

# Object Representation

# 3D Object Representation

- So far we have used the notion of expressing 3D data as points(or vertices) in a Cartesian or Homogeneous coordinate system.
- We have simplified the representation of objects as a series of line segments representing the edges between vertices on a polygon object.
- Differing render engines use differing types of methods of representing 3D objects in CG applications:
  - Fundamentally, most render engines reduce surfaces to polygon meshes.
  - Some surface shading algorithms require geometry in the execution.
  - The resolution of these meshes depends on the renderer and the implementation of the surface in the renderer.

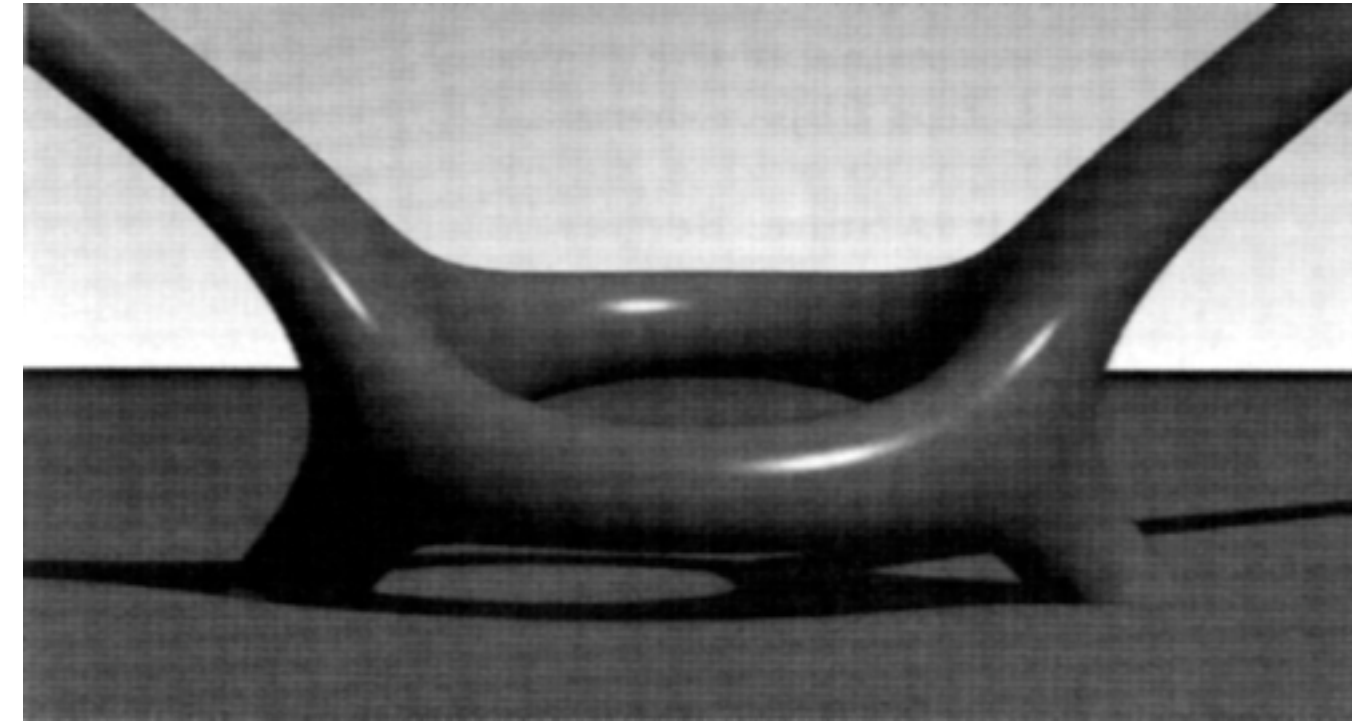
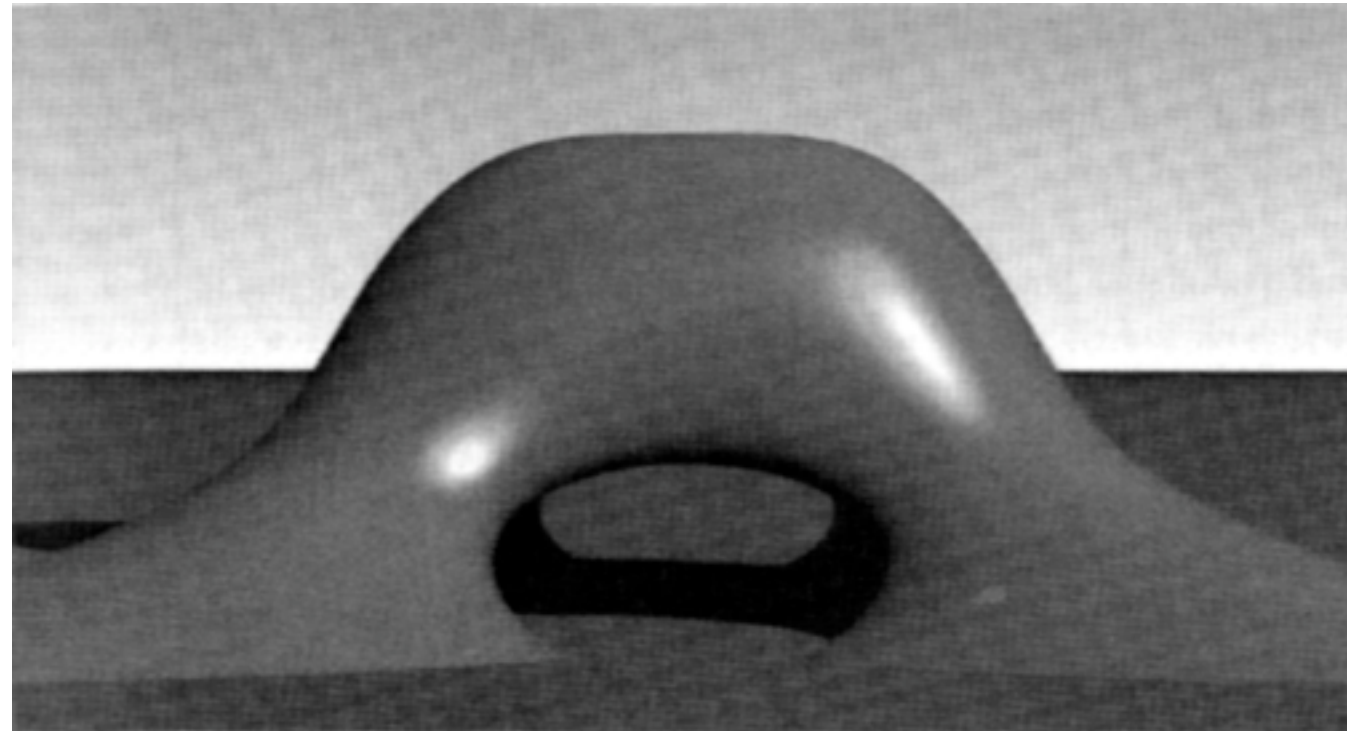
# Types of Surface

- Computer graphics applications deal with the notion of approximating the surface.
- We have already covered Polygonal surfaces in some depth.

# CSG

- Constructive Solid Geometry popular in CAD packages to obtain a “solid” representation.
  - CSG system uses a succession of Boolean operations on
  - primitive objects to gradually build up geometry
  - not as tangible to the user.
- Some applications allow the use of CSG.
- Current systems adopt less than solid approaches
  - only object surfaces are represented.

# Mathematical Representation



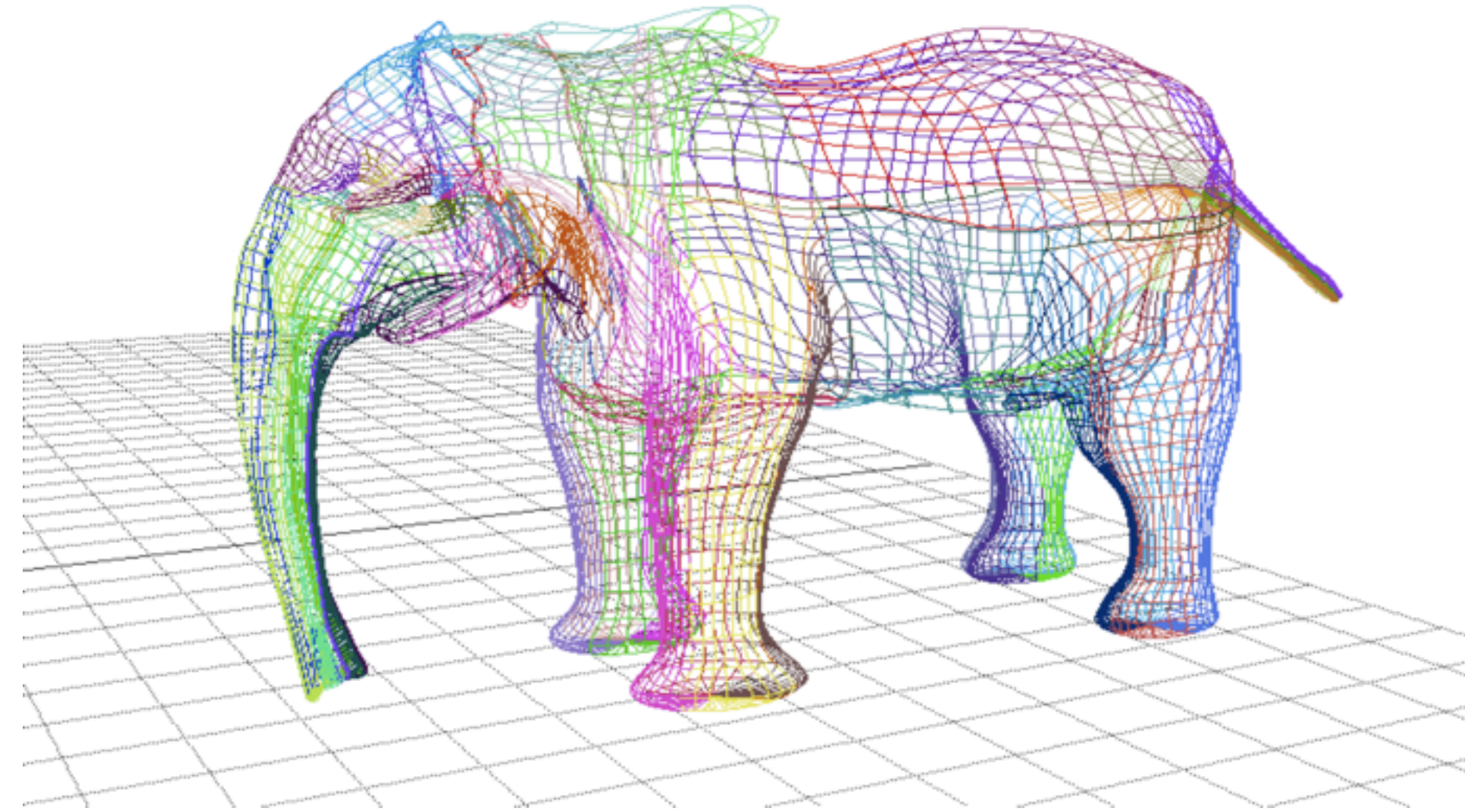
- Mathematical equations can be used directly to create object surfaces.
- The above examples are the visualisation of different polynomial equations.
- It would be hard though to explicitly describe objects in this manner to incorporate all of the detail required that everyday objects have.
- Some applications allow these to be used in conjunction with CSG

# Parametric Representation

- Parametric representation is an approximation of a surface using curves to generate patches.
- By manipulating the curves and therefore the patches, explicit objects can be formed.
- By separating the representation of an object into many different patches, a surface can be described parametrically.

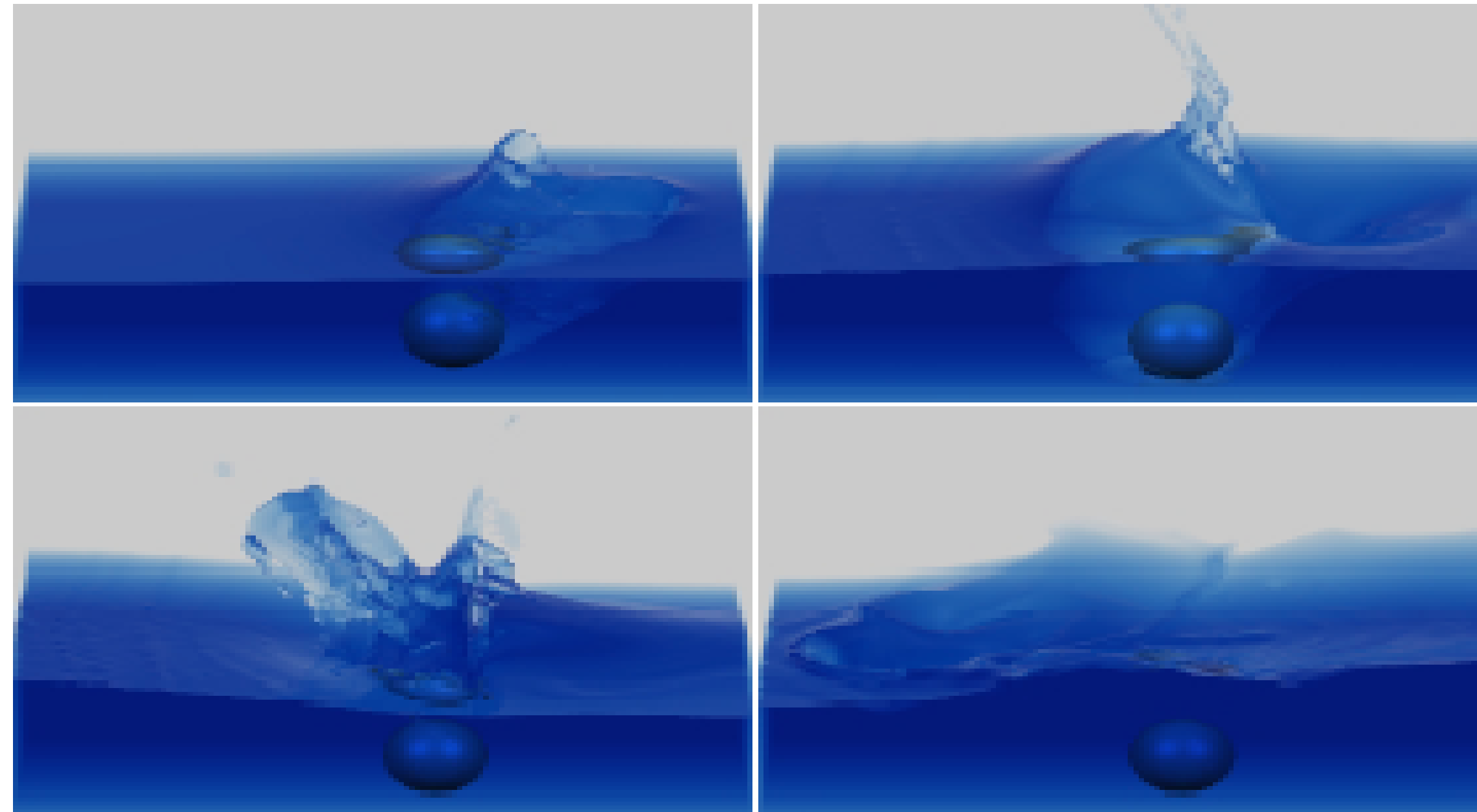


# Parametric Representation



- Although patches present an alternate method for surface representation most computer graphics applications convert patch based objects to polygons prior to rendering.

# Rendering Approaches



- Implicit surfaces are utilised where explicit definitions cannot be practically used.
- E.g. in gaseous elements, Clouds and fluids.
- Methods such as Eulerian grids, voxels, blobbies, particles, volumetric renderers etc.

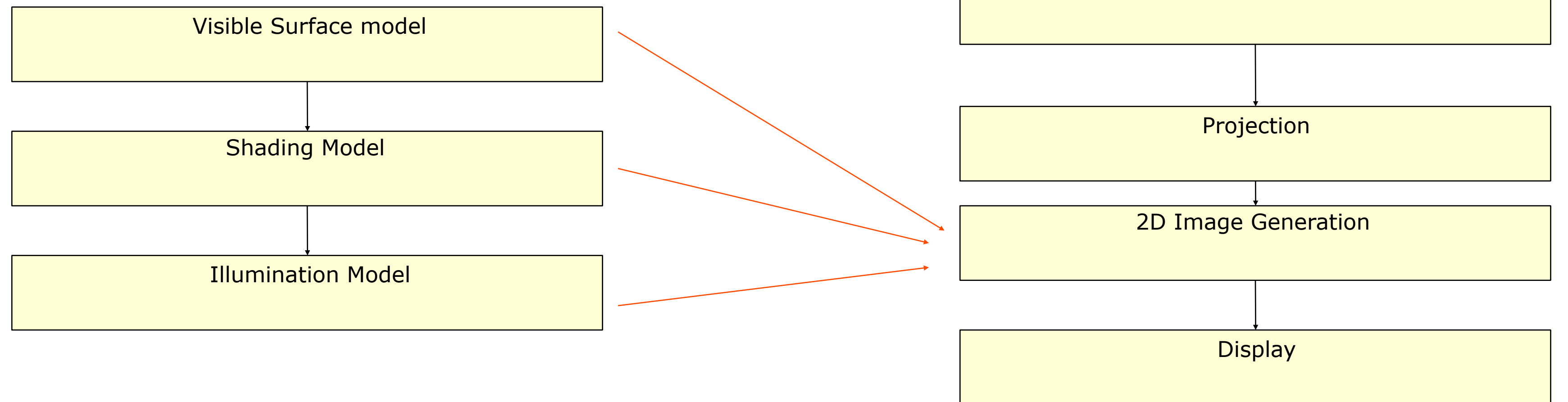


# Illumination Models

- Most CG algorithms are optimised for polygon data (ideally planar triangles).
- Geometric shapes can be manipulated using the notion that 2 vertices form a line segment – e.g. line clipping algorithms.
- The approximation of meshes varies:
  - screen resolution is important
- If the polygon samples (micro polygons) are smaller or the same size as the pixel area on the viewing plane, then the surface is described in a manner such that appears as a perfect representation at that given resolution.
- The user usually has some control over surface approximation, more in cases where the surface is not a primitive supported by the renderer.
- The usage of the mesh is important and the approximation may vary over distance from the camera: e.g. a surface used to describe an ocean.

# Visualising Objects

Before we tackle the issue of *image generation*, we must concern ourselves with the information required for such processes.

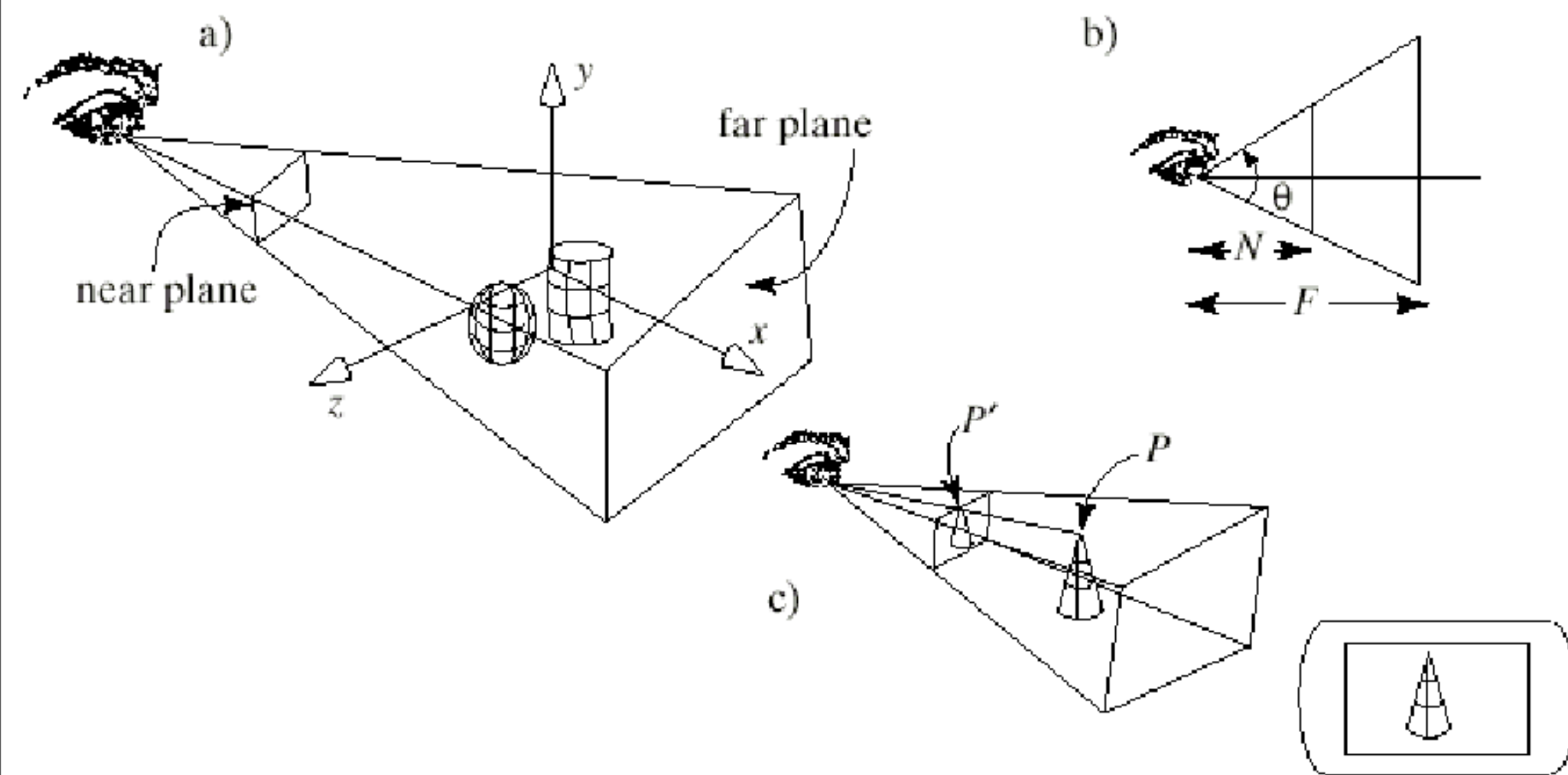


# Image Synthesis



- The process of image synthesis is the part of the pipe line that finally turns our object information into actual imagery.
- There are 3 main aspects of this process:
  1. Rasterisation : mapping the polygons onto the pixels of the image.
  2. Hidden Surface Removal:the elimination of polygons not seen and polygons that are occluded in the output view.
  3. Shading the polygons be it from flat shading to full blown illumination modelling.

# The Virtual Camera



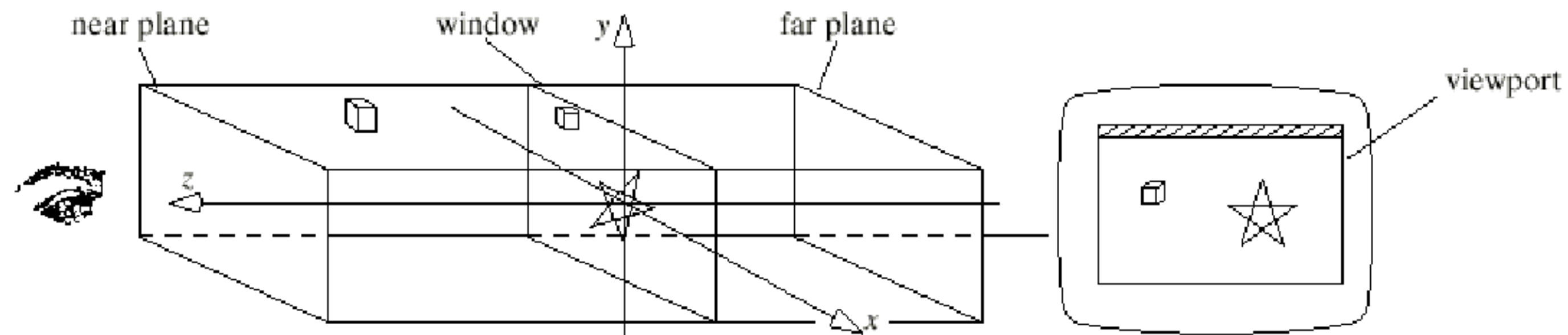
- The Camera is defined by using a parallelepiped as a view volume with two of the walls used as the near and far view planes
- Typically a view volume is also created using an angle, the screen dimensions and near and far planes

# Drawing 3D scenes

- When drawing in 3D the main emphasis is on transforming objects in order to orient and position them as a desired 3D scene
- To do this we use matrices and transform all our vertices by these matrices
- Typically each object will also have its own transformation matrix to do local transformations.
- When using realtime systems (like our application viewports) each of the vertices of our model are processed through a series of these matrices, usually called the Model, View Project (MVP) matrices.
- Renderers usually have a slightly different way of doing this but the principles will be the same



# Overview of the Viewing process

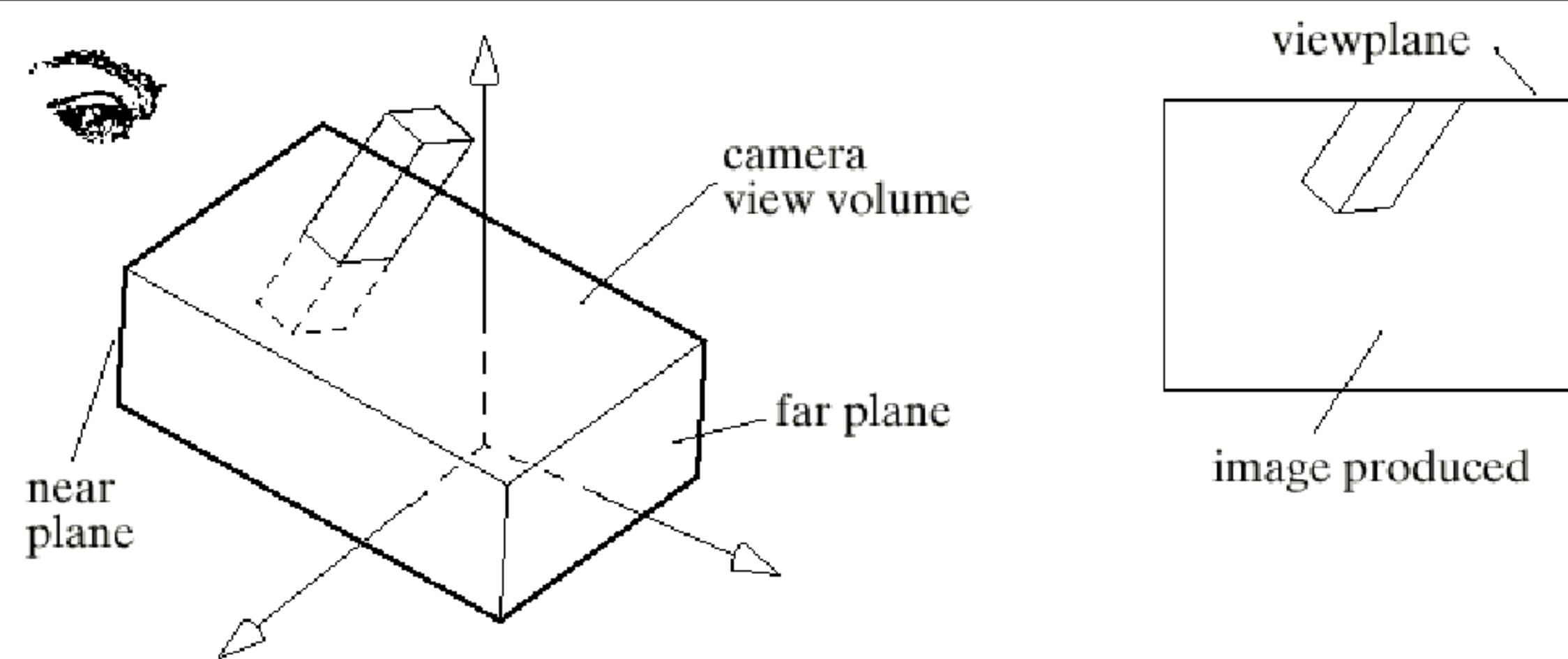


- When using “orthographic” views (such as side, top etc in our apps) we are actually using a special case of 3D projection.
- The view volume of the camera is a rectangular parallelepiped, whose four sidewalls are determined by the border of the window and whose other two walls are determined by a near plane and a far plane

# Overview of the Viewing process II

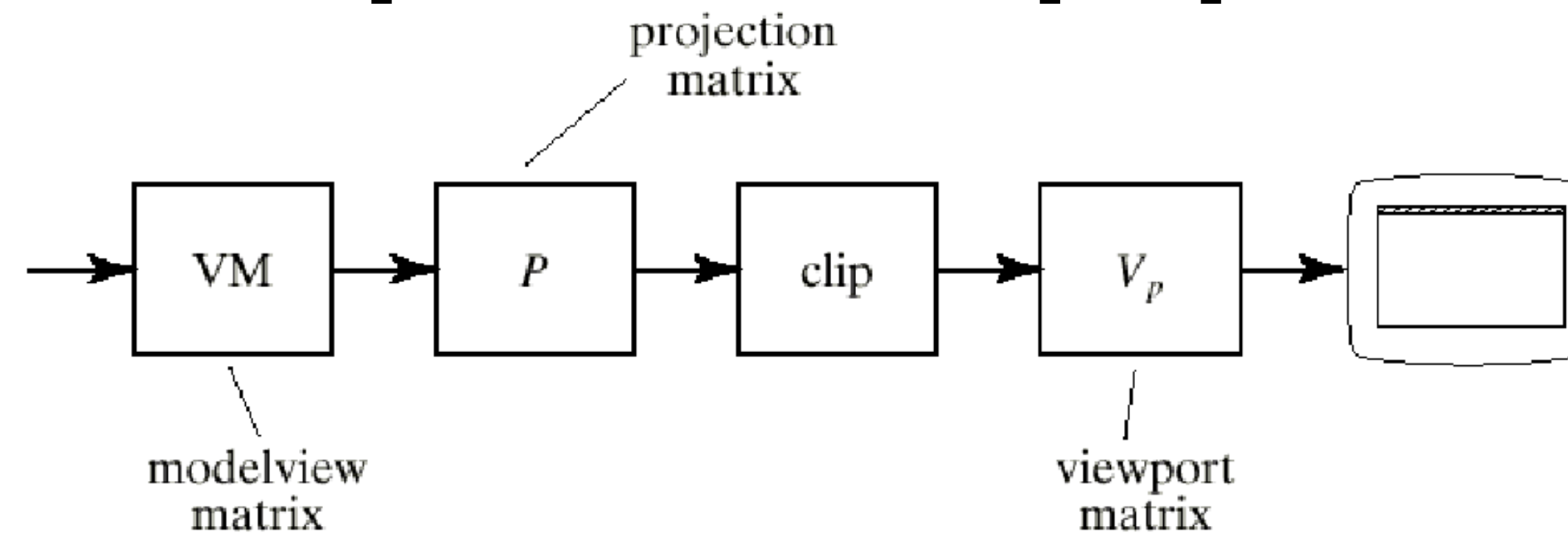
- Points lying inside the view volume are projected onto the window along lines parallel to the z axis.
- This is equivalent to simply ignoring the z components of these points so that the 3D point  $(x_1, y_1, z_1)$  projects to  $(x_1, y_1, 0)$
- A separate viewport transformation maps the projected points from the window to the viewport on the display device

# 3D parallel projection



- The image above shows a camera immersed in a scene, the scene consists of a block, part of which lies outside the view volume
- Most applications will then process all of the vertices by the following matrices before rendering.
  - The modelview matrix
  - The projection matrix
  - The viewport matrix

# OpenGL pipeline



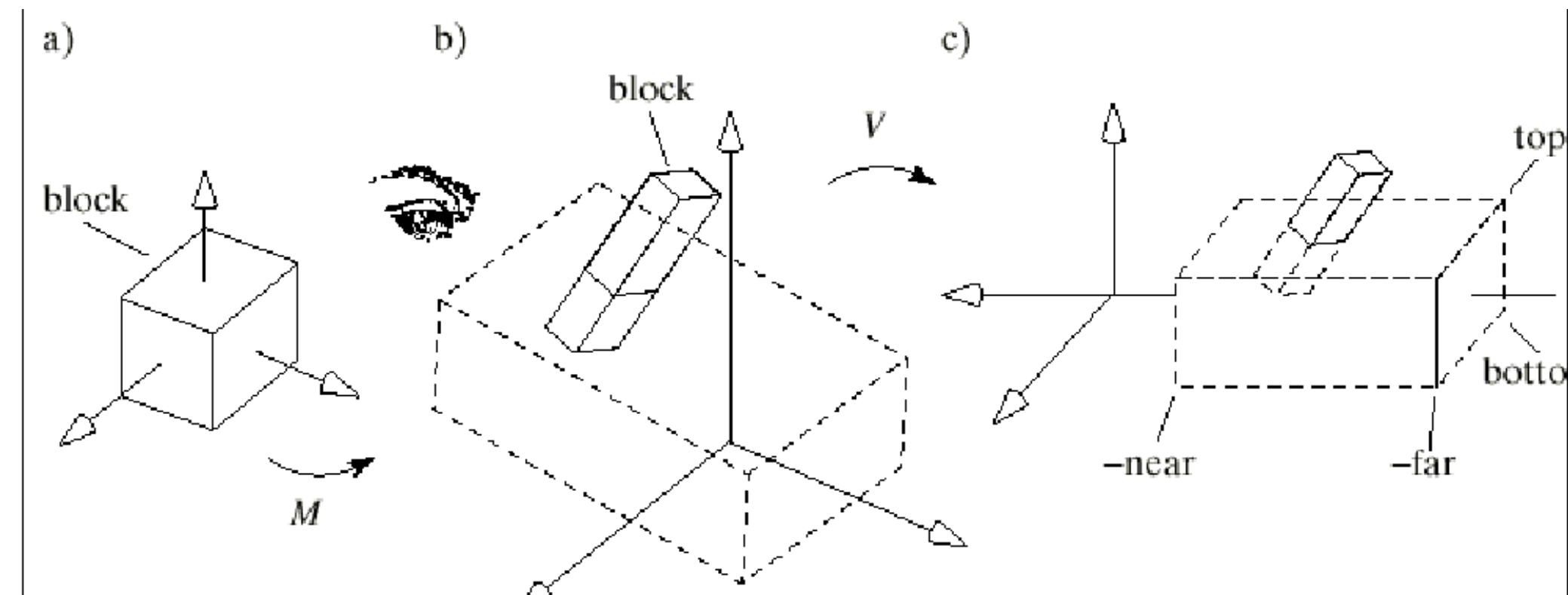
- Each object is made up of a number of vertices, each vertex can be added to a scene by the graphics calls (OpenGL for Maya and Houdini)
- Each vertex is multiplied by the various matrices shown, it is then clipped if necessary
- Any parts that survive clipping are then mapped onto a viewport

# The modelview matrix I

- The modelview matrix provides what we call the CT (Current Transform).
- It combines two effects
  - the sequence of modelling transformations applied to objects
  - and the transformation that orients and positions the camera in space
- Although the model view matrix is a single matrix in the actual pipeline it is easier to think of it as the product of two matrices :-
  - a modeling matrix  $M$
  - a viewing matrix  $V$
- First the modeling matrix is applied and then the viewing matrix so the modelview matrix is in fact the product  $VM$



# The modelview matrix II

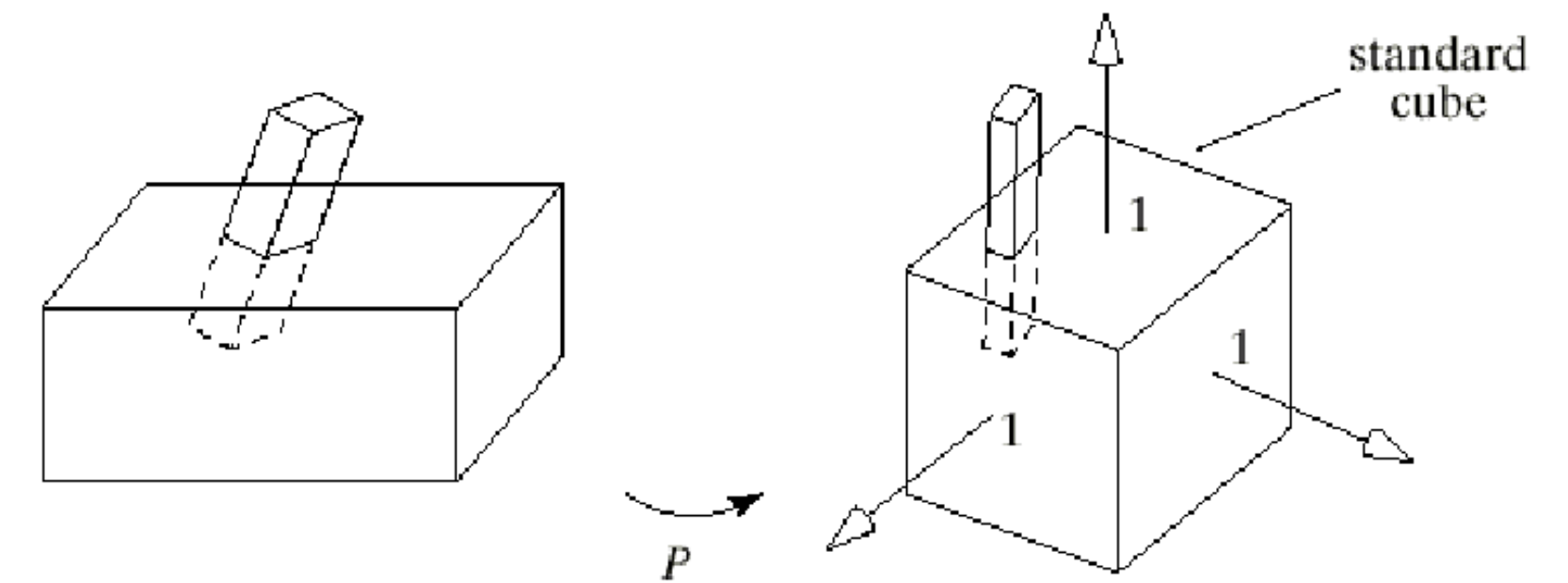


- a) shows a unit cube centred at the origin,
- A modelling transformation based on  $M$  scales, rotates and translates the cube into the block shown in part b)
- part b) also shows the relative position of the camera view volume
- The  $V$  matrix is now used to rotate and translate the block into a new position.
- The specific transformation is that which would carry the camera from its position in the scene to its “generic” position, with the eye at the origin and the view volume aligned with the z-axis as shown in part c)

# The modelview matrix III

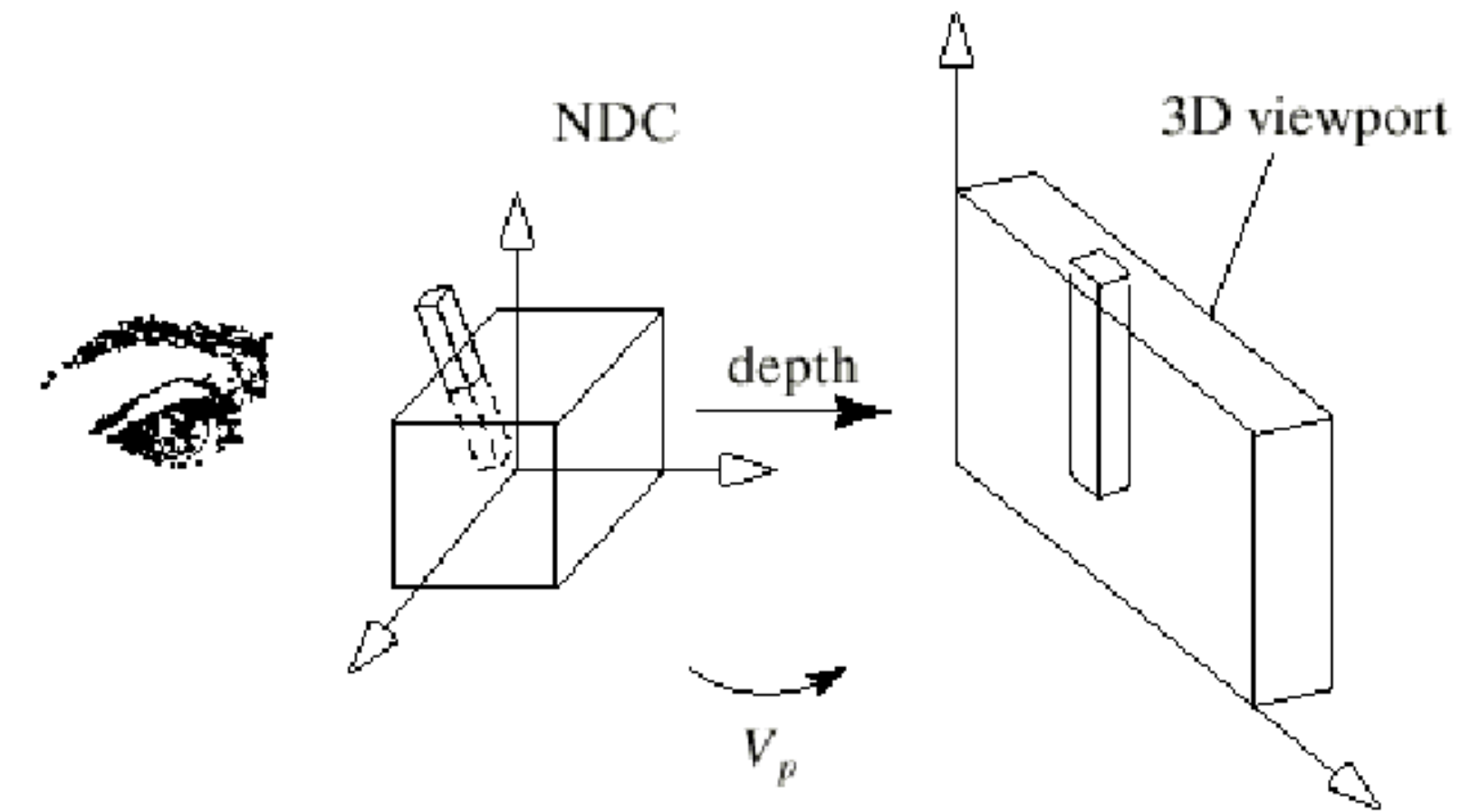
- The vertices of the block are now positioned so that projecting them onto a plane such as the near plane yields the proper values for displaying the projected image.
- So the matrix  $V$  in fact effects a change of coordinates of the scene's vertices into the camera's coordinate system (eye coordinates)
- In the camera's coordinate system the edges of the view volume are parallel to the  $x$ ,  $y$  and  $z$  axis
- The view volume extends from left to right in  $x$  from bottom to top in  $y$  and from -near to -far in  $z$
- When the vertices of the original cube have passed through the entire modelview matrix they are located as show in part c)

# The projection matrix



- The projection matrix scales and shifts each vertex in a particular way, so that all those vertices that lie inside the view volume will lie inside a standard cube that extends from  $-1$  to  $1$  in each dimension
- This matrix effectively squashes the view volume into the cube centered at the origin, which is a particularly efficient boundary to clip against
- Scaling the block in this fashion might badly distort it but this distortion can be compensated for in the viewport transformation
- The projection matrix also reverses the sense of the  $z$ -axis so that an increase in the value of  $z$  now represents increasing values of the depth of a point from the eye.
- clipping is now performed

# The viewport matrix



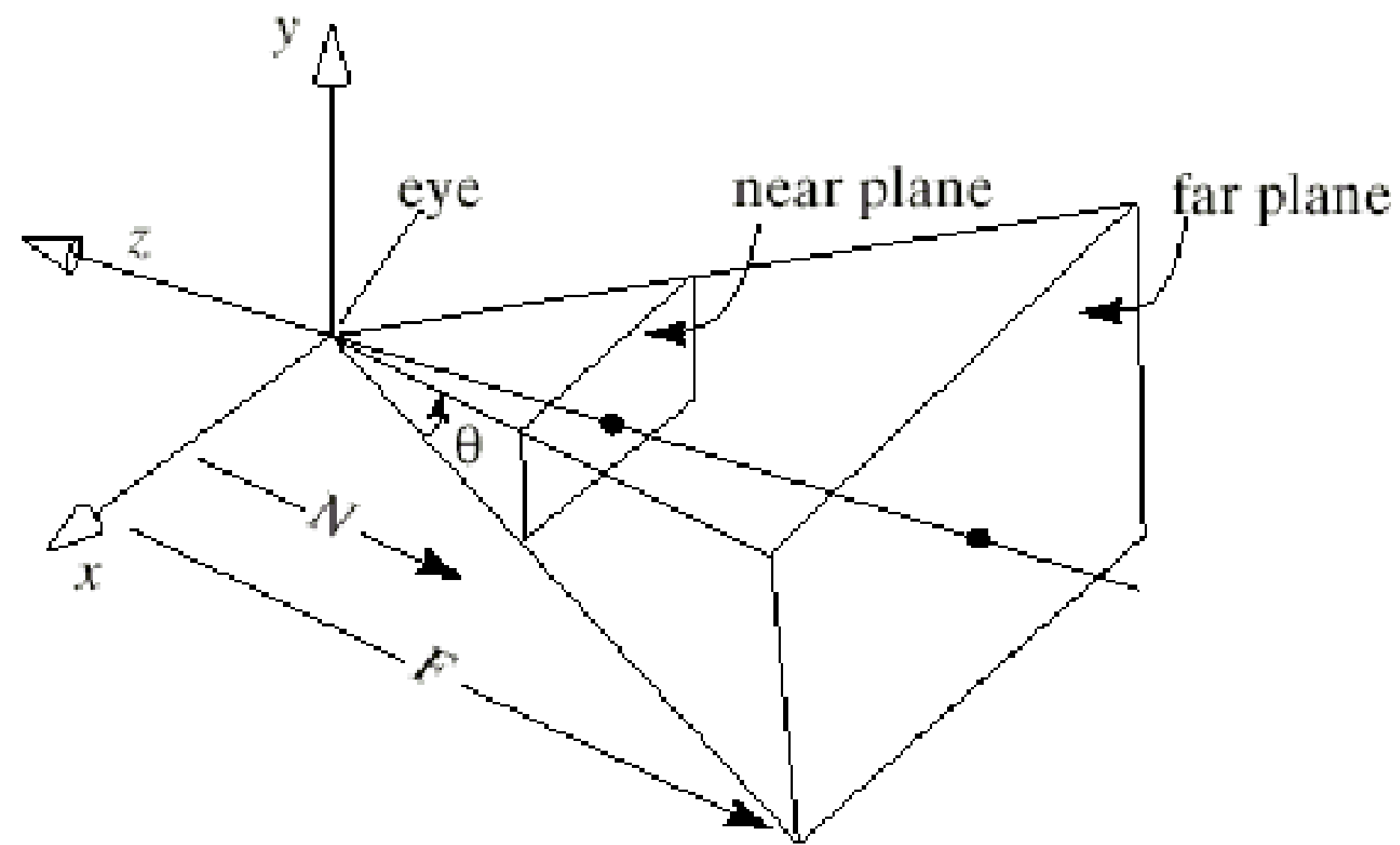
- The viewport matrix takes the surviving elements of the block and maps it to a 3D viewport
- The matrix maps the standard cube into a block shape whose x and y values extend across the viewport (in screen co-ordinates)
- and a z component which extends from 0 to 1

# Perspective Camera

- The Camera has an eye positioned at some point in space.
- Its view volume is a portion of a rectangular pyramid, whose apex is at the eye.
- The opening of the pyramid is set by the viewangle  $\theta$  (part b of figure)
- Two planes are defined perpendicular to the axis of the pyramid : the near and the far plane.
- Where these planes intersect the pyramid they form rectangular windows which have an adjustable aspect ratio.
- We then clip points which lie outside the view volume. Points lying inside the the view volume are projected onto the viewplane to a corresponding point  $P'$
- With a perspective projection the point  $P'$  is determined by finding where a line from the eye to  $P$  intersects the viewplane.




# Setting the View Volume



- Most applications have parameters for setting the realtime camera (not the renderable one)
- This lets you set clipping planes for realtime visualisation as well as the FOV
- Sometimes this will also transfer to the renderable camera but this depends upon the renderer

# Setting The Perspective

- The parameter  $fov$ , shown as  in the figure is given in degrees and sets the angle between the top and bottom walls of the pyramid.
- The parameters  $w$  and  $h$  sets the aspect ratio of any window parallel to the  $xy$ -plane
- The value  $N$  is the distance from the eye to the near plane, and  $F$  is the distance from the eye to the far plane.  $N$  and  $F$  should be positive.

# Perspective Projection

$$f = \cot\left(\frac{fovy}{2}\right)$$

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Projections is calculated using a single matrix
- For example a simple Perspective projection can be generated by using the matrix on the right
- This can be used to generate all different types of perspective and effects, especially in games etc

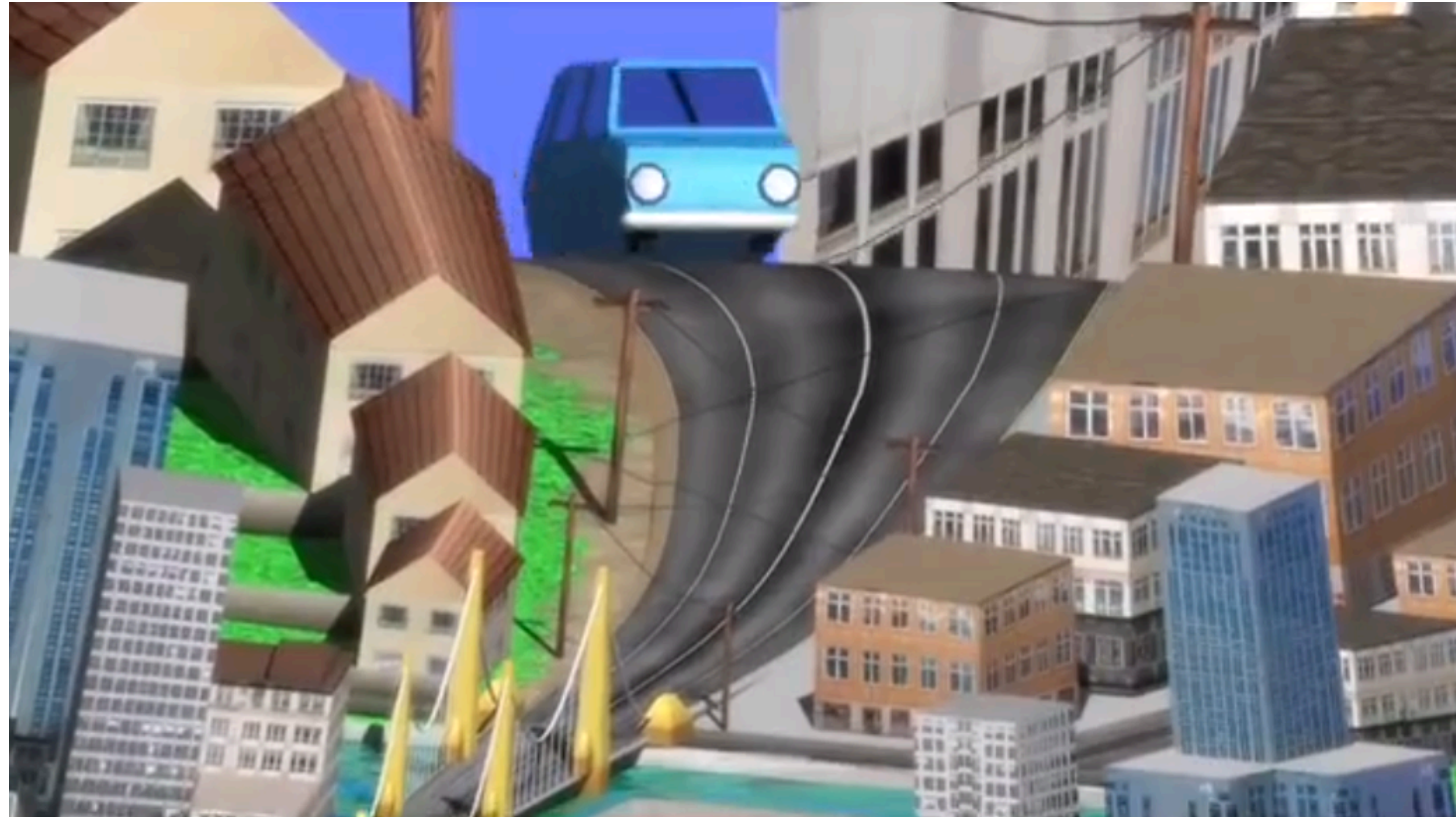
# Orthographic Projection

Scale

$$\begin{bmatrix} \frac{2.0}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2.0}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2.0}{f-n} & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & -\frac{f+n}{f-n} & 1 \end{bmatrix}$$

Translate

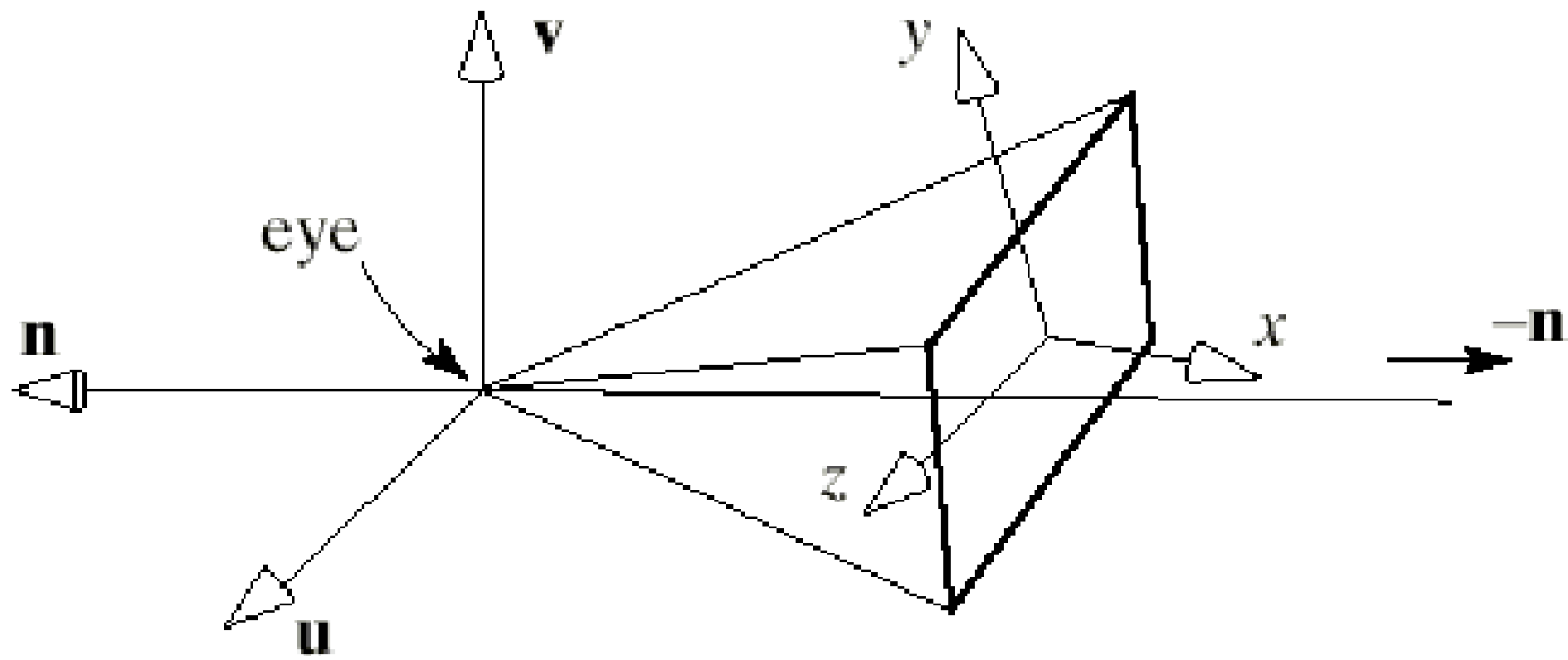
# Reverse Perspective



<http://www.youtube.com/watch?v=jdMSEJwbCCY>

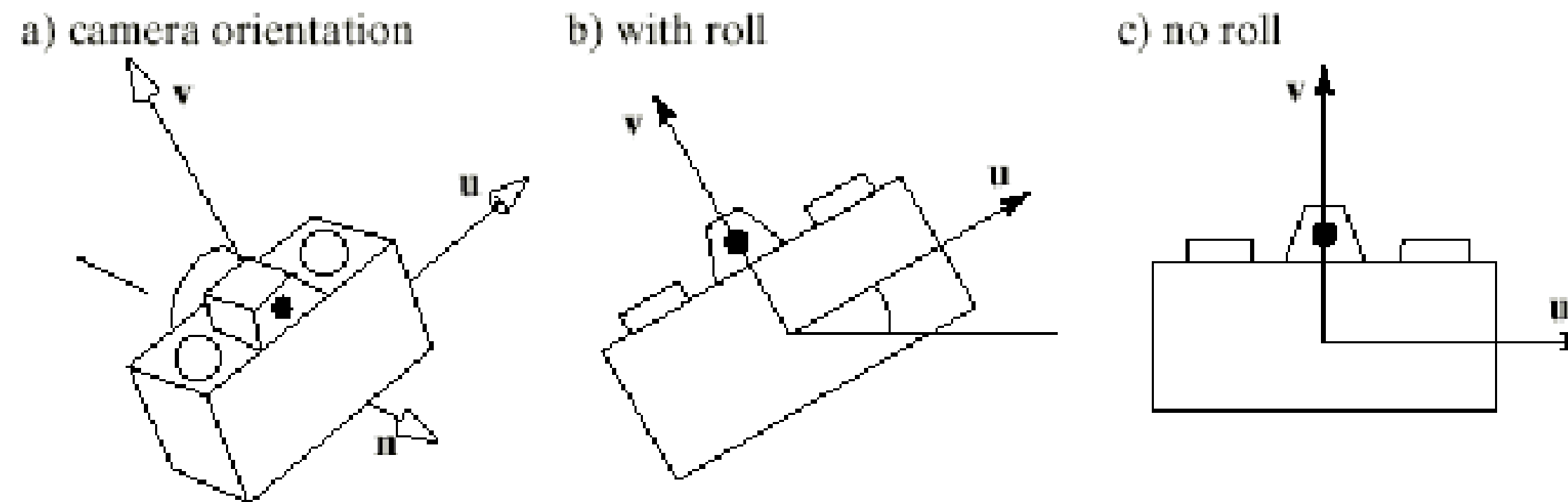


# The General Camera with Arbitrary Orientation and Position



- It is useful to attach an explicit co-ordinate system to the camera as shown in the figure above.
- This co-ordinate system has its origin at the eye and has three axes, usually called the  $u$ -,  $v$ -, and  $n$ -axis which define the orientation of the camera
- The axes are pointed in directions given by the vectors  $\mathbf{u}$ ,  $\mathbf{v}$ , and  $\mathbf{n}$
- Because, by default, the camera looks down the negative  $z$ -axis, we say in general that the camera looks down the negative  $n$ -axis in the direction  $-\mathbf{n}$
- The direction  $\mathbf{u}$  points off “to the right of” the camera and the direction  $\mathbf{v}$  points “upward”

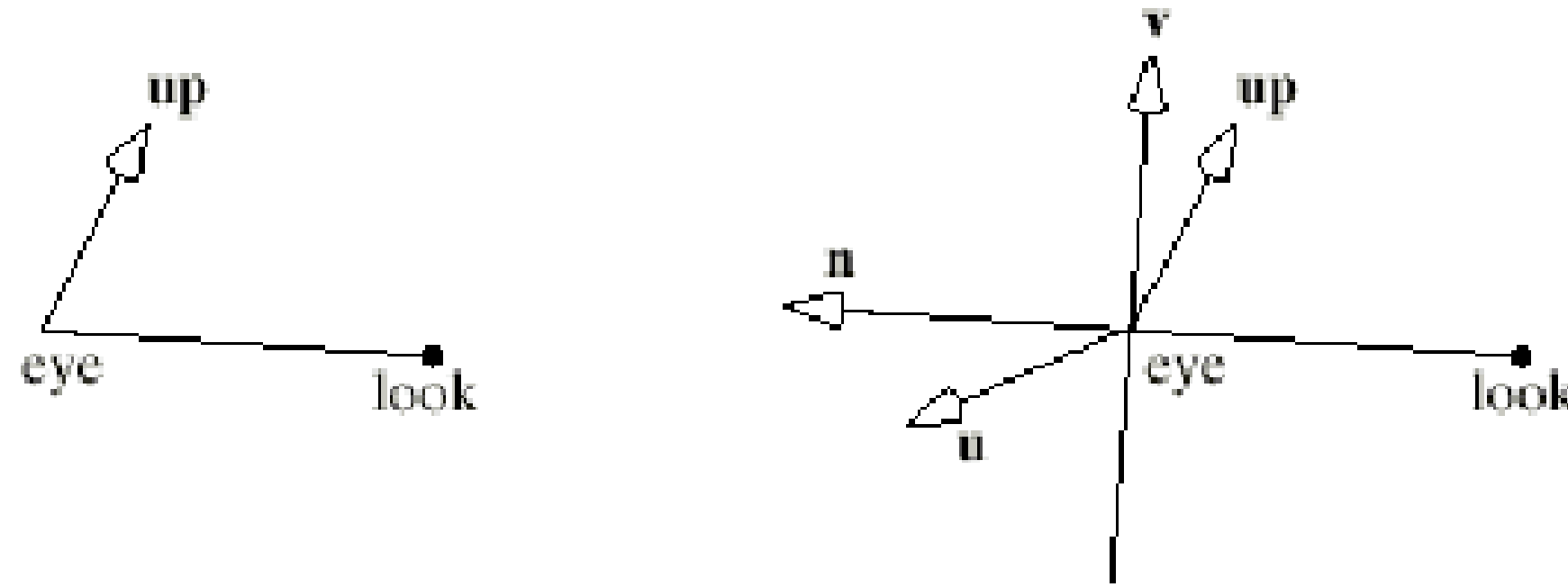
# Camera



- The above figure shows a camera with the same co-ordinate system attached to it
- It has **u-**,**v-** and **n-**axes and an origin at position eye
- b) shows the camera with a roll applied to it
- c) shows the camera with zero roll or “no-roll” camera
- The u-axis of a no-roll camera is horizontal, that is perpendicular to the y-axis of the “world”
- Note that a no-roll camera can still have an arbitrary **n** direction, so it can have any pitch or heading.



# Calculating the View Matrix



- What are the directions of  $\mathbf{u}$ ,  $\mathbf{v}$  and  $\mathbf{n}$  vectors with given values for eye, look and up
- If given the locations of eye, look and up, we immediately know that  $\mathbf{n}$  must be parallel to the vector **eye-look**, as shown above. so we can set  **$\mathbf{n} = \text{eye} - \text{look}$**
- We now need to find a  $\mathbf{u}$  and a  $\mathbf{v}$  that are perpendicular to  $\mathbf{n}$  and to each other.
- The  $\mathbf{u}$  direction points “off to the side” of a camera, so it is natural to make it perpendicular to **up** which the user has said is the “upward” direction.

# Calculating the View Matrix

- An easy way to build a vector that is perpendicular to two given vector is to form their cross product, so we set  $\mathbf{u} = \mathbf{up} \times \mathbf{n}$
- The user should not choose an  $\mathbf{up}$  direction that is parallel to  $\mathbf{n}$ , because  $\mathbf{u}$  then would have zero length.
- We choose  $\mathbf{u} = \mathbf{up} \times \mathbf{n}$  rather than  $\mathbf{n} \times \mathbf{up}$  so that  $\mathbf{u}$  will point “to the right” as we look along  $-\mathbf{n}$
- With  $\mathbf{u}$  and  $\mathbf{n}$  formed it is easy to determine  $\mathbf{v}$  as it must be perpendicular to both and is thus the cross product of  $\mathbf{u}$  and  $\mathbf{n}$  thus  $\mathbf{v} = \mathbf{n} \times \mathbf{u}$
- Notice that  $\mathbf{v}$  will usually not be aligned with  $\mathbf{up}$  as  $\mathbf{v}$  must be aimed perpendicular to  $\mathbf{n}$  whereas the user provides  $\mathbf{up}$  as a suggestion of “upwardness” and the only property of  $\mathbf{up}$  that is used is its cross product with  $\mathbf{n}$

# Calculating the View Matrix

- To summarise, given eye look and up, we form

$$\mathbf{n} = \mathit{eye} - \mathit{look}$$

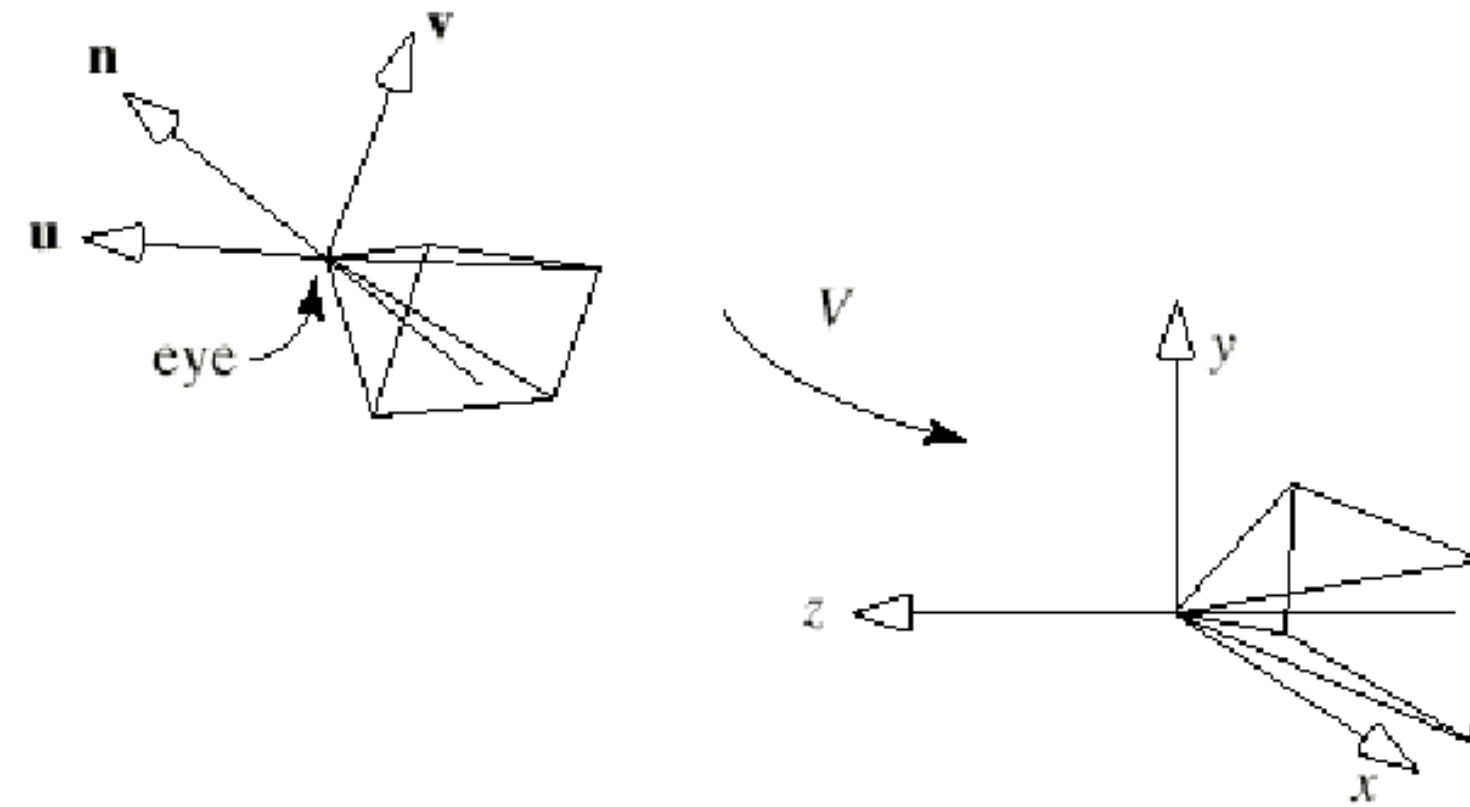
$$\mathbf{u} = \mathbf{up} \times \mathbf{n}$$

and

$$\mathbf{v} = \mathbf{n} \times \mathbf{u}$$

and then normalise the vectors to unit length.

# ModelView Matrix



- The ModelView is the product of two matrices the matrix  $\mathbf{V}$  that accounts for the transformation of the world point into camera coordinates.
- and  $\mathbf{M}$  that embodies all of the modelling transformations applied to the points.
- We can now multiply the two and generate our points
- Because the job of the  $\mathbf{V}$  matrix is to convert world co-ordinates to camera co-ordinates it must transform the camera's coordinate system into the generic position for the camera as shown in the figure.

# ModelView Matrix

- This means that  $V$  must transform  $eye$  into the origin,  $\mathbf{u}$  into the vector  $\mathbf{i}$ ,  $\mathbf{v}$  into  $\mathbf{j}$ , and  $\mathbf{n}$   $\mathbf{k}$
- The easiest way to define  $V$  is to use the following matrix

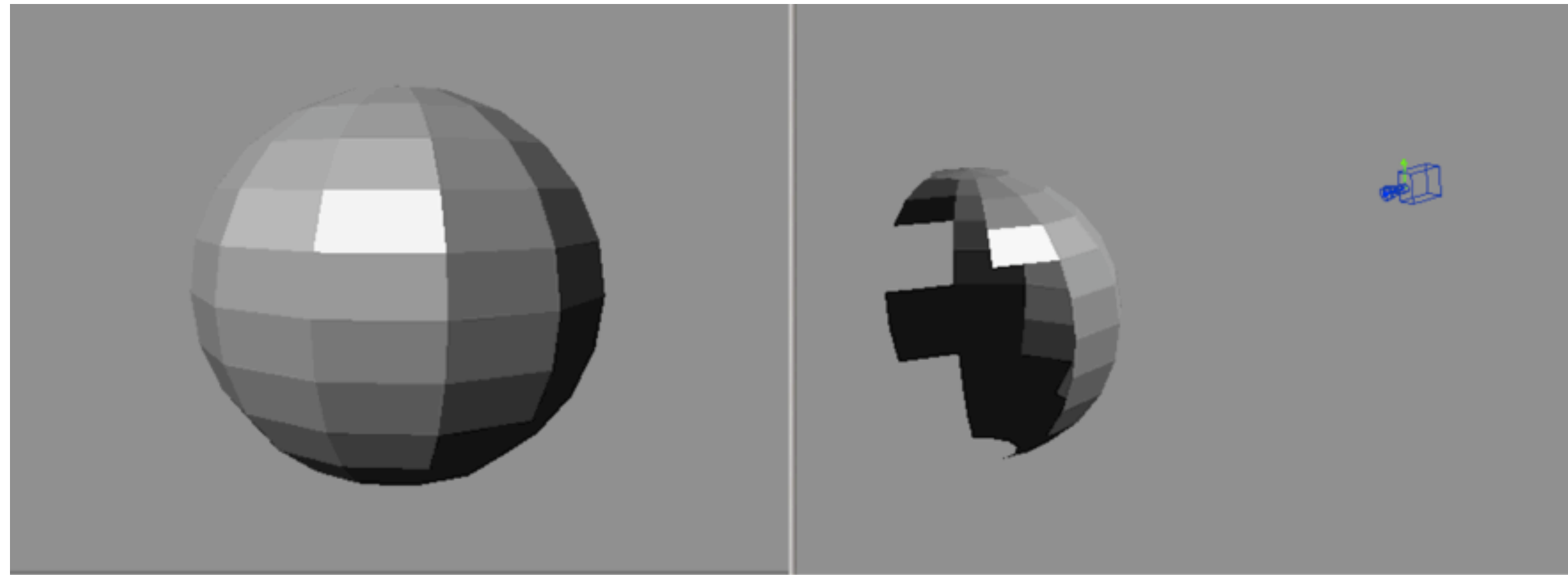
$$V = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

Where  $(d_x, d_y, d_z) = (-eye \bullet \mathbf{u}, -eye \bullet \mathbf{v}, -eye \bullet \mathbf{n})$

# Visible Surface Algorithms

- Visible surface algorithms (or hidden surface elimination algorithms) basically do one thing:
- They calculate what is visible to the viewer and what is not.
- This process is very computationally expensive.
- The process has the potential to be called for every single pixel in an image.
- For example, for a standard PAL sized raster image, there are 720 x 576 pixels.
- This results in a staggering 414,720 pixels.
- Efficiency is paramount – especially in real-time applications.

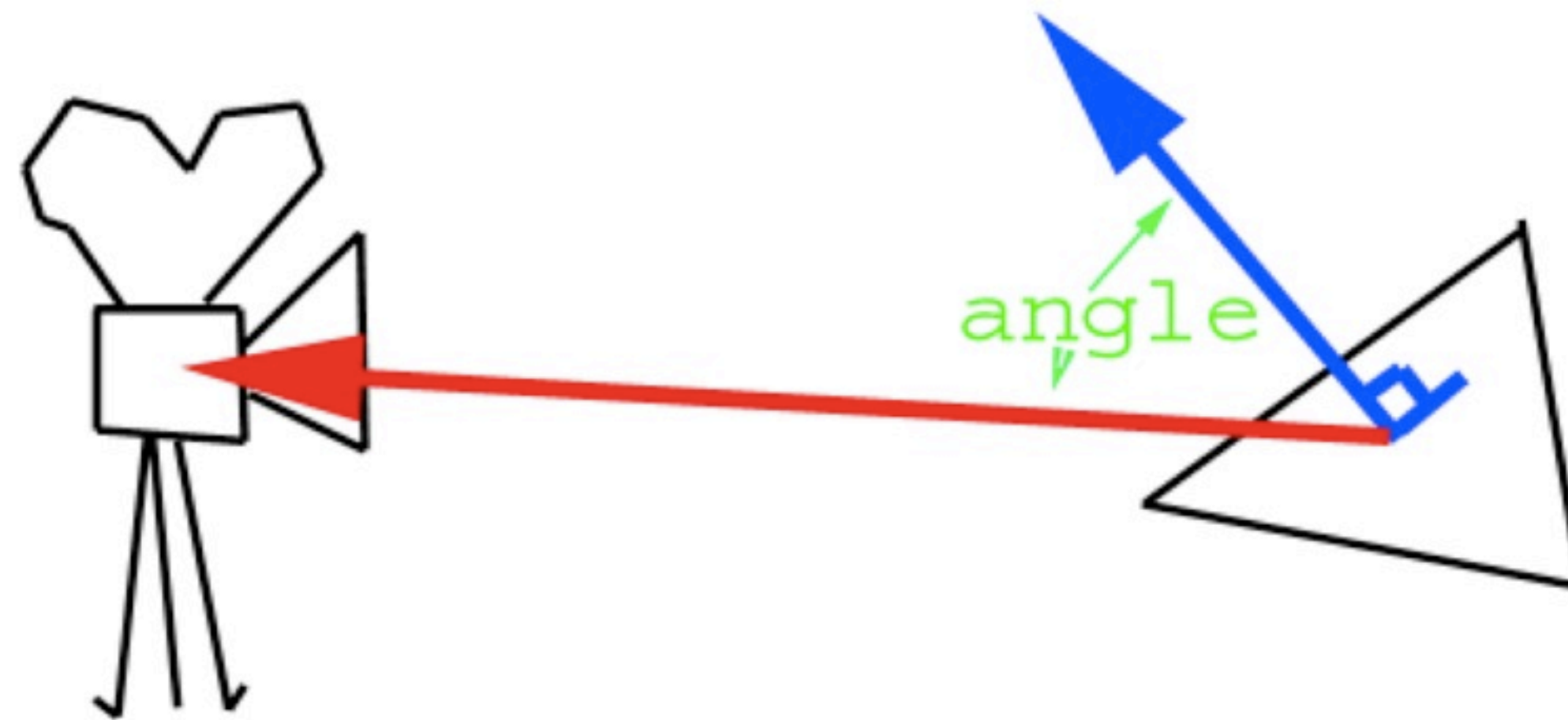
# Back Face Culling



- One of the simplest visible surface determination algorithms is culling, or disregarding all polygons that face away from the camera.
- Most software packages make the option of rendering back faces available to the user.
- Note : By Default this option is usually on

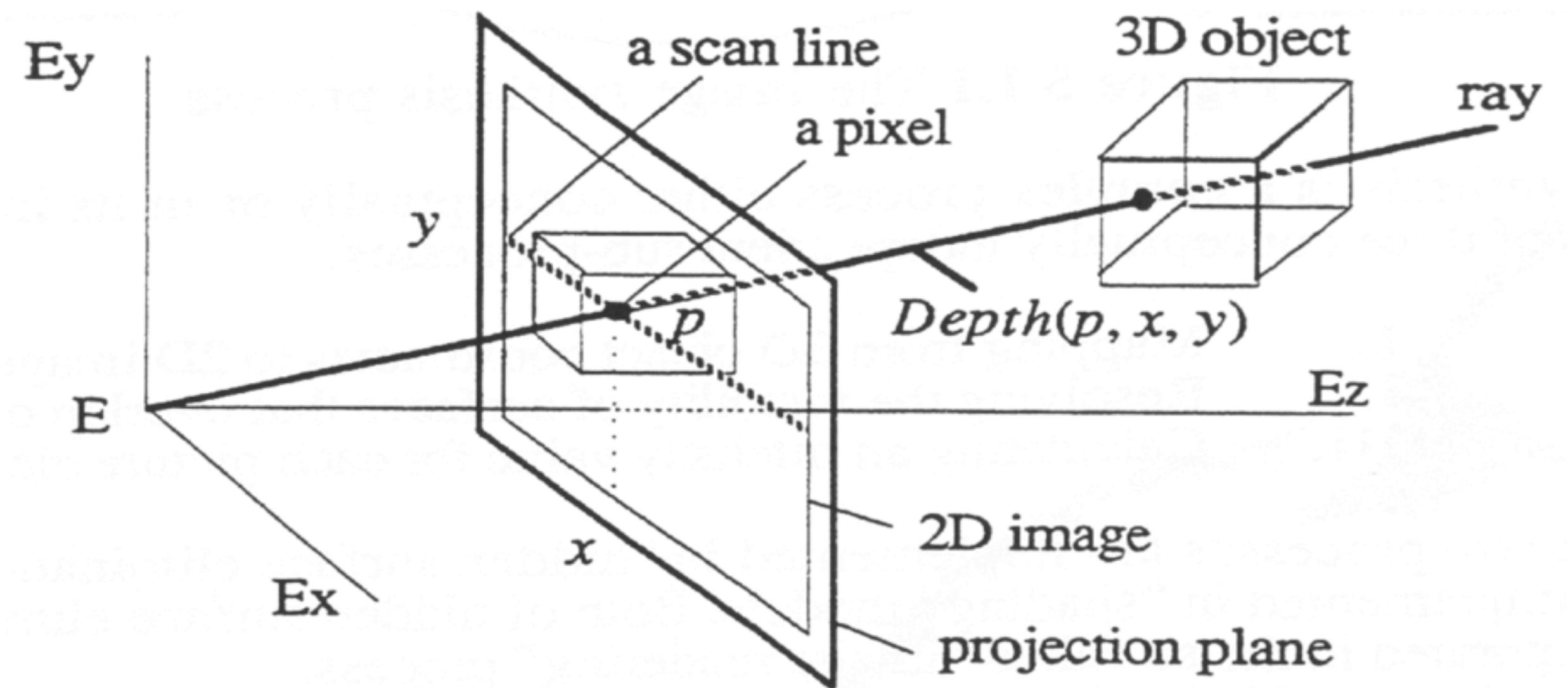


# Back Face Culling



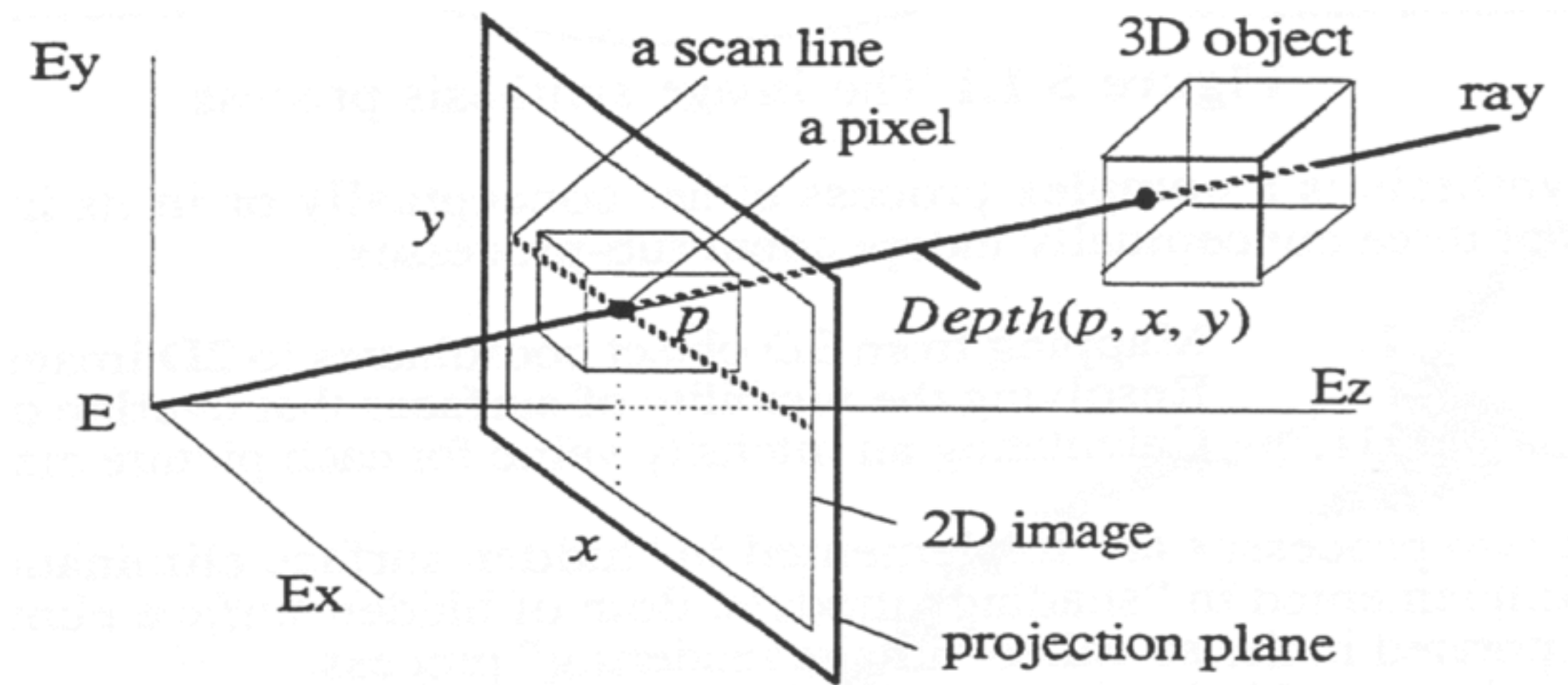
- The dot product is used to eliminate polygons, the normal of a polygon and a ray vector from the polygon to the camera
- From this, we can use simple rules to cull the polygon:
- If the viewing angle of the polygon is greater than 90 degree, cull it.
- Otherwise proceed with using polygon.

# Visible Surface Calculations



- The basic system for calculating the surfaces visible deals with the depth of the objects in the viewing volume
- remember we retained the depth value of point data in our projection systems.
- If multiple surfaces intersect the ray, then all the surfaces need analysing.
- The surface which is nearest to the Eye is accepted as the viewable surface hence the depth requirement.

# Visible Surface Algorithms



- The first process determines which of  $n$  objects is visible at every pixel in the image:

**For each pixel in the image**

**for every object in the scene ( $n$ )**

calculate distance from the Eye to the intersection of the ray and the object.

**next**

return colour value of the nearest object

**next**

This process requires a loop through each pixel ( $p$ ) and in that a loop through each object( $n$ ):  $np$  calculations.

For a PAL image with 20 objects in a scene, that's 8,294,400 object calls

# Visible Surface Algorithms

- The second approach is more subtle:
- For every object in a scene ( $n$ ), compare all the other objects in the scene and eliminates areas of the object that are occluded.
- In essence it's a clipping algorithm based on the current view:

**For each object in the scene( $n$ )**

**for every object in the scene ( $n$ )**

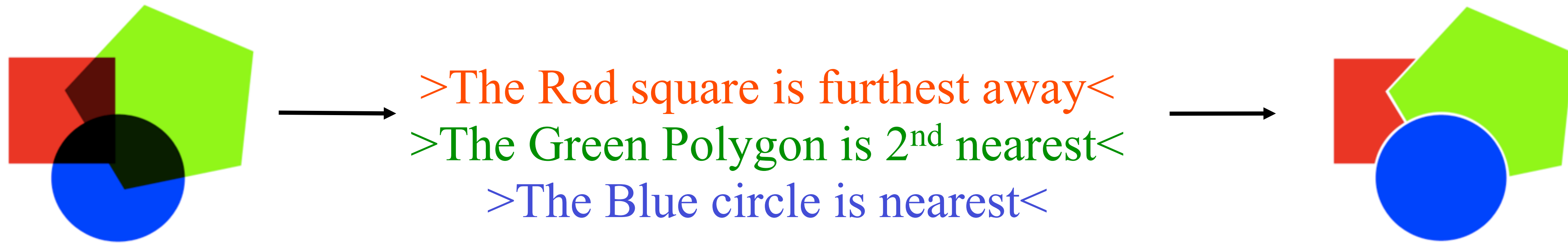
calculate areas occluded by the 2<sup>nd</sup> loop object and clip

**next**

draw the current object

**next**

# Example



In essence, this algorithm employs only  $n^2$  calls.



# Types of basic algorithm

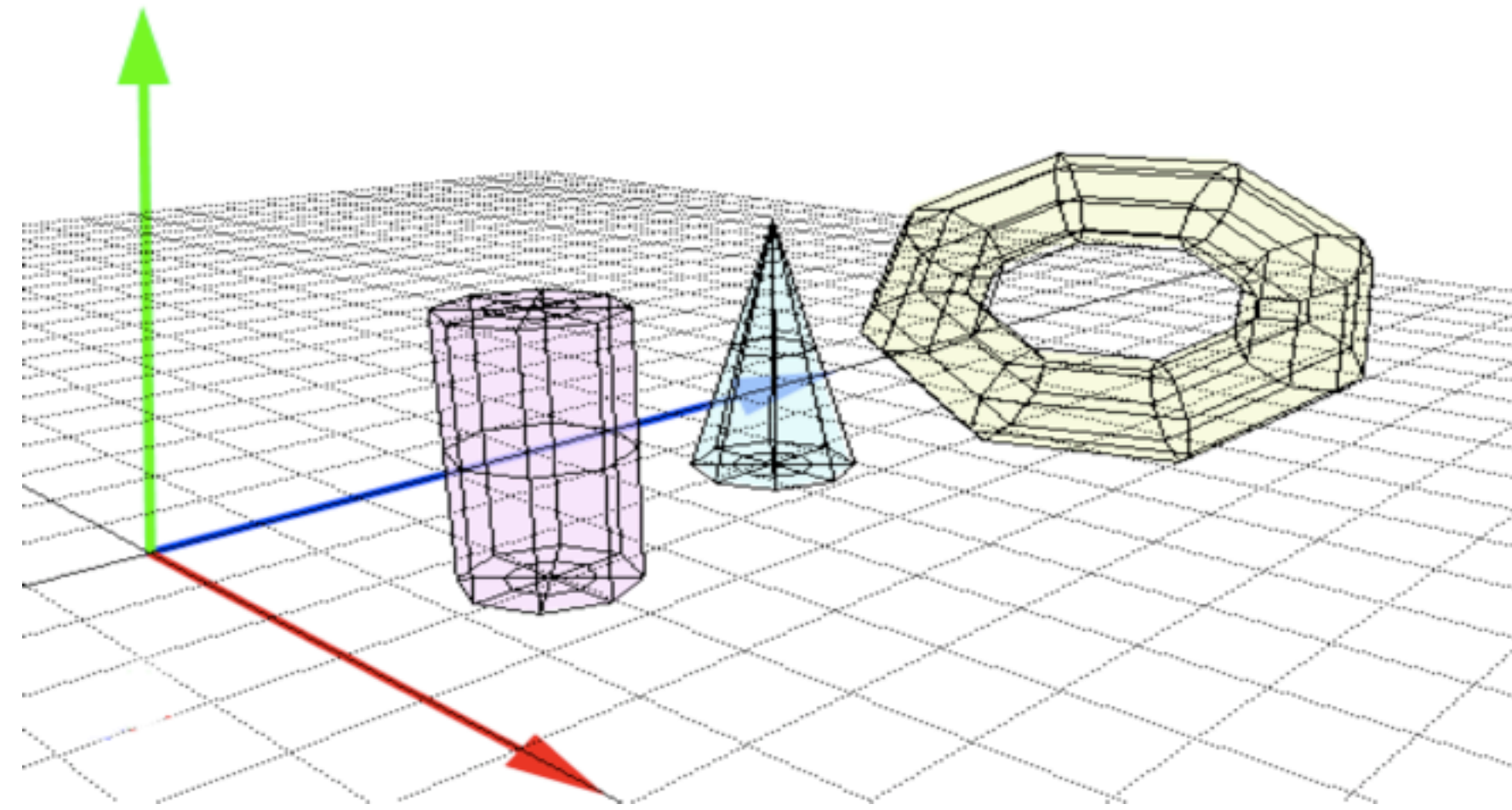
- In essence the 2 types of algorithm can be labelled:
- The first, dealing with pixel positions is called an image precision algorithm.
- The second, dealing with object relationships is called an object precision algorithm.
- Image precision algorithms are directly related to the raster size of the output image. (resolution dependant)
- Object precision algorithms are at first entirely based on objects relationships.

# Types of Algorithms

- Image precision algorithm can be thought of as a sampling algorithm the objects are sampled to the desired output image resolution.
- The image cannot be scaled without the need to re-calculate the entire image precision algorithm again.
- Object precision algorithms are independent of the output resolution.
- We still have to address the image resolution in respect to drawing the pixels but it is a secondary function independent of the process.
- Object precision algorithms can be called prior to the image generation.
- The results are unaffected by any transformation on the actual output image itself.



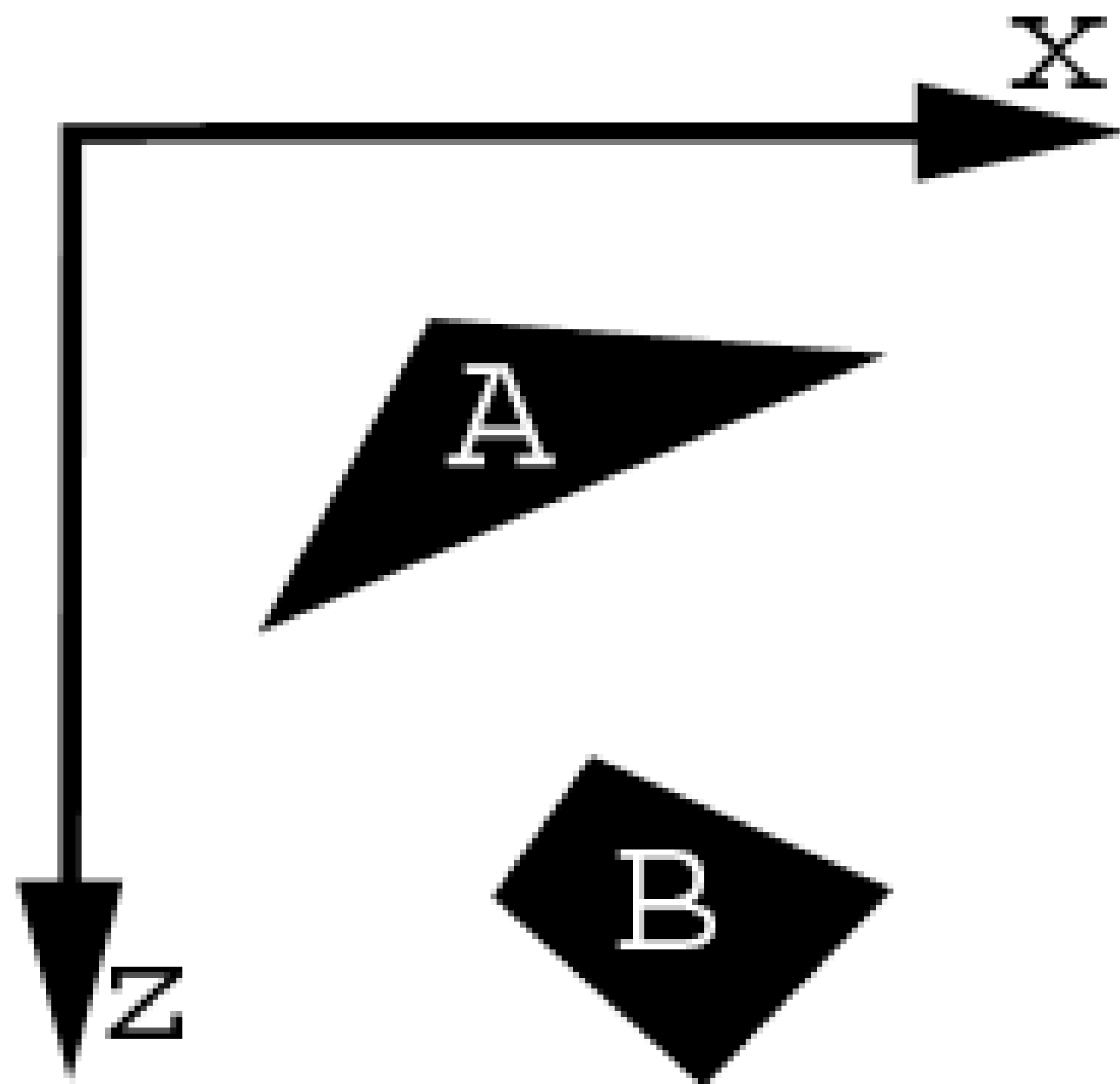
# List Priority Algorithms



- This method of algorithm performs a series of evaluations on the objects to be viewed in order to ascertain a list of polygons sorted in bias of depth.
- Once a list is generated, the objects are rendered in descending depth order.
- As the pixel colours are calculated and stored in the colour buffer, pixels nearer the camera will be drawn last and thus overwrite data that is further away.

# List Priority Algorithms

- Despite being simplistic in idea, the actual execution of the algorithm presents obstacles.



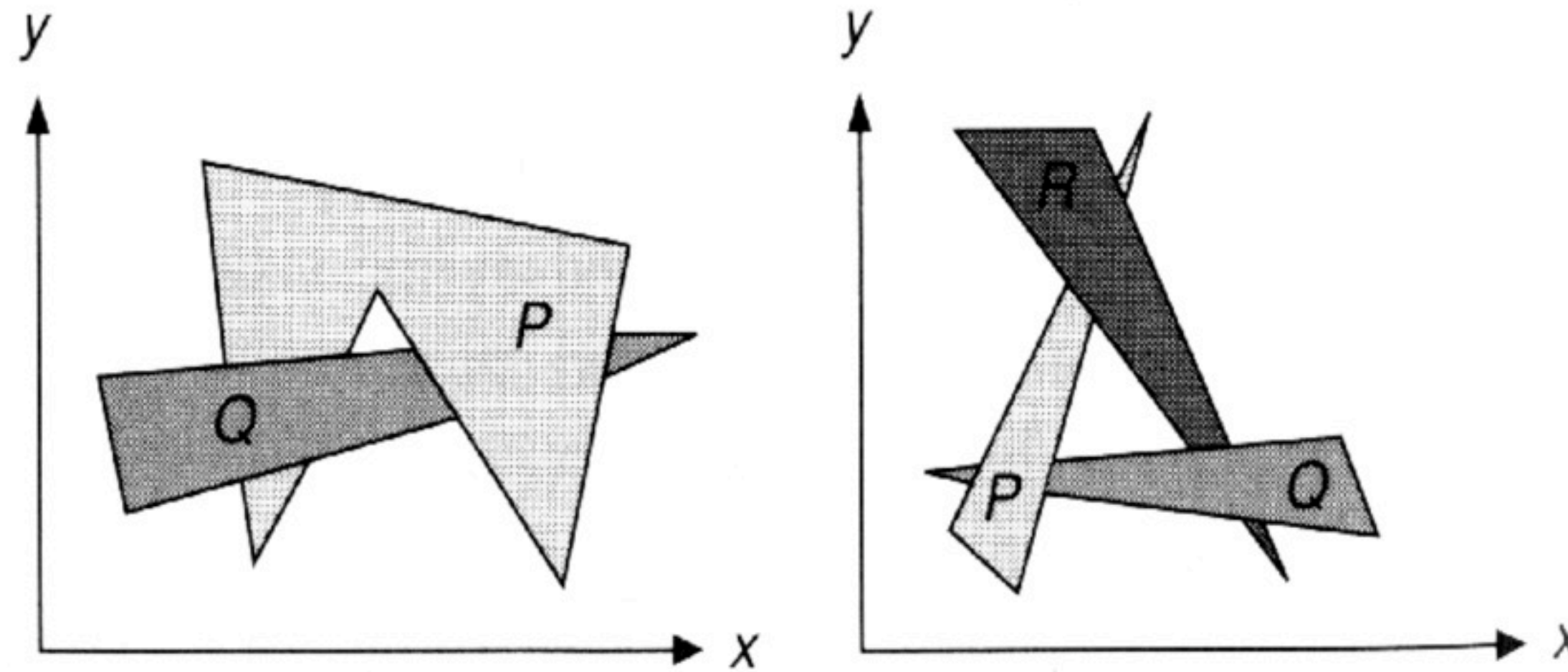
In the case to the left,  
all of the faces of object A  
*are nearer than*  
all of the faces of object B.

Object B will be drawn first,  
then object A will be drawn.

Any faces of object A that occlude object B will automatically overwrite the colour buffer.

# List Priority Algorithms

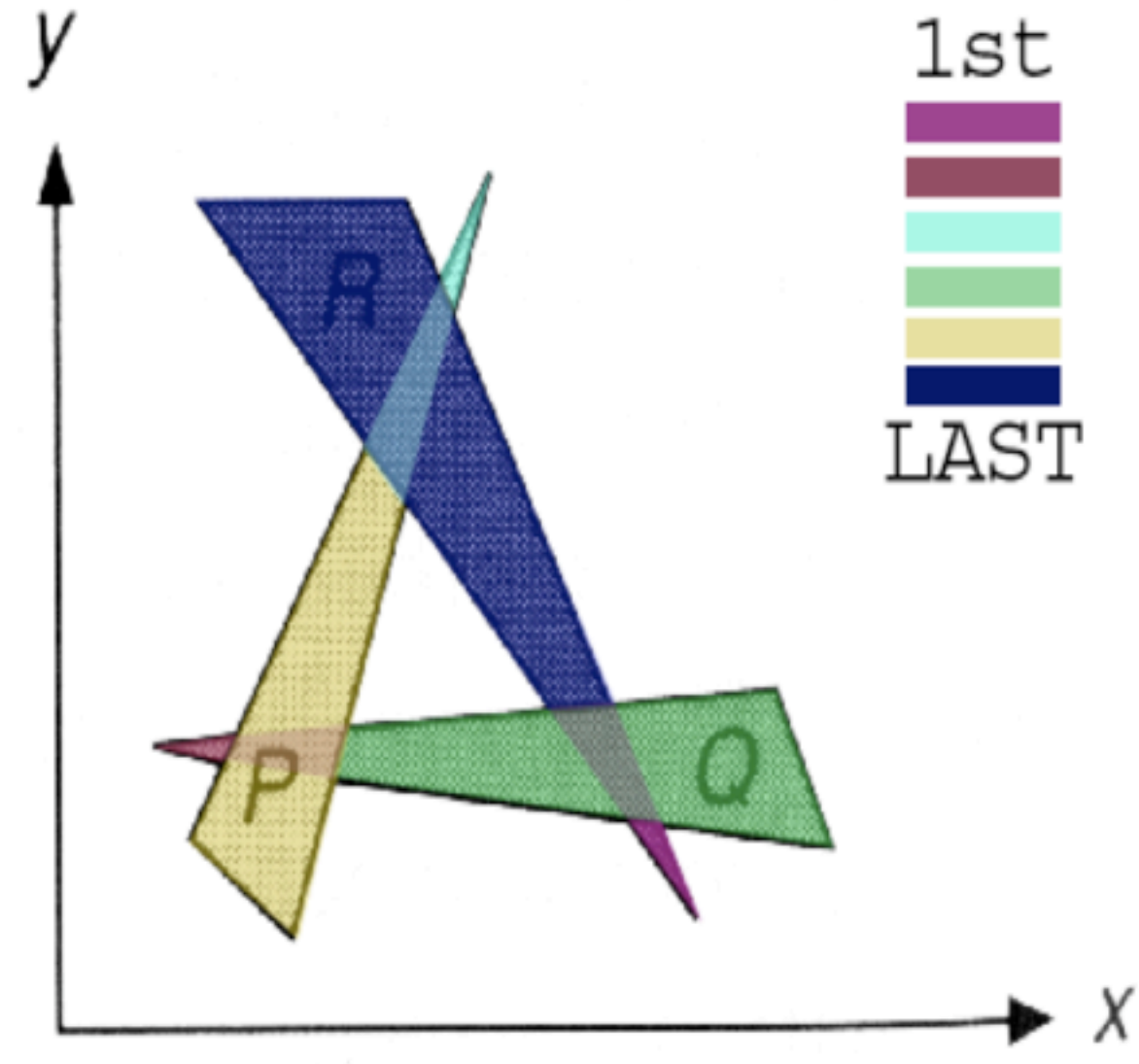
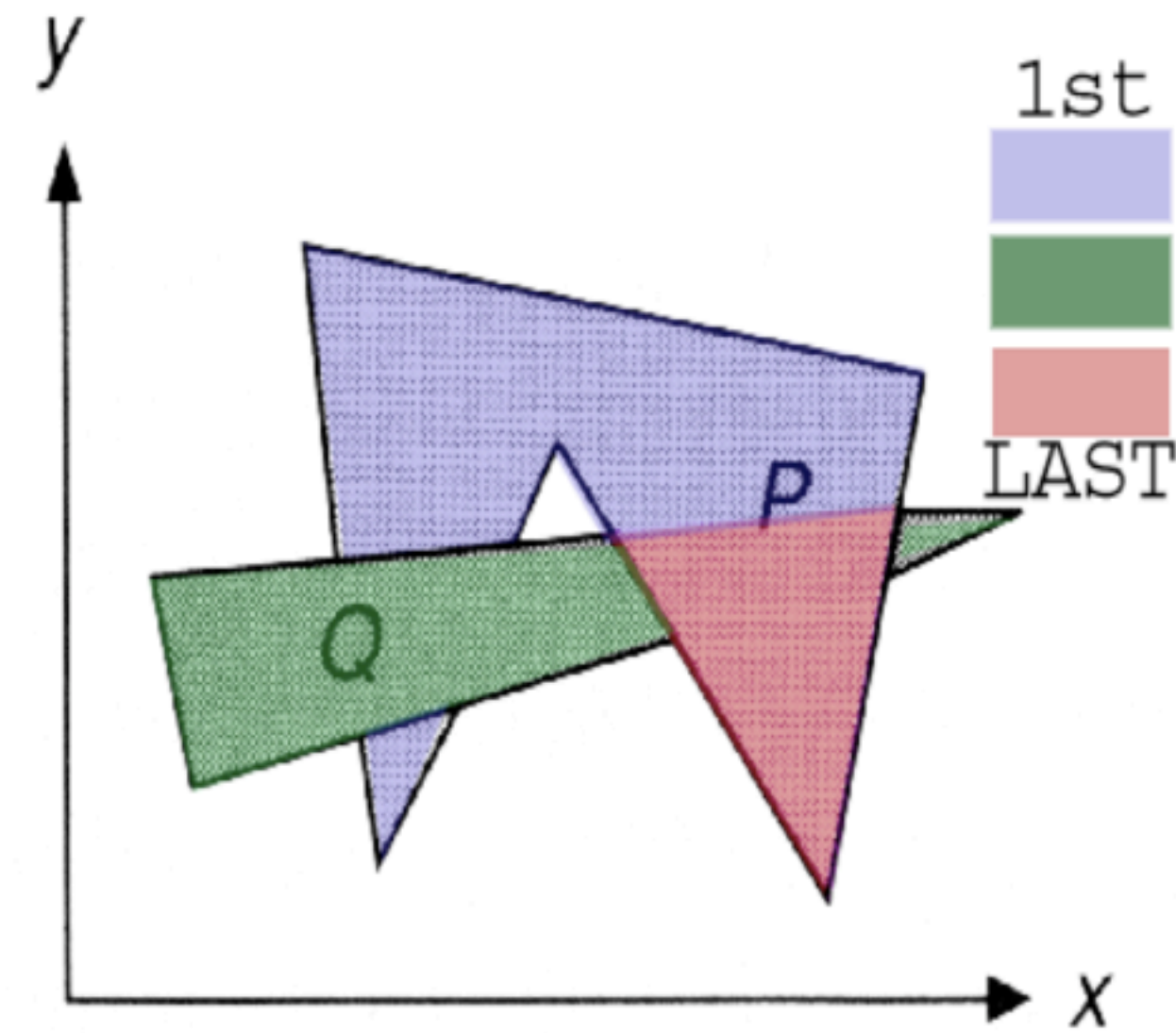
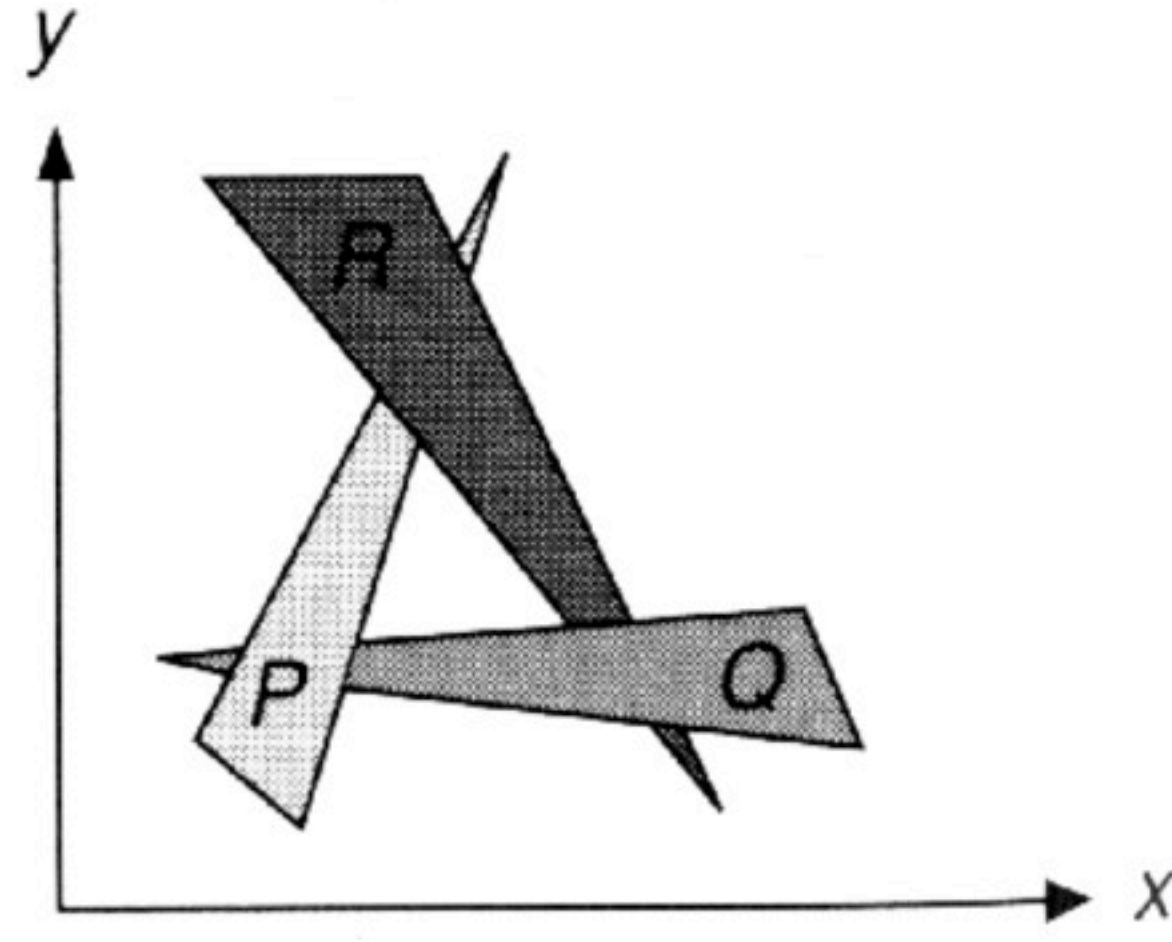
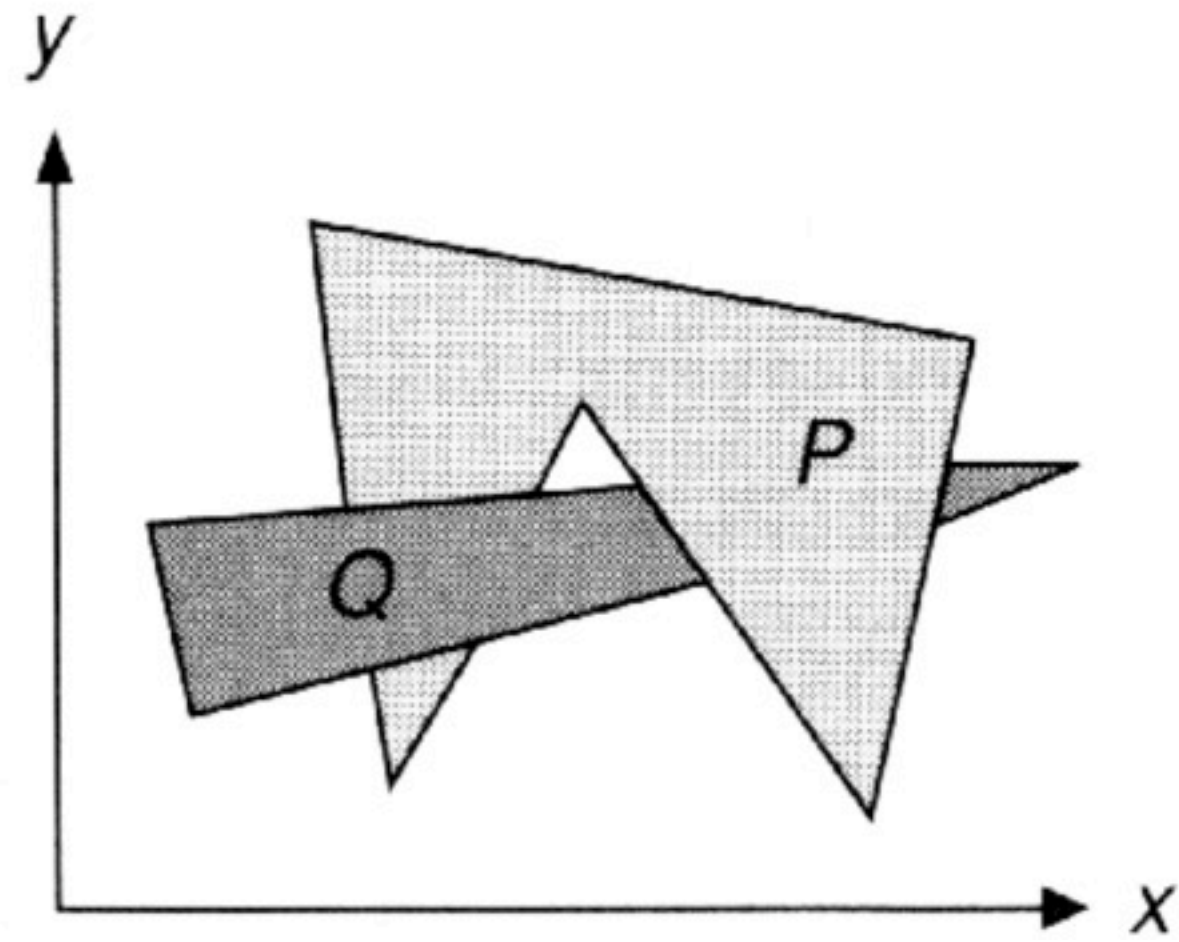
- Such a distinction between objects cannot be made in some cases.



In these cases, the algorithm splits the polygon faces down further so that a linear order can be generated.



# List Priority Algorithms

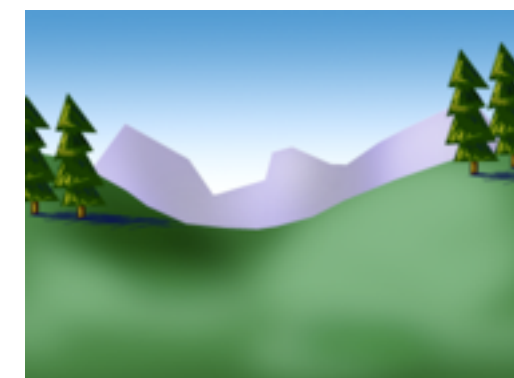


# List Priority Algorithms

- List priority algorithms are a combination of image precision methods and object precision methods.
- Depth comparisons and object splitting is done in an object operation, whilst the actual image scan conversion relies on the display device for its operation.

# The Depth Sort (Painters) Algorithm

- The Depth Sort Algorithm, by Newell, Newell and Sancha, it is affectionately termed as the painting algorithm
- (as an artist paints over parts of the image to put objects nearer to the viewer)



# Basic Depth Sort Algorithm

- The basic depth sort algorithm works on the following three steps:
  1. Sort all polygons to the farthest z coordinate of each.
  2. Resolve any ambiguous polygons where the z distances overlap – splitting polygons if necessary.
  3. Scan convert each polygon in ascending order of farthest z coordinate (i.e. back to front)

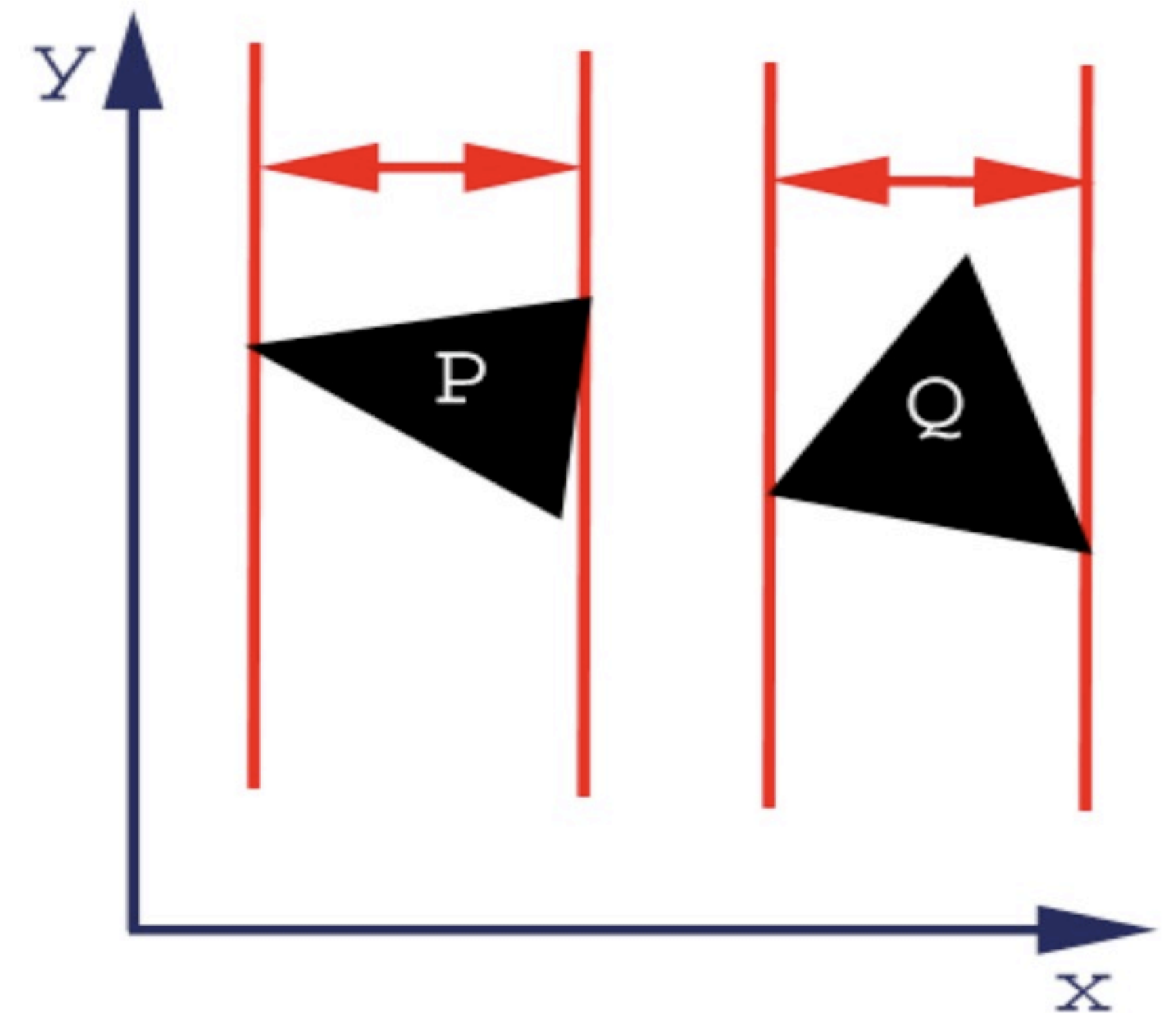


# Basic Depth Sort

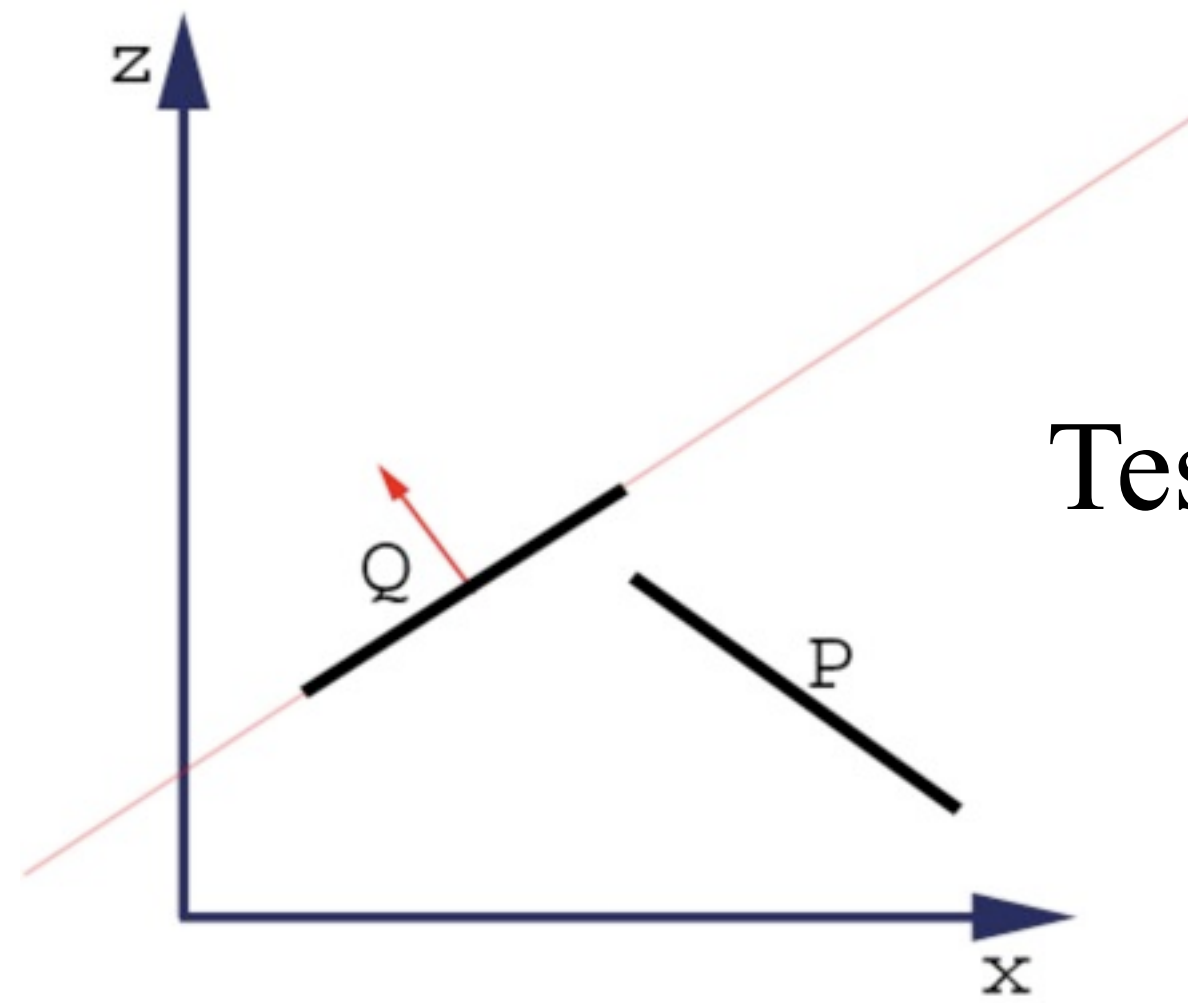
- For a polygon to successively be given a list priority, it must be tested against up to 5 operations levels of complexity.
- As soon as a single test is passed, the test polygon is deemed NOT to obscure the other polygon.
- If a polygon passes a test against all other polygons, then the polygon can be given a list priority.

# Basic Depth Sort

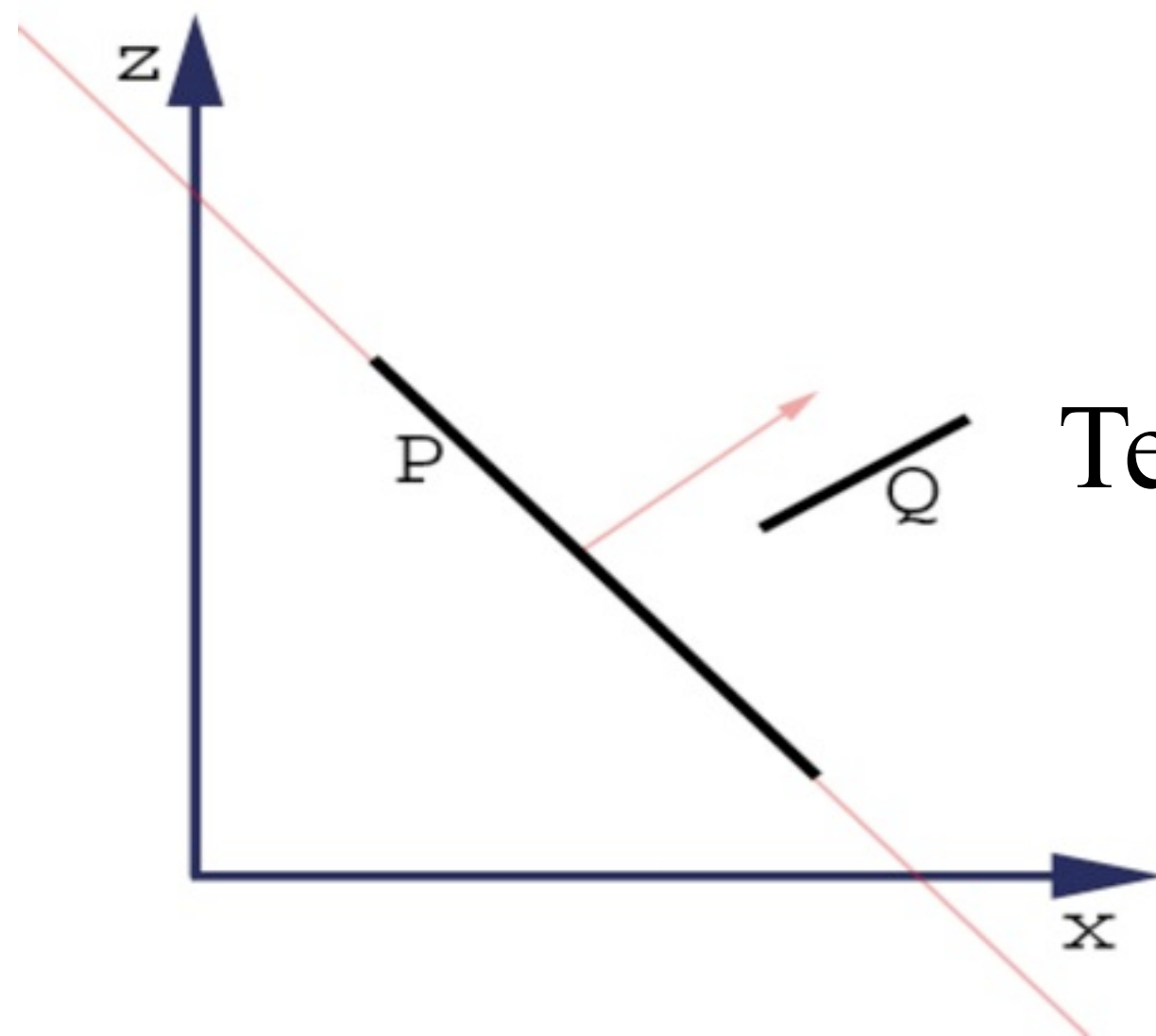
- For example, if we are testing polygon P against polygon Q
- Test 1 : Do the polygons' x extents not overlap
- And similarly ...
- Test 2 : Do the polygons' y extents not overlap?



# Basic Depth Sort



Test 3: Is P entirely on the opposite side of Q's plane from the viewport?

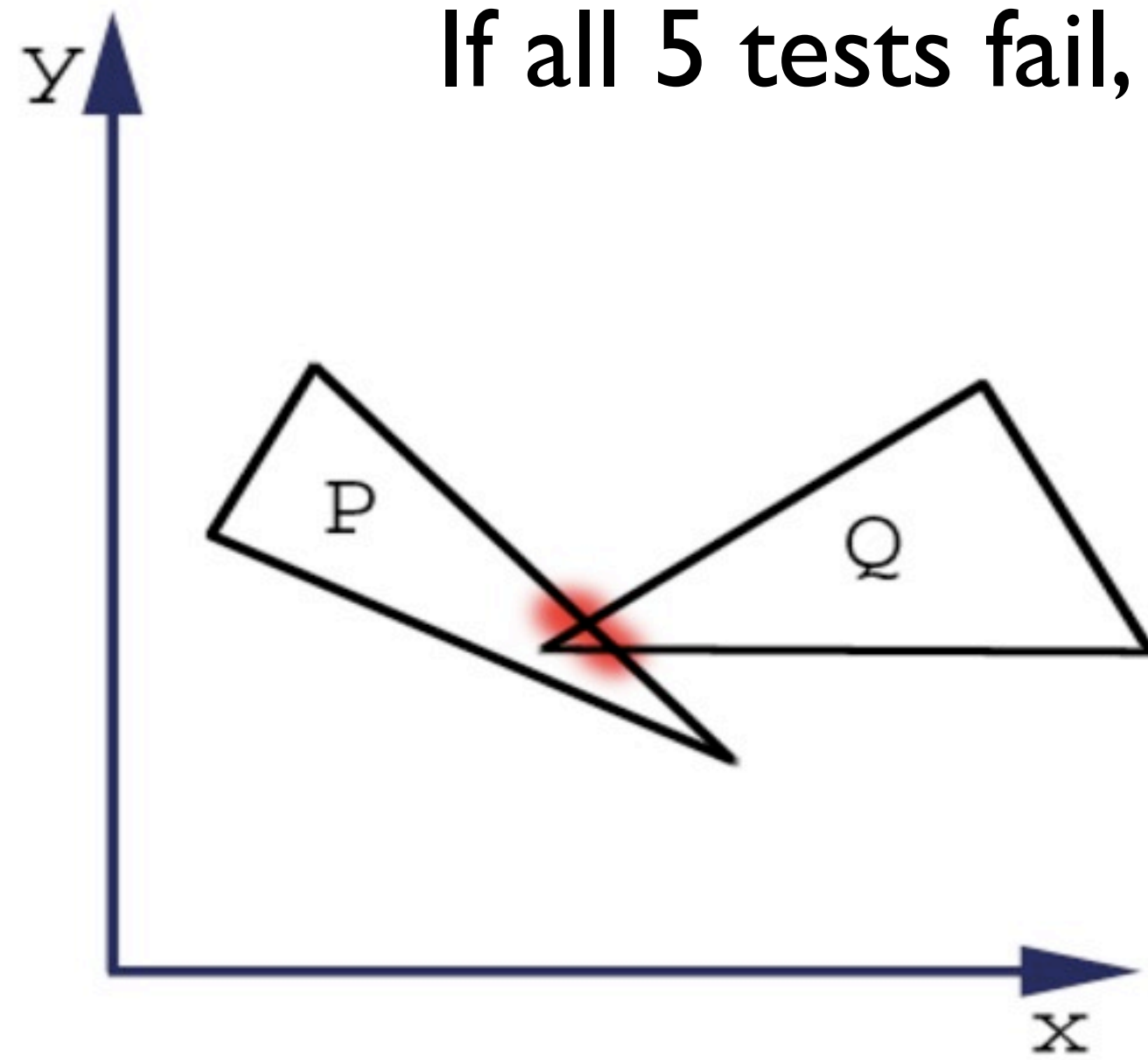


Test 4: Is Q entirely on the same side of P's plane from the viewport?

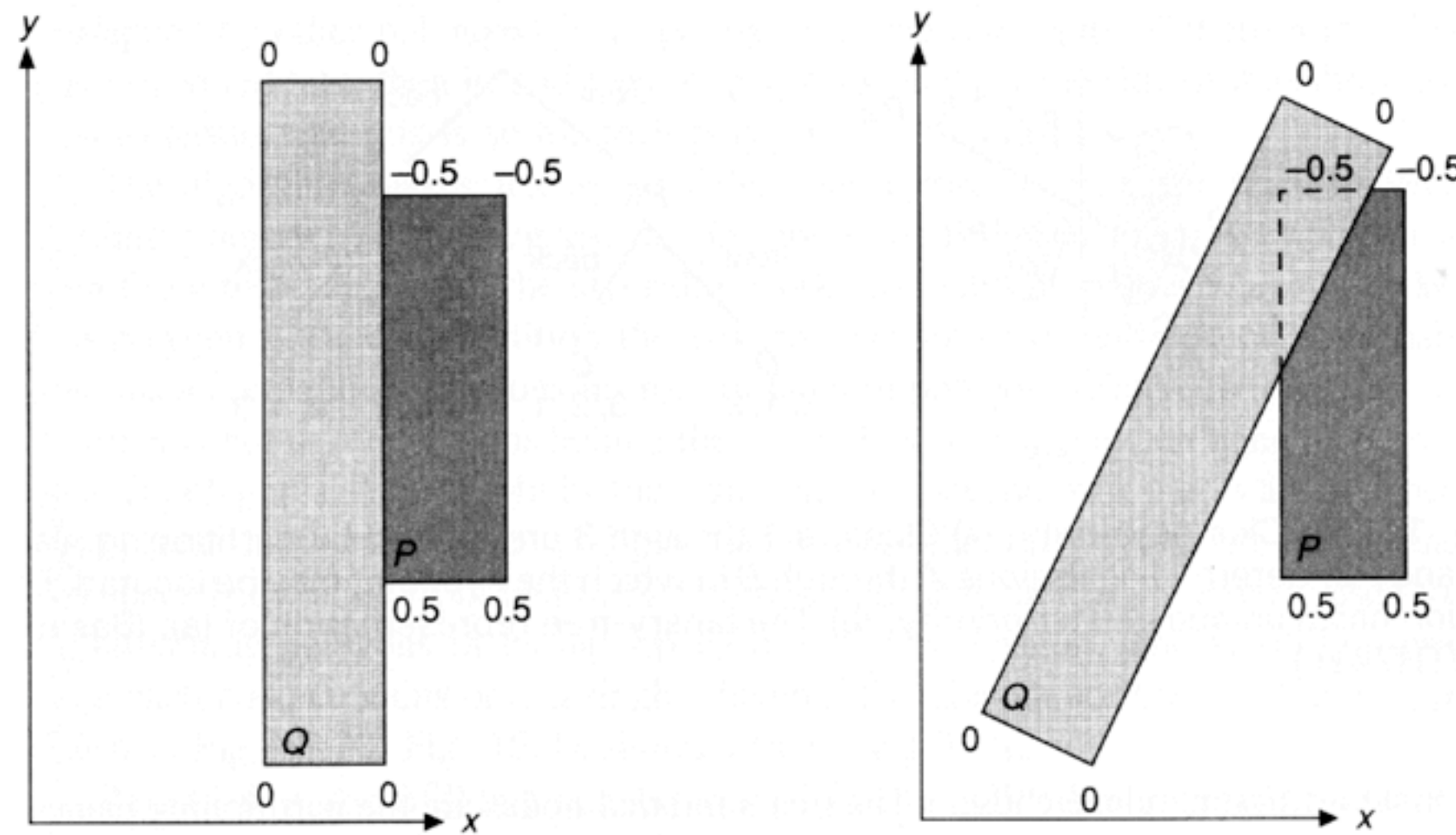
# Basic Depth Sort

Test 5: Does the projections of the polygons onto the  $(x,y)$  plane not overlap?  
*This can be determined by comparing the edges of one face against another – similar in essence to 2D line clipping we explored earlier.*

If all 5 tests fail, then we can assume that P actually obscures Q.



# Example



The examples above are annotated only their depth value.

The image on the left passes test 1 – the x extent of each face do not overlap.

Consider the rotation of the left face to resemble the right picture.

**Test 1 fails** – the x extent of each face overlap.

**Test 2 fails** – the y extent of each face overlap.

**Test 3 and Test 4 fail** – neither face is wholly in one half space of the other.

**Test 5 fails** – the face edges overlap.

It is quite possible for a situation where all 5 tests fail on a polygon.

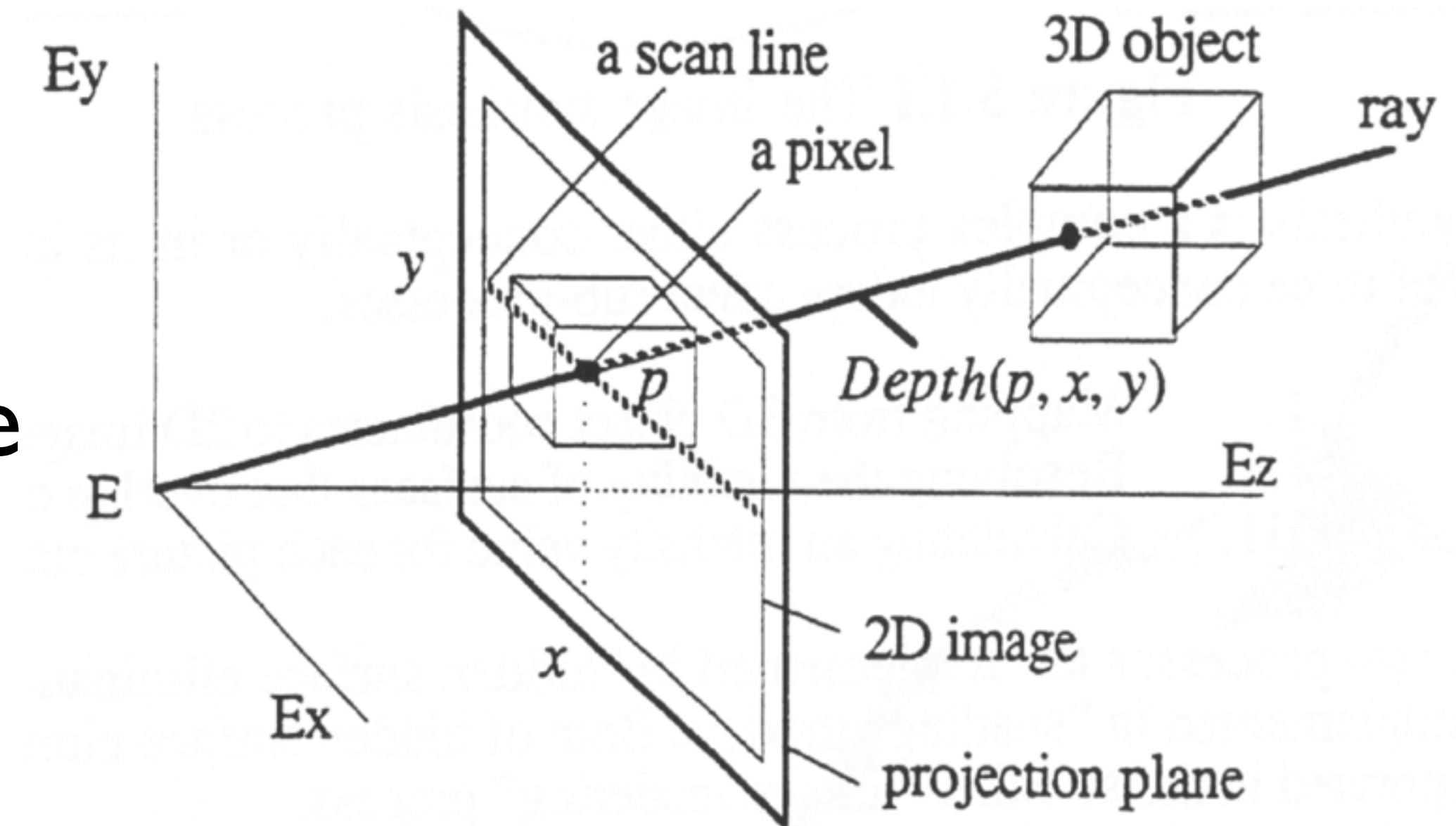
# Does Depth Sorting Work?

- If the polygons are small the cyclic overlap errors are diminished
- Larger polygons generate larger visible errors
- Nobody uses depthsort:
- Its nasty to resolve cyclic overlaps!
- Many games use list priority sorting without fixing perspective overlap



# 1974 Catmull Z Buffer Algorithm

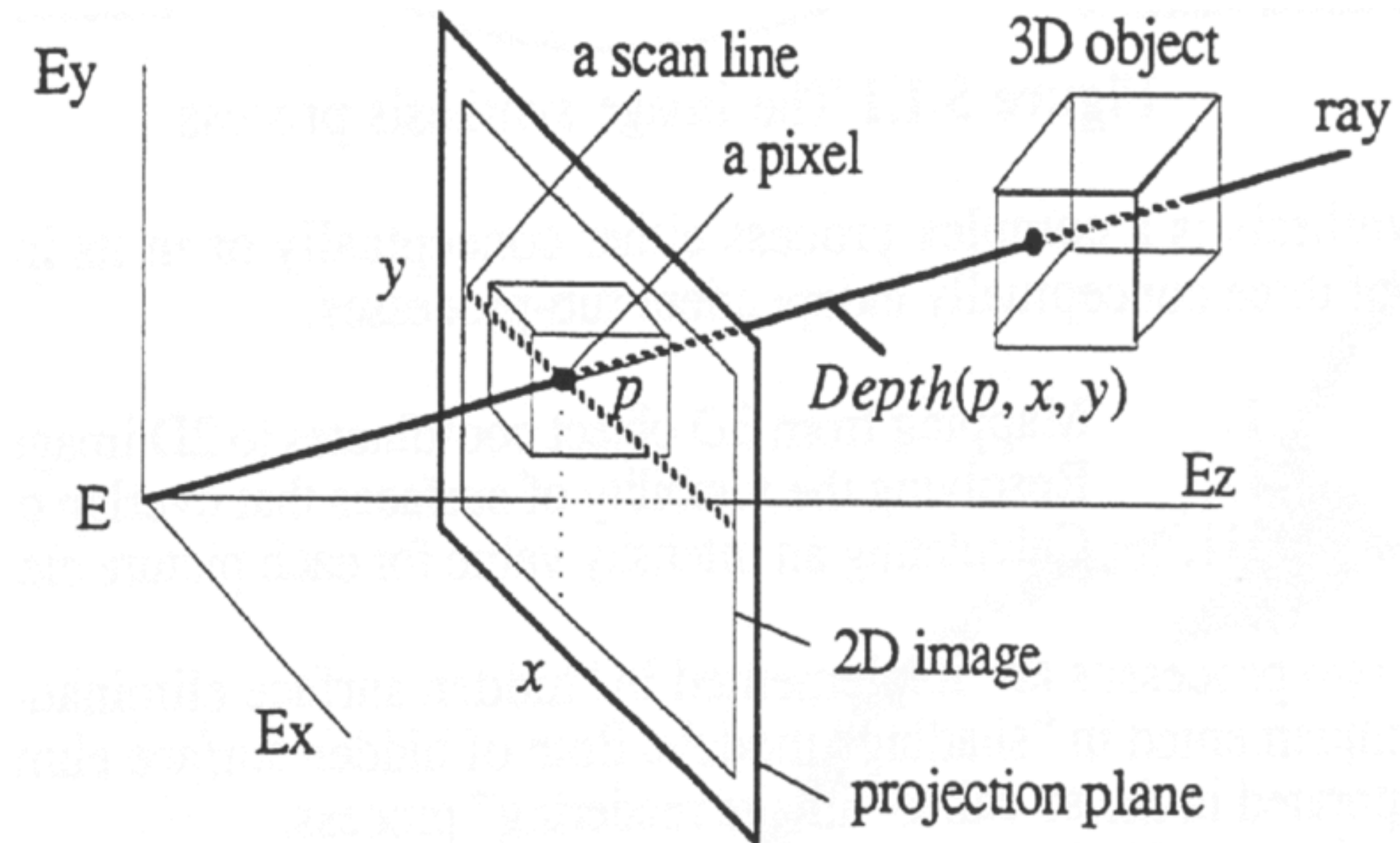
- This is the simplest image-precision algorithm.
- It generates not only a colour buffer for the colour of the pixels,
- but a z buffer as well, for the depth of each pixel.
- The colour buffer is initialised to the background colour defined.
- The z buffer is initialised to a very large distance.





# Z Buffer

- Each polygon is scanned in turn against each pixel in the projection
- If pixel depth is no farther away than the current depth stored then the colour returned from the pixel polygon is placed in the colour buffer, and its respective depth in the depth buffer.

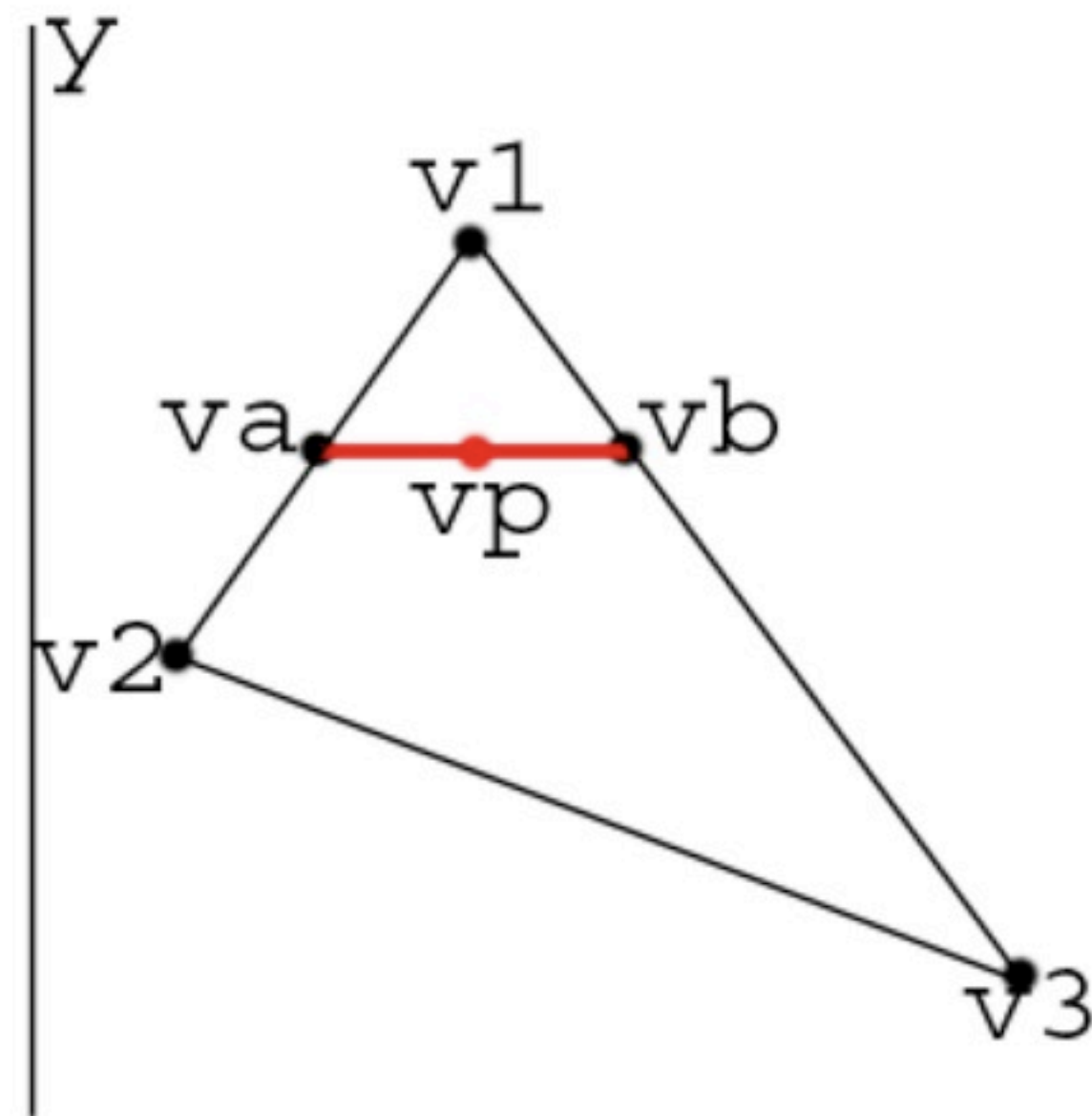


# Catmull Algorithm

- As such, the Catmull algorithm does not require any object component sorting.
- The hardest problem with the Catmull algorithm is working out what pixels are covered by the faces.
- If the faces are all triangular then this becomes easier.

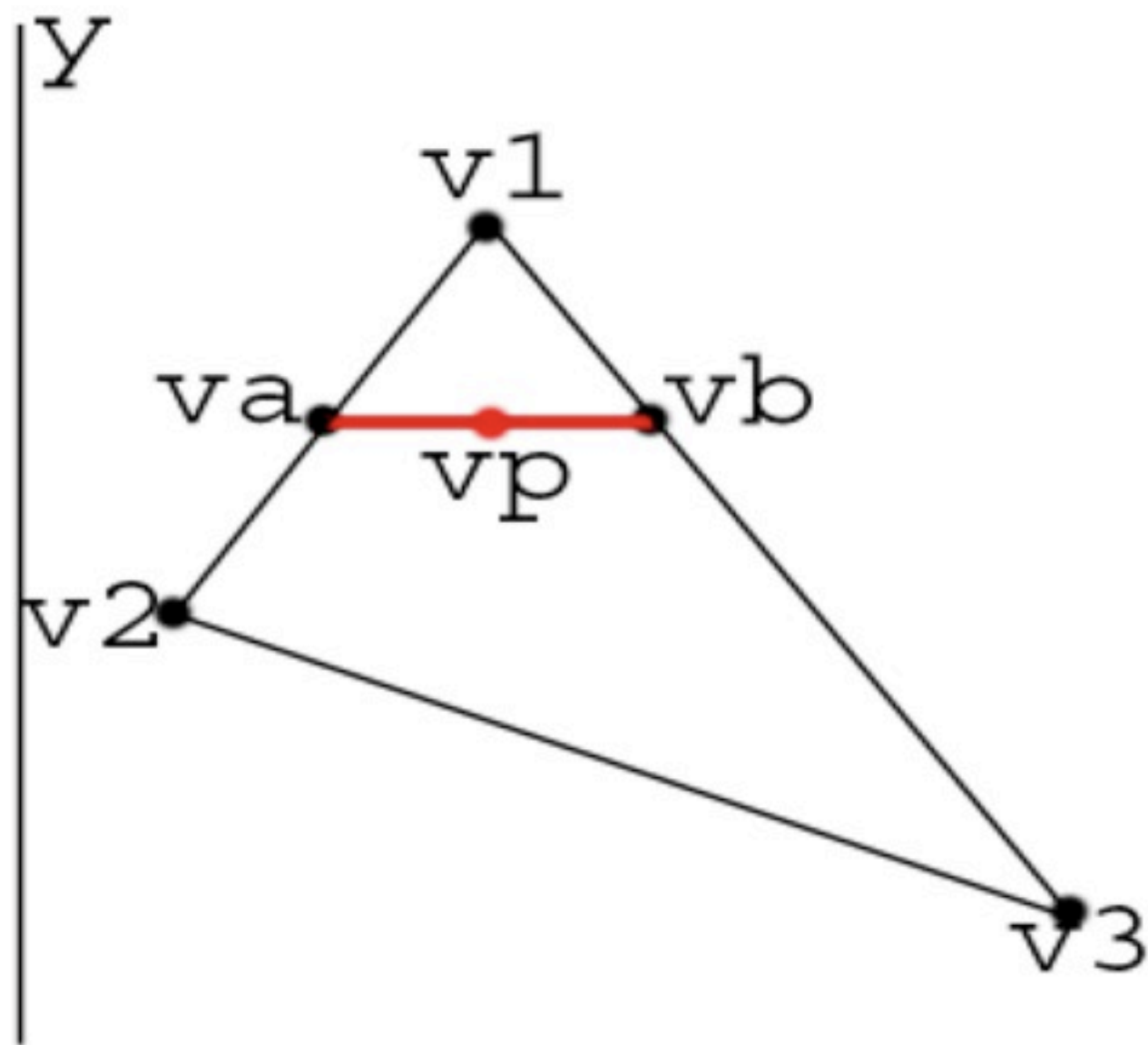
```
d(x,y) = very large  
c(x,y) = bg colour  
For each face  
  For each pixel (x,y) covering the face  
    depth = depth of face at x,y  
    if depth < d(x,y) then  
      c(x,y) = colour of face at x,y  
      d(x,y) = depth  
    next  
  next  
next
```

# Calculating the pixels in a triangle



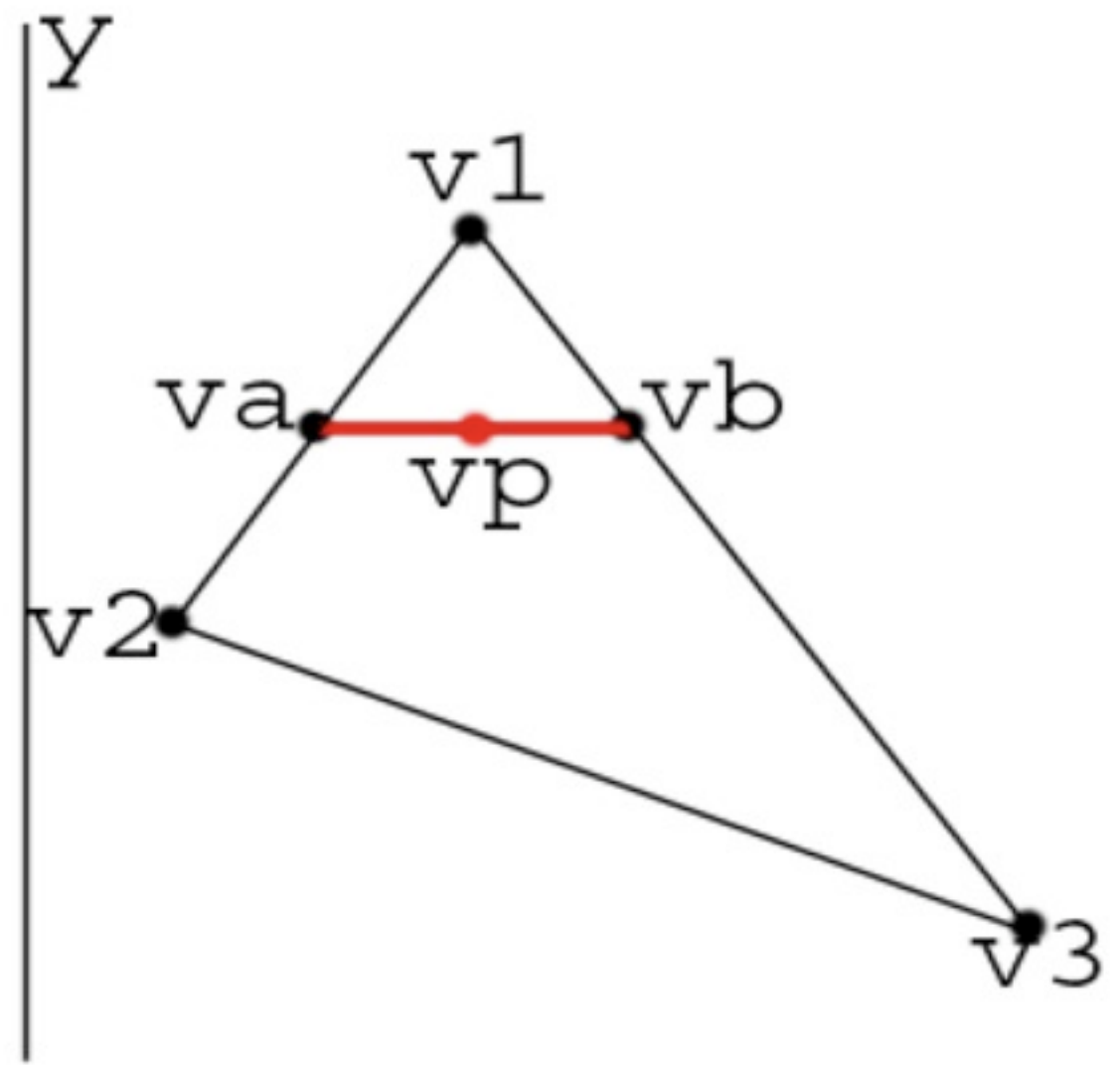
- We have a triangular face generated by 3 known points ( $v_1, v_2, v_3$ ).
- We can generate a scanline from the top of the triangle to the bottom, i.e.  $v_{1y}$  to  $v_{3y}$ .
- We explicitly describe the scanline  $y$  position – at  $v_{ay}$  in the example above.
- The horizontal pixels on the indicated scanline run from point  $v_{ax}$  to  $v_{bx}$  – unknown.

# Calculating the pixels in a triangle



- To Calculate the unknown element we can use the following equation

$$va = v1 - \left[ \frac{v1y - vay}{v1y - v2y} \times (v1 - v2) \right]$$



$$va = v1 - \left[ \frac{v1y - vay}{v1y - v2y} \right] \times (v1 - v2)$$

Annotations with arrows pointing to the equation:

- start** points to  $v1$
- scaling ratio** points to the fraction  $\frac{v1y - vay}{v1y - v2y}$
- vector of distance** points to  $(v1 - v2)$

$V_{ay}$  tends to  $V_{2y}$

$$va = v1 - \left[ \frac{v1y - v2y}{v1y - v2y} \right] \times (v1 - v2)$$

$$va = v1 - 1 \times (v1 - v2)$$

$$va = v1 - v1 + v2$$

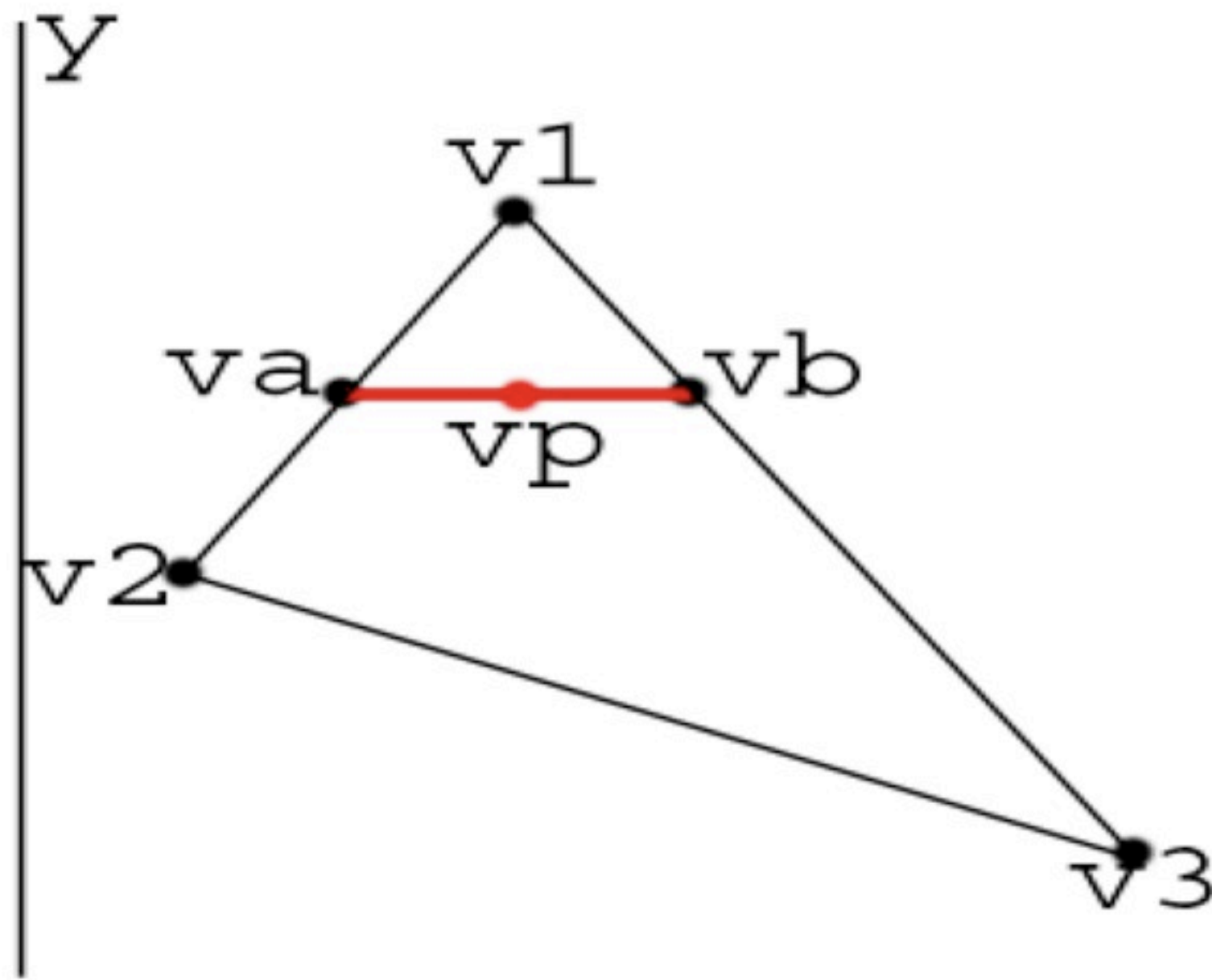
$$va = v2$$

$V_{ay}$  tends to  $V_{1y}$

$$va = v1 - \left[ \frac{v1y - v1y}{v1y - v2y} \right] \times (v1 - v2)$$

$$va = v1 - 0 \times (v1 - v2)$$

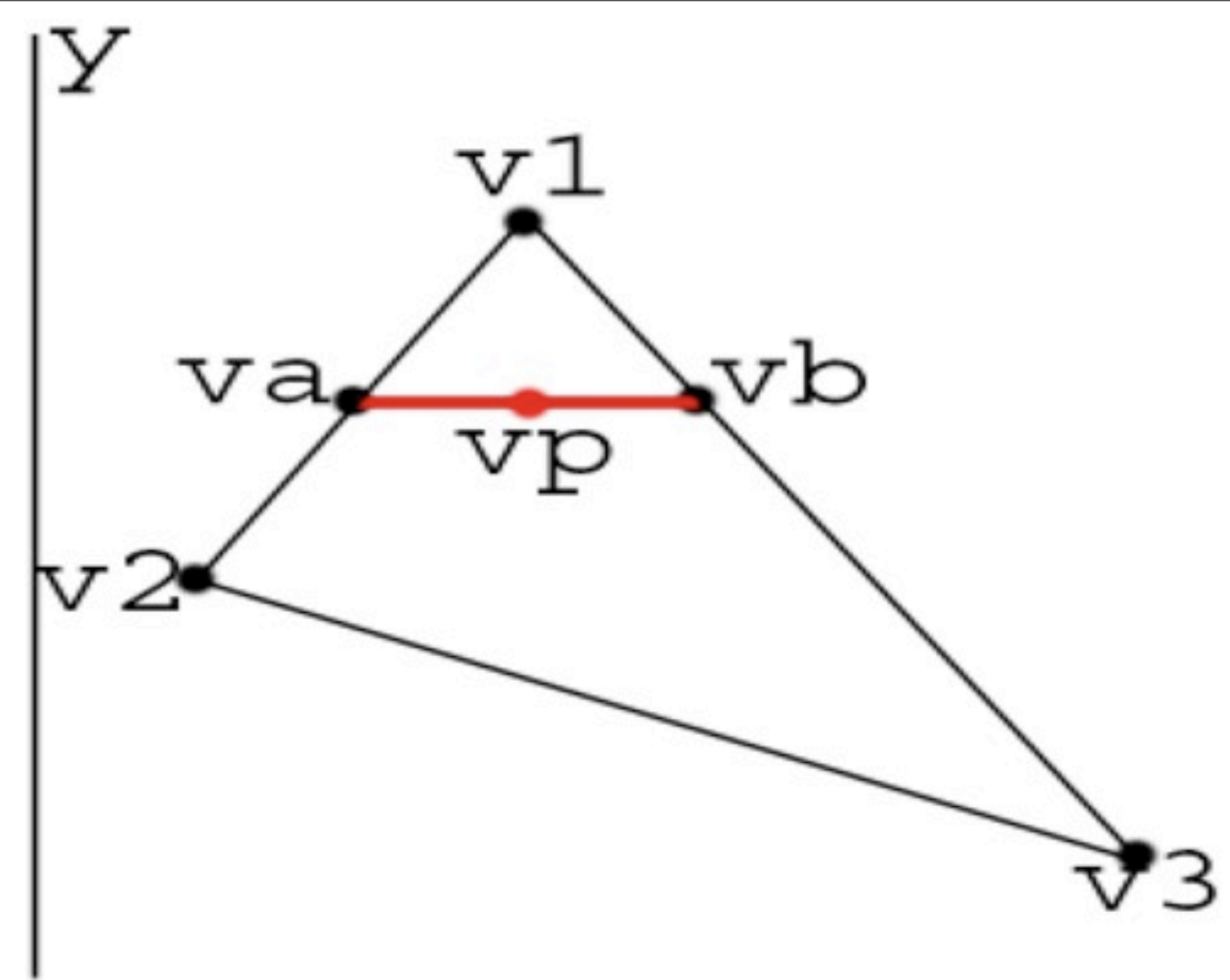
$$va = v1$$



We can also state that...

$$v_b = v_1 - \left[ \frac{v_1 y - v_a y}{v_1 y - v_3 y} \right] (v_1 - v_3)$$





Hence for x components:

$$vax = v1x - \left[ \frac{v1y - vay}{v1y - v2y} \right] (v1x - v2x)$$

$$vbx = v1x - \left[ \frac{v1y - vay}{v1y - v3y} \right] (v1x - v3x)$$

Thus solving the start ( $v_{ax}$ ) and end ( $v_{bx}$ ) x positions for a given scanline vertical position.

# Pros and Cons of the Z Buffer

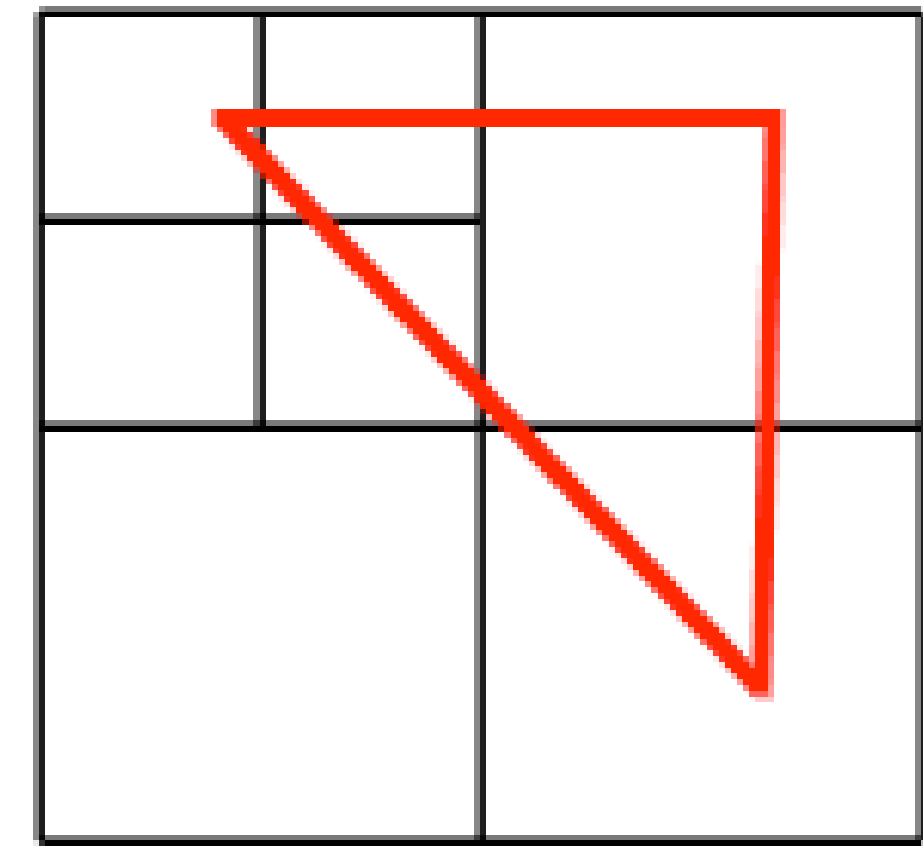
- Pros
  - No object sorting, comparison or splitting required – straight forward approach
- Cons
  - At least twice the memory overhead-
  - more memory for zdepth buffer to prevent depth aliasing
  - but we have so much these days or we could scan it in strips...
  - We need the z values! (We may require them anyway later for shaders etc)
  - We may perform more than one polygon draw per pixel

# Z Buffer

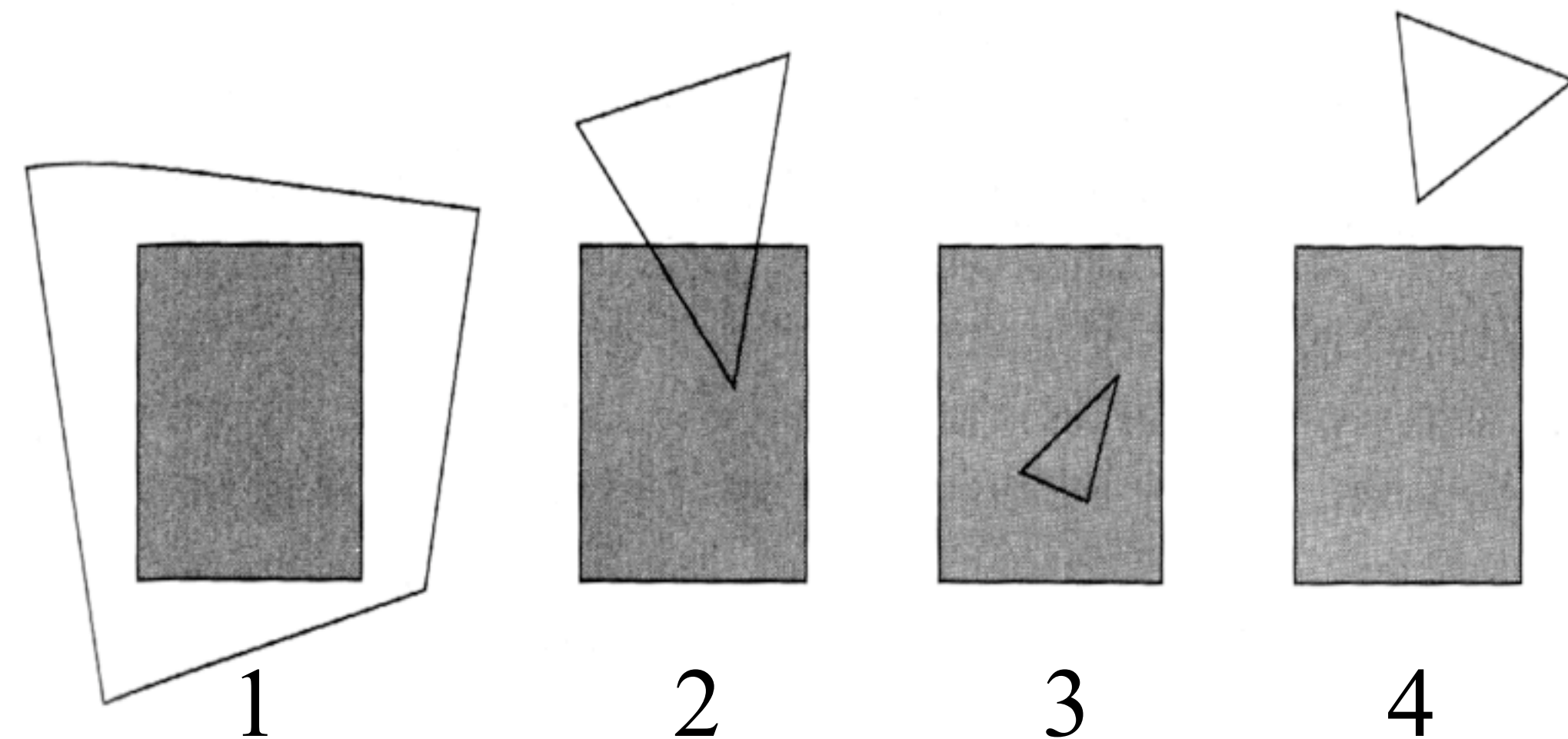
- The basic algorithm before gives us a colour buffer and a z-depth buffer
- The z buffer algorithm does not require only polygonal data : one of its strengths
- The z buffer algorithm is an image precision algorithm and as such can produce aliasing.
- More variations have been made :
  - Catmul '78
  - Carpenter in '84 A-Buffer

# Screen Subdivision-Divide & Conquer

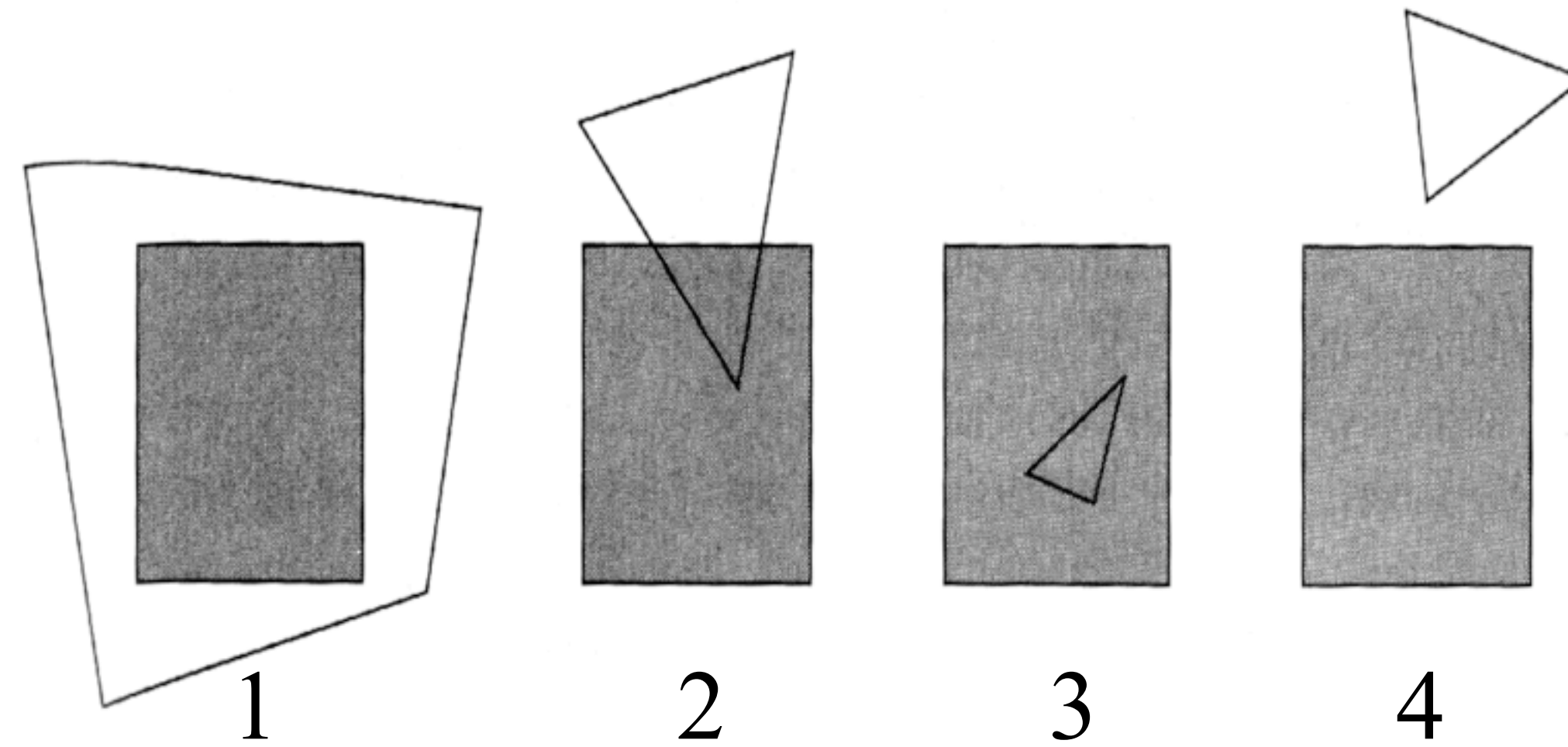
- An improvement to the basic algorithm is to subdivide the screen into “buckets” to aid efficiency.
- When studying the relationship between regions and polygons, we can come to 4 distinct cases:



- 1 A polygon completely surrounds the region.
- 2 A polygon and the region intersect.
- 3 The region surrounds a polygon.
- 4 A polygon and the region are far apart.



# Subdivision Methods



- This allows us to analyse the situation:
- In case 1, all of the pixels in the region can proceed visibility and testing.
- In case 4, none of the pixels in the region contribute to the visibility of the polygon and it is discarded.
- In cases 2 and 3, the region is subdivided to smaller regions and the process starts again.



# Example Algorithm

Subdivide screen into X and Y regions

For y = Y to 1

For x = 1 to X      ... We loop thru each pixel in the region

z\_buffer(x,y)=Zmax      ...Set the zbuffer to be very large

For p = 1 to NOP      ...we loop thru all the polygons in the scene

If case 4 then goto next polygon      ...the poly has no effect

If case 2 or 3 then subdivide bucket and start again

...we need to subdivide the region more

If case 1 then

z = depth(p,x,y)      ... get the depth of the current polygon and store in z

if z < z\_buffer(x,y)      ... is the polygon nearer than the buffer data?

z\_buffer(x,y)=z      ...replace z\_buffer data with current z

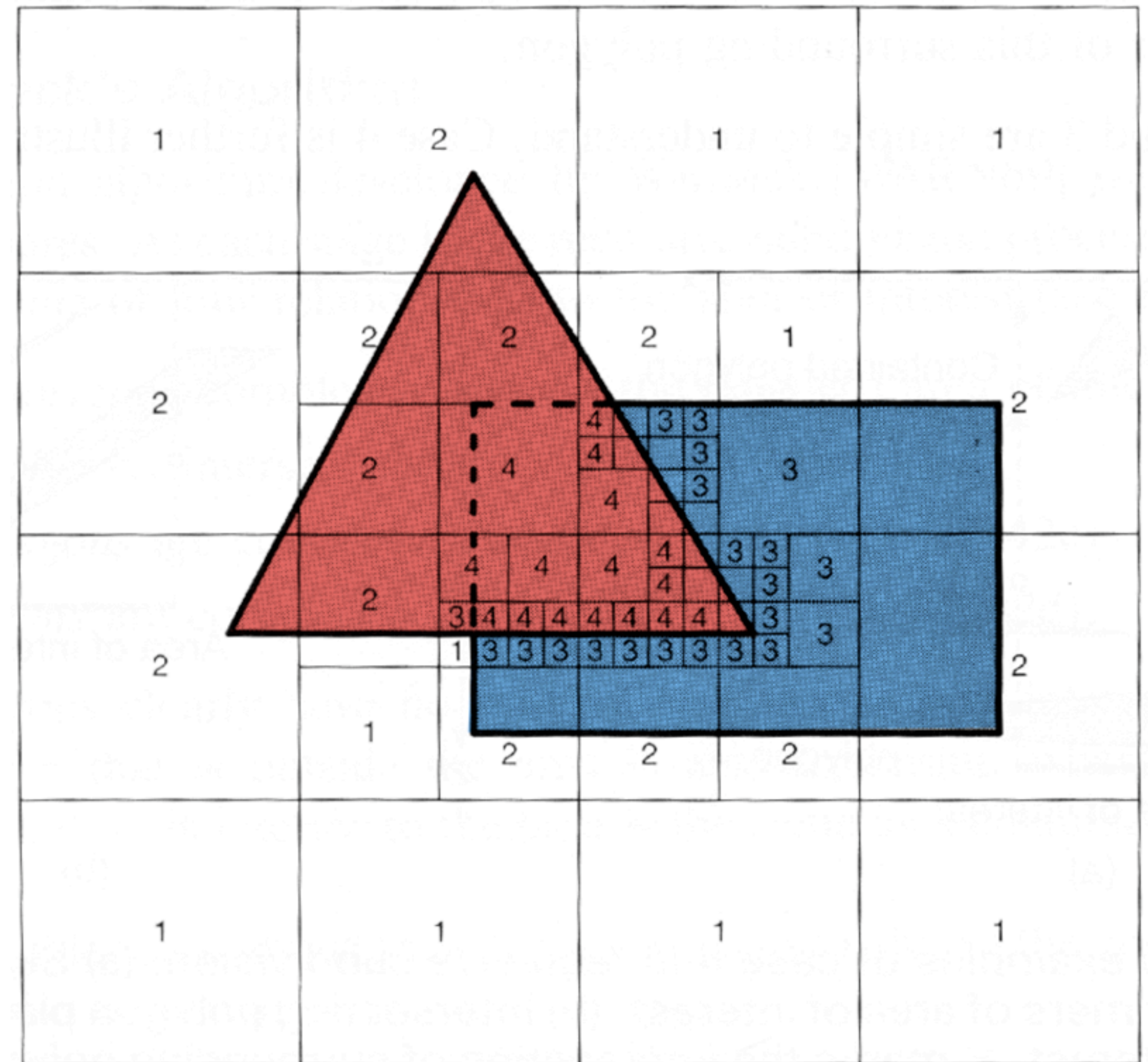
polygon =      ... the current polygon for this pixel is p

proceed with shading current pixel with polygon *polygon*



# Buckets

- Along the edges of polygons, the regions must be subdivided repeatedly until the region becomes a single pixel in size. For complex scenes, this algorithm is quite inefficient.
- Along the edges of polygons, the regions must be subdivided repeatedly until the region becomes a single pixel in size. For complex scenes, this algorithm is quite inefficient.



# References

- Computer Graphics With OpenGL, F.S. Hill jr, Prentice Hall (most images from the instructors pack of this book)
- Basic Algebra and Geometry. Ann Hirst and David Singerman. Prentice Hall 2001
- "Essential Mathematics for Computer Graphics fast" John Vince Springer-Verlag London
- "Geometry for Computer Graphics: Formulae, Examples and Proofs" John Vince Springer-Verlag London 2004

# References

- **Computer Graphics: Principles and Practice in C (2nd Edition) (Systems Programming Series) Foley, van Dam et Al**