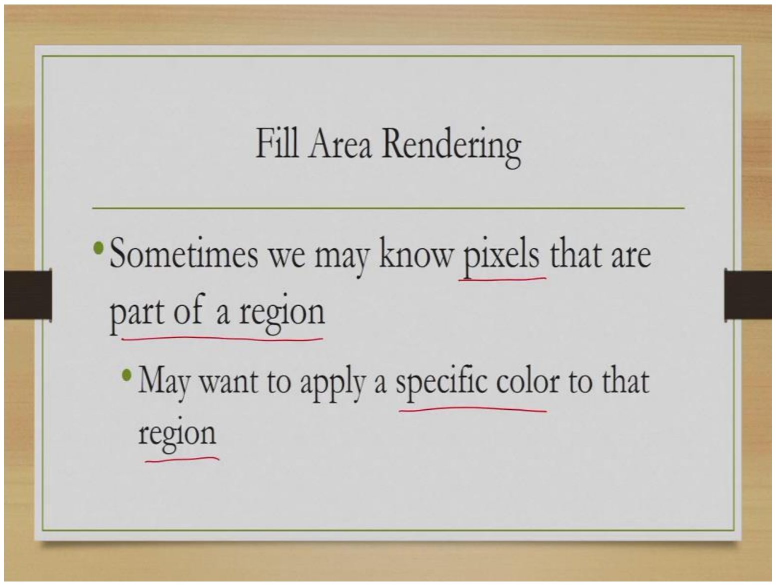


Fill Area Rendering

- So far we discussed determination of pixels that define a line or circle boundary

What we have discussed so far, how to determine pixels that define a line or a circle boundary.

(Refer Slide Time: 03:24)

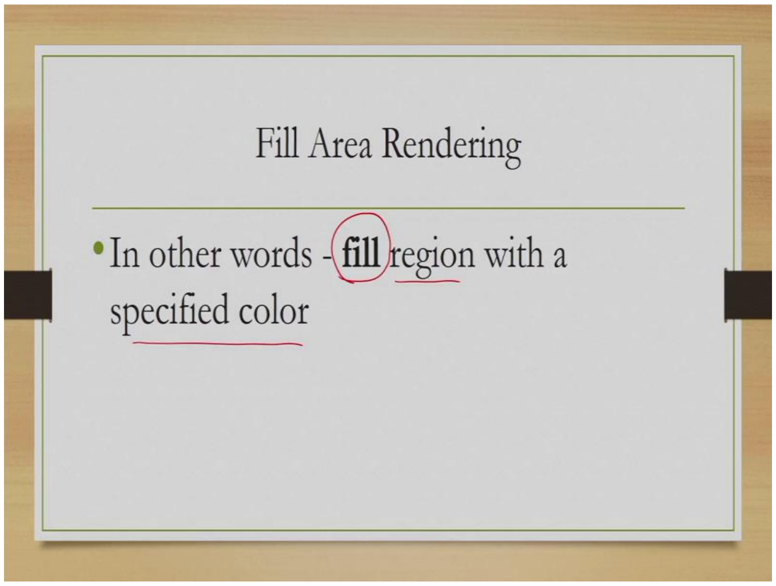


Fill Area Rendering

- Sometimes we may know pixels that are part of a region
  - May want to apply a specific color to that region

Sometimes that may not be the case, sometimes we may know pixels that are part of a region. And we may want to apply a specific colour to that whole region. So, earlier what we did? We determined pixels that are part of a single line or the circle boundary, but sometimes there may be situations where we may go for assigning colours to region rather than a line or a boundary.

(Refer Slide Time: 04:09)

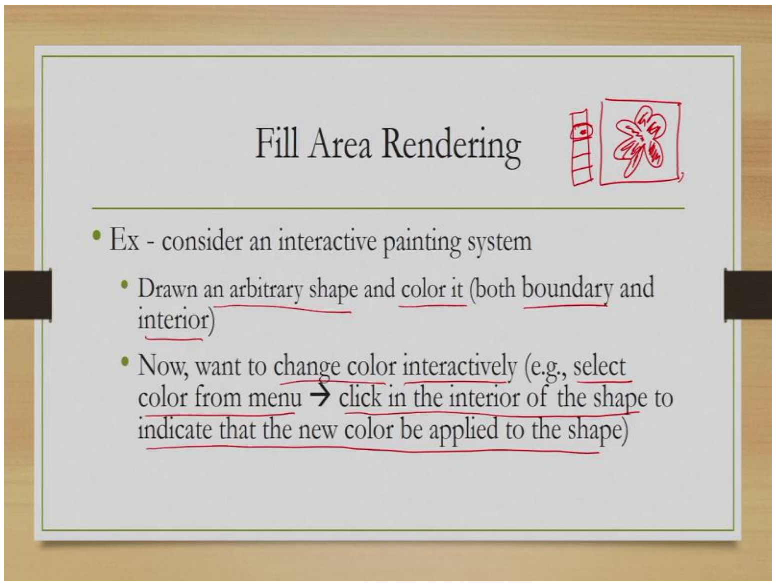


Fill Area Rendering

- In other words - **fill** region with a specified color

Now, that is the same as saying that we want to fill a region with a specified colour. So, that is fill area rendering, one of the topics of our discussion today. So, when we are talking about fill area rendering we are referring to a region and our objective is to fill that entire region that means the pixels that are part of that region with a specified colour. This is in contrast to what we have learned earlier where our objective was to find out pixels and of course assign colours to them which are part of a line or the boundary of a circle.

(Refer Slide Time: 05:02)



Fill Area Rendering

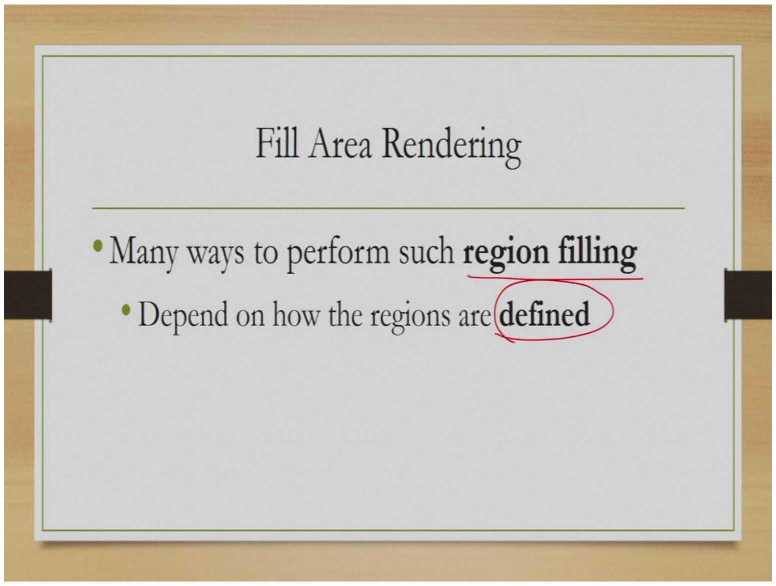
- Ex - consider an interactive painting system
  - Drawn an arbitrary shape and color it (both boundary and interior)
  - Now, want to change color interactively (e.g., select color from menu $\rightarrow$ click in the interior of the shape to indicate that the new color be applied to the shape)

Let us, try to understand this concept with an example. Consider an interactive painting system, so in that system what we do? We may try to draw any arbitrary shape and then we may wish to assign some colours to that shape, that means assign colours inside the boundary of that set. Also, we may want to change the colour, so first thing is we may want to colour it, colour that arbitrary shape that we have drawn.

Now, when we say we are trying to colour some shape that means we want to colour the boundary as well as the interior. We may also want to change colour and that too interactively that means select some colour from a menu, click in the interior of the shape to indicate that the new colour to be applied to that shape. If you have used some interactive painting system, then you maybe already familiar with these things.

For example, suppose this is our canvas and here we have drawn a shape something like this, then there may be a menu of colour or say colour palette, so we may choose this menu, say for example this colour click our mouse pointer or touch some point inside this shape and then the centre colour is applied in the interior of this shape. So, that is interactive colouring of a shape. And here as you can see, we are concerned about colouring a region rather than only the boundary, unlike what we did when we were trying to determine pixels as well as their colours for lines or circle boundaries

(Refer Slide Time: 07:26)

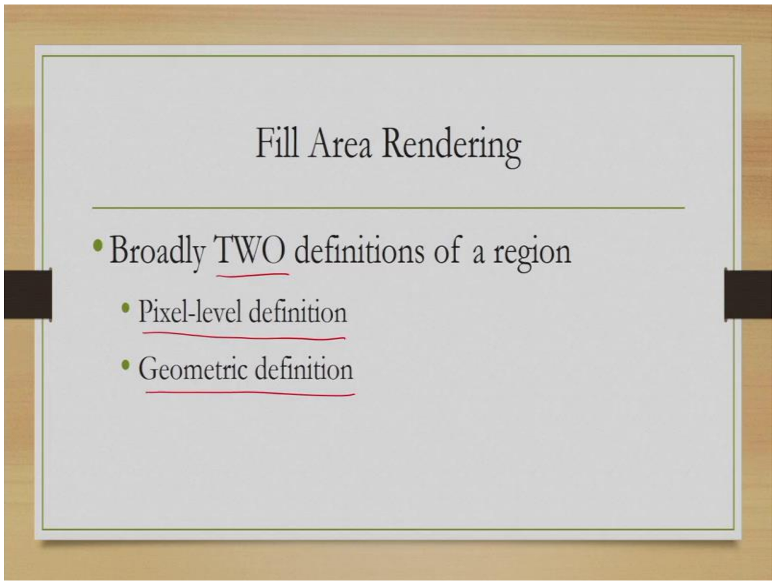


Fill Area Rendering

- Many ways to perform such **region filling**
  - Depend on how the regions are **defined**

The question is how we can perform such colouring or region filling? Now, that depends on how the regions are defined, so there can be different ways to define a region and depending on that definition we can have region filling approaches.

(Refer Slide Time: 07:50)

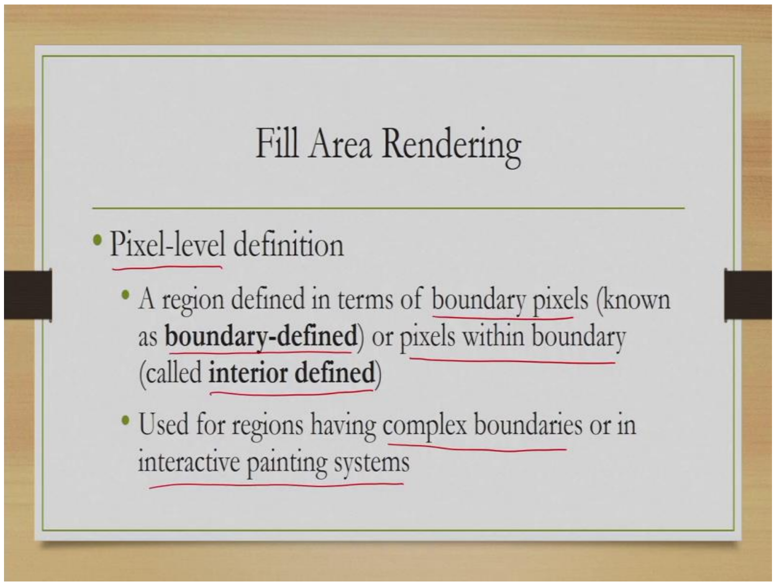


Fill Area Rendering

- Broadly TWO definitions of a region
  - Pixel-level definition
  - Geometric definition

Broadly there are two definitions of a region, one is pixel level definition one is geometric definition.

(Refer Slide Time: 08:02)



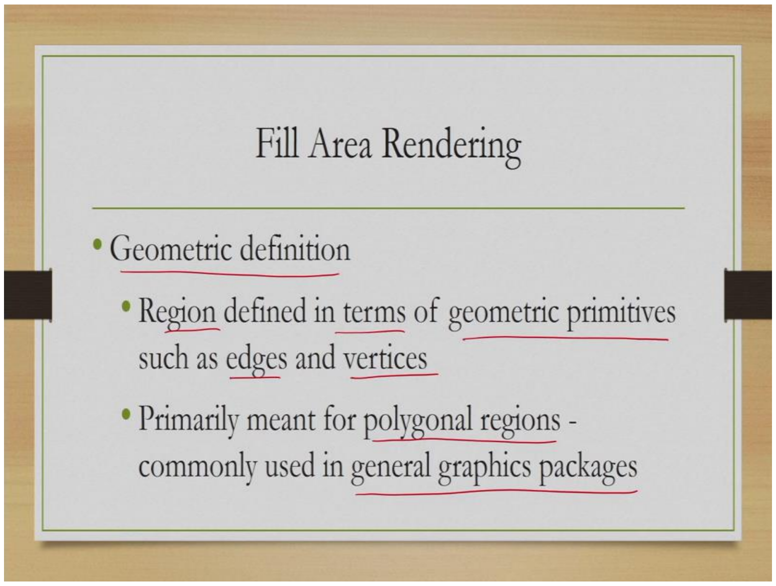
Fill Area Rendering

- Pixel-level definition
  - A region defined in terms of boundary pixels (known as **boundary-defined**) or pixels within boundary (called **interior defined**)
  - Used for regions having complex boundaries or in interactive painting systems

In case of a pixel level definition we define a region in terms of pixels that means we may define the region in terms of boundary pixels or we may define the region in terms of pixels within a boundary. In the first case when we are defining a region in terms of boundary pixels or the set of pixels that define the boundary such definition is called boundary defined.

In the other case we do not explicitly define a boundary but set of pixels that defines the whole region in that case we call it interior defined. So, such pixel definitions are useful when we are dealing with regions having complex boundaries or as we have just seen applications such as interactive painting systems. So, for complex shapes, it is difficult to deal with the boundary, so their pixel level definition may be useful. Also in interactive systems pixel level definitions are very useful.

(Refer Slide Time: 09:28)



Fill Area Rendering

- Geometric definition
  - Region defined in terms of geometric primitives such as edges and vertices
  - Primarily meant for polygonal regions - commonly used in general graphics packages

The other type of fill area definition is geometric definition, here we define a region in terms of the geometric primitives such as edges and vertices this we have already seen before during our object representation techniques. Now, this particular approach is primarily meant for polygonal regions. And these definitions are commonly used in general graphics packages, which we have already mentioned earlier.

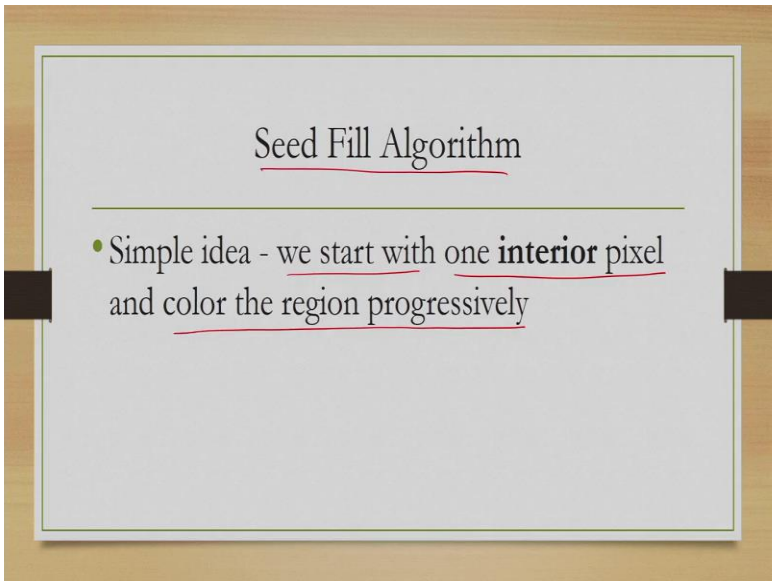
So, essentially geometric definitions means, defining a region in terms of geometric primitives such as edges, vertices, if you may recollect we have discussed such things during our discussion on objective representation where we used vertex list, edge list to define objects or regions. And when we are dealing with geometric definitions, they are primarily meant to define regions that are polygonal in shape.

(Refer Slide Time: 10:52)

Seed Fill Algorithm

Now, with this knowledge of two broad definitions of regions, let us try to understand the different region filling scan conversion algorithms. We will start with one simple approach that is called seed fill algorithm. So, what it does let us try to understand.

(Refer Slide Time: 11:16)

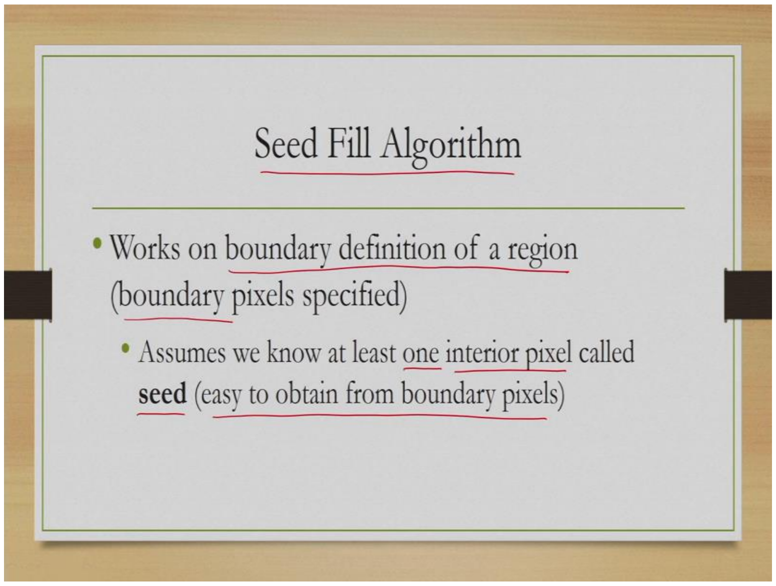


Seed Fill Algorithm

- Simple idea - we start with one **interior** pixel and color the region progressively

So, the idea is very simple for a seed fill algorithm, we start with one interior pixel and colour the region progressively, that is the simple idea.

(Refer Slide Time: 11:33)

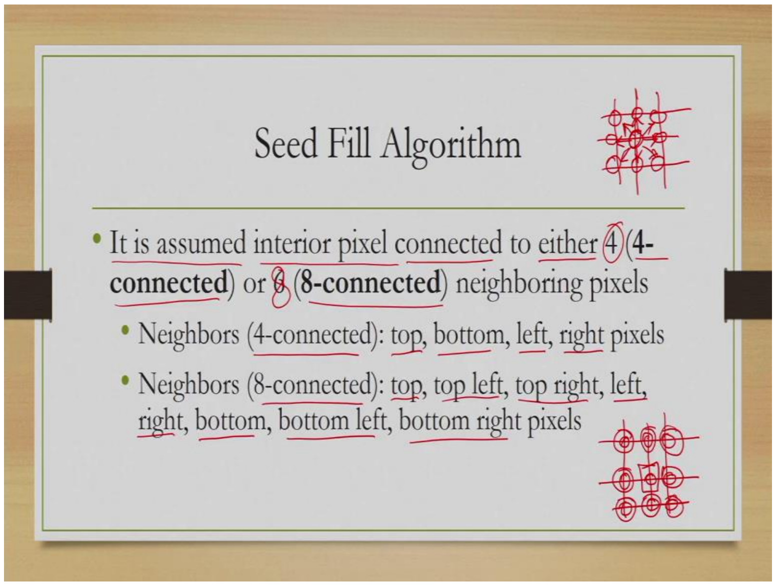


Seed Fill Algorithm

- Works on boundary definition of a region (boundary pixels specified)
  - Assumes we know at least one interior pixel called **seed** (easy to obtain from boundary pixels)

Clearly here we are assuming a pixel level definition particularly, a boundary definition of a region, where the boundary pixels are specified. And we also assume that we know at least one interior pixel, now that pixel is called the seed pixel and if we know the boundary pixels we can decide on any seed pixel, it is easy, because we are dealing with a seed pixel, so the algorithm is named seed fill algorithm. So, we have a seed pixel and we have boundary definitions of the region in terms of pixels.

(Refer Slide Time: 12:26)



Seed Fill Algorithm

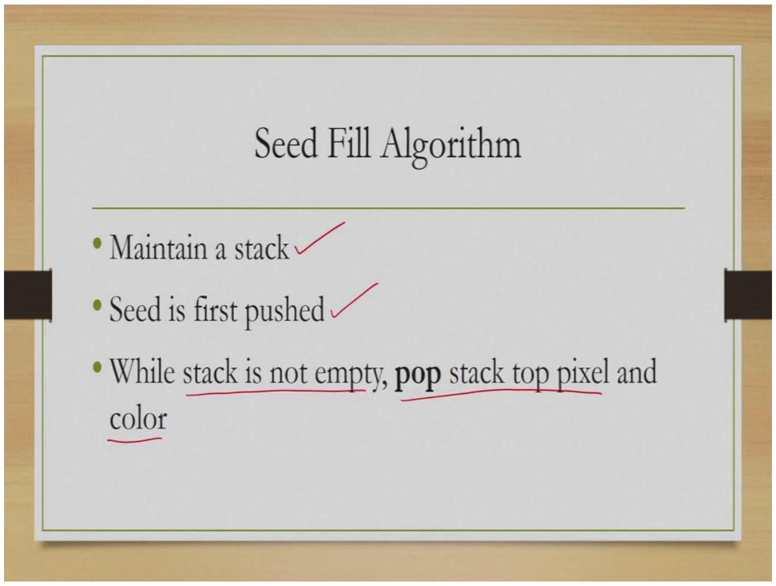
- It is assumed interior pixel connected to either 4 (**4-connected**) or 8 (**8-connected**) neighboring pixels
  - Neighbors (4-connected): top, bottom, left, right pixels
  - Neighbors (8-connected): top, top left, top right, left, right, bottom, bottom left, bottom right pixels

Next in this algorithm, it is also assumed that interior pixels are connected to other pixels in either of the two ways, either they can be connected to 4 pixels which is called 4 connected or they can be connected to 8 pixels, which is called 8 connected. Now, these are the neighbouring pixels for example suppose this is a seed pixel and there are pixels around, if this is the grid, where the circle show the pixels, then these pixels can be assumed to be connected to either 4 neighbouring pixels or all 8 neighbouring pixels.

Accordingly, the nature of connection is called 4 connected or 8 connected. So, when we are talking of 4 connected, we essentially assume that let us redraw the figure again, suppose these are the pixels these intersection points of the grid, this is one pixel, now in case of 4 connected the 4 neighbouring pixels are defined as top, bottom, left and right that means this is the top, this is bottom, this is right and this is left.

Whereas when we are dealing with 8 connected pixels, we are dealing with the 8 neighbours top, top left, this is the top left, then top right here, then left, right, bottom, bottom left this is here and bottom right here. So, either of these connections we can assume. And accordingly the algorithm is executed.

(Refer Slide Time: 15:05)

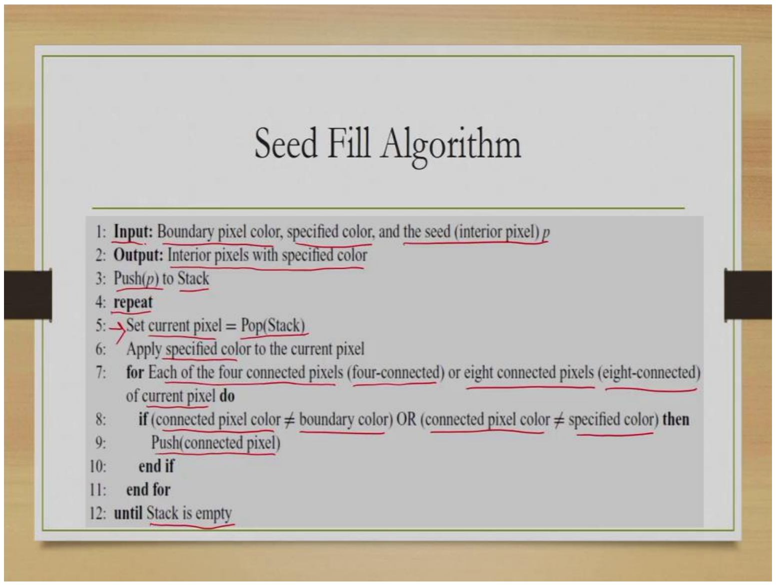


Seed Fill Algorithm

- Maintain a stack ✓
- Seed is first pushed ✓
- While stack is not empty, **pop** stack top pixel and color

So, the basic idea is simple we maintain a stack, the seed pixel is first pushed into the stack and then a loop executed till the stack is not empty. Now, in each step, we pop the stack top pixel and assign the desired colour to that pixel.

(Refer Slide Time: 15:37)



Seed Fill Algorithm

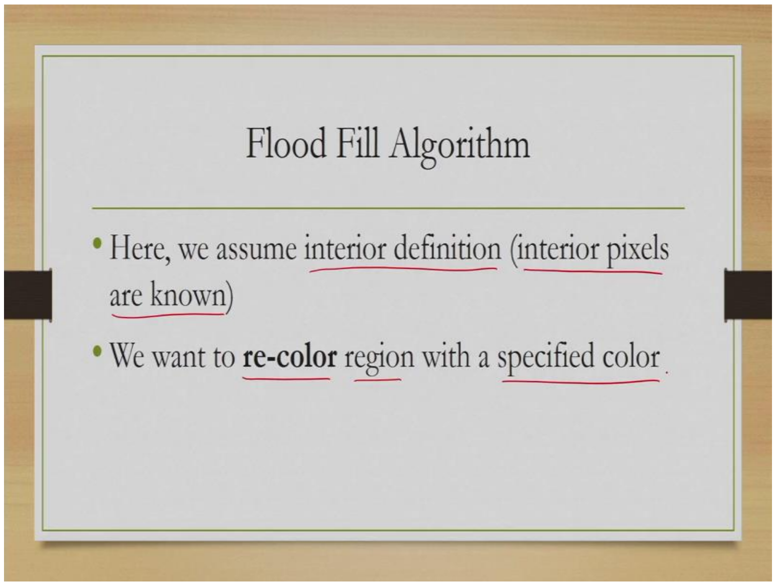
```
1:  Input: Boundary pixel color, specified color, and the seed (interior pixel) p
2:  Output: Interior pixels with specified color
3:  Push(p) to Stack
4:  repeat
5:    Set current pixel = Pop(Stack)
6:    Apply specified color to the current pixel
7:    for Each of the four connected pixels (four-connected) or eight connected pixels (eight-connected) of current pixel do
8:      if (connected pixel color $\neq$ boundary color) OR (connected pixel color $\neq$ specified color) then
9:        Push(connected pixel)
10:     end if
11:   end for
12: until Stack is empty
```

The algorithm is shown here. So, what is our input? The boundary pixel colour, specified colour, which we want to assign to the region and the seed or interior pixel, anyone seed pixel and the output is the interior pixels with specified colour that is our objective. We start with pushing the seed pixel to a stack and then we enter into a loop where we set the current pixel to be the stack top pixel by popping it from the stack, apply specified colour to that pixel, then we make use of the connected property.

So, if we are assuming that it is a 4 connected pixel, then for each of the 4 connected pixels or if we are assuming that it is 8 connected pixel then for each of the 8 connected pixels of the current pixel, what we do? We check if connected pixel colour is not equal to boundary colour that means we have not reached the boundary or the connected pixel colour is not equal to the specified colour that means we are yet to assign it any colour, then we push it to the stack.

So, for each pixel we push either 4 connected pixels to the stack or 8 connected pixels to the stack depending on the nature of connectedness that we are assuming. And then we come back here and the loop continues still the stack is empty that means we have reached the boundary or we have assigned colours to all the interior pixels. That is the simple idea of the seed fill algorithm. Next we will discuss another approach which is called flood fill. The idea is almost similar with some minor variations. Let us see how it works.

(Refer Slide Time: 18:03)

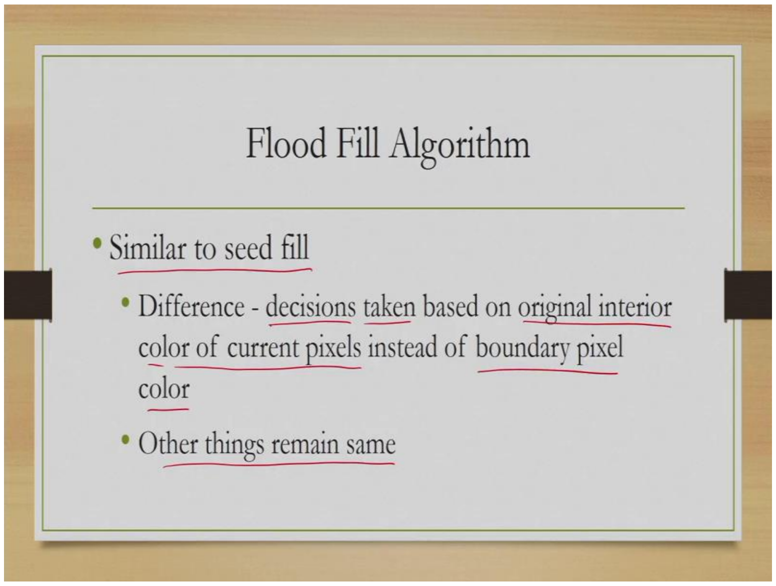


Flood Fill Algorithm

- Here, we assume interior definition (interior pixels are known)
- We want to **re-color** region with a specified color

Now, in case of flood fill algorithm, we assume a different definition which is an interior definition that means the interior pixels are known. Earlier, we assumed boundary definition with only one interior pixel, that is the seed pixel. Now, here we are assuming interior definition, that means all the interior pixels are known. And our objective is to colour or recolour the region with a specified colour.

(Refer Slide Time: 18:38)

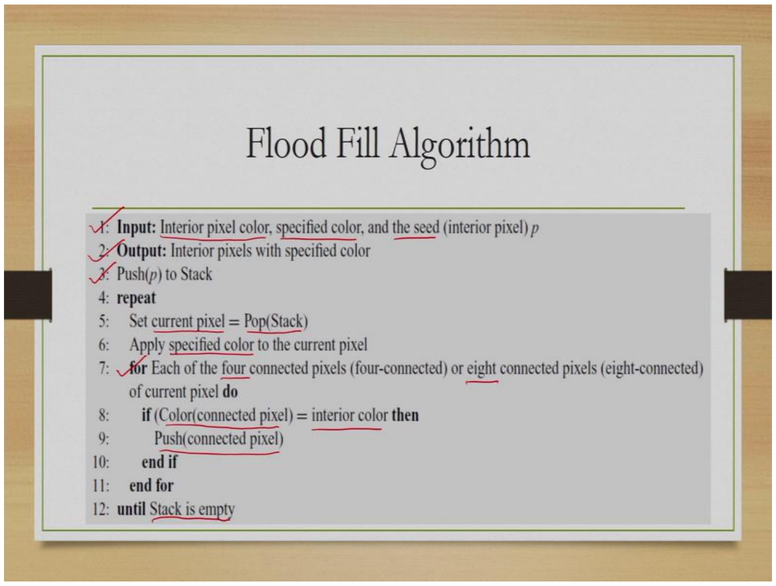


Flood Fill Algorithm

- Similar to seed fill
  - Difference - decisions taken based on original interior color of current pixels instead of boundary pixel color
  - Other things remain same

The idea is similar to seed fill, with some difference. Now, in this case the decisions are taken based on original interior colour of the current pixels instead of boundary pixel colour. Other things remain the same, that means using a stack and utilizing the stack elements in a particular way, colouring them in a particular way remains the same.

(Refer Slide Time: 19:17)



Flood Fill Algorithm

```
1: Input: Interior pixel color, specified color, and the seed (interior pixel) p
2: Output: Interior pixels with specified color
3: Push(p) to Stack
4: repeat
5:   Set current pixel = Pop(Stack)
6:   Apply specified color to the current pixel
7:   for Each of the four connected pixels (four-connected) or eight connected pixels (eight-connected)
     of current pixel do
8:     if (Color(connected pixel) = interior color then
9:       Push(connected pixel)
10:    end if
11:  end for
12: until Stack is empty
```

So, the algorithm is shown here again input is the interior pixel colour, specified colour and the one interior pixel or seed pixel, it is even more easy here because we already know the interior pixels and we can randomly pick up one pixel and the output is after assigning colours to all the pixels the set of pixels.

Now, we push the seed pixels to stack and as before we enter a loop, first we pop the stack and set it to the current pixel applying specified colour, then assuming connectedness as we did before, we deal with either 4 or 8 connected pixels and for each pixel we do the check, now here the check is slightly different as compared to what we did earlier. Here we check if the colour of the connected pixel is the interior colour.

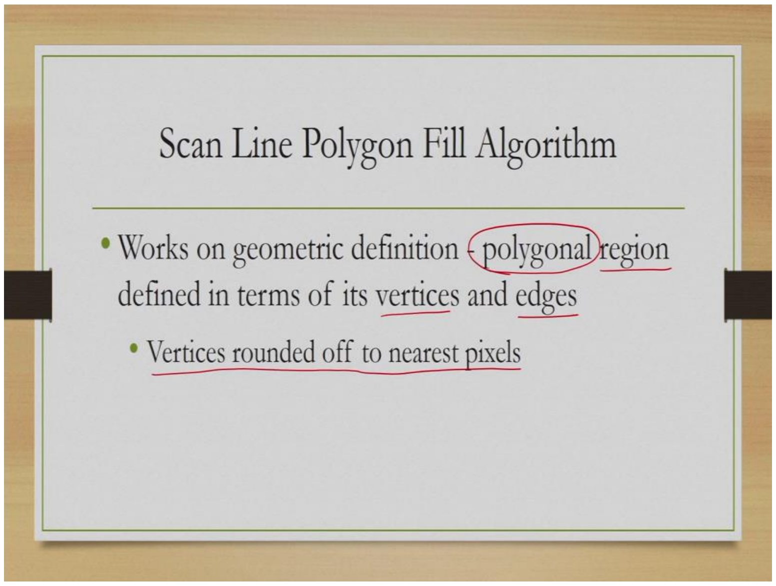
Only in that case we push the connected pixel, because here we cannot check for boundary colour, there is no boundary specified. And then we continue like before till stack is empty. So, in both the cases we start with a seed pixel, but in one case we are dealing with a boundary definition of pixels, in other case we are dealing with an interior definition of region in terms of pixels. And accordingly our algorithm changes slightly otherwise broad idea remains the same.

(Refer Slide Time: 21:29)

Scan Line Polygon Fill Algorithm

We will discuss a third approach, which relies on geometric definition this is called scan line polygon fill algorithm. So, earlier approaches seed fill or flood fill depend on pixel definitions. In case of scan line polygon fill algorithm, we depend on geometric definition.

(Refer Slide Time: 21:56)

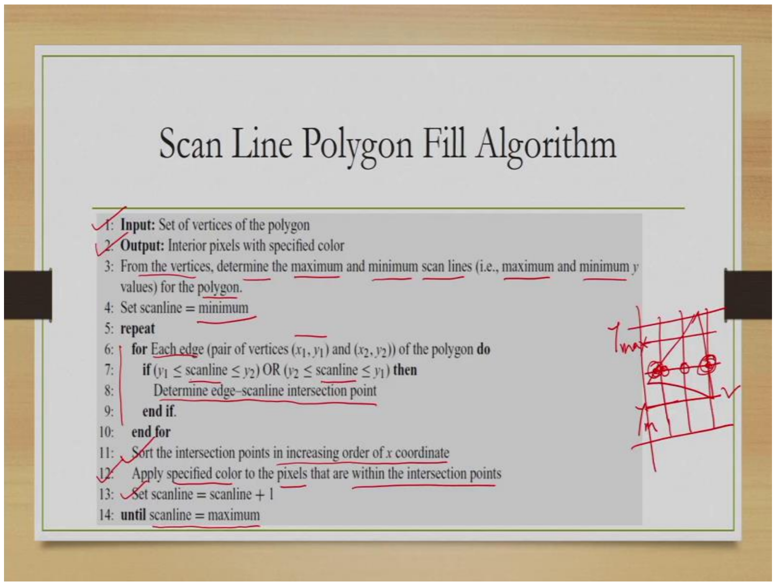


Scan Line Polygon Fill Algorithm

- Works on geometric definition - polygonal region defined in terms of its vertices and edges
  - Vertices rounded off to nearest pixels

So, here we assume that the region is defined in terms of its vertices and edges, of course here the implicit assumption is that the region is polygonal and the vertices are rounded off to the nearest pixels. These are the things that we assume.

(Refer Slide Time: 22:23)



Scan Line Polygon Fill Algorithm

```
1: Input: Set of vertices of the polygon
2: Output: Interior pixels with specified color
3: From the vertices, determine the maximum and minimum scan lines (i.e., maximum and minimum y
   values) for the polygon.
4: Set scanline = minimum
5: repeat
6:   for Each edge (pair of vertices $(x_1, y_1)$ and $(x_2, y_2)$) of the polygon do
7:     if $(y_1 \leq$ scanline $\leq y_2)$ OR $(y_2 \leq$ scanline $\leq y_1)$ then
8:       Determine edge–scanline intersection point
9:     end if.
10:  end for
11:  Sort the intersection points in increasing order of x coordinate
12:  Apply specified color to the pixels that are within the intersection points
13:  Set scanline = scanline + 1
14: until scanline = maximum
```

We will first discuss the algorithm and then try to understand it in terms of an illustrative example. So, here the input is set of vertices and the output is the pixels, interior pixels with specified colour. From the vertices what we do is determine the maximum, the minimum scan lines, that means the maximum and minimum y values for the polygon.

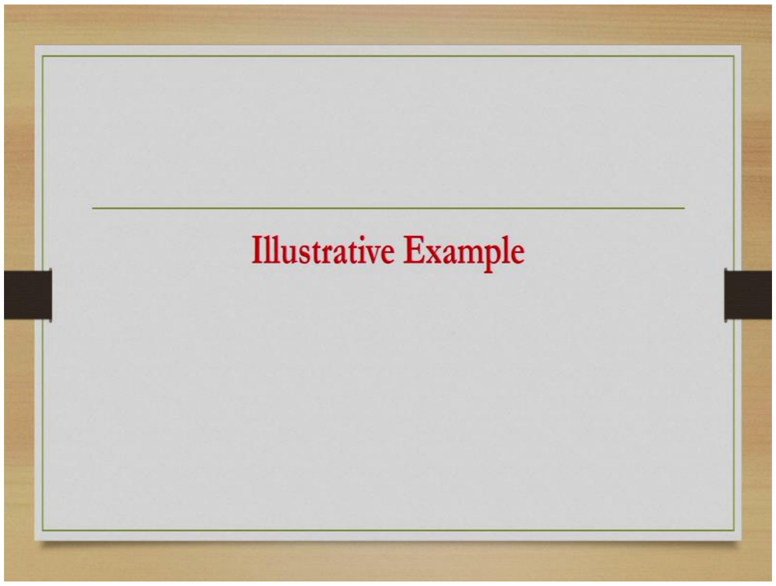
So, for example here suppose this is our pixel grid and we have shape like this, so we need to know the minimum y which is here y_{min} and maximum y which says here y_{max}. So, this maximum and minimum first we determine, then we start from the minimum scan line, that is the lowermost one here.

And then we enter into a loop and continue in the loop until we reach the maximum scan line as shown in this loop condition. So, in the loop what we do? For each edge or the pair of vertices of the polygon if go for a check if the scan line is within a certain range defined by the y coordinates of the edge then we determine the edge scan line intersection point.

After these steps what we do? We sort the intersection points in increasing order of x coordinates, that means we first try to determine the intersection points then we sort them in increasing order, then apply specified colour to the pixels that are within the intersection points all intermediate pixels we apply the colour and then we go to the next scan line, that is the broad idea.

So, first we determine the minimum and maximum we start with minimum and continue the processing till the maximum scan line is reached. In each step of the processing or each loop execution what we do? We determine these two intersection points of the edge with the scan lines to get the two extremes on a single scan line and then assign specified colour to all the pixels that are within these extremes, that is the simple idea. Let us, now try to understand it in terms of one example.

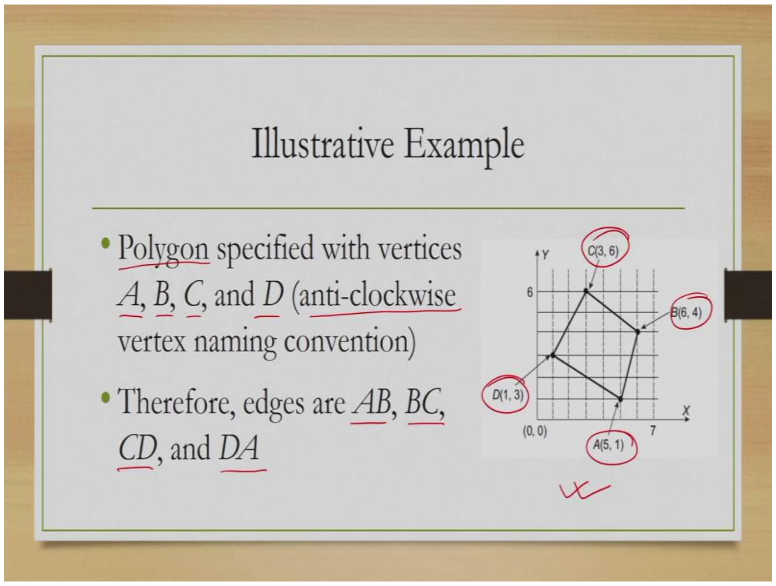
(Refer Slide Time: 25:58)



Illustrative Example

We will go through one illustrative example to get more clarity on the algorithm.

(Refer Slide Time: 26:07)

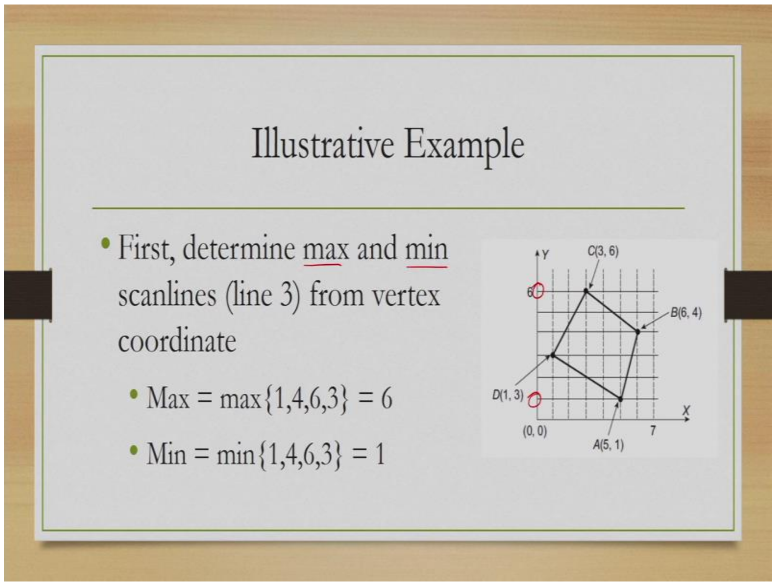


Illustrative Example

- Polygon specified with vertices *A*, *B*, *C*, and *D* (anti-clockwise vertex naming convention)
- Therefore, edges are *AB*, *BC*, *CD*, and *DA*

Let us consider this figure. Here there is a polygon or fill area specified with 4 vertices A, B, C and D as shown here. Now, we followed an anti-clockwise vertex naming convention, so there are 4 edges AB, BC, CD and DA.

(Refer Slide Time: 26:46)



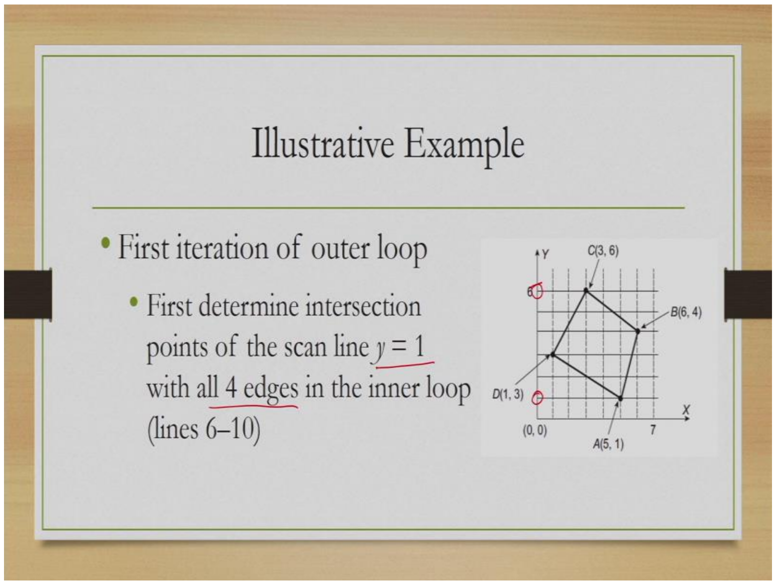
Illustrative Example

- First, determine max and min scanlines (line 3) from vertex coordinate
  - Max = max{1,4,6,3} = 6
  - Min = min{1,4,6,3} = 1

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
(0, 0)
7
X
A(5, 1)

Now first, we determine the minimum and maximum extent of the scan lines. Here it is 1 is the minimum as you can see here and 6 is the maximum scan line, this we determine as the first step.

(Refer Slide Time: 27:10)



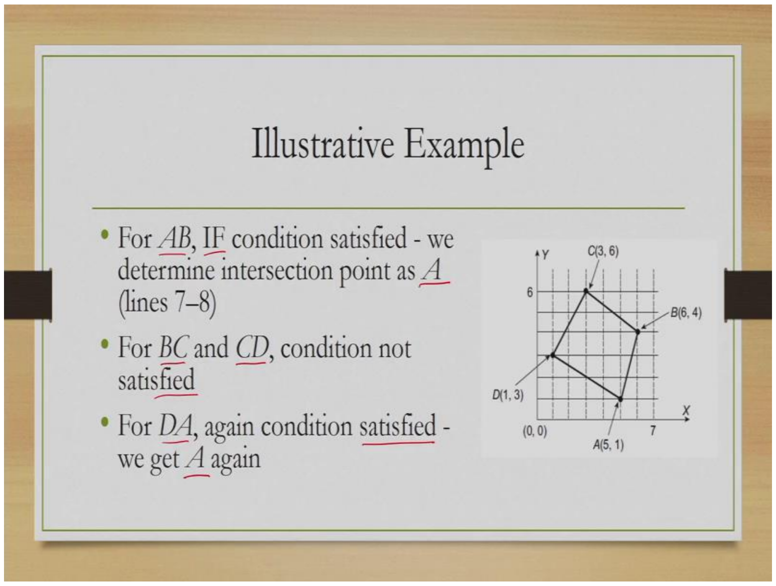
Illustrative Example

- First iteration of outer loop
  - First determine intersection points of the scan line $y = 1$ with all 4 edges in the inner loop (lines 6–10)

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
(0, 0)
7
X
A(5, 1)

Then we start the loop. So, we start from 1 and continue till 6 and in each execution of the loop we process one scan line. So, when we are starting with 1, so our objective is to determine the intersection points of the scan line y equal to 1 with all 4 edges in the inner loop of the algorithm that is lines 6 to 10.

(Refer Slide Time: 27:44)



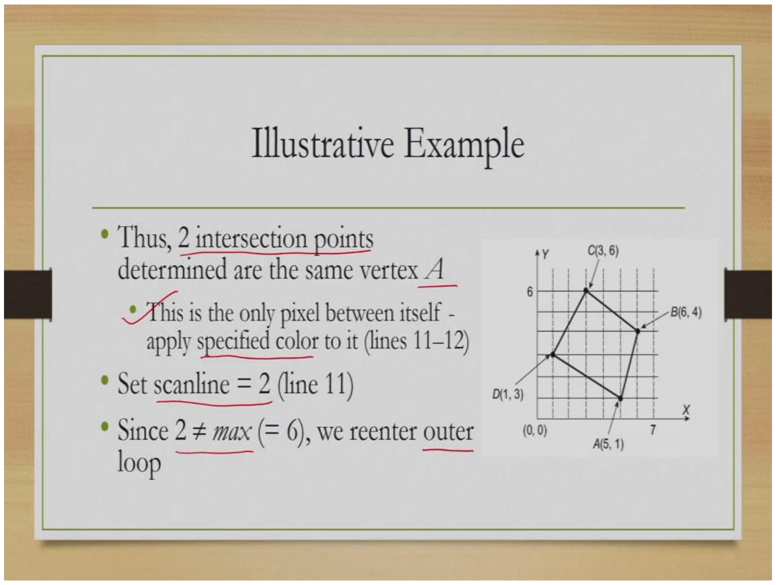
Illustrative Example

- For *AB*, IF condition satisfied - we determine intersection point as *A* (lines 7–8)
- For *BC* and *CD*, condition not satisfied
- For *DA*, again condition satisfied - we get *A* again

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
(0, 0)
7
X
A(5, 1)

If you execute the lines, you will find that for the edge AB the if condition is satisfied and the intersection point is A, for BC and CD edges the condition is not satisfied, again for DA the condition is satisfied and we get A again, so the intersection point is only A.

(Refer Slide Time: 28:15)



Illustrative Example

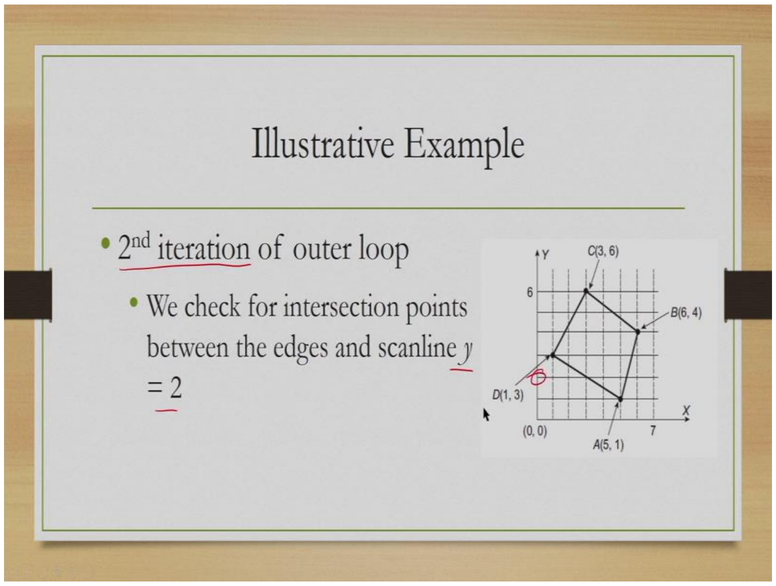
- Thus, 2 intersection points determined are the same vertex *A*
  - This is the only pixel between itself - apply specified color to it (lines 11–12)
- Set scanline = 2 (line 11)
- Since 2 ≠ *max* (= 6), we reenter outer loop

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
(0, 0)
7
X
A(5, 1)

Since it is already a vertex, there cannot be any intermediate pixels. So, we get 2 intersection points, which is the same vertex A, thus it is the only pixel and we apply the specified colour.

Then we go to the next iteration by setting scan line equal to 2 and checking that 2 is not the maximum scan line that is 6, so we execute the loop again.

(Refer Slide Time: 28:57)

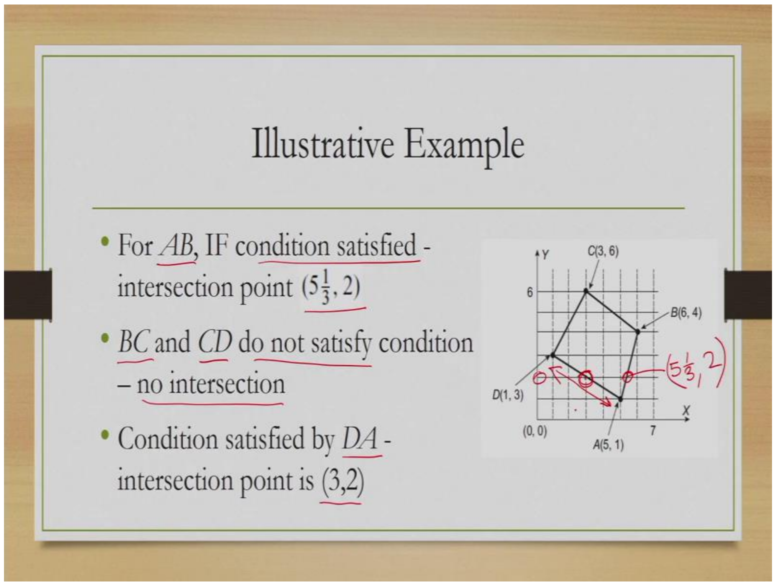


Illustrative Example

- 2nd iteration of outer loop
  - We check for intersection points between the edges and scanline $y = 2$

In the second iteration of the loop, what we do? We check for intersection points as before with the edges and the scan line y equal to 2, that is this scan line.

(Refer Slide Time: 29:14)



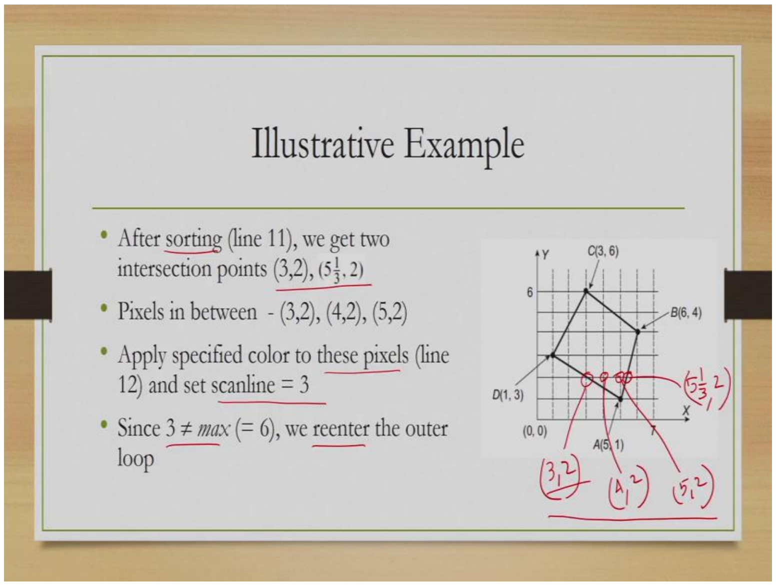
Illustrative Example

- For AB, IF condition satisfied - intersection point $(5\frac{1}{3}, 2)$
- BC and CD do not satisfy condition – no intersection
- Condition satisfied by DA - intersection point is (3,2)

Now, for y equal to 2 and if we check the edges we see that for AB if condition is satisfied that means there is an intersection point, using the edge line equation and the scan line equation we can get the intersection point as this one, this point, for BC and CD the if condition does not satisfy for BC and CD, so there is no intersection.

And for DA the condition satisfies, again. So, this is for AB intersection point, this is for DA intersection point and this DA intersection point we can find to be (3, 2) by using the line equation for the edge as well as the scan line equation. So, this point is one intersection point, this point is another intersection point.

(Refer Slide Time: 30:41)



Illustrative Example

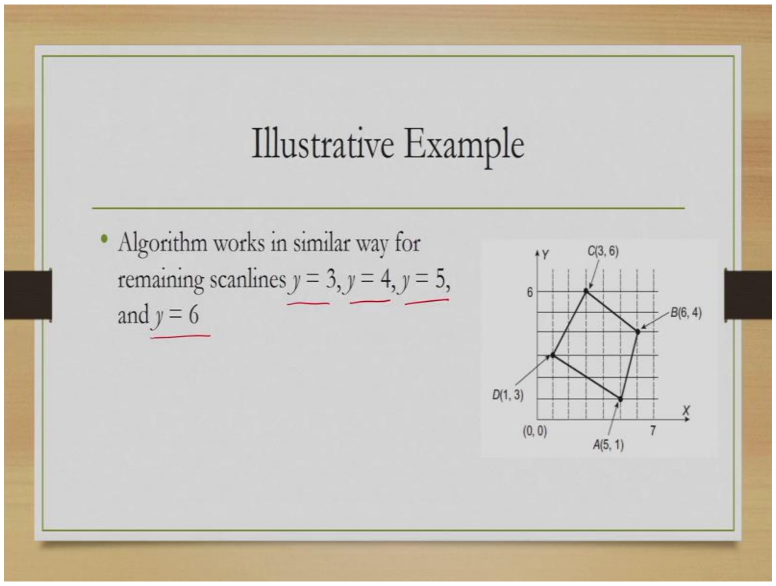
- After sorting (line 11), we get two intersection points (3,2), $(5\frac{1}{3}, 2)$
- Pixels in between - (3,2), (4,2), (5,2)
- Apply specified color to these pixels (line 12) and set scanline = 3
- Since $3 \neq max$ (= 6), we reenter the outer loop

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
X
(0, 0)
A(5, 1)
7
$(5\frac{1}{3}, 2)$
(3, 2)
(4, 2)
(5, 2)

Then we perform a sorting as mentioned in the algorithm and get these two intersection points in a sorted order as this one. So, this is one intersection point and this is the other intersection point. In between pixels are there as you can see so this itself is an in between pixel, then we have this pixel, which is (4, 2) and then we have (5, 2).

Note that the other intersection point is not a pixel in itself because it involves a real number as coordinate. So, we found out the 3 pixels, which are there in between the two intersection points, then we apply specified colour to these pixels. Then we reset the scan line to 3 and check whether 3 is the maximum scan line, it is not so we re-enter the loop again.

(Refer Slide Time: 32:15)

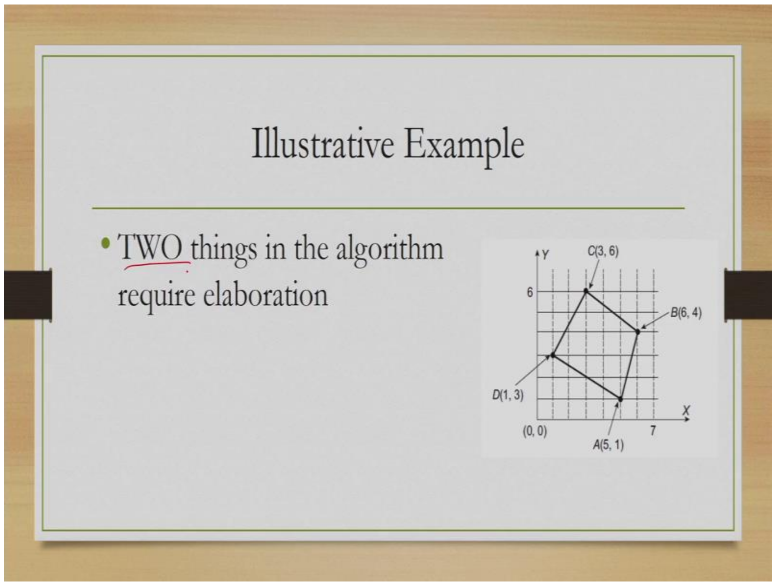

Illustrative Example

- Algorithm works in similar way for remaining scanlines $y = 3$, $y = 4$, $y = 5$, and $y = 6$

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
X
(0, 0)
7
A(5, 1)

And in a similar way we process the scan lines y=3, y=4, y=5 till y=6. So, that is the idea of the algorithm. So, to summarize here in this scan line polygon fill algorithm, what we do? We assume a geometric representation of the region, polygonal region in terms of edges or vertices. Then for each scan line which is there between the minimum and maximum scan lines, what we do? We determine the intersection points of the edges with those scan lines, sort them to get two extreme intersection points, identify the in between pixels and colour those pixels. And this we do for all the scan lines that are there between the minimum and the maximum scan lines.

(Refer Slide Time: 33:29)



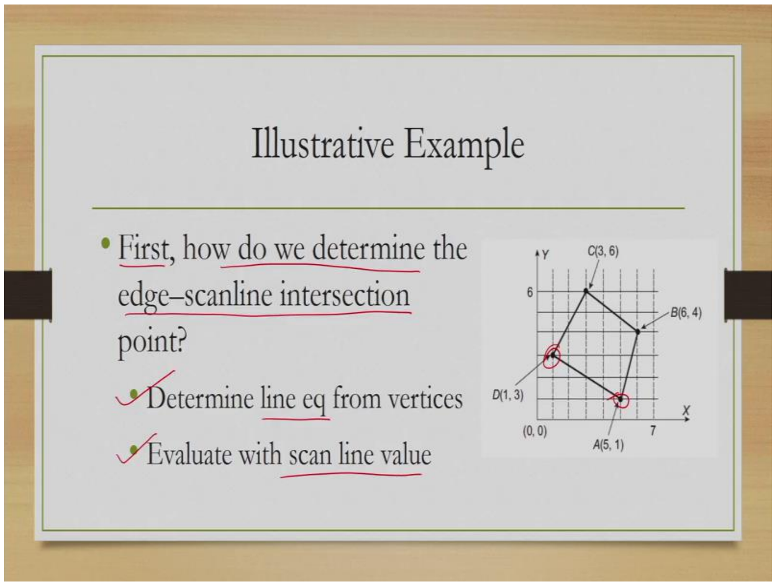
Illustrative Example

- TWO things in the algorithm require elaboration

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
(0, 0)
7
X
A(5, 1)

Now, there are two things that require some elaboration in the algorithm.

(Refer Slide Time: 33:38)



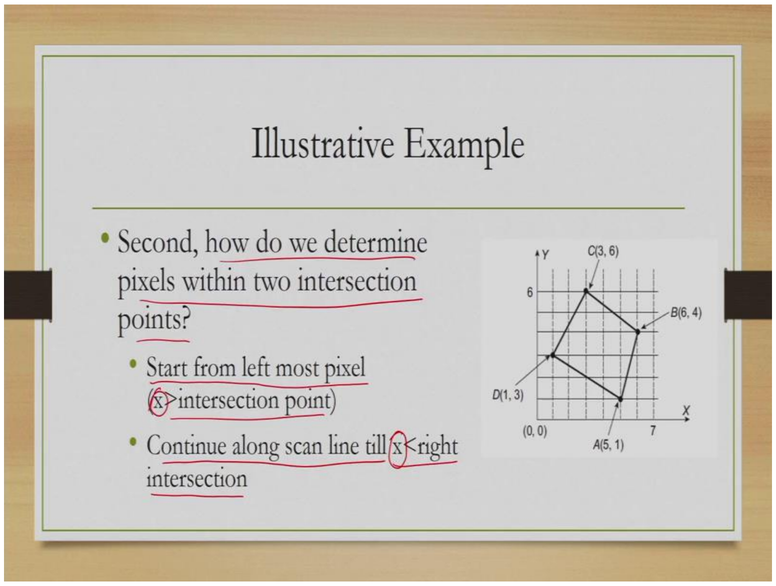
Illustrative Example

- First, how do we determine the edge–scanline intersection point?
  - Determine line eq from vertices
  - Evaluate with scan line value

Y
C(3, 6)
6
B(6, 4)
D(1, 3)
(0, 0)
7
X
A(5, 1)

First is how do we determine the 8 scan line intersection point, that I think all of you know that we can use the line equation which we can determine by the two endpoints and we can use the scan line equation to get the intersection point. So, this line equation can be evaluated with the scan line value to get the intersection point, which is a very simple approach and I think all of you may already know how to do that.

(Refer Slide Time: 34:20)



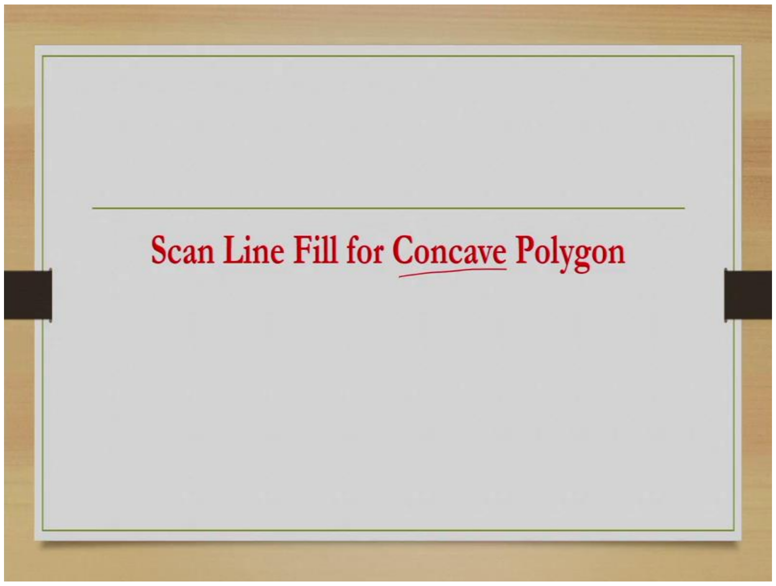
Illustrative Example

- Second, how do we determine pixels within two intersection points?
  - Start from left most pixel (x > intersection point)
  - Continue along scan line till x < right intersection

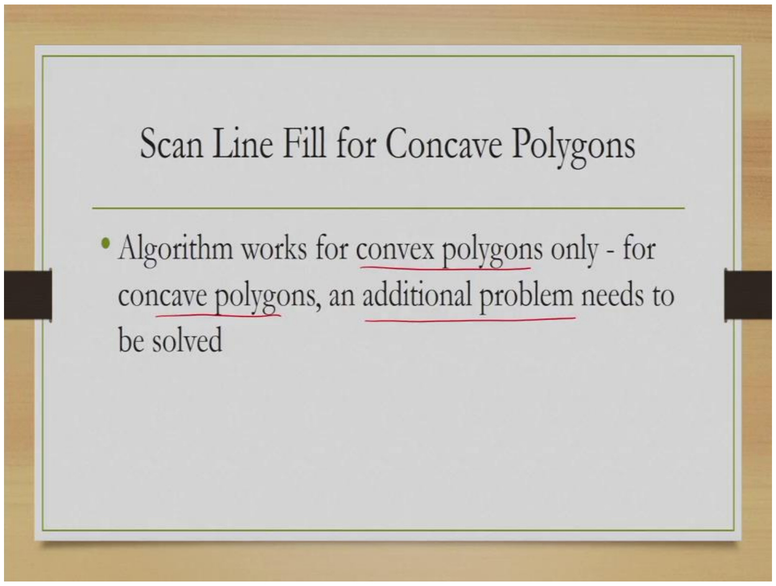
Y
C(3, 6)
6
B(6, 4)
D(1, 3)
X
(0, 0)
7
A(5, 1)

Secondly how do we determine pixels within two intersection points? This is again very simple, we start from the left most pixel which is either the intersection point or just next to the intersection point and then continue along the scan line till we get a pixel value which is less than the right intersection point. Pixel x coordinate we check in all these checks. So, both are simple, but the second point that is how to determine pixels within two intersection points is not as simple as it appears. And why that is so?

(Refer Slide Time: 35:27)



Scan Line Fill for Concave Polygon

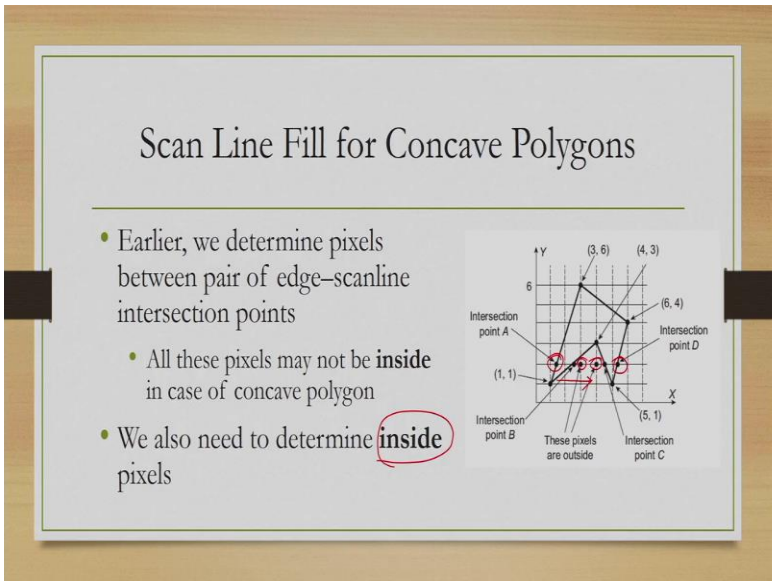


Scan Line Fill for Concave Polygons

- Algorithm works for convex polygons only - for concave polygons, an additional problem needs to be solved

If we assume we have a concave polygon, then there is a problem. So, far whatever explanation is given is based on the assumption that we are dealing with convex polygons. For concave polygons it is not so easy to determine the intermediate pixels, an additional issues are there which needs to be solved before we determine the interior pixels.

(Refer Slide Time: 36:16)

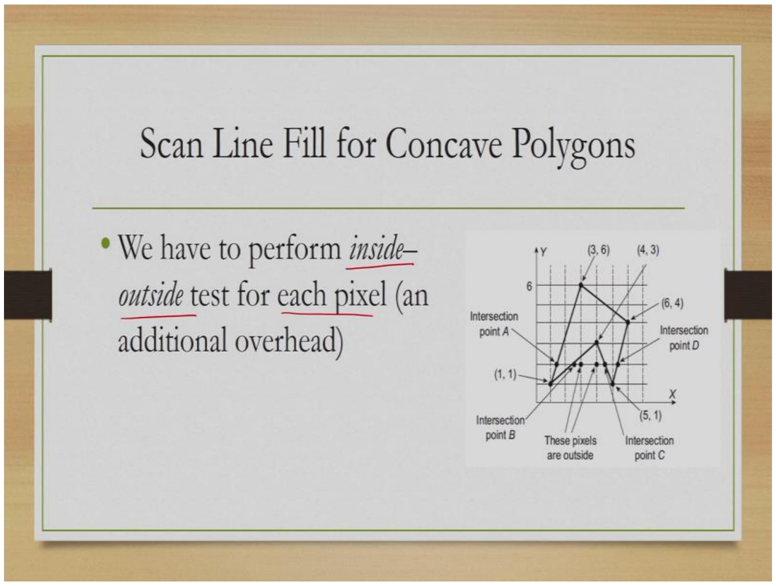


Scan Line Fill for Concave Polygons

- Earlier, we determine pixels between pair of edge–scanline intersection points
  - All these pixels may not be **inside** in case of concave polygon
- We also need to determine **inside** pixels

Let us, take an example, here in this figure as you can see this is a concave polygon, so when we are dealing with these two extreme intersection points, some pixels are outside the polygon, although if we follow the approach that we outlined earlier that we will just move along this line to get all the intermediate pixels, then outside pixels will also be treated as interior pixels, which definitely we do not want. So, we need to go for some different approach when we are dealing with concave polygons. What we need to do is, we need to explicitly determine inside pixels, pixels that are inside the polygon. As you can see from the figure that is not so obvious for concave polygons.

(Refer Slide Time: 37:24)



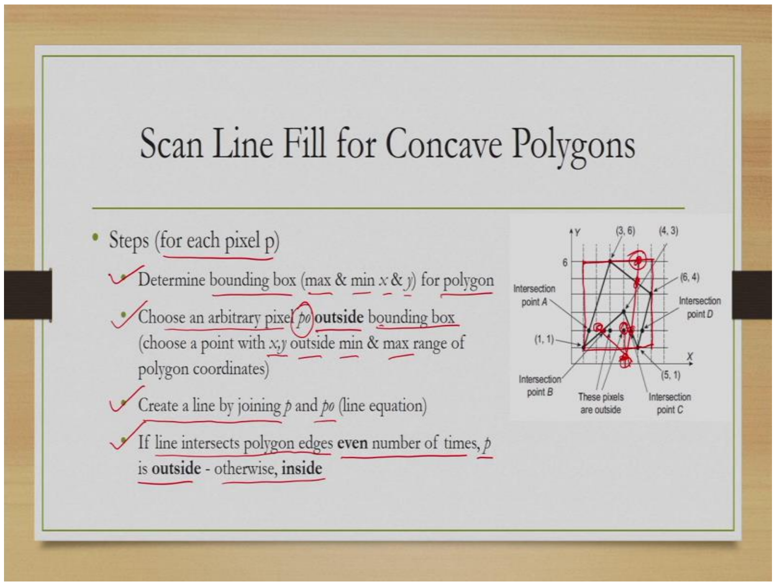
Scan Line Fill for Concave Polygons

- We have to perform *inside–outside* test for each pixel (an additional overhead)

Y
(3, 6)
(4, 3)
6
(6, 4)
Intersection point A
Intersection point D
(1, 1)
X
(5, 1)
Intersection point B
These pixels are outside
Intersection point C

So, in this case we need to perform an inside outside test for each pixel which is of course an additional overhead.

(Refer Slide Time: 37:39)



Scan Line Fill for Concave Polygons

- Steps (for each pixel p)
  - Determine bounding box (max & min *x* & *y*) for polygon
  - Choose an arbitrary pixel *po* **outside** bounding box (choose a point with *x,y* outside min & max range of polygon coordinates)
  - Create a line by joining *p* and *po* (line equation)
  - If line intersects polygon edges **even** number of times, *p* is **outside** - otherwise, **inside**

Y
(3, 6)
(4, 3)
6
(6, 4)
Intersection point A
Intersection point D
(1, 1)
X
(5, 1)
Intersection point B
These pixels are outside
Intersection point C

And how we can do that? So, for each pixel p what we can do is determine the bounding box of the polygon that is the maximum and minimum x and y values of the polygons vertices. This is the first step, then in the second step we choose an arbitrary pixel, let us denote it by p0, outside the bounding box.

So, in this case, this can be our bounding box, as you can see it covers all vertices of the polygon. Then we choose one pixel outside this bounding box somewhere say here, that means choose a point (x, y) which is outside the min and max range of the polygon coordinates. In the third stage we create a line by joining p and p0.

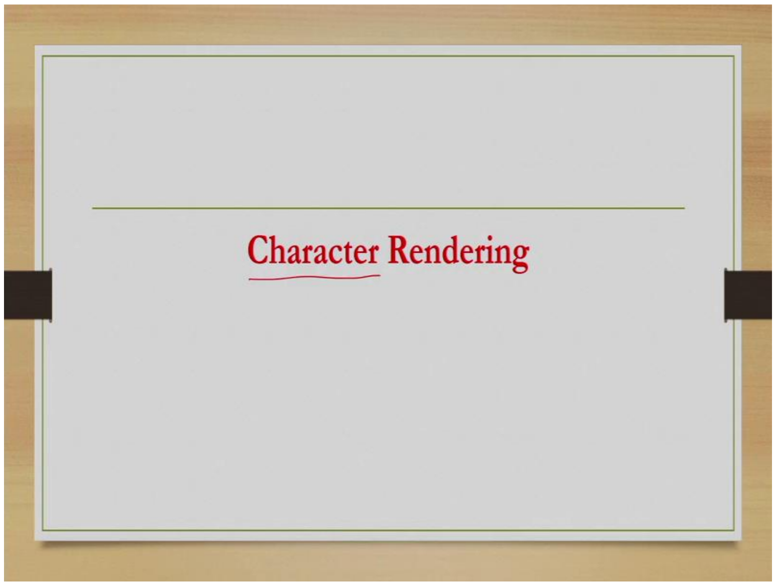
So, we create a line between the pixel that is inside that is our pixel of concern and the point that is outside the bounding box. In the final stage we go for some checks, if the line intersects the polygon edges even number of times then p is outside, otherwise it is inside. So, as you can see suppose we have a pixel here and these two pixels if we join it intersects one time, that is odd number of time, so this pixel is inside.

Whereas, if we are dealing with a pixel here and we are joining this, so as you can see here intersection is twice that is even number of times, so these pixel is outside. Similarly, these pixels if we join these two lines, we see that it does not intersect the polygon edges, so that is 0, so in that case also it is outside.

But of course all these checks takes time, so it is an additional overhead when we are dealing with concave polygons. Otherwise, scan line polygon fill algorithm for convex polygons is quite simple. So, we have discussed different region fill algorithms, both for pixel level definitions as well as geometric definitions.

For pixel level definitions we learned about seed fill algorithm and flood fill algorithm, both are similar with minor variation. For geometric definitions we learned about scan line polygon fill algorithm, this algorithm is quite simple and straightforward when we are dealing with convex polygon, but requires additional checks when we are dealing with concave polygons. Now, let us try to understand how characters are displayed on the computer screen. So, how do we render characters?

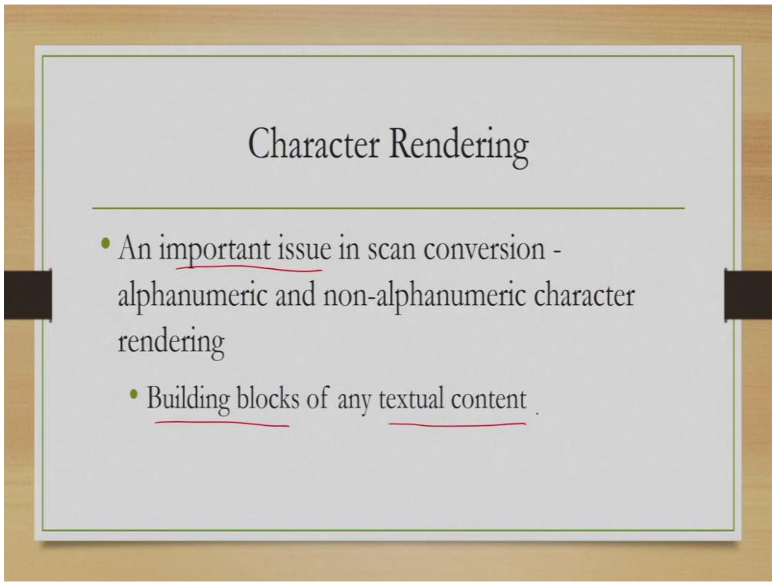
(Refer Slide Time: 41:28)



Character Rendering

Here, character means alphanumeric characters.

(Refer Slide Time: 41:35)

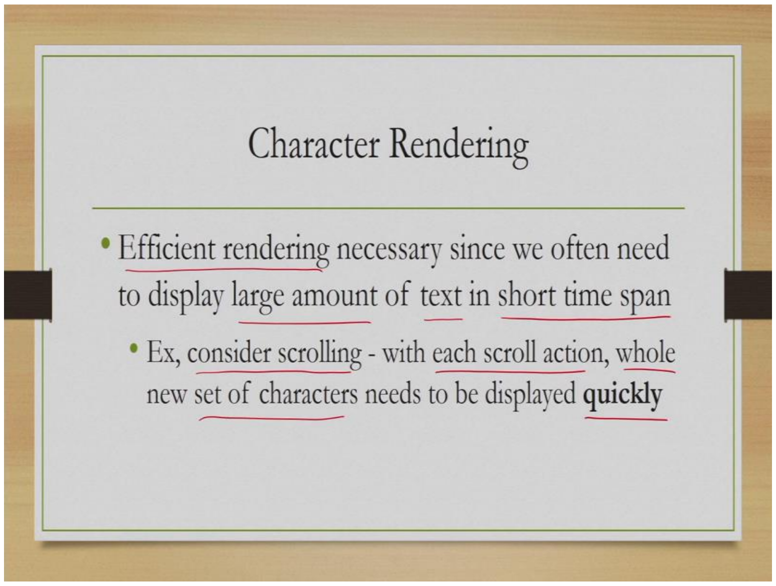


Character Rendering

- An important issue in scan conversion - alphanumeric and non-alphanumeric character rendering
  - Building blocks of any textual content

Now, character rendering is of course as we all know is an important issue, for example consider this slide, here we see lots of alphanumeric characters displayed. So, clearly it is an important issue and this is the building blocks of any textual content, so any application that deals with displaying texts must have support for character rendering. And how it is done?

(Refer Slide Time: 42:15)

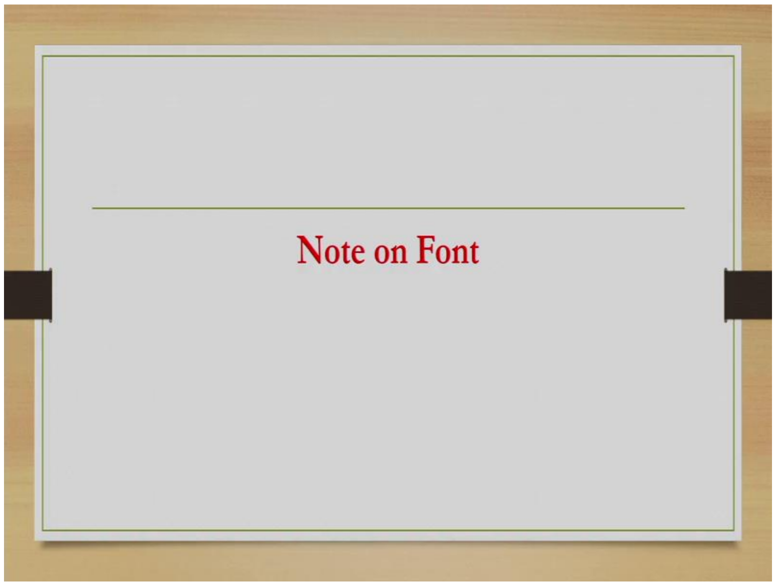


Character Rendering

- Efficient rendering necessary since we often need to display large amount of text in short time span
  - Ex, consider scrolling - with each scroll action, whole new set of characters needs to be displayed **quickly**

As we all know when we are dealing with some text processing application, typically large amount of text needs to be displayed in short time span. For example, if we consider scrolling, now with each scroll action the whole set of characters is redrawn and that has to be done very quickly. So, efficient rendering of characters is a very important issue in computer graphics.

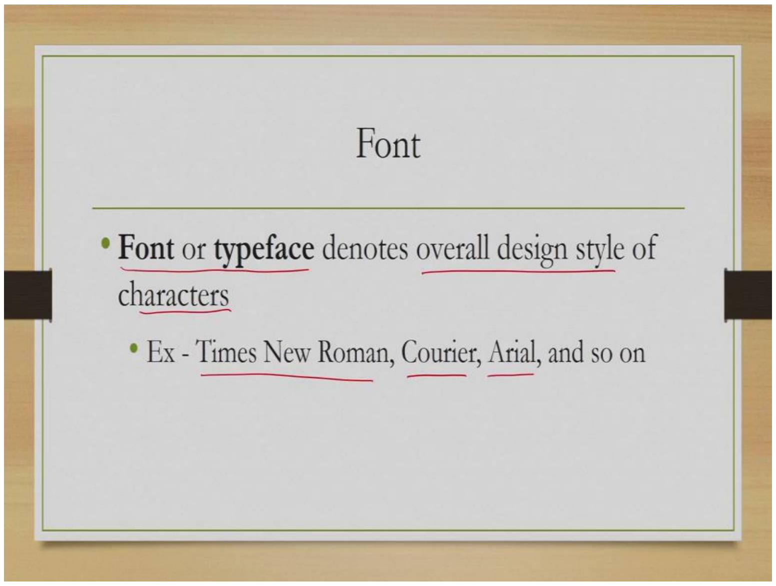
(Refer Slide Time: 43:05)



Note on Font

Now, before we try to understand character rendering, let us have a quick look at the idea font and we already know probably already heard of this term, so let us see what is a font and how it is dealt within computer graphics.

(Refer Slide Time: 43:22)

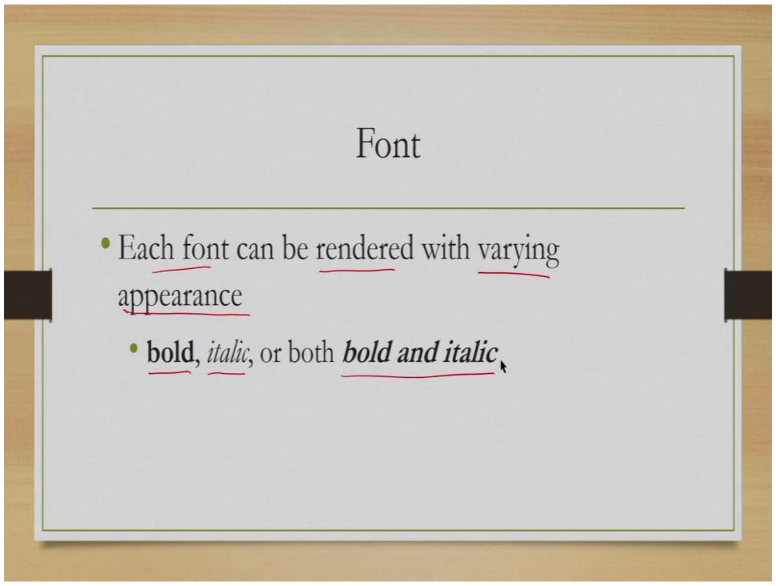


Font

- **Font** or **typeface** denotes overall design style of characters
  - Ex - Times New Roman, Courier, Arial, and so on

So, when we are talking of font, font or typeface denotes overall design style of characters and there are many such fonts as probably all of us are using every day such as Times New Roman, Courier, Arial and so on. So, these fonts or typefaces indicate the design style of the characters how they look.

(Refer Slide Time: 43:58)

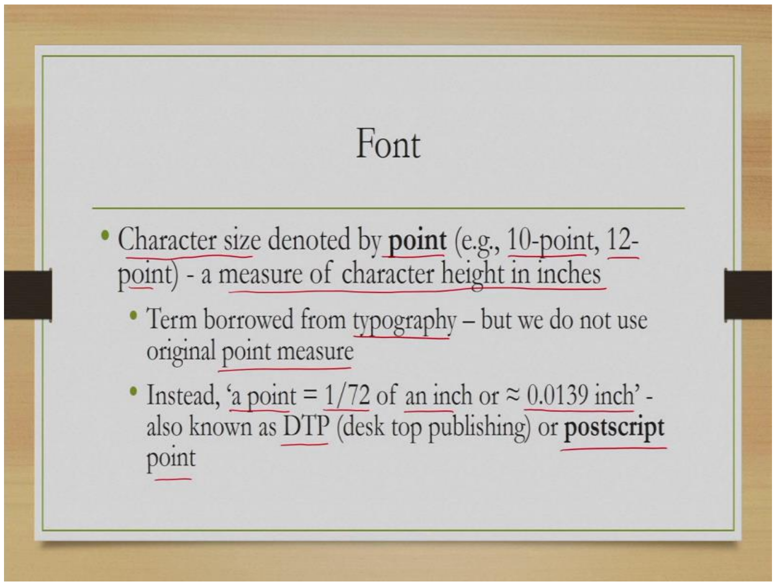


Font

- Each font can be rendered with varying appearance
  - **bold**, *italic*, or both ***bold and italic***

Now, each font can be rendered with varying appearance. So, appearance may be different, it may be bold, it may be italicised or it may be both bold and italicised.

(Refer Slide Time: 44:17)



Font

- Character size denoted by **point** (e.g., 10-point, 12-point) - a measure of character height in inches
  - Term borrowed from typography – but we do not use original point measure
  - Instead, 'a point = 1/72 of an inch or ≈ 0.0139 inch' - also known as DTP (desk top publishing) or **postscript** point

Along with that there is another concept called size, how big or small the character appears on the screen. So, that is denoted by point, for example a 10-point font, 12-point font and so on which is a major of character height in inches and this term is borrowed from typography, but in computer graphics we do not use the original measure.

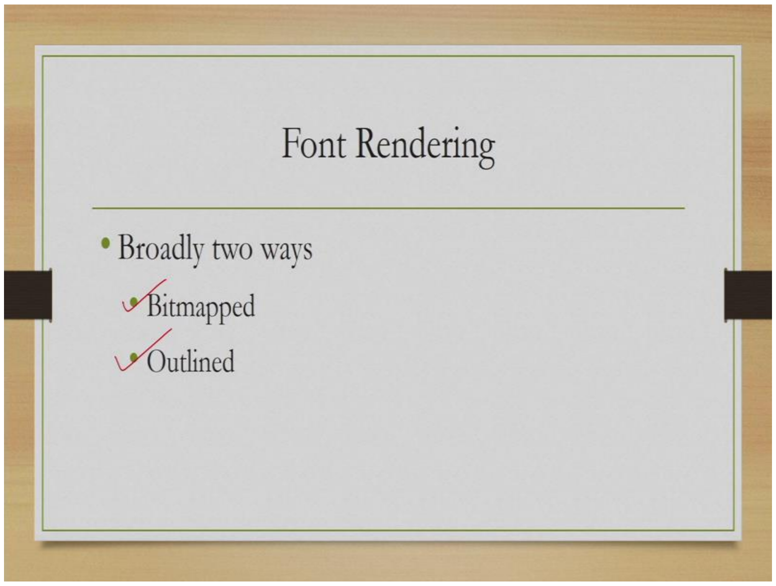
Instead, we assume that point is equivalent to 1/72 of an inch or approximately 0.0139 inch. And this is also known as DTP or desktop publishing or postscript point. So, when we are talking of a point, we assume that it is 1/72 of an inch or approximately 0.0139 inch, which indicates the height of the character.

(Refer Slide Time: 45:33)

Font Rendering

Now, with that basic knowledge, let us try to understand how characters are rendered.

(Refer Slide Time: 45:41)

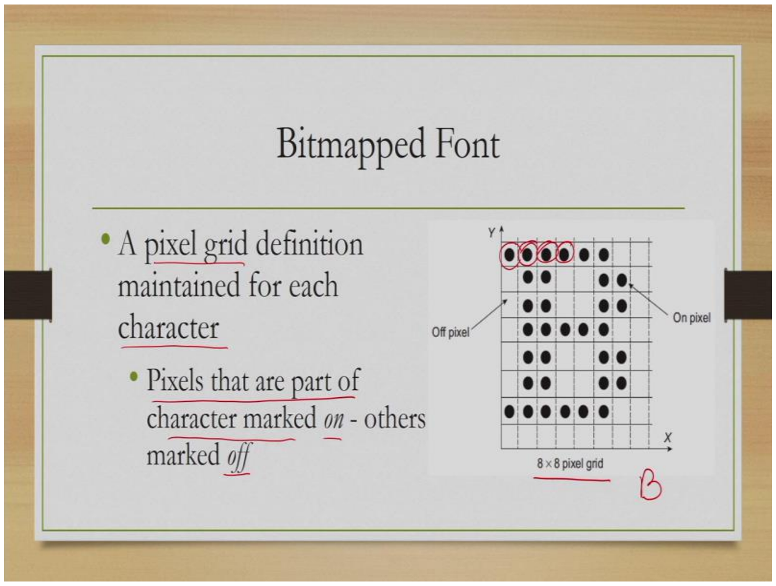


Font Rendering

- Broadly two ways
  - ✓ Bitmapped
  - ✓ Outlined

So, there are broadly two ways of rendering characters, one is bitmapped one is outlined.

(Refer Slide Time: 45:53)



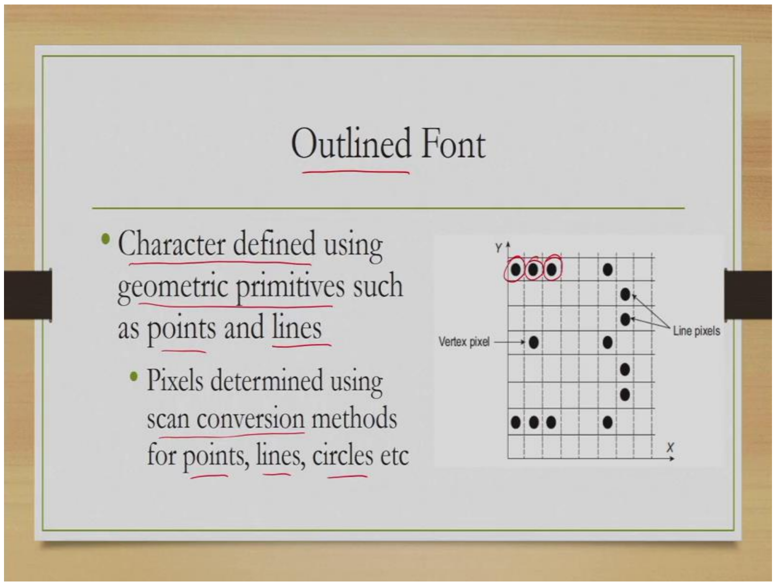
Bitmapped Font

- A pixel grid definition maintained for each character
  - Pixels that are part of character marked *on* - others marked *off*

Y
X
Off pixel
On pixel
8 × 8 pixel grid
B

In case of bitmapped font, we define a pixel grid for each character, for example consider this 8 by 8 pixel grid and we can define the grid for character B in capital, where the pixels that are part of the characters are marked ON and others are OFF. So, when the grid is rendered for B, only those pixels will be illuminated other pixels will not be illuminate. The black circles here indicate the ON pixels as you can see here and the white boxes indicate the OFF pixels. So, we can have this type of grid for each character when we are dealing with bitmapped font.

(Refer Slide Time: 46:59)



Outlined Font

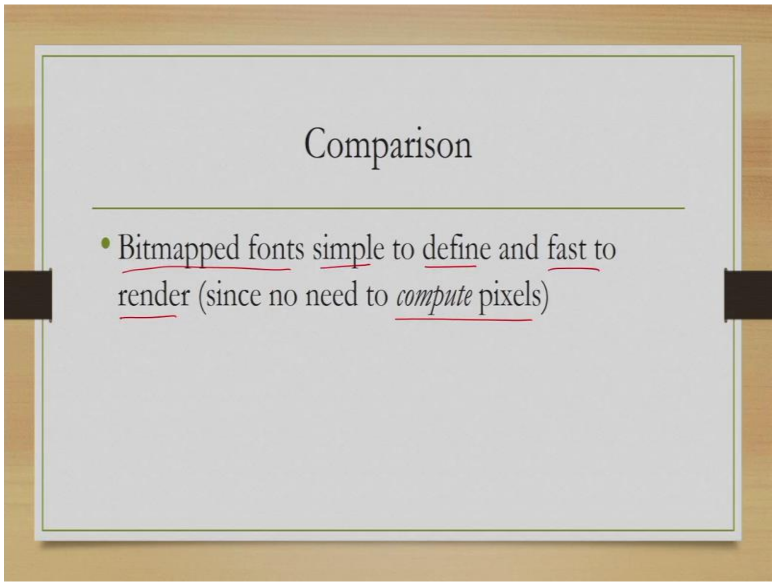
- Character defined using geometric primitives such as points and lines
  - Pixels determined using scan conversion methods for points, lines, circles etc

Y
X
Vertex pixel
Line pixels

In contrast when we are dealing with outline font the approach is totally different, here characters are defined using geometric primitives such as points and lines. So, few pixels maybe provided and then other pixels will be determined by using scan conversion techniques for points, lines and circles.

So, essentially few pixels are provided and using those pixels the computer will draw the primitives such as lines or circles to construct a character like creating an image. So, in case of bitmapped font, we already specify all the pixels whereas in case of outline font we do not specify all the pixels, few pixels are specified and using those the overall shape is computed or created by following the scan conversion techniques.

(Refer Slide Time: 48:18)

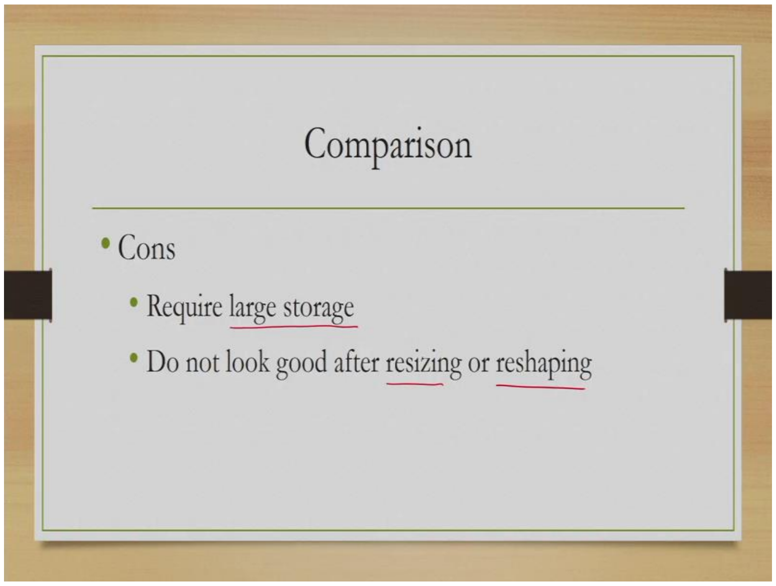


Comparison

- Bitmapped fonts simple to define and fast to render (since no need to *compute* pixels)

Clearly bitmapped fonts are simple to define and first to render because here no computation is involved. We do not need to compute any pixels they are already specified, but it has some problems.

(Refer Slide Time: 48:37)

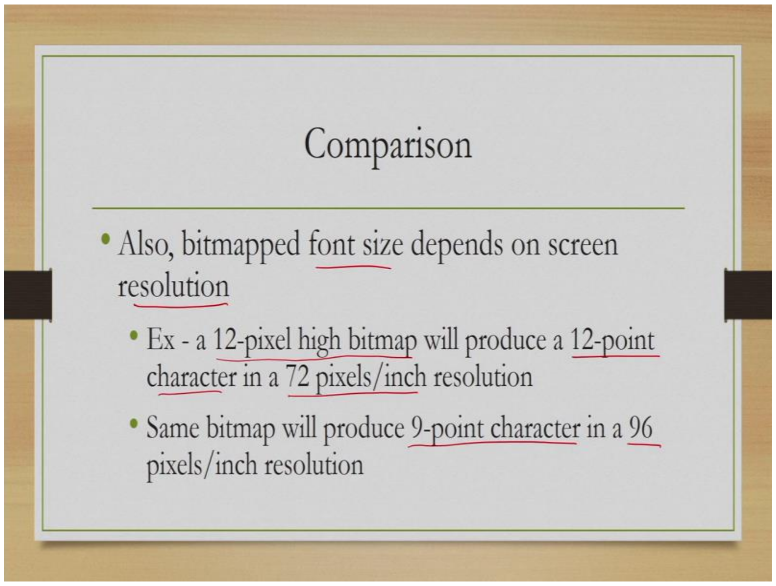


Comparison

- Cons
  - Require large storage
  - Do not look good after resizing or reshaping

Obviously, it will require additional storage, large amount of storage, because for each character we are storing a pixel grid information and then if we want to resize or reshape to generate different stylish effect of the font then that is not easily possible with bitmapped definitions and the resulting font may appear to be bad.

(Refer Slide Time: 49:22)

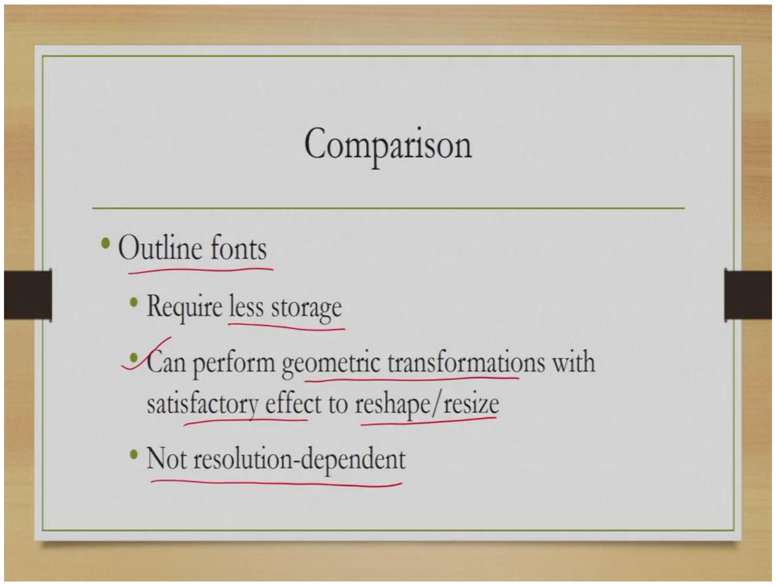


Comparison

- Also, bitmapped font size depends on screen resolution
  - Ex - a 12-pixel high bitmap will produce a 12-point character in a 72 pixels/inch resolution
  - Same bitmap will produce 9-point character in a 96 pixels/inch resolution

The third concern is the font size depends on screen resolution, because we are fixing the pixel grid. So, if the resolution changes then the rendering may not look good. For example, suppose we have defined a 12 pixels high bitmap, it will produce a 12-point character in a 72 pixels per inch resolution. Now, if we change the resolution to 96 pixels per inch then the same bitmap will produce 9-point character which we may not want. So, depending on resolution the outcome may change.

(Refer Slide Time: 50:18)

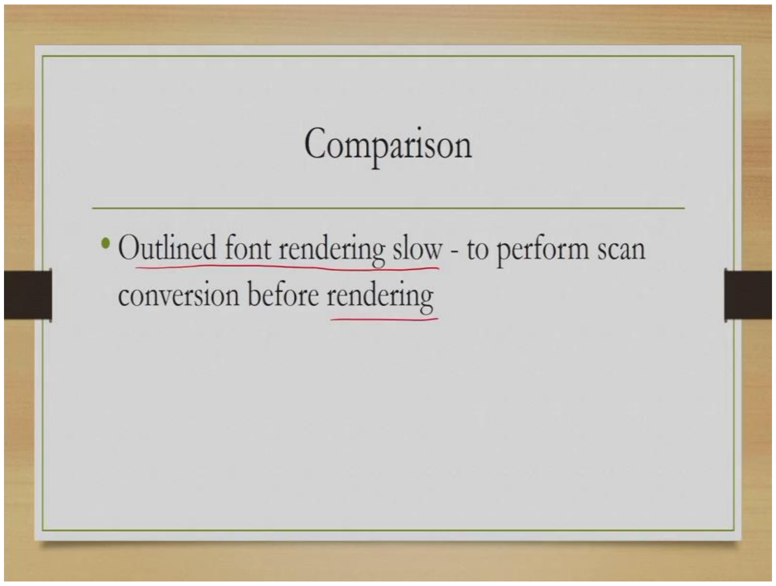


Comparison

- Outline fonts
  - Require less storage
  - Can perform geometric transformations with satisfactory effect to reshape/resize
  - Not resolution-dependent

On the other hand, outline fonts compute the intermediate pixels, they do not store everything, so it requires less storage, it can perform geometric transformations with satisfactory effect to reshape and all resize, so the distortion will be less and it is not resolution dependent.

(Refer Slide Time: 50:50)

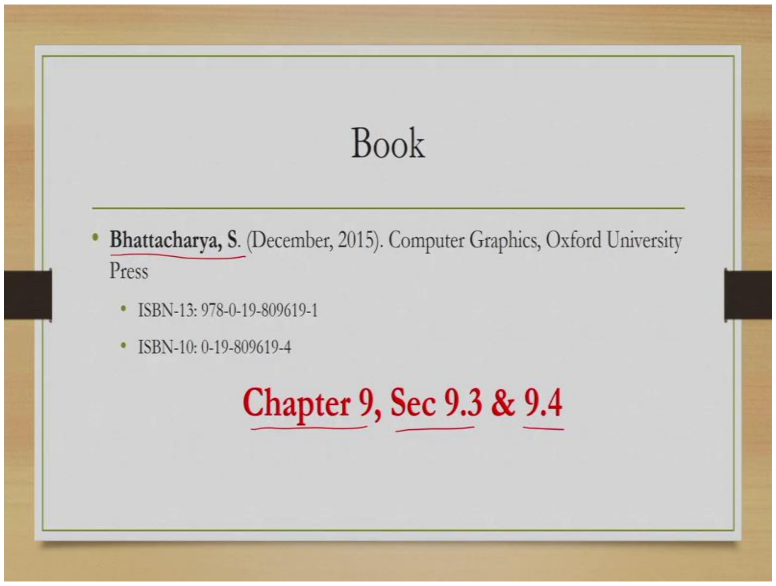


Comparison

- Outlined font rendering slow - to perform scan conversion before rendering

But on the other hand rendering such fonts is slow, which is quite obvious because computations are involved, we need to create the shape, we need to scan convert the shape before rendering. So, due to such computations rendering is slower compared to bitmap fonts. So, both the approaches have their positive and negative sides and depending on the resources used and the outcome desired we can choose one particular approach.

(Refer Slide Time: 51:34)



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 9.3 & 9.4

Whatever I have discussed today can be found in this book, you may go through chapter 9 section 9.3 and 9.4 for more details on the topics that we have covered today. In the next lecture, we will discuss an interesting issue in scan conversion which is called aliasing effect till then thank you and goodbye.