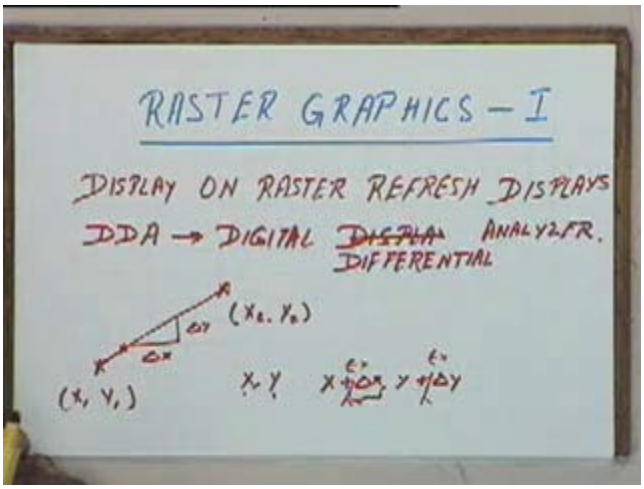
Computer Aided Design
Dr. Anoop Chawla
Department of Electrical Engineering
Indian Institute of Technology, Delhi
Lecture No. # 03
Raster Graphics – I

We will be having the first of a series of lectures on raster graphics. In raster graphics we will essentially be taking of display on raster refresh displays. In the last lecture I had briefly described what are raster refresh displays or raster refresh terminals or raster refresh type of CRT's.

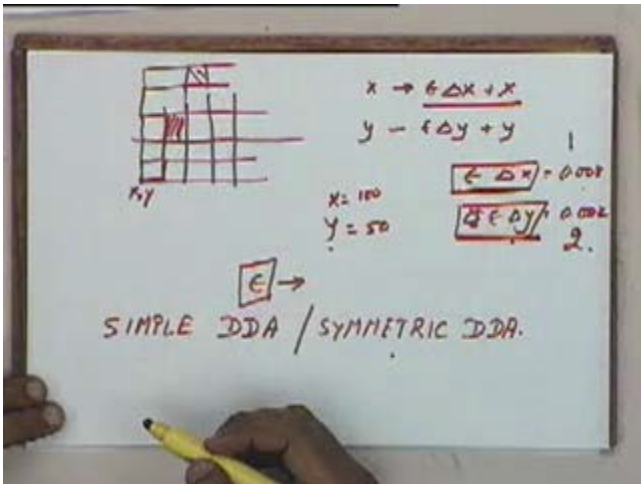
(Refer Slide Time: 00:01:09 min)



Now in the first display method that we will be talking of is called a DDA or it stands for digital differential analyzer. The basic idea in this method of drawing lines is that if you have to draw a line from a point x_1 y_1 to a point x_2 y_2, we can easily compute the slope of this line which is $\frac{\Delta y}{\Delta x}$. Therefore if we know that the particular pixel, this particular pixel is a pixel x y if this pixel has already been displayed then the next pixel to be displayed should be a pixel given by $x+\Delta x$ and $y+\Delta y$ or some amount proportional to Δ x. So what we will normally say, it should be ε*Δ x and ε*Δ y.

So if we start from the first pixel here that pixel is let's say the pixel x y then we will display the pixel which will be given by $x+\varepsilon*\Delta x$ and $x+\varepsilon*\Delta y$. As long as we keep incrementing in proportion to the, we keep incrementing x and y by amounts proportional to Δx and Δ y. We will always get a line which will be in this direction. So that's the basic idea in this.

(Refer Slide Time: 00:03:38 min)

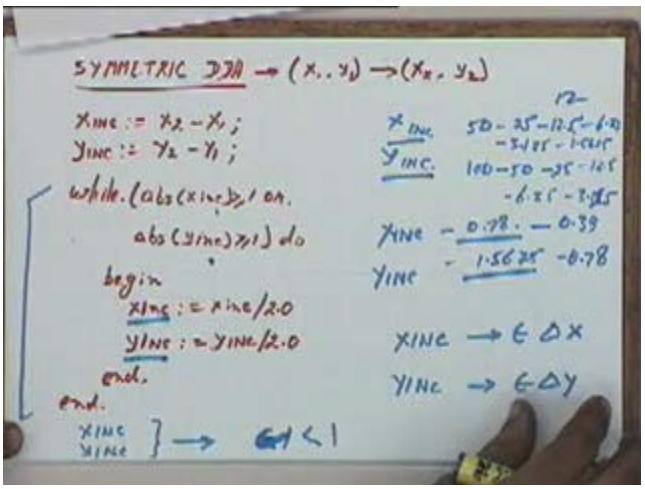


But the problem that comes is let's say this is one pixel, we just a make a small grid. If the first pixel that has been displayed is this one, so this is my x y. Now I increment x and y, $x = \varepsilon * \Delta x + x$ and $y = \varepsilon * \Delta y + y$. If I take my ε to be very small then this amount will still be the same as x because x and y are bound to be integers. So let's say x is let's say 100 and y is let's say 50 and I take $\varepsilon * \Delta x$ o be very small let's say 0.001 and I take $\varepsilon * \Delta y$ to be let's say 0.002. Even though I will increment x and y by these amounts, I will still be displaying the same pixel because 100 plus 0.001 will mean that x will get rounded off to 100 again. So I will end up displaying the same pixel again. So I will keep on displaying the same pixel till I am able to increment x by 1 or y by 1. So this amount $\varepsilon * \Delta x$ and $\varepsilon * \Delta y$, these amounts have to be chosen very carefully.

If they are very small then we will end up displaying the same pixel and again and again and we will be doing wasteful computation. The algorithm will become unnecessarily slow. If I take $\varepsilon * \Delta x$ or $\varepsilon * \Delta y$ to be very big then maybe my line will not be continuous. If I take $\varepsilon * \Delta x$ to be 1 and $\varepsilon * \Delta y$ to be 2, I am displaying this pixel and the next pixel to be displayed will be this one. Similarly the next one to be displayed would be this pixel, this line will not look continuous.

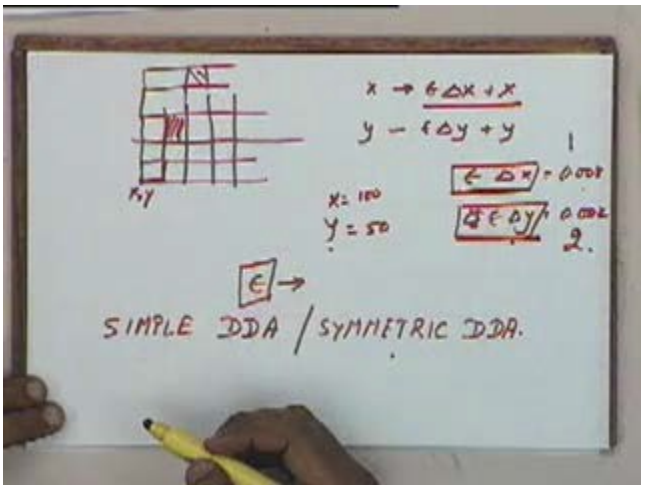
So the important question is how do we chose the value of $\varepsilon * \Delta x$ and $\varepsilon * \Delta y$. The choice of ε will govern the quality of the algorithm whether we are able to draw good lines or not that depends purely on the choice of ε. So we have two algorithms, one is called the simple DDA. Again DDA stands for digital differential analyzer and the other is called the symmetric DDA. The only difference between these two algorithms is the choice of ε. We will just go through both these algorithms and see how they will compare and what kind of results they come up with.

(Refer Slide Time: 00:07:34 min)



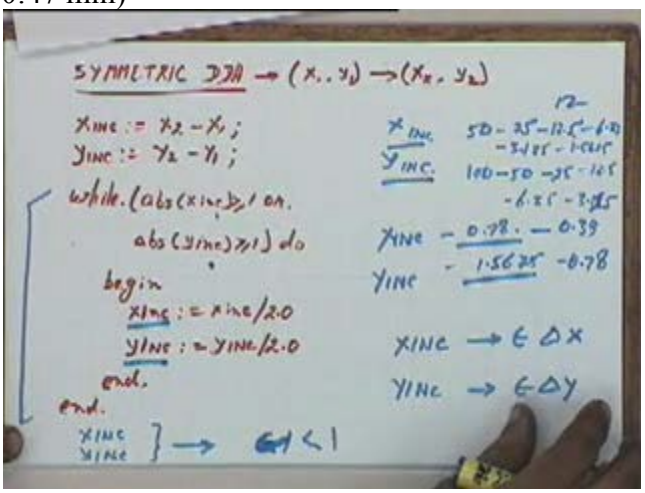
We will start with the symmetric DDA and we will write this algorithm for drawing a line from a point (x_1, y_1) to a point (x_2, y_2). The first step would be to compute this Δx and Δy. So let's say we will call this Δx as x increment and that will be equal to $x2 - x1$, we will say y increment will be equal to $y2 - y1$ y_2. Then the way we compute this $\varepsilon * \Delta x$ and $\varepsilon * \Delta y$ would be that lets say if I take this x increment and y increment and I keep dividing them by 2. I say x increment is initially 50, y increment is 100. I keep dividing them by 2, 50 will become 25, 100 will become 50, this will become 12.5, this will become 25, this will become 6.25, this will become 12.5 then 3.125, 6.25, this will become 1.5625 and this will be 3.5 and this will finally become 0.78 something and this will become 1.5625.

(Refer Slide Time: 00:09:47 min)



We will keep dividing both of them by 2, till one of them becomes less than 1. The idea is if you saw what was happening here, if this value of $\varepsilon * \Delta x$ becomes very small then we are doing wasteful computation, if it is very large we are not displaying the pixels correctly. So this value $\varepsilon * \Delta x$ or $\varepsilon * \Delta y$, at least one of them has to be less than one. If both of them are greater than one, we will end up missing some pixels, so at least one of them should be less than one. At the same time it should not be very much less than one, if it is very much less than one then again we will be doing a lot of wasteful computation because if any of them is less than half let's say then the same pixel will be displayed again and again.

(Refer Slide Time: 00:10:47 min)

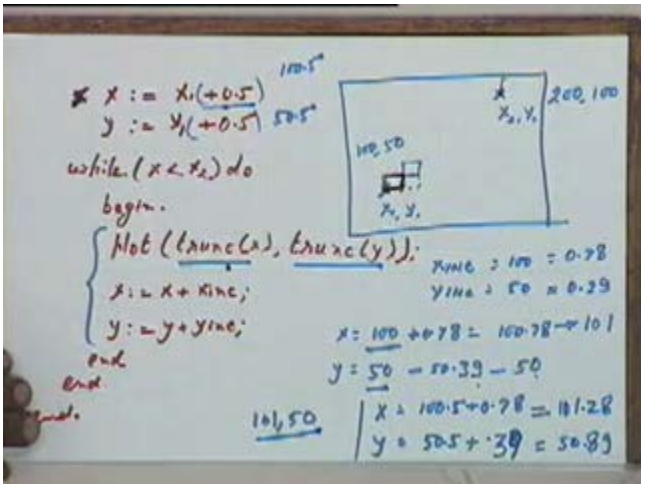


So therefore what we do is we will take x increment y increment, keep dividing both of them by two till at least one of them is less than one. So, we will write a piece of code for that. While… you are all familiar with the Pascal, so I am writing in a somewhat Pascal like syntax. So essentially what we are doing is in this while loop, while either absolute value of x increment or absolute value of y increment is greater than one. If any one of them are greater than one we will divide both of them by 2 and we will keep repeating this process till both of them become less than or equal to one. I think I said it wrong earlier. Either of them. Again. Either of them becomes less than one. Both of the operator took at the greater. While either of them is greater than one I will keep dividing, so till both of them become less than one, I will keep dividing. At this point x increment is 0.78 and y increment is 1.5625. This condition is still satisfied, absolute value of x increment is greater than 1 or absolute value of y increment is greater than 1. So I will again carryout this loop which means this will become 0.39 and this will become 0.78. So I keep dividing till both of them become less than 1. I think I said it wrong earlier. Till both of them become less than one. And the first time I find that both of them are less than one, I will come out of this while loop. Is that all right? So at the end of this loop, at the end of this while

loop my x increment and y increment both these variables will have values less than or equal to 1.

In fact not equal to one, so I put an equal to here that both of them will be less than one. And in terms of what I have written earlier $\varepsilon * \Delta x$ and $\varepsilon * \Delta y$, my x increment is nothing but $\varepsilon * \Delta x$ and the variable y increment is nothing but $\varepsilon * \Delta x$. Ok. Now in this both x increment and y increment have to be declared as real numbers, they cannot be integers. If they are integers then when I am dividing, I divide 25 by 2, it will not be 12.5. It will get truncated and maybe it will be 12 because of that there will be an error and the slope of the line will not be correct. So both x increment, excuse me and y increment have to be declared as real numbers. Now after this stage we have the values of x increment and y increment. Now we can start the display part of it.

(Refer Slide Time: 00:15:52 min)



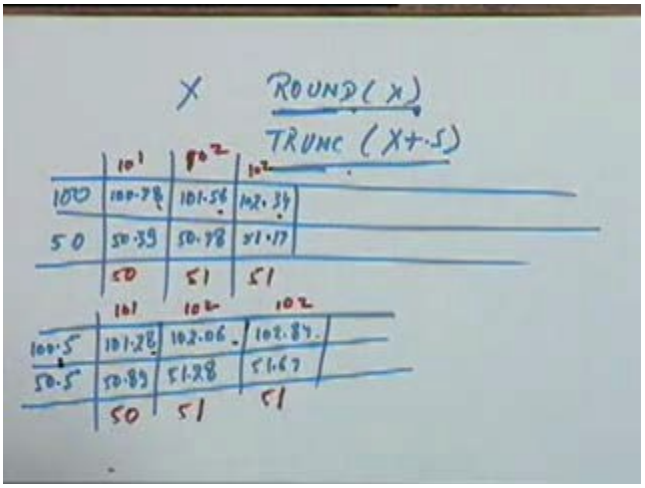
So what we will do is we will say x will be initialized to x_1. What we are doing now is this is our screen, this is our starting point, and this is the end point. The starting point is x_1 , y_1, the ending point is x_2 ,y_2. So the variables x, y would be the variables of the current displayed pixel or the variables corresponding to the current displayed pixel. So the first pixel will display will be the pixel x_1 y_1 and the second pixel will display will be x_1 y_1 incremented by x increment and y increment, so there should be some increment in this direction and so on. So we will initialize x = x_1 and y = y_1. Now I said x = x_1 but I will come back to it, for certain reasons we will be actually putting it equal to plus 0.5, both of them. I will come back to it, why do we put a plus 0.5, when we write down the full algorithm then I will explain why we are adding a 0.5 here. Then I have put x, y equal to this pixel, I have to keep displaying till I reach my last pixel. So we say while x is less than x_2, do begin. Ok .While x is less than x_2 we are doing this, these three statements are being repeated in a loop. The first is plot truncate of x and truncate of y.

Let's say I have to draw a line from a point 150 to a point 200, 100. So initially my x increment is equal to 100 and y increment is equal to 50, I will keep dividing by 2. So finally I will get this to be probably 0.39 and this would come out to be 0.78. I am initializing $x = x1 + 0.5$, for the time being let's take it to be x_1, so my x is 100 and y is 50. First time I will plot x y, so pixel number 100, 50 will get displayed. Second time x will get incremented by 0.78, so it will become 100 plus 0.78 which is equal to 100.78 and y will become 50.39 but now this screen is an array of pixels. In an array I can only address integer locations. So instead of 100.78, I have to address pixel number 101. I will round it off and instead of 50.39, I will address pixel number 50. So the pixel to be displayed after the pixel number 100, 50 will be the pixel 101, 50. Is that understood? Ok. If the pixel to be displayed is 101, 50 that is this is 100, 50; 101, 50 will be this pixel.

Now instead of rounding off if you notice, I have put truncation over here. If I am truncating 100.78 will actually become 100 but instead of x_1 I have put x_1 plus 0.5. So initially pixel 100, 50 so x became 100.5 and y became 50.5. In the first cycle I truncated both these numbers and displayed 100 and 50, displayed the pixel number numbered as 100, 50. In my second time when I come here my x will be 100, I will write here again, x will be 100.5 plus 0.78 and y will be 50.5 plus 0.39. Sir why did you save the, if we use the command round off without initializing the value of x and y by increasing 0.5. That's right if we do not put this 0.5 here then we will replace truncation by rounding off. The same thing.

It will be the same thing but since we are putting 0.5 here, we are putting truncation instead of rounding off. That is exactly what I was trying to come to. Now x will become 101.28 and this will become 50.89. Now when I truncate these two I will still get 101, 50 which is this. Now the result of adding 0.5 here and then truncating is the same as if I don't add 0.5 here and do a rounding off. Is that appreciated?

(Refer Slide Time: 00:23:02 min)



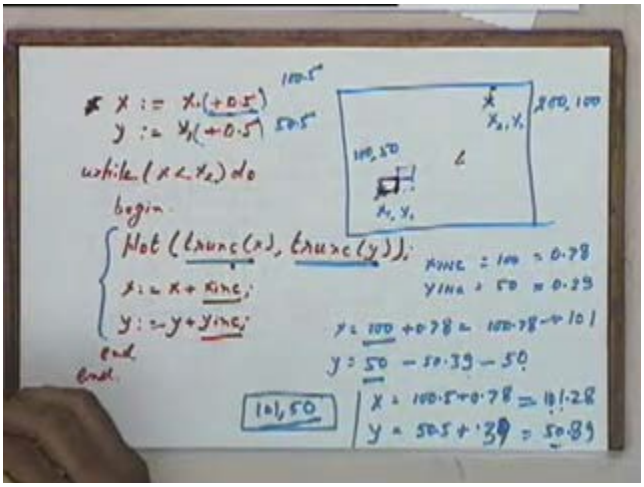
In general if you take any number x, round off x would be the same as truncation of x plus 0.5. This you can check up for any number. That is why here we are adding 0.5 in the beginning and

then truncating every time and the reason we prefer to do this is truncation is much faster than rounding off. Truncation takes much more time sorry the other way round. Rounding off is very slow and truncation is faster. So we prefer to truncate rather than rounding it out. Sir but this is the only thing for the first two pixel, the rounding off and truncation will mean coincide. No, but for the rest afterwards then they will. For all of them it will coincide. No, for example you take third point that maybe different. No let's just. If you would be adding 0.5 out there, the equation of line will be changing like we can't add inside the loop. Just a sec.

In one case we will start with 100, 50, we are starting with 100, 50 this will become 100.78, 50.39. Then the next time it will become, you add 0.78 again 101.56, this will become 50.78. Then it will become, we will add 0.78 again so this will become 102.34. Is that right? And this will become 51.17 and so on. In this case we are supposed to round off. So this one becomes 101, this will become 50, this will be 102, this will be 51, this will still be 102 and this will be 51. Is that okay? Now I have added 0.5 in the beginning itself, so this becomes 100.5 and 50.5, this will be 101.28 and 50.89. This one will be 102.06 and this will be 51.28. This will be 102.84 and this will be 51.67. Now when I truncate this will become 101, this will become 50, this will become 102, this will be 51. This will become 102 and this will be 51. The results are the same.

See basically when I have added 0.5 in the beginning that 0.5 is carried over in all the iterations. All these numbers will be 0.5 higher than these numbers. I have added 0.5 only in the beginning but that effect will remain in all the iterations, in all the cycles let's say. So initially this number is 0.5 higher than this. This is also 0.5 higher than this, this is also 0.5 higher than this. So, this effect will go on in all the iterations. As a result if I take any of these numbers, the result of truncating them will be same as what I will get from there. Is that okay?

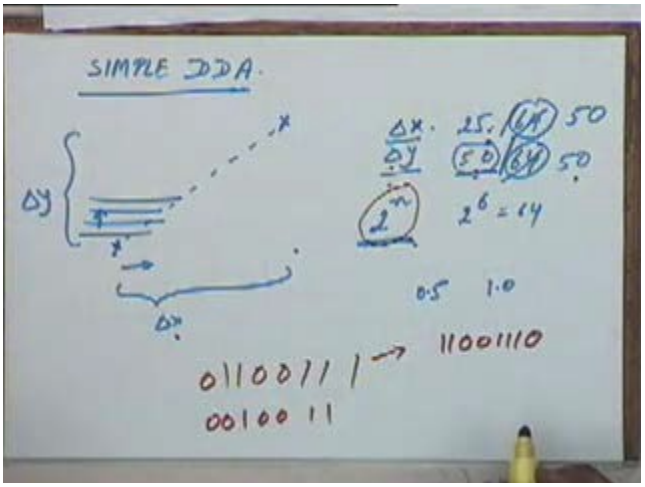
(Refer Slide Time: 00:27:18 min)



So coming back to this algorithm, what we have done is we started with the pixel x y and we have displayed pixels in the direction of the slope by adding x increment and y increment every time and instead of rounding it off, we have added 0.5 in the beginning and then we are truncating it every time. And however we have found out x increment and y increment. We took

$\Delta x = x2 - x1$ and $\Delta y = y2 - y1$ and kept on dividing them by 2 till both of them were less than one. So this is referred to as a symmetric DDA algorithm or symmetric digital differential analyzer. Any question on this symmetric DDA algorithm?

(Refer Slide Time: 00:28:23 min)



Now we will see the algorithm which is referred to as a simple DDA. If we have to draw a line from this point to this point, in the symmetric DDA what we did was we took increments in the x direction and y direction depending on the multiples of two that means we had taken Δx and Δy. Essentially what we did was we took the higher of the two, let's say if the higher of the two is 50, we found out some number 2 to the power n which is bigger than 50. Some multiple of 2 which is bigger than 50 and we divided both of them by that because you are dividing both Δx and Δy by 2 every time. So if you are taking 50, so $2^6 = 64$.

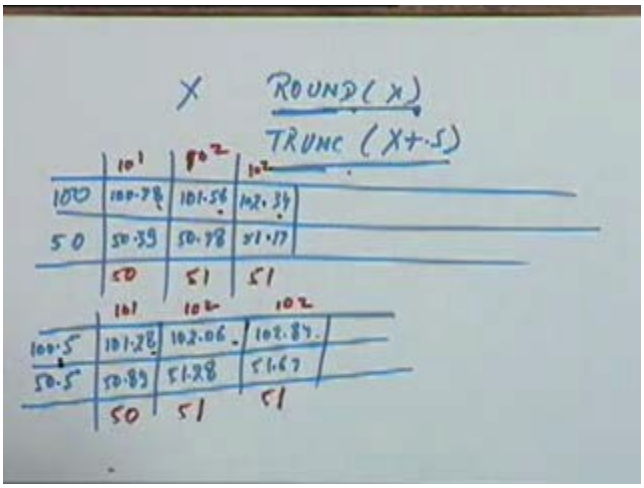
So we are dividing this by 64 and Δx let's say if it is 25, we are dividing that also by 64. So this Δx and this Δy are both being divided by 2^n. In a simple DDA instead of dividing it by 2^n, we will divide it by higher of Δx or Δy. So that out of Δx and Δy at least one of them should be equal to one, between Δx and Δy if let's say one is 25, the other is 50, we will divide both of them by 50. You are not clear? My total increment in the x direction, let's say is 25, my total increment in the y direction, let's say is 50. Instead of dividing it by 64, now we will divide it by 50. So that the increment that I will take in the x direction will be 25 by 50 or 0.5 and the increment that I will take in the y direction will be 50 by 50 or 1.0.

As a result either Δx or Δy will always be equal to one, one of the two will always be one. In the symmetric DDA case both of them are necessarily less than one. In this case one of them has to be equal to one. The basic reason behind doing this is that if my increment in both the direction is less than one then it is possible that I will end up displaying the same pixel again.

While if one of them at least is equal to one then that will not happen because in one direction at least, I am incrementing by an amount equal to one.

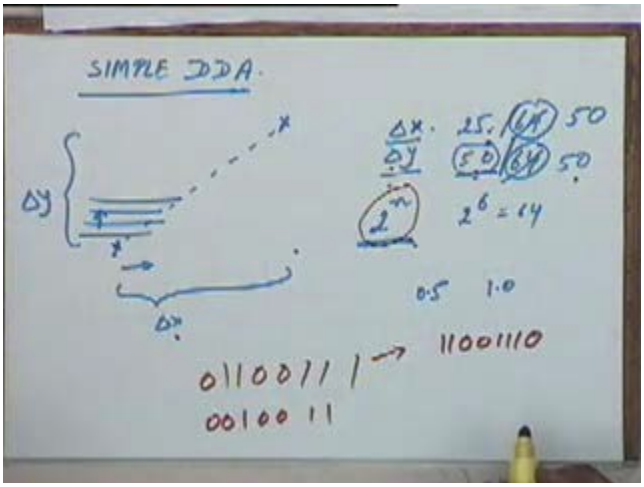
Of course in a symmetric DDA also we will never end up displaying the same pixel again. That will not happen in the symmetric DDA either. Somebody said something? That was happening sir. Again. That was happening in the last example, in the last three points that you have taken the last two.

(Refer Slide Time: 00:32:34 min)



So these two points were being the same, you are right. In the symmetric DDA these two points can be the same because the increment was less than one but if my increment is more than one, at least one of the two increments is more is equal to one then the same point will never get displayed again because at least one of the pixels will shift. The next pixel to be displayed either in the x direction or in the y direction, it will be incremented by one.

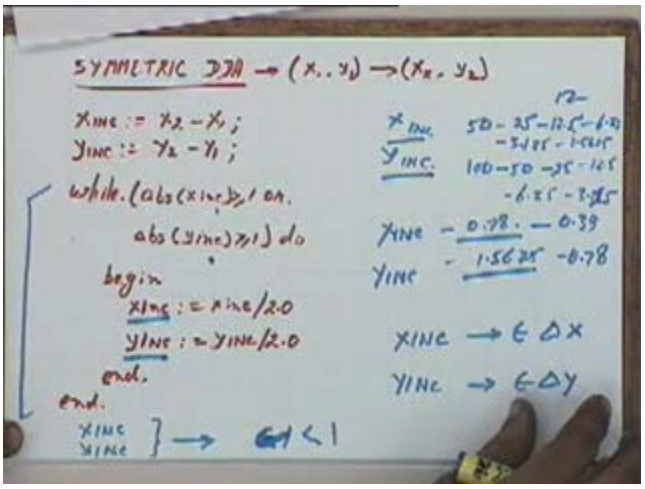
(Refer Slide Time: 00:33:09 min)



So in this case if my Δy is equal to 1, in that case every time I will take a pixel in the new row. Every time I am increasing Δy by 1, Δx may or may not increase but Δy will

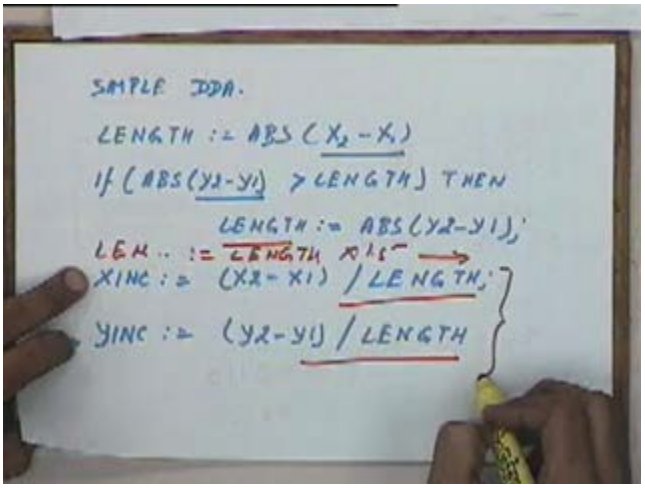
definitely increase that is a simple DDA. So we will just write down the algorithm for this and see how it compares.

(Refer Slide Time: 00:33:46 min)



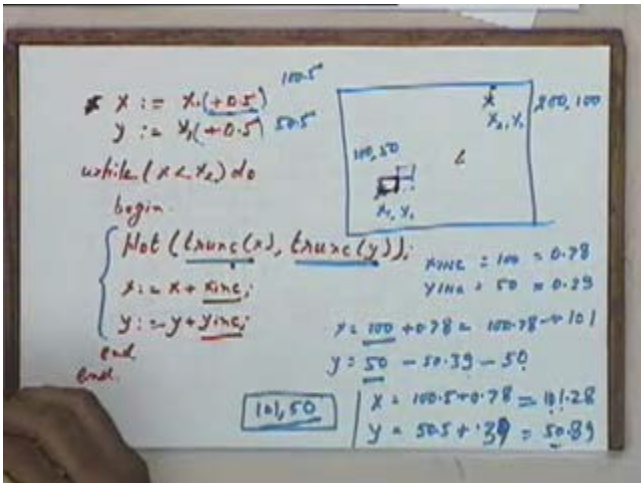
If you look at the symmetric DDA, this is the algorithm we had written. Now in contrast to this we will retain these two steps and this part will now change. In fact these will also not be required.

(Refer Slide Time: 00:34:07 min)



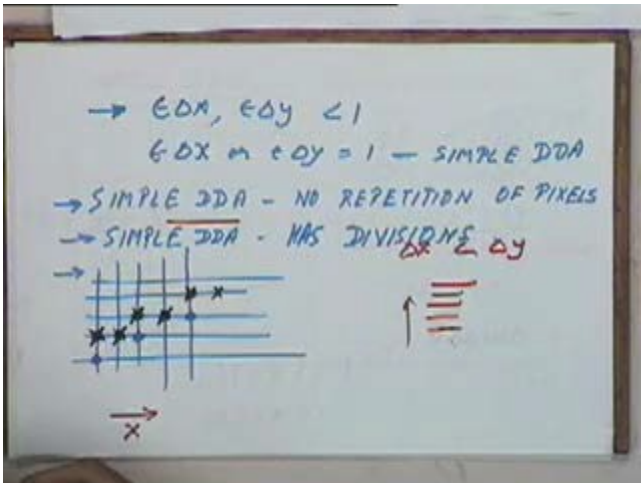
What we will do is simple DDA. We will take a variable let's say length which will be to start with let's say absolute value of $x2-x1$. Then we will say, if absolute value of $y2-y1$ is greater than length, then length will be equal to absolute value of $y2-y1$. Essentially between $x2-x1$ and $y2-y1$ whichever is greater in magnitude that we are assigning equal to length, ok and then we will say x increment will be equal to $x2-x1$ divided by length and y increment will be equal to $y2-y1$ divided by length. Is that okay?

(Refer Slide Time: 00:35:56 min)



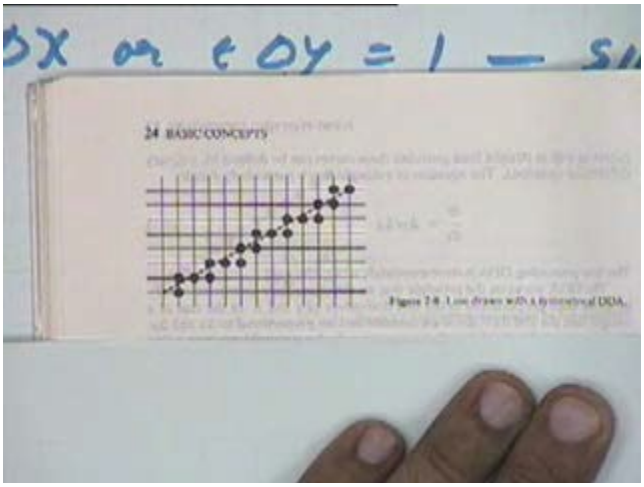
Once you have calculated x increment and y increment then the rest of the algorithm will remain the same. Basically this loop for displaying the line is still the same. We have already found out what is the value of x increment and y increment, we know the starting point, we repeated in the same loop and we will able to draw the complete line.

(Refer Slide Time: 00:36:35 min)



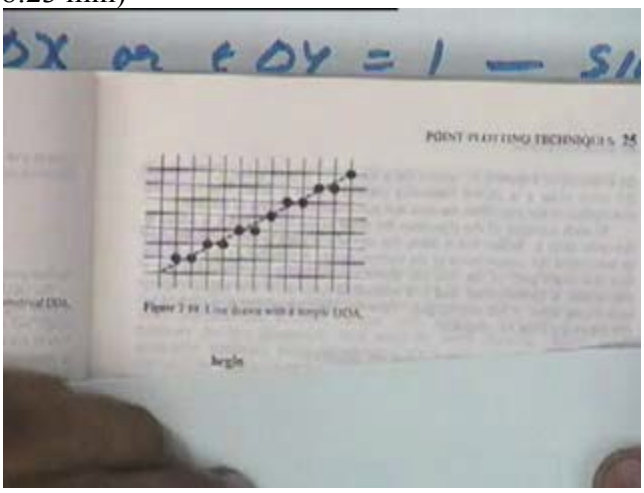
The difference between these two algorithms, the first difference is essentially the value of $\varepsilon\Delta x$. In symmetric DDA both $\varepsilon\Delta x$ and $\varepsilon\Delta y$ are less than one. In a simple DDA either $\varepsilon\Delta x$ or $\varepsilon\Delta y$ will be equal to one. One of them will always be equal to one, this is in the case of simple. As a result in a simple DDA there will be no repetition of points or repetition of pixels, the same pixel will not be displayed again. In fact I will just show you some lines drawn by the two algorithms.

(Refer Slide Time: 00:38:03 min)



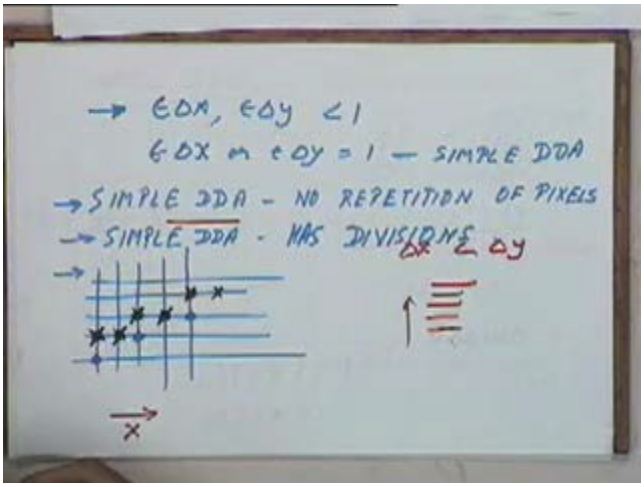
Are you able to see it? This figure, can you see it clearly? This is the line drawn by a symmetrical DDA.

(Refer Slide Time: 00:38:23 min)



In contrast to this, a line draw by the simple DDA will look something like this. Is it visible? I hope so.

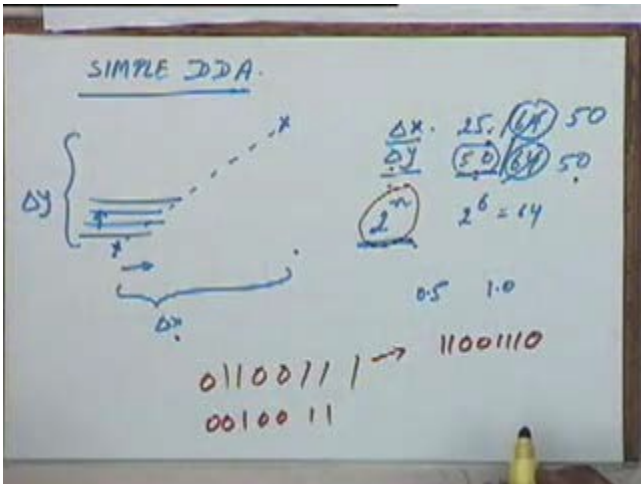
(Refer Slide Time: 00:38:46 min)



If you notice the difference between the two. The points in simple DDA are less. Yes, the number of point in simple are less. The blue circle that I have made, those are the lines or the pixels selected under the symmetric DDA. The black crosses are the one selected under the simple DDA. These pixels are extra in the case of symmetric DDA. So the symmetric DDA draws more pixels than the simple DDA, the simple DDA always draws less number of pixels and the reason is let's say Δx is less than Δ y that means the line is traveling more in the y direction. If the line is traveling more in the y direction then in the y direction, you are at least incrementing by one pixel every time. So in the y direction you will always be moving by one pixel. So in every iteration, you are going to a different row of pixels that need not be the case in symmetric DDA.

In the y direction also your increment will be less than one. So on the same line, you will end up drawing sometimes two pixels before going on to the next line and that is what is happening in this case. Here in the x direction, the slope is more. The movement is more in the x direction. So in the simple DDA every time you move in the x direction, every time you draw a new pixel you will at least move in the x direction by one. So that is why you can in see the black crosses and you go from one pixel to a second pixel, you are always moving in the x direction at least by one pixel while that is not true in the case of the symmetric DDA. This is one pixel drawn, the next pixel drawn is here. So in the x direction you have not moved at all. Similarly here one pixel was drawn and the next pixel drawn is this one. Why this choice of 2^{64} . Sir or powers of 2, why don't we have just 1.5 times the length. That's a good question.

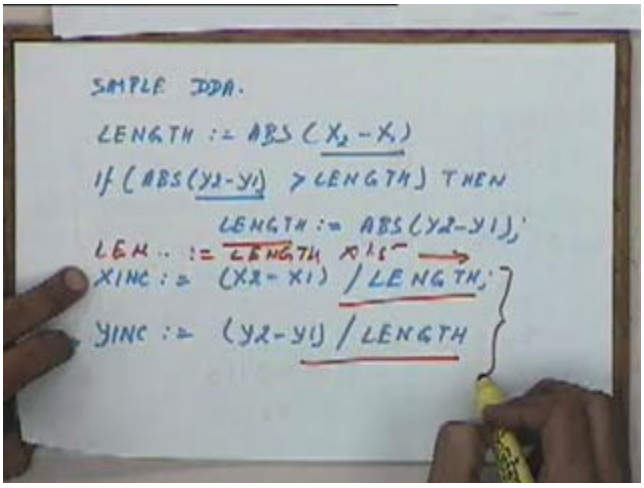
(Refer Slide Time: 00:42:16 min)



Essentially what he has asked is let's say $\Delta x = 25$ and $\Delta y = 50$. We kept on dividing by 2 in every iteration till both the numbers became less than one. Why divide it by 2? Anyone trying to make a guess on that, why should you divide by two every time? What I am saying is that instead of why divide 2, like two is obvious because like left shifting it will give a faster computation. But using the same algorithm at the end if found the length we just multiply the length by 1.5. I will say that divide by that will be a first this because this is in a loop that one won't be in a loop. I will explain both the points.

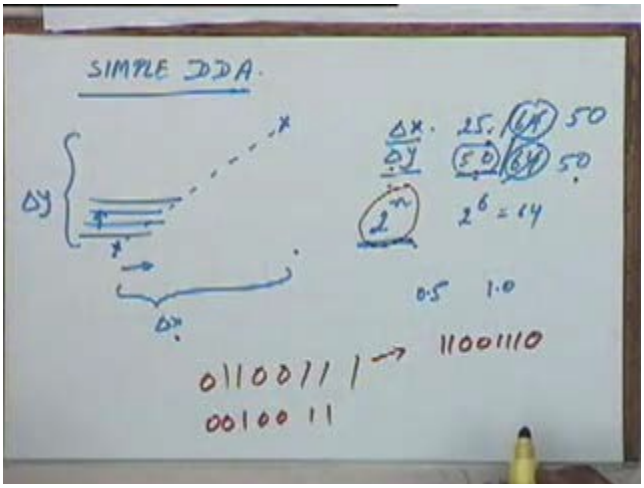
See the first reason, the first part of the question is why we are dividing by two every time and the reason is given himself that is division by 2 is simpler than division by any other number. We are trying to divide any number by 2, let's say the number is you want to divide it by 2. All that you have to do is a right shift. If you want to multiply it by 2, all that you do is a left shift. That means if you want to divide this by 2 let's say if this is your number, this will just become something like this, this is this number divided by 2. If you want to multiply this number by 2, all that you will do is a left shift and this number will become like this. So division by two is much easier to implement than division by any other number. Therefore we choose a multiple of two and we keep dividing by two every time. The second thing he is saying is that why divide by 2 at all.

(Refer Slide Time: 00:31:42 min)



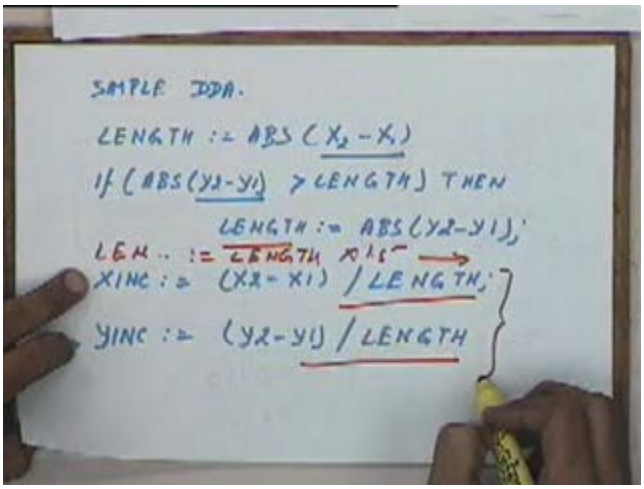
In the simple DDA, we found out the value of length, at this stage. All that we need to do is just say length is equal to length multiplied by 1.5 or some number. If you notice this value of length will give you a different set of pixels than what the symmetric DDA gives. It will not give the same set of pixels.

(Refer Slide Time: 00:44:55 min)



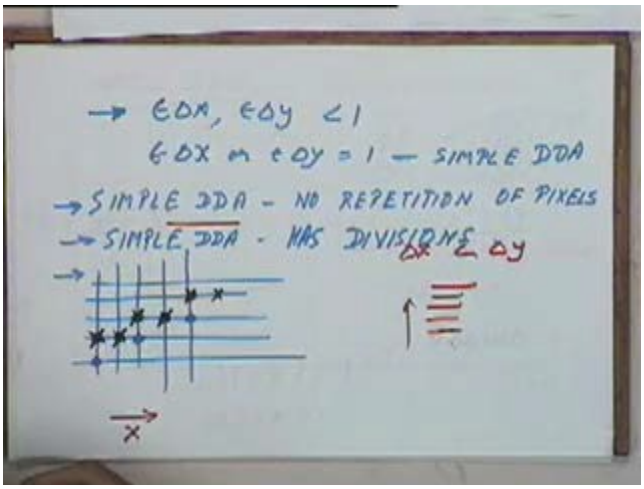
Because when you are multiplying by, this figure $2^6=64$ is not equal to 1.5 times length. No, but why do we get. Why should we have a line which is the same as a symmetrical algorithm? There is no reason for you to have the same line as such.

(Refer Slide Time: 00:45:17 min)



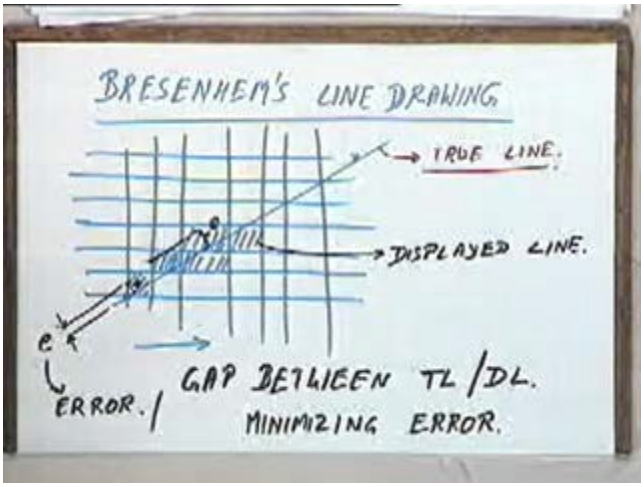
But what I am saying is just you can do something like this, you will get a different set of pixels. Whether this is more efficient than the symmetric DDA that I won't say for the simple reason that we are having what is called divisions over here. We are having two divisions. Divisions are more expensive than dividing by 2. So this algorithm is likely to be less sufficient than the symmetric DDA. The advantage that symmetric DDA has over a simple DDA is its speed because there is no division involved and we are only doing a right shift sorry left shift. So the advantage of the simple DDA is the speed. If you use this as modified algorithm then we lose that advantage. Yes,this algorithm will also work. You will get a set of pixels which will be able to display that line but the disadvantage compared to the symmetric DDA would be the loss in speed.

(Refer Slide Time: 00:46:32 min)



So we are just comparing the two algorithms. We said in simple DDA there is no repetition of pixels but simple DDA has divisions. Divisions make the algorithm much slower.

(Refer Slide Time: 00:47:01 min)



We will now see another algorithm which is called the Bresenhem's line drawing algorithm. We will describe the basic principle of this algorithm today and then we will go on to the details in the next lecture. I say this is the array of pixels and you want to draw a line from one point here to some point here. Now this line is what your line should actually look like, so we will call this as the true line. Now if you actually select a set of pixels let's say this is one pixel I select, maybe this is the next pixel I select, this is another pixel, this is one pixel, this is one pixel and so on. When I am selecting this pixel actually if I consider center, this is the point that has been selected or if I am talking of the starting plane this is the place, this is the point which has been selected. The distance of this point from this line is this. Ok. When I select the next pixel let's say this one, the pixel that has been selected is at this location, I am talking of at the start of the pixel, this is the point. So here the distance of this location from this true line is this much. So, the set of pixels that are actually displayed these constitute what we call as the displayed line, this straight line that I have drawn that is the true line.

Now the Bresenhem's line drawing algorithm works on the principle that the distance between these two lines should be the minimum. This distance e that is actually the error that is there between the displayed line and the true line. So this distance e is the error or we will say gap between true line and displayed line and the Bresenhem's line drawing algorithm works on the principle of minimizing this error. Again please. Is somehow like root means square distance minimizing. We will come to that. Whether we consider the root mean square distance, the perpendicular distance or the distance along the x axis or the distance along y axis we will come to that soon.

Essentially what we do is just to give a brief idea, we take the direction of maximum movement and we measure the error in a direction perpendicular to that. We will come to that later on. So the basic idea is that we try to minimize this error. In the previous algorithm simple DDA or the

symmetric DDA, we were not talking of this error at all. We were just displaying pixels in the direction of the slope. We have not even defined the error; we are not bothered about whether there is any error between the displayed line and the true line. So we will see this algorithm in detail in the next class. We will stop here now and next time we will see this algorithm in more detail.