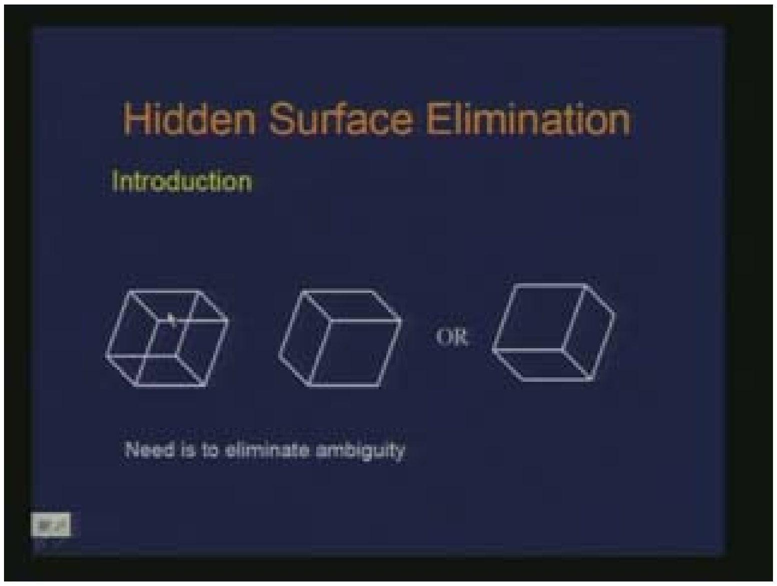
Introduction to Computer Graphics

Dr. Prem Kalra
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 28
Hidden Surface Elimination

Today we are going to talk about hidden surface elimination. The motivation is that as we have seen in the case of clipping what does clipping offer? It actually eliminates the portion of the scene which is not contained in the viewing frustum in 3D. Similarly, there are issues like when you want to display the scene there are portions of the object or there are objects which are hidden by other objects and you would not like to display them. So eliminating what is hidden is what we studied under hidden surface elimination. That basically requires you to determine the visibility of the objects or part of the objects which you display. Sometimes it is referred has visibility detection or visibility determination. If we are considering an option of displaying a 3D object in its wireframe form that means I basically display the edges of the polygon using lines. This is what I have in the wireframe.

(Refer Slide Time: 03:55)



Hidden Surface Elimination

Introduction

Need is to eliminate ambiguity

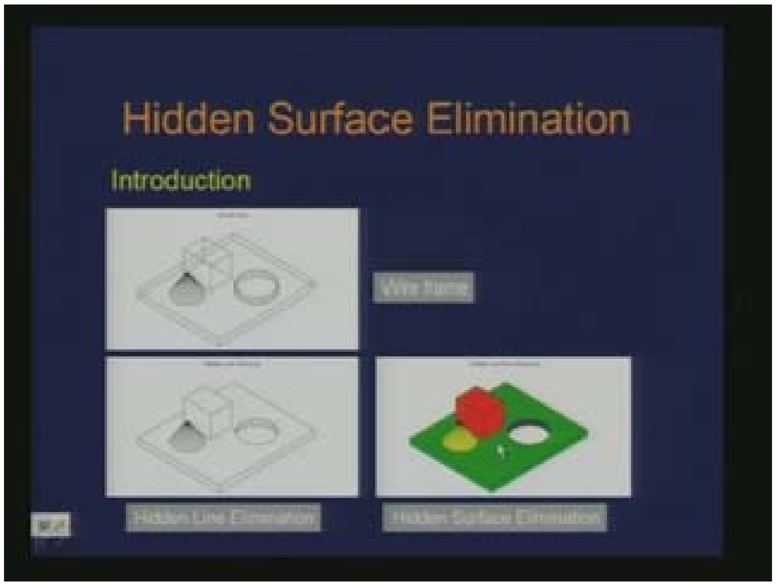
Clearly it is not known whether the object is this or the object is this unless I perform the hidden line elimination that means the lines which are hidden are not displayed. So if this is the situation I know that this is actually in front this line on the face corresponding to this line is in front so I get this. Similarly if this is the situation then I know that this face or the lines corresponding to this face is in front. So this way of removing the ambiguity is obtained through hidden line or surface elimination.

Let us look at the various methods or ways in which we perform this. One of the methods we already saw was ray tracing. Ray tracing does it by its inherent nature of rendering.

You shoot the ray then you find out the intersection and the closest intersection point is the point you render and that already takes into account for the visibility because that is the front most point.

Here is another example, you have this wireframe scene so these are the various lines which are displayed. So one can look at the problem in terms of hidden line elimination where you are plotting or displaying the lines. This was of particular important when in the earlier days these plots were drawn on the scenes of 3D objects and only the line drawings were done. Or you may be actually referring to the hidden surface elimination where you are looking at the face or the polygon or the surface of which you determine this visibility. So, one may be solving hidden line elimination or hidden surface elimination. In today's graphics we generally do hidden surface elimination. We will look into at least one of the methods which refer to hidden line elimination but we will talk about hidden surface elimination in detail.

(Refer Slide Time: 06:25)

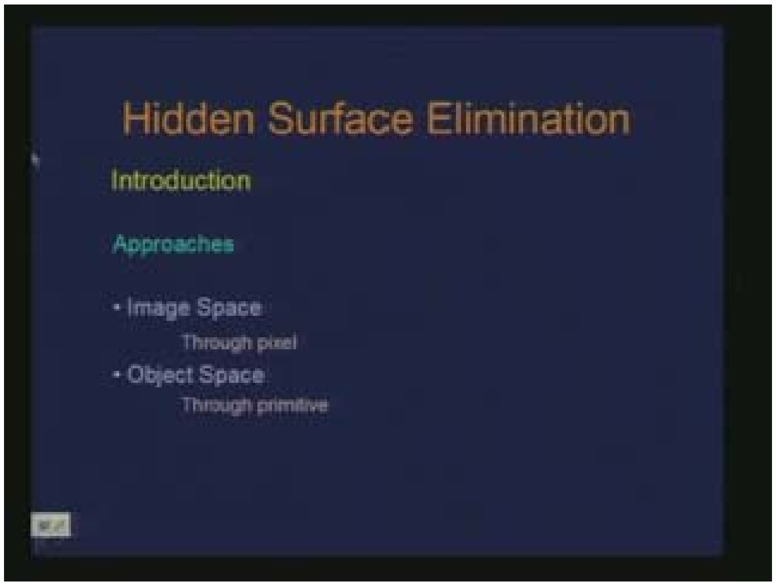


Hidden Surface Elimination

Introduction

When we talk about the approaches for hidden surface elimination one can see it in two ways. There could be two ways to look at the problem depending on how you are performing the computation or where you are performing the computation.

(Refer Slide Time: 07:29)



Hidden Surface Elimination

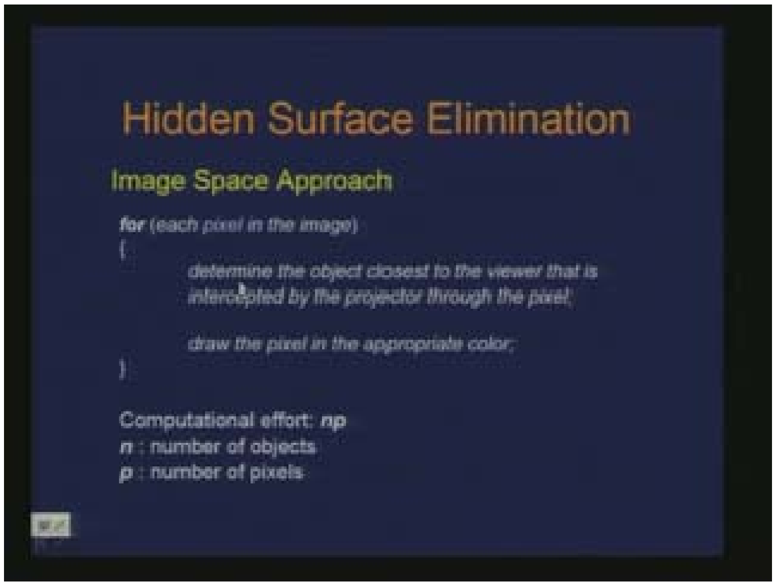
Introduction

Approaches

- Image Space
  Through pixel
- Object Space
  Through primitive

One could be image base or image space method. So here what we are doing is we are basically determining the visibility per pixel so it is a through pixel. For a given pixel you want to determine what is visible in the scene which is known as image space approach. The approach is object space approach where you try to see among the objects in the scene or the parts of the object for the object itself what is the portion or the objects which are visible or which are occluding other objects. So the computation which you are performing is in object space.

(Refer Slide Time: 09:29)



Hidden Surface Elimination

Image Space Approach

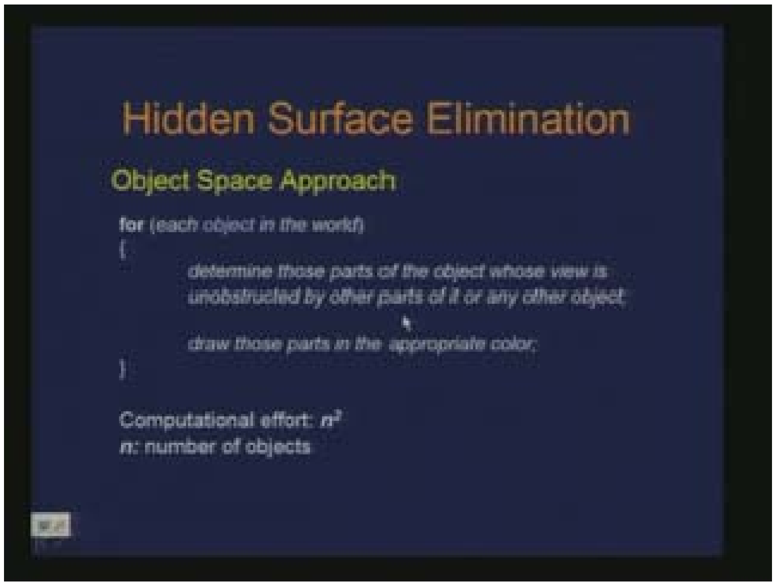
```
for (each pixel in the image)
{
        determine the object closest to the viewer that is
        intercepted by the projector through the pixel;

        draw the pixel in the appropriate color;
}
```

Computational effort: *np*
n : number of objects
p : number of pixels

In image space approach what we are doing is for each pixel in the image we determine the object closest to the viewer intercepted by the projector which is coming from the center of projection or the eye through the pixel and then we draw the pixel in the appropriate color. That is what happens in image space approach. So the precision of the computation is the pixel. So, if you want to look at the computational effort which is given here, if I have n as the number objects in the scene, p as the number of pixels of the screen which needs to be rendered then the computational effort is of the order n(p).

For each pixel I am basically determining out of the n objects which part of the object or which portion of the object or which point of the object needs to be rendered. So, that is the order of computational effort.

(Refer Slide Time: 11:01)



Hidden Surface Elimination

Object Space Approach

```
for (each object in the world)
{
        determine those parts of the object whose view is
        unobstructed by other parts of it or any other object;

        draw those parts in the appropriate color;
}
```

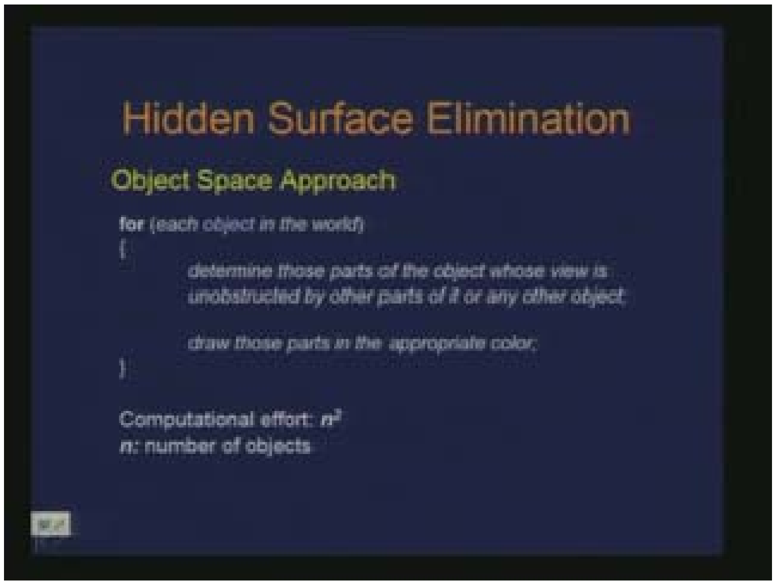
Computational effort: n^2
n: number of objects

On the other hand, if we are talking about object space approach here we are looking at, for each object in the world determine those parts of the objects whose view is unobstructed by other parts of it or any other object and then draw those parts in the appropriate color. So the precision in which you will be working will be the precision with which you define the object. And if you look at the computational effort since we are assuming n to be the number of objects it is going to be of the order n power 2 because for every object you are going to see what happens with respect to the other object. Therefore this is sort of an overview of the two approaches namely the image space approach and the object space approach.

Some of the methods of hidden surface elimination:

One of the methods is more from the historical perspective, and actually speaking is hidden line elimination is this floating horizon algorithm. Typically we are looking at a surface which is defined in a functional form like f(x, y, z) is equal to 0 and one can do the plot of this function considering the slices at different z. For instance, for one constant value of z I can make a plot of this so find out (x, y). Therefore, x is another function (g, y, z) and similarly y is another function some (h, x, z) just to have a 2D plot. So, if I do this then basically the function may look like this and all these values z_1 z_2 z_3 z_4 z_5 are these constants or values where the section of the surface is considered.

(Refer Slide Time: 11:02)



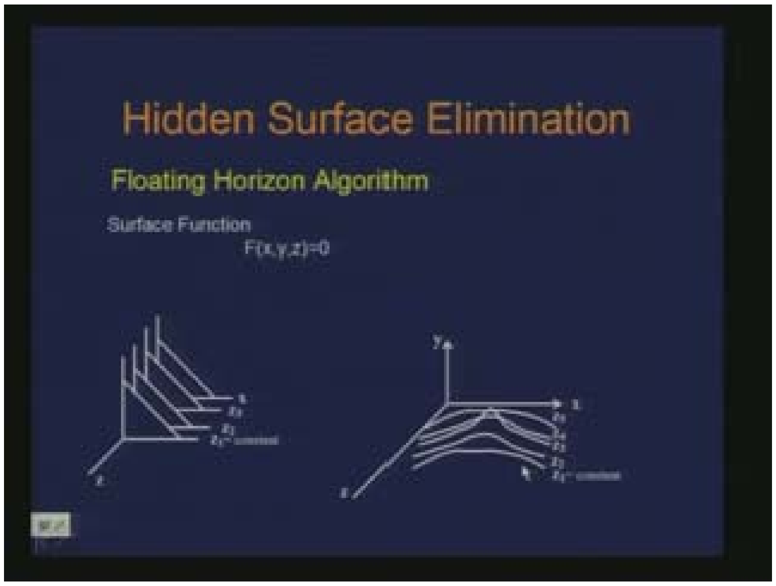
Hidden Surface Elimination

Object Space Approach

```
for (each object in the world)
{
        determine those parts of the object whose view is
        unobstructed by other parts of it or any other object;

        draw those parts in the appropriate color;
}
```

Computational effort: n^2
n: number of objects

(Refer Slide Time: 13:46)



Hidden Surface Elimination

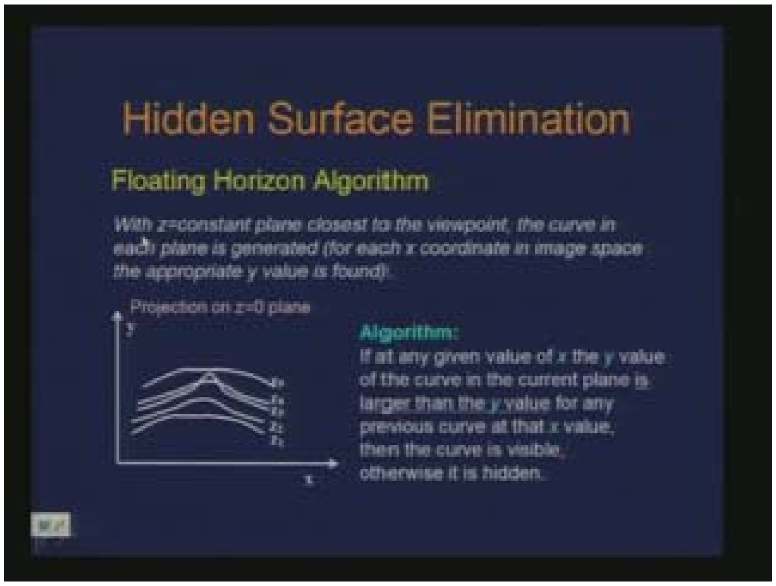
Floating Horizon Algorithm

Surface Function

F(x,y,z)=0

And these are the corresponding plots of the surface for a particular value of z. So now what we trying to do is, figure out what portion in that function should be visible. And there is an implicit order by which we are looking at this from front to back. That means the front most z is z_1 then z_2 and so on. So, if I am doing a plot in the order of front to back then how should I make sure that the hidden line or hidden curves with respect to these plots are not plotted or not displayed.

(Refer Slide Time: 17:24)



Hidden Surface Elimination

Floating Horizon Algorithm

With z=constant plane closest to the viewpoint, the curve in each plane is generated (for each x coordinate in image space the appropriate y value is found).

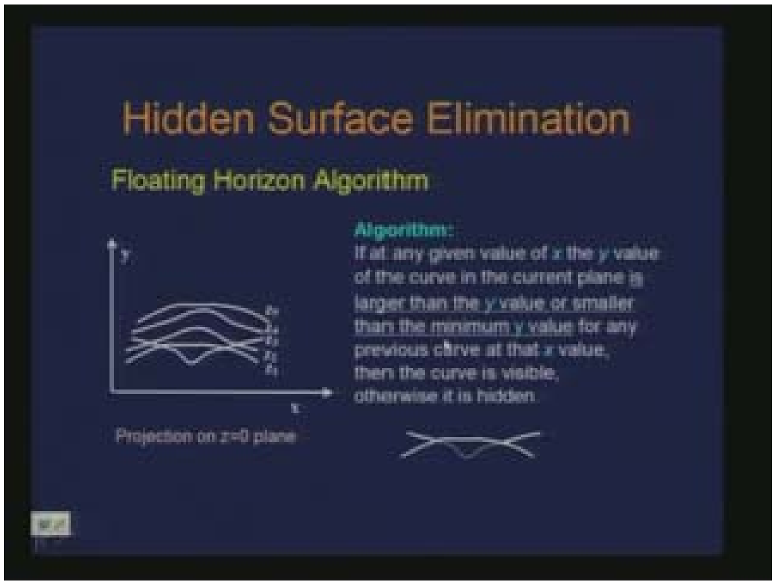
Projection on z=0 plane
y
z_5
z_4
z_3
z_2
z_1
x

Algorithm:
If at any given value of *x* the *y* value of the curve in the current plane is larger than the *y* value for any previous curve at that *x* value, then the curve is visible, otherwise it is hidden.

Now what we have is z equal to constant which is a plane closest to the viewpoint which is the front to back order, the curve in each plane is generated that means for each x coordinate in image space the appropriate y value is found. This is what is meant by generation. Therefore we are looking at the projection on z is equal to 0 plane. This is what your generation of curves is going to be z_1 z_2 z_3 z_4 and z_5. So when you look at this one can propose an algorithm. What is happening is if you look at the plot here at this point for z is equal to 4 this is the portion or the particular point which is actually hidden.

So one can propose an algorithm where one says that if at any given value of x the y value of the curve in the current plane, so z_3 is already plotted the previously plotted plane and now we are plotting the curve for z_4 going from front to back. Then we are saying that the value of the curve in the current plane is larger than the y value for any previous curve at that x that the curve is visible otherwise it is hidden. So for this part of the curve it will turn out to be hidden. So it is a very simple algorithm. Now the question is does it work always?

(Refer Slide Time: 21:03)



Hidden Surface Elimination

Floating Horizon Algorithm

Algorithm:
If at any given value of x the y value of the curve in the current plane is larger than the y value or smaller than the minimum y value for any previous curve at that x value, then the curve is visible, otherwise it is hidden.

Projection on z=0 plane

What happens if instead of having a peak here there is a valley? Then you are in trouble it does not work. But actually it requires a very small modification even to account for that.

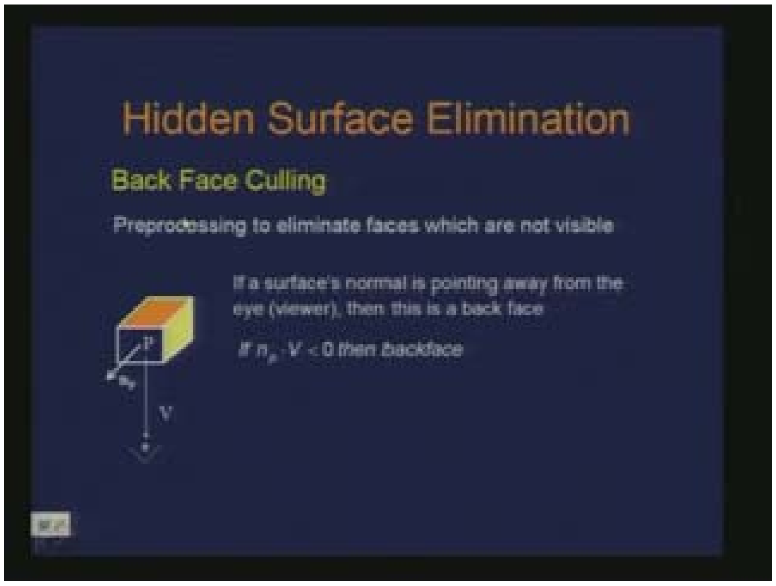
This is the scenario here, you had a plot here for z_1, plot here for z_2 and this is the plot for z_3 where you observe that it actually dips down and according to the algorithm which we just looked at these would not have been plotted, these would have been declared as hidden.

Therefore, what you do is you modify this and say that if at any given value of x the y value of the curve in the current plane is larger than the y value or smaller than the minimum y value for any previous curve at that x value then the curve is visible. This was the value of y earlier. So you also need to look at whether it is smaller than the minimum y value for that x. So in this situation if this point is to be considered then you would find this is smaller than the minimum y value and therefore you display it when the curve is visible. So in a sense what you are looking at is some sort of a horizon which is defined for the y values which you keep shifting. This minimum y value is nothing but some sort of a horizon which keeps floating and then you always compare to that horizon. That is why it is also called as floating horizon.

Of course there are other smaller issues which are also concerning because much of it depends on the sampling of your screen. Any time you want to display a point then you will need to sample this screen and then you will figure out whether this point lies there or not and then you may actually require do an interpolation.

These are not continuous things but these are actually discrete points when they come to the image. But at a gross level this is what the algorithm is. So here we are basically dealing with in line elements because we are just plotting the lines of the curves.

(Refer Slide Time: 23:34)



Hidden Surface Elimination

Back Face Culling

Preprocessing to eliminate faces which are not visible

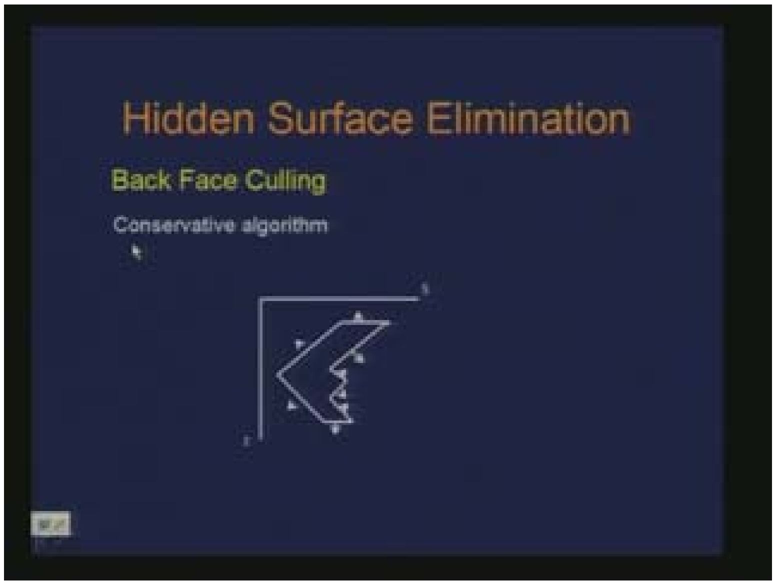
If a surface's normal is pointing away from the eye (viewer), then this is a back face

If $n_p \cdot V < 0$ then backface

Now let us look at another hidden surface elimination which you have also seen in some other context known as the back face culling. What is back face culling? Back face calling is something which helps you declaring that this polygon of the face in an object is facing backwards so you need not display it. So, to determine whether a particular face is a back face or not it is a simple test which you look at with respect to the normal of the face you have, for example point p is the face you have and this is your viewer here so V defines the direction from the face to the viewer and all you are talking of is taking a dot product of this normal with respect to the vector V and check the sign of it.

If it is less than 0 that means if it is negative then you are saying that the surface normal is actually pointing away from the viewer or it is facing backwards and therefore there is no display. It is a very simple way of eliminating the back faces. This is actually a pre processing step. This does not take away all the faces which are possibly not visible. It is actually a conservative kind of an algorithm.

(Refer Slide Time: 26:02)



Hidden Surface Elimination

Back Face Culling

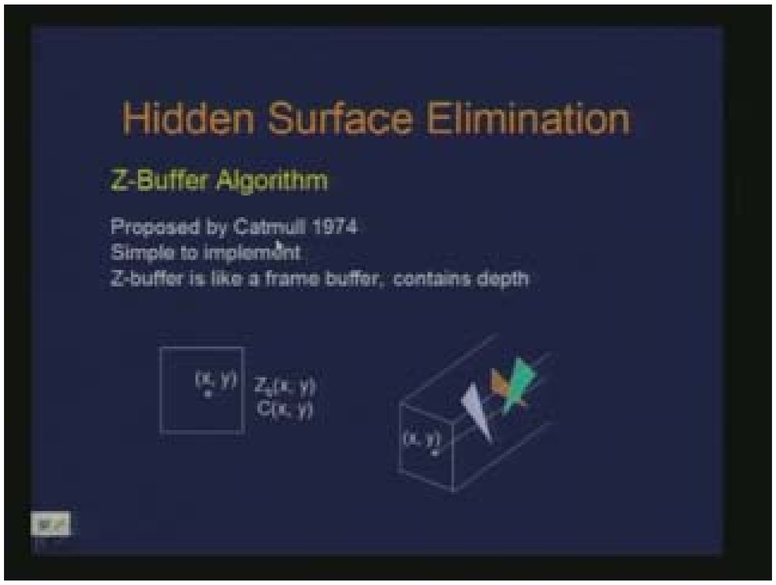
Conservative algorithm

For instance, if I have these faces I have plotted them in 2D (z, x) section and all these arrows are basically showing the normals on those faces. The viewing direction is somewhere there so I am looking at that in this direction from here to here. So what will happen using the back face culling is some of these faces will be marked as back faces and therefore should not be displayed. These green ones are basically the back faces. But if you look at, this is a face which is still not visible because it is actually obstructed by this so it is not considered and that is why we say that the algorithm is conservative.

Therefore it is a pre-processing step you may require to flag out the faces which should not be displayed and then may be you need another part of some other hidden surface element. Similarly it is actually only partially visible. So those things are not determined by this. It just declares whether a particular face is a back face or not. But it is a very simple and straightforward way.

The next technique which is very popular is the Z-buffer algorithm. It was actually proposed by Catmull back in 1974. It is a very old method but still it is used and is extremely popular. It is very simple to implement.

(Refer Slide Time: 29:38)


Hidden Surface Elimination

Z-Buffer Algorithm

Proposed by Catmull 1974
Simple to implement
Z-buffer is like a frame buffer, contains depth

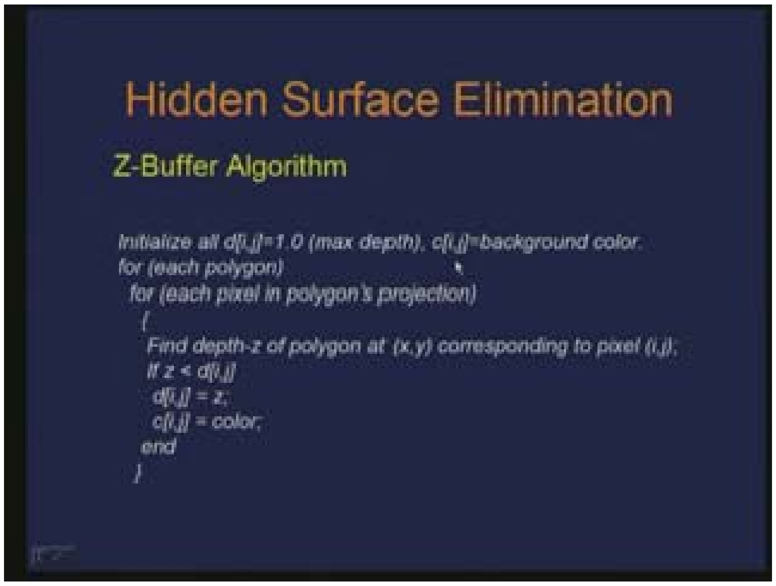
What happens here is that you consider a Z-buffer which is something like a frame buffer which maintains the depth of the current polygon or the primitive which is being drawn in that buffer. Basically it contains the information of depth. Sometimes it is also called as depth buffer instead of Z-buffer. Strictly speaking it may not be the z of the world you have but it is actually the depth depending on how you are viewing it. So if I have this buffer here for a given pixel located at (x, y) the Z-buffer value is there as Zb(x, y). And similarly you would have the information about color of the pixel in the frame buffer which is C(x, y) just as the frame buffer contains the color information you have this Z-buffer containing the depth information.

What is going to happen is you have a scene like this which is containing these polygons, triangles, this is the scene I have and this is what I have the viewing plane somewhere here so when I say that you maintain the information of depth for each pixel at (x, y) now what will happen is basically take a polygon and plot that, plot means perform the projection and do the plotting of that polygon and with respect to this pixel I tag the depth for the z. Then I go to the next polygon and do the plot of it and change the depth if required.

That means the depth which was stored if the currant polygon is less depth than that I change the depth I change the value in the buffer. So I update the value of the depth in the buffer and I move on to the other polygon and so on. So at the end of the day I have for the pixel (x, y) the depth corresponding to the polygon which needs to be plotted and simultaneously I am updating the frame buffer or the color buffer. So again it is a very simple way of handling intersections.

First of all we initialize this buffer which we call as d buffer which contains the depth to some maximum depth it could be 1 or it could be infinity depending on what range you are looking at. And the corresponding color buffer or the screen buffer you may assign to the back ground color.

(Refer Slide Time: 32:26)



Hidden Surface Elimination

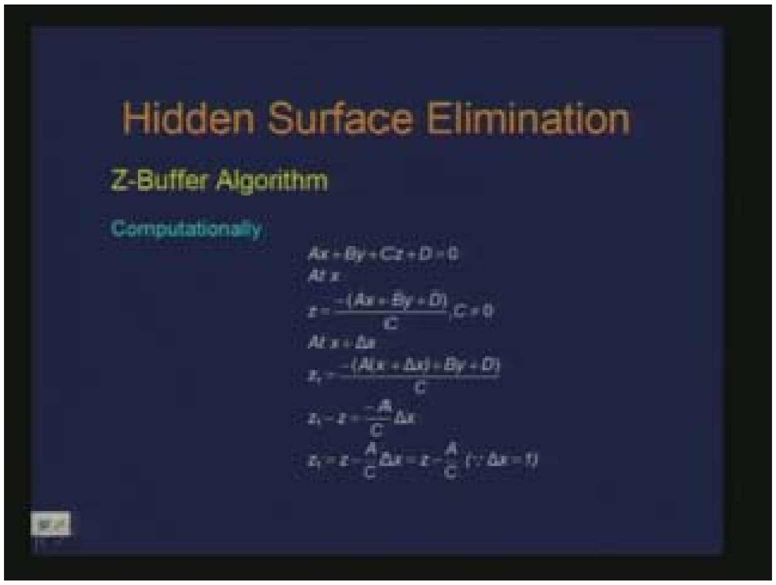
Z-Buffer Algorithm

```
Initialize all d[i,j]=1.0 (max depth), c[i,j]=background color.
for (each polygon)
 for (each pixel in polygon's projection)
  {
   Find depth-z of polygon at (x,y) corresponding to pixel (i,j);
   If z < d[i,j]
    d[i,j] = z;
    c[i,j] = color;
   end
  }
```

So nothing is found to be displayed and we display it with the background color. So what we are doing is basically we are plotting each polygon within each polygon for each pixel in polygons projection. That means we are doing a scan conversion of the polygon find depth z of polygon at (x, y) corresponding to pixel i, j) and if this z is less than the stored value of d[I, j] I update the depth buffer and the color buffer or the frame buffer.

So computationally what is happening? When we are talking about scan conversion of the polygon we still need to find out the Z at each point of the polygon. So let us try to see what is involved computationally? If I have this polygon defined as A_X plus B_Y plus C_Z plus D is equal to 0 at X so let us say I am talking about a scan at a given value of X.

(Refer Slide Time: 35:53)



Hidden Surface Elimination

Z-Buffer Algorithm

Computationally

$$Ax + By + Cz + D = 0$$

At x

$$z = \frac{-(Ax + By + D)}{C}, C \neq 0$$

At $x + \Delta x$

$$z_1 = \frac{-(A(x + \Delta x) + By + D)}{C}$$

$$z_1 - z = \frac{-A}{C}\Delta x$$

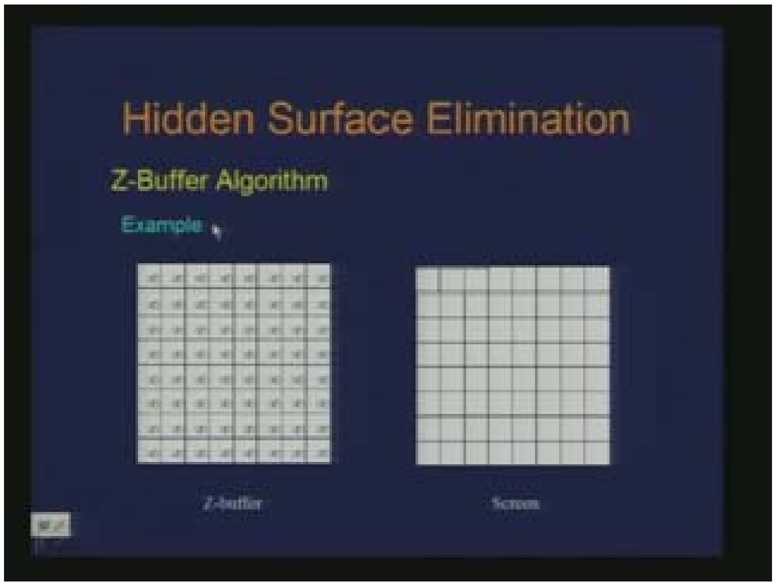
$$z_1 = z - \frac{A}{C}\Delta x = z - \frac{A}{C} \quad (\because \Delta x = 1)$$

So at given value of y I am looking at various axes just as the way we do scan conversion. So I compute the value of z of this x which is this and assume that C is not equal to 0.

Now what happens to the next point which needs to be drawn? At x plus delta x, I again compute the value of z which is z_1 just by substituting in place of x x plus delta x so this is what I get as z_1. Now if you look at the difference between z_1 and z the current z and previous z along this scanline is nothing but this all the other terms are canceled and the only term which is left is this.

So if I look at from the point of view of finding$_1$ from z which was previously computed it is just an incremental change to that out z z so z_1 is equal to z minus A by C times delta x and as we would go pixel by pixel. That means I am going to look at from the next pixel as to what happens then this delta x is nothing but unity so it turns out that the new z is nothing but the previous z minus A by C. So I can actually design an incremental algorithm for computing z. I do not have to explicitly do this every time so this is along the scanline.

(Refer Slide Time: 36:41)



Hidden Surface Elimination

Z-Buffer Algorithm

Example

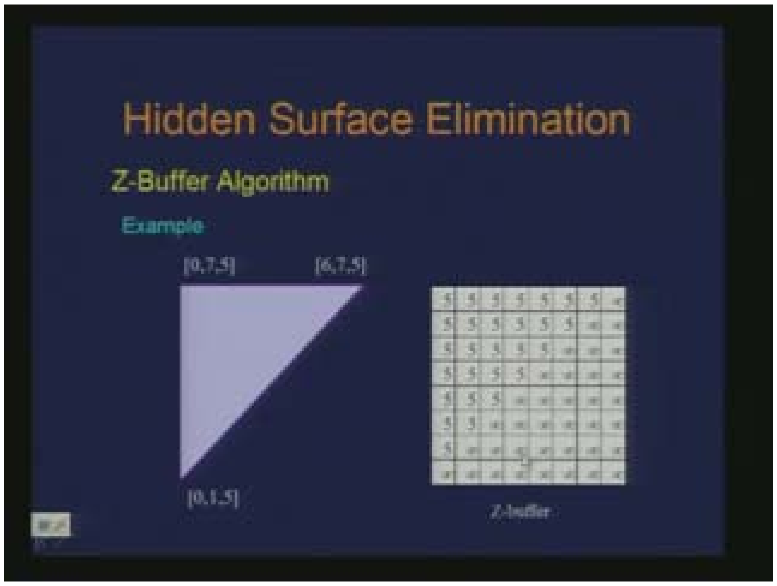
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

Z-buffer

Screen

Here is an example where I consider a buffer of the size 8 into 8 and I assign a maximum depth of infinity which is some big value to this Z-buffer. And correspondingly I have a screen or the display or the frame buffer which I am going to use for color which is this white color. So this is the initialization of my Z-buffer algorithm.

(Refer Slide Time: 37:39)



Hidden Surface Elimination

Z-Buffer Algorithm

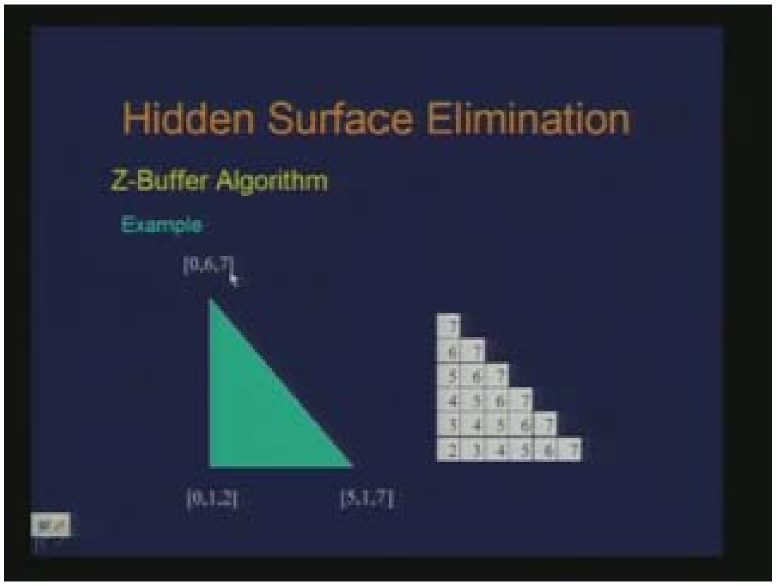
Example

5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
5	5	5	∞	∞	∞	∞	∞
5	5	∞	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

Z-buffer

Now I have this polygon which is to be redrawn. This is the polygon number one which is plotted. So what we notice here is that the z of these entire points is the same which is 5. So if I just I have to look at what happens to the corresponding Z-buffer these values become 5. So here the unity is a unit cell here. So all I am saying is that these are the updated z values in the buffer computed form here.

(Refer Slide Time: 38:45)



Hidden Surface Elimination

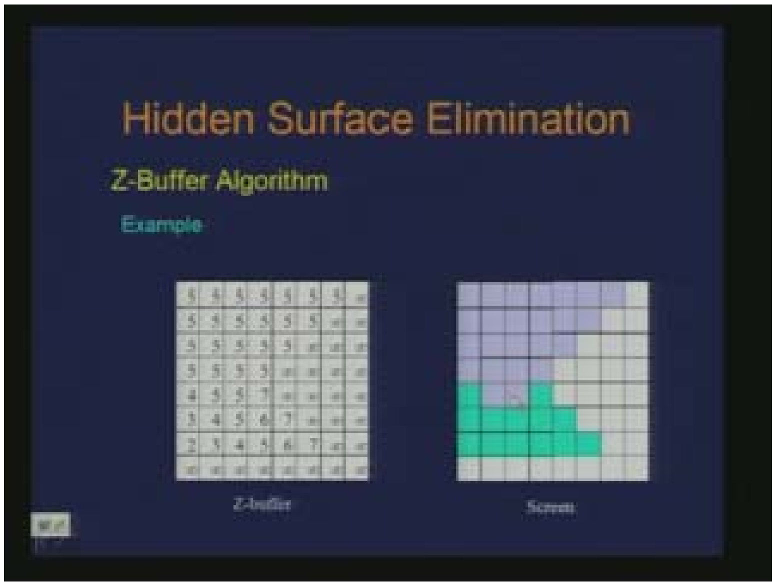
Z-Buffer Algorithm

Example

Now I have another polygon which is this which has got the points as (0, 6, 7), (5, 1, 7) here and (0, 1, 2) here so here we have different depth at this point. Now I need to do this computation of z for each of the points of the cell of the buffer.

So this would give me these as the z values so it is 2 3 4 5 6 7 here 3 4 5 6 7 here and so on. So clearly you see that this is referring to the depth of the diagram here. Now when I do the plot of this I need to update my Z-buffer. When I do that this is what happens. These are the updated values of Z-buffer. Hence, up to this nothing changes and these are the values which are changed. So when I do that this is what happens, these are the updated values of Z-buffer so up to this nothing changes and these are the values which are changed. So this is the current Z-buffer and accordingly I have this color buffer which contains the information of the color of the respective polygon. So notice here that this value and this value turn out to be same for both. And depending on what condition you put, I have basically put the condition strictly as less than and only then I update.

(Refer Slide Time: 40:14)



Hidden Surface Elimination

Z-Buffer Algorithm

Example

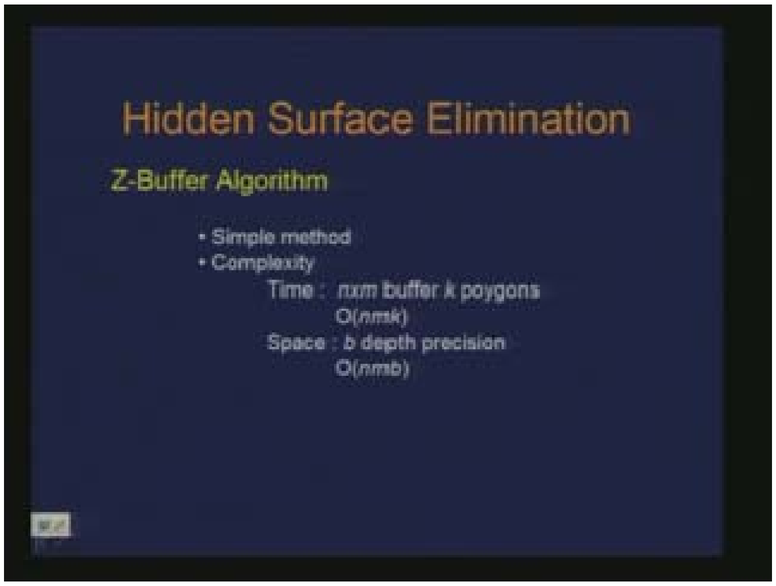
5	5	5	5	5	5	5	∞
5	5	5	5	5	5	∞	∞
5	5	5	5	5	∞	∞	∞
5	5	5	5	∞	∞	∞	∞
4	5	5	7	∞	∞	∞	∞
3	4	5	6	7	∞	∞	∞
2	3	4	5	6	7	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞

Z-buffer

Screen

So this would carry from this previous polygon and therefore the color of the previous polygon. So this is the way you can perform your Z-buffer algorithm, it is fairly simple.

(Refer Slide Time: 41:53)



Hidden Surface Elimination

Z-Buffer Algorithm

- Simple method
- Complexity

  Time : *n*x*m* buffer *k* poygons
  O(*nmk*)

  Space : *b* depth precision
  O(*nmb*)

Now if you just look at from the complexity point of view the complexity in terms of time, if I have the size of the buffer as n into n and then there are k polygons in the same then the order is O(nmk) where this is the size of the buffer and k is number of polygons. And if you just go back and look at the algorithm there is a for loop for polygons and then there is for loop for every pixel.

In terms of space if I consider b as the precision or the number of bits I assign for depth then it is simply n m times this length of the depth, this is the storage we require. Clearly this is very well supported in the hardware implementation. You are performing operations at each cell or pixel of the buffer and you can do a very fast hardware computation and that is why it is still very popular.

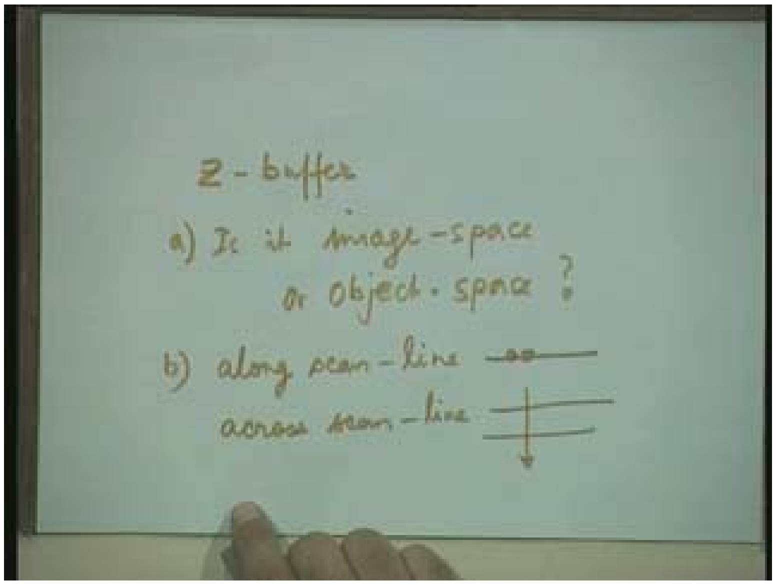
Do you see the use of Z-buffer kind of an algorithm somewhere which we have seen in ray tracing for finding out shadows? What is Z-buffer doing? The Z-buffer is basically maintaining depth. So the question is you need to figure out the shadows in ray tracing of may be otherwise also. You may not be using ray tracing to figure out shadows but ray tracing is one of the ways. So what you do is you basically shoot a ray from the object where the hit point is asked is that are there any similarities in terms of computing the shadow and doing Z-buffer.

Conceptually it is exactly the same because you are replacing the viewer by the light source and you could still maintain a buffer which is sometimes called as shadow buffer instead of Z-buffer for finding out whether a particular point is in shadow or not, so that again requires a pre processing. What you are trying to do is you may actually put some sort of a grid on top of the light source which is a buffer and when you shoot rays with respect to each of the cell in that grid you can figure out whether this cell causes a shadow or not and that is what you are trying to address. So, again you can use similar mechanism like a Z-buffer to form a shadow buffer.

We are going to look at some other hidden surface elimination method. So as supposed in floating horizon we actually went from front to back but a natural thing to do is back to front. We keep drawing those things which come in the front.

Quiz questions:
We have the Z-buffer.
a) Is it image space or object space?
b) We have seen computationally what happens along scanline. So along scanline we have an incremental algorithm. What happens across scanline that means from one scanline to another scanline. This is what I was referring to from here to here and here I am referring to here to here. Does there an incremental algorithm exist?

(Refer Slide Time: 51:02)


z - buffer

a) Is it image-space or object-space ?

b) along scan-line

across scan-line

The b part is that there is that incremental computation along this scanline so I can compute z_1 in terms of the previous z computed just by changing incrementally. The question is can I do this also from one scanline to another scanline.