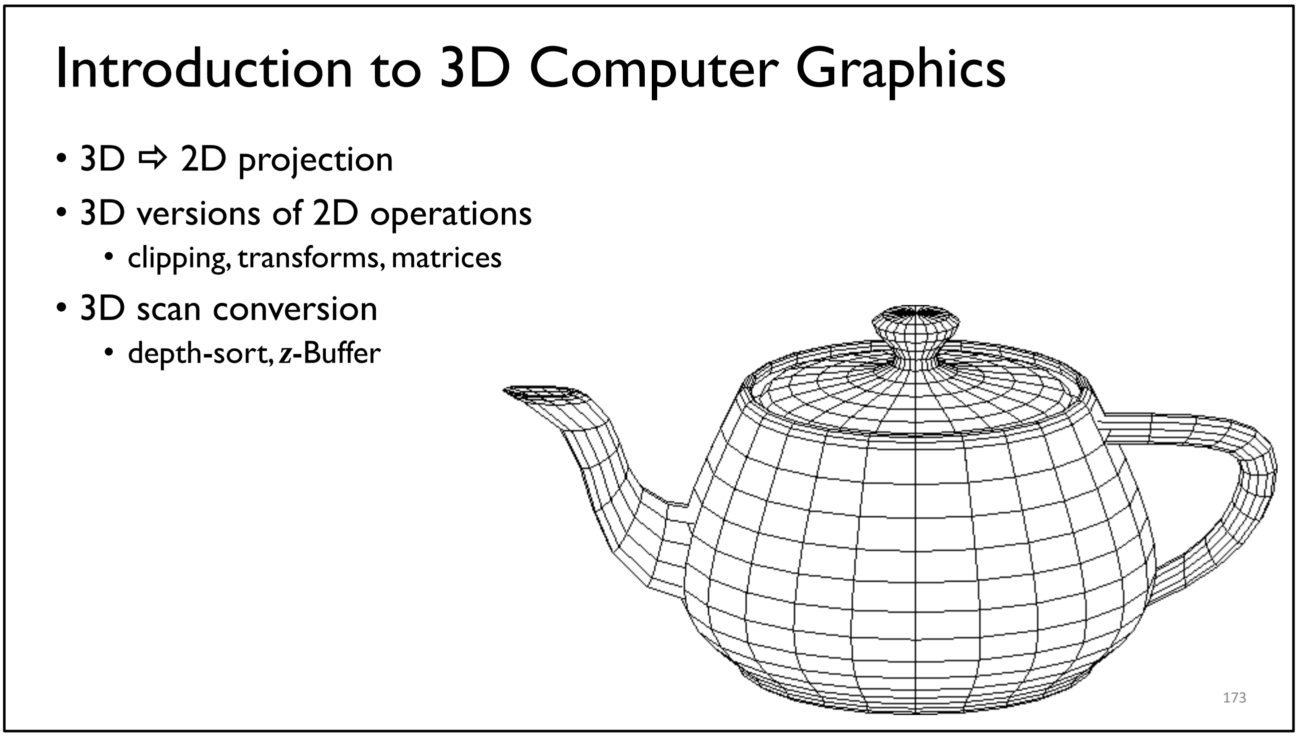
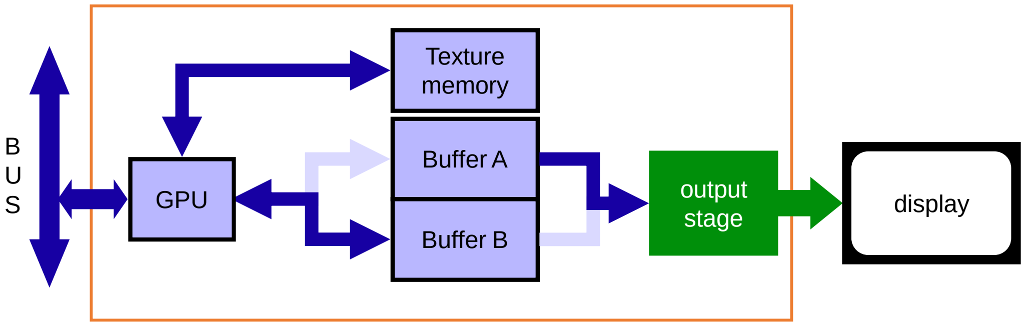


Introduction to 3D Computer Graphics

- 3D ⇨ 2D projection
- 3D versions of 2D operations
  - clipping, transforms, matrices
- 3D scan conversion
  - depth-sort, *z*-Buffer

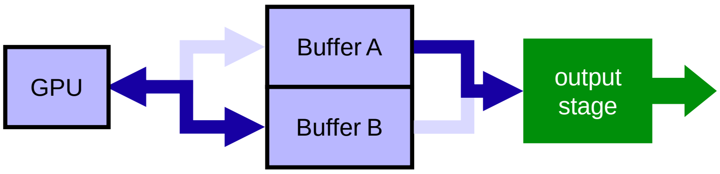
Modern graphics cards

- most graphics processing is done on a separate graphics card
- the CPU communicates primitive data over the bus to the special purpose Graphics Processing Unit (GPU)
- video memory (dual-ported Dynamic RAM (DRAM)) used for storing the image to be output and textures

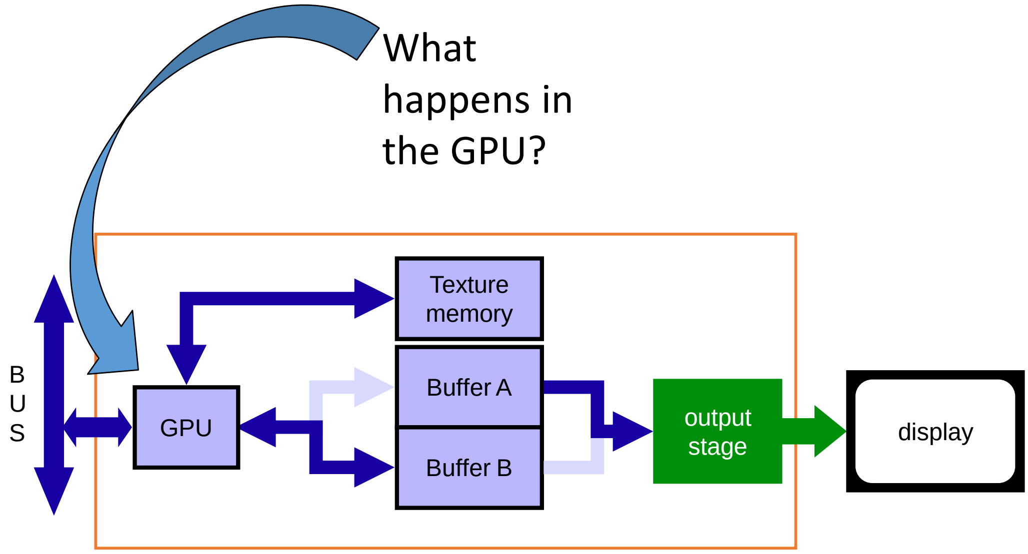


Double buffering

- if we allow the currently displayed image to be updated then we may see bits of the image being displayed halfway through the update
  - this can be visually disturbing, especially if we want the illusion of smooth animation
- double buffering solves this problem: we draw into one frame buffer and display from the other
- when drawing is complete we flip buffers

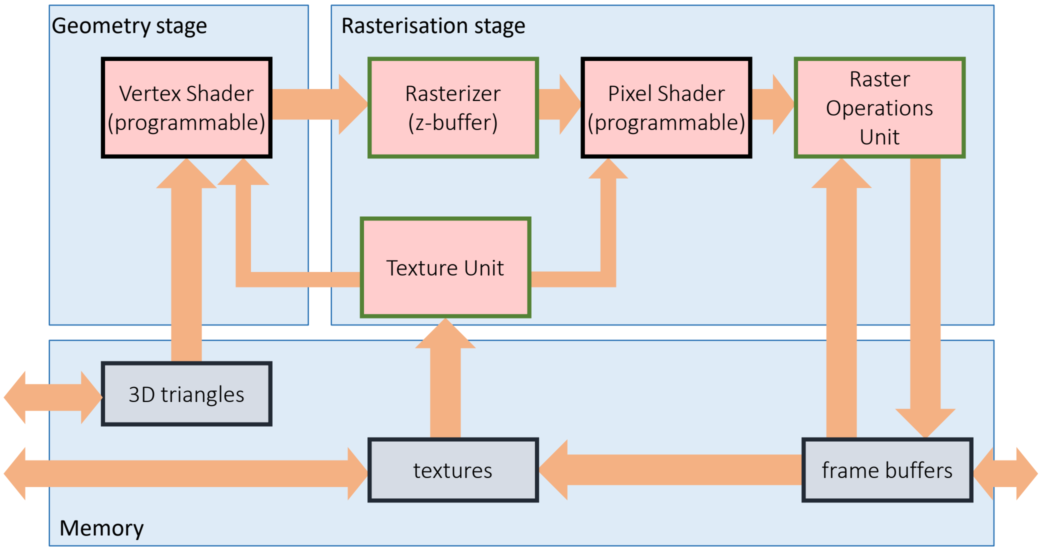


Graphics card architecture



What happens in the GPU?

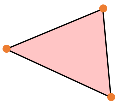
A graphics card architecture

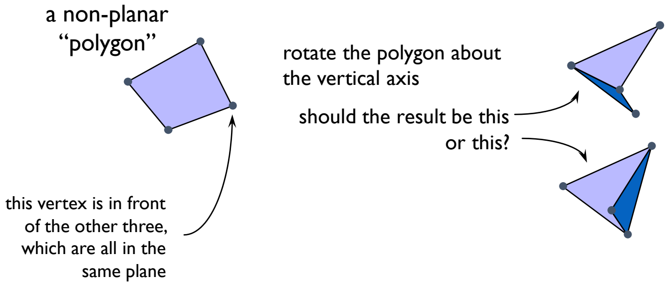


based on nVIDIA's GeForce 6 architecture

Surfaces in 3D: polygons

- 3 vertices (triangle) must be planar
- > 3 vertices, not necessarily planar





a non-planar “polygon”

this vertex is in front of the other three, which are all in the same plane

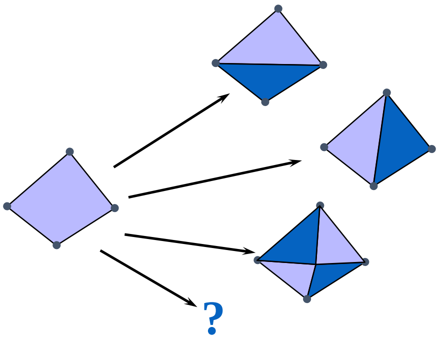
rotate the polygon about the vertical axis

should the result be this

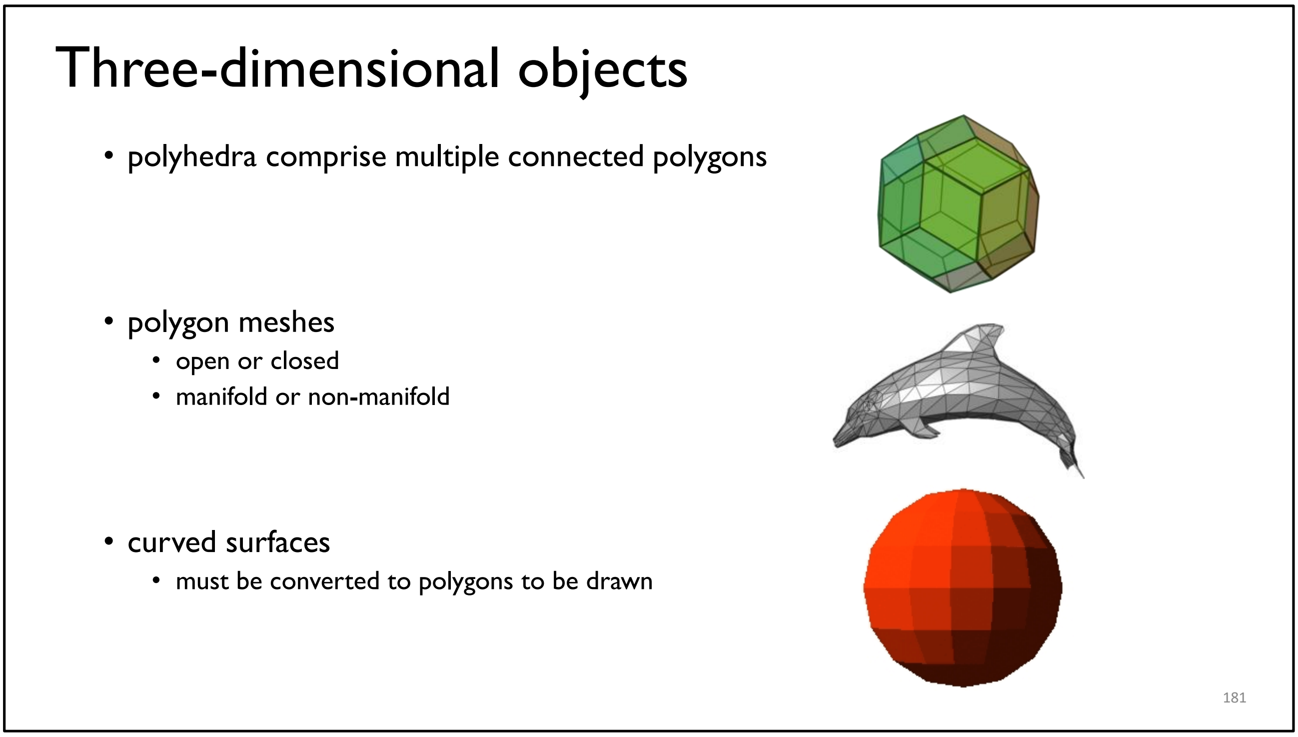
or this?

Splitting polygons into triangles

- some graphics processors accept only triangles
- an arbitrary polygon with more than three vertices isn't guaranteed to be planar; a triangle is

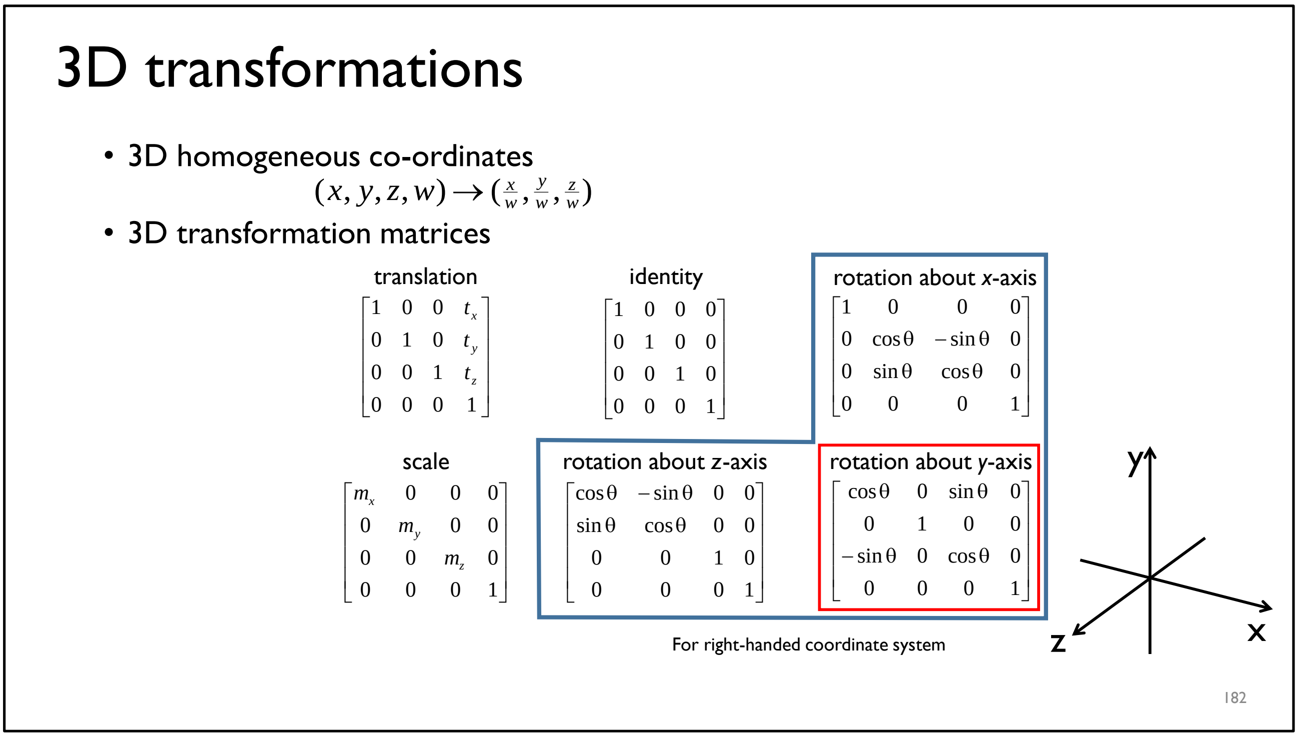


which is preferable?



Three-dimensional objects

- polyhedra comprise multiple connected polygons
- polygon meshes
  - open or closed
  - manifold or non-manifold
- curved surfaces
  - must be converted to polygons to be drawn



3D transformations

- 3D homogeneous co-ordinates

  $$(x, y, z, w) \rightarrow (\tfrac{x}{w}, \tfrac{y}{w}, \tfrac{z}{w})$$

- 3D transformation matrices

translation

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

identity

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about *x*-axis

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

scale

$$\begin{bmatrix} m_x & 0 & 0 & 0 \\ 0 & m_y & 0 & 0 \\ 0 & 0 & m_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about *z*-axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

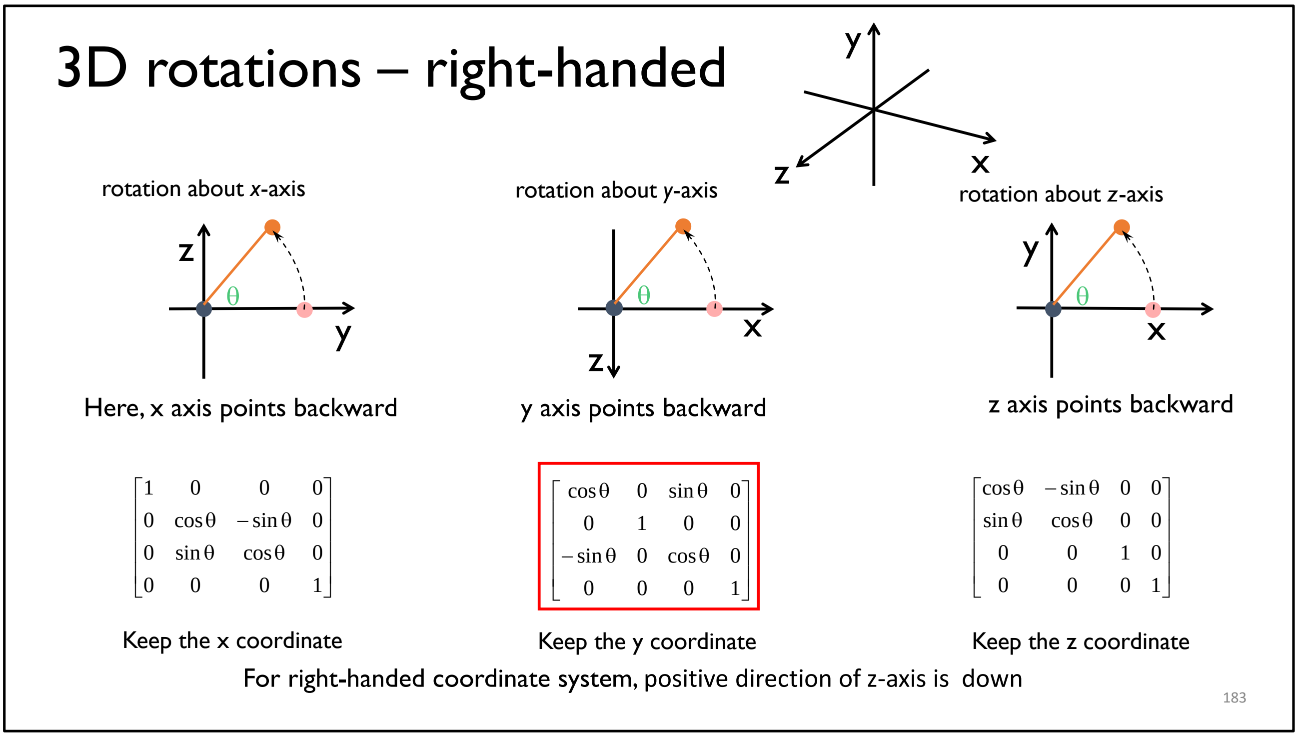
rotation about *y*-axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For right-handed coordinate system

In any 2D coordinate system when you choose two of xyz-axis, if you want a counter-clockwise rotation angle theta, for y-z (rotate about x-axis, means rotate about the origin in the yz-plane) and x-y, you will find the relationship between the two axis is the same with how we normally define a 2D coordinate system, where when the first axis is pointing right, the second should be pointing up.

However, in a right-handed 3D system, for x-z plane, (when we want to rotate about the y-axis), you can see when x is poiting right, z's positive direction is pointing down.



3D rotations – right-handed

rotation about *x*-axis

Here, x axis points backward

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Keep the x coordinate

rotation about *y*-axis

y axis points backward

$$\boxed{\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}$$

Keep the y coordinate

rotation about z-axis

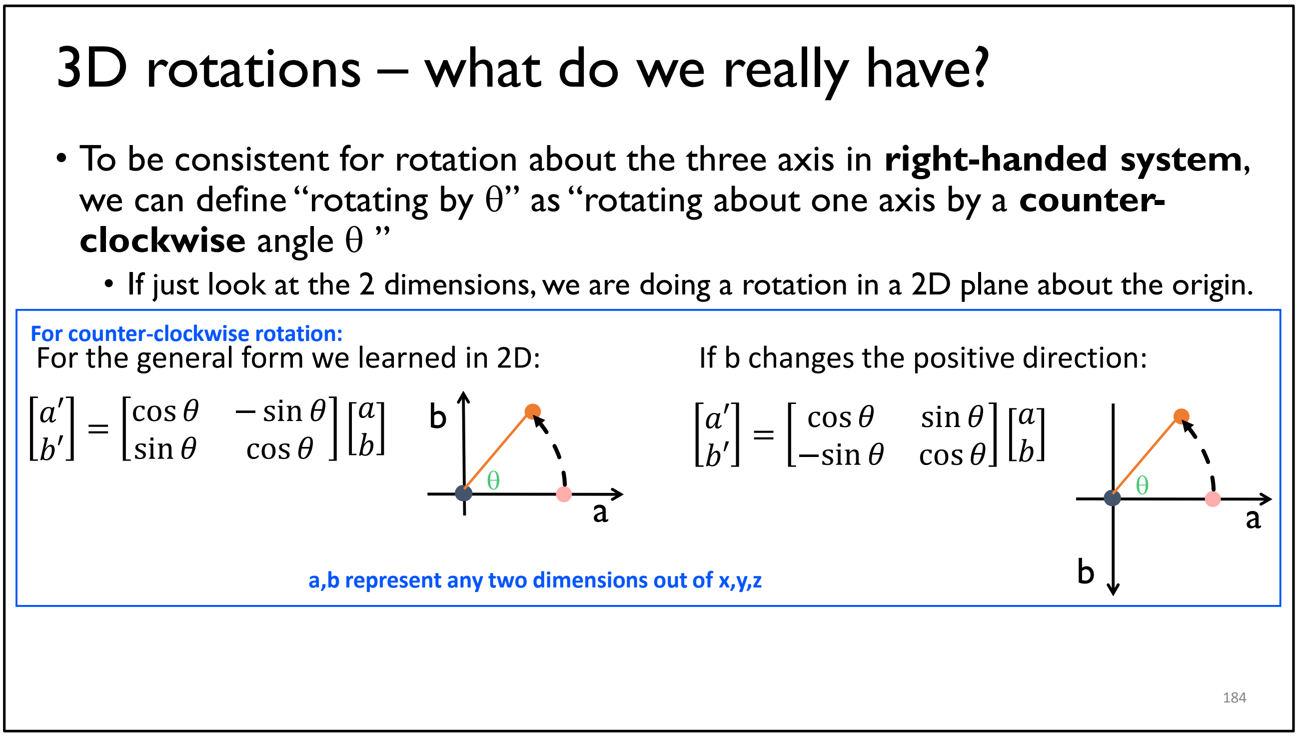
z axis points backward

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Keep the z coordinate

For right-handed coordinate system, positive direction of z-axis is down

X is forward, O is backward



3D rotations – what do we really have?

- To be consistent for rotation about the three axis in **right-handed system**, we can define "rotating by θ" as "rotating about one axis by a **counter-clockwise** angle θ"
  - If just look at the 2 dimensions, we are doing a rotation in a 2D plane about the origin.

For counter-clockwise rotation:

For the general form we learned in 2D:

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

If b changes the positive direction:

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

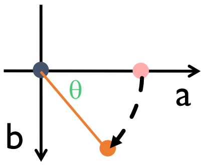
a,b represent any two dimensions out of x,y,z

Sin(-a) = -sin(a)

3D rotations – what do we really have?

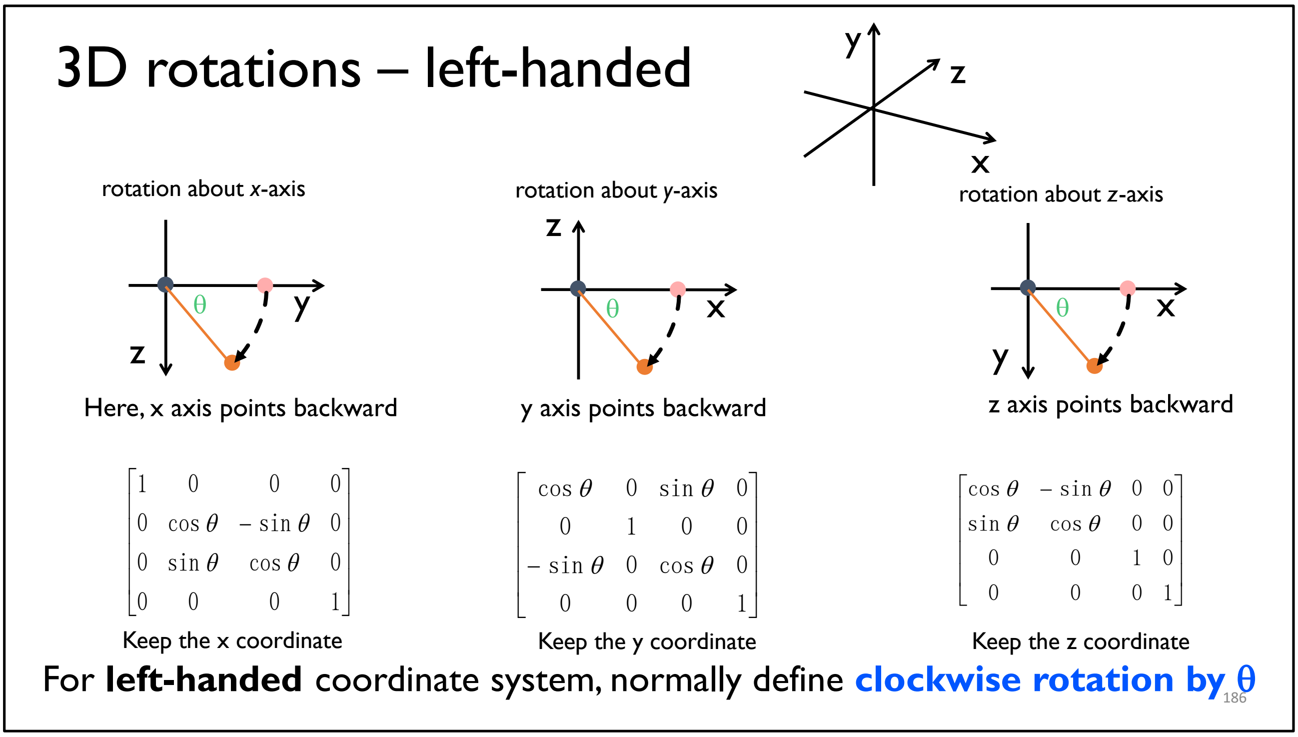
- But if we use the **same** rotation matrix no matter where the second axis points, we get a general form to rotate from one axis to another axis:

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$



We actually rotate by θ from the positive direction of the first axis to the positive direction of the second axis

Sin(-a) = -sin(a)
Cos(-a) = cos(a)
Trigonometry



3D rotations – left-handed

rotation about *x*-axis

Here, x axis points backward

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Keep the x coordinate

rotation about *y*-axis

y axis points backward

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Keep the y coordinate

rotation about z-axis

z axis points backward

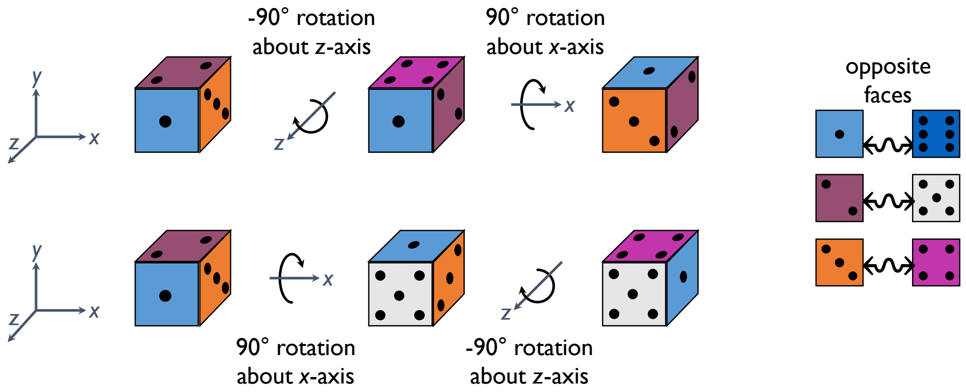
$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Keep the z coordinate

For **left-handed** coordinate system, normally define **clockwise rotation by θ**

Test your understanding in left-handed coordinate system.

3D transformations are not commutative



-90° rotation
about *z*-axis

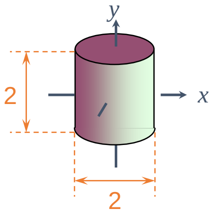
90° rotation
about *x*-axis

opposite
faces

90° rotation
about *x*-axis

-90° rotation
about z-axis

A transformation example I

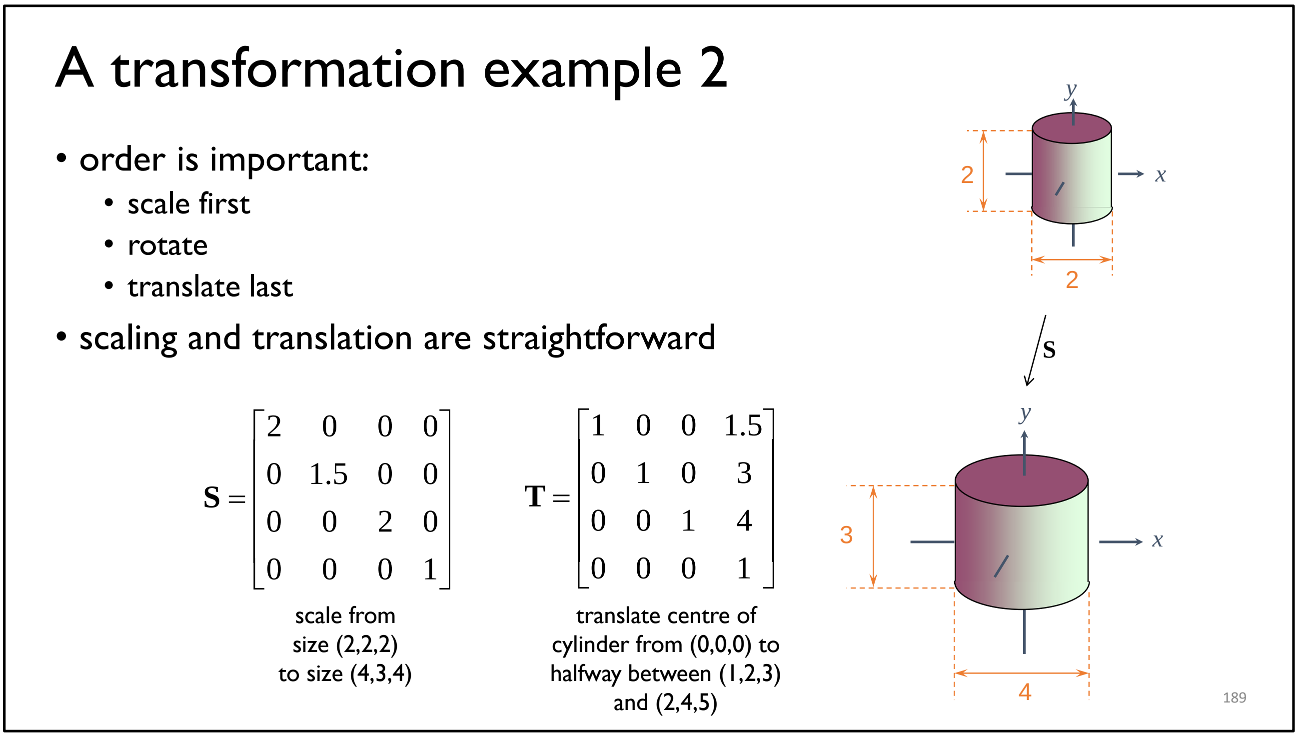


- the graphics package Open Inventor defines a cylinder to be:
  - **centre at the origin, (0,0,0)**
  - **radius 1 unit**
  - **height 2 units, aligned along the *y*-axis**

- this is the only cylinder that can be drawn, *but* the package has a complete set of 3D transformations

- we want to draw a cylinder of:
  - **radius 2 units**
  - **the centres of its two ends located at (1,2,3) and (2,4,5)**
    - its length is thus 3 units

- what transforms are required? and in what order should they be applied?



A transformation example 2

- order is important:
  - scale first
  - rotate
  - translate last
- scaling and translation are straightforward

$$\mathbf{S} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

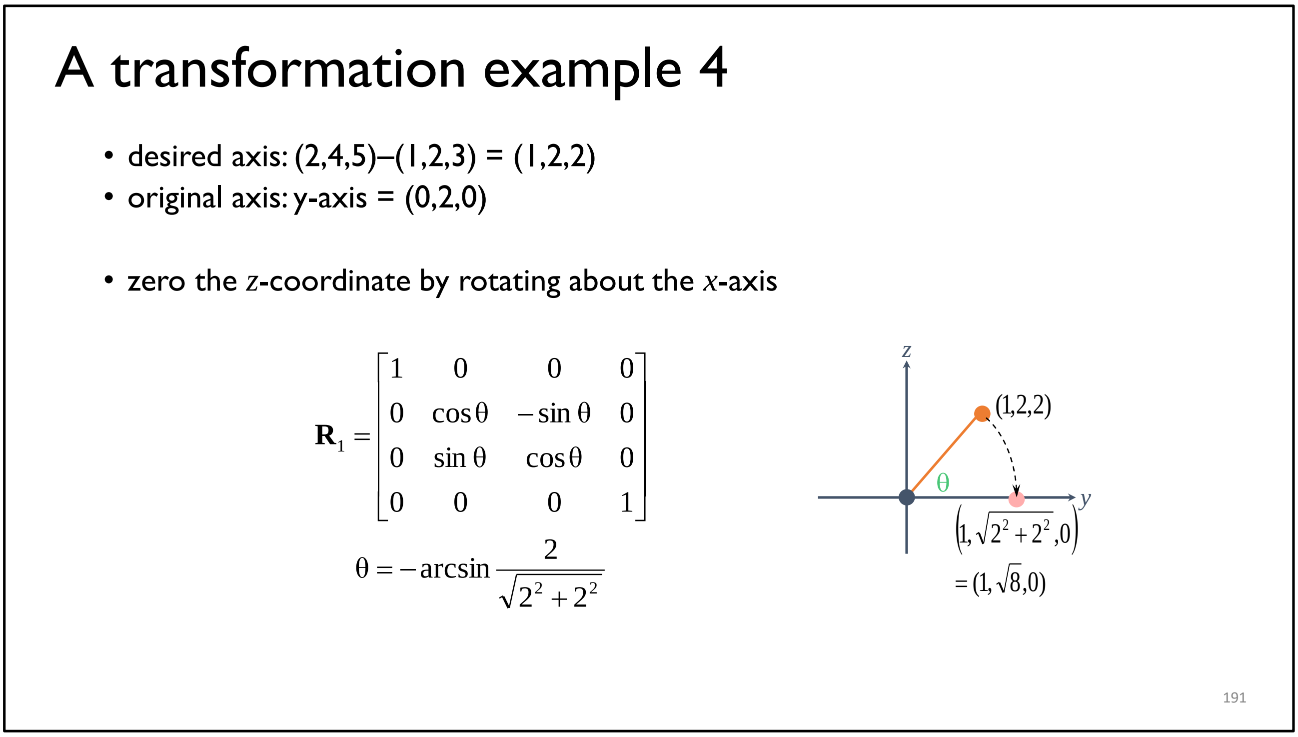
scale from size (2,2,2) to size (4,3,4)

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 1.5 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate centre of cylinder from (0,0,0) to halfway between (1,2,3) and (2,4,5)

A transformation example 3

- rotation is a multi-step process
  - break the rotation into steps, each of which is rotation about a principal axis
  - work these out by taking the desired orientation back to the original axis-aligned position
      - **the centres of its two ends located at (1,2,3) and (2,4,5)**
  - desired axis: (2,4,5)–(1,2,3) = (1,2,2)
  - original axis: y-axis = (0,1,0) – (0,-1,0) = (0,2,0)



A transformation example 4

- desired axis: (2,4,5)–(1,2,3) = (1,2,2)
- original axis: y-axis = (0,2,0)

- zero the z-coordinate by rotating about the x-axis

$$\mathbf{R}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

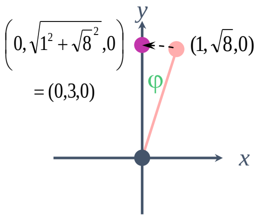
$$\theta = -\arcsin\frac{2}{\sqrt{2^2+2^2}}$$

A transformation example 5

- then zero the x-coordinate by rotating about the z-axis
- we now have the object's axis pointing along the y-axis

$$\mathbf{R}_2 = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\varphi = \arcsin \frac{1}{\sqrt{1^2 + \sqrt{8}^2}}$$



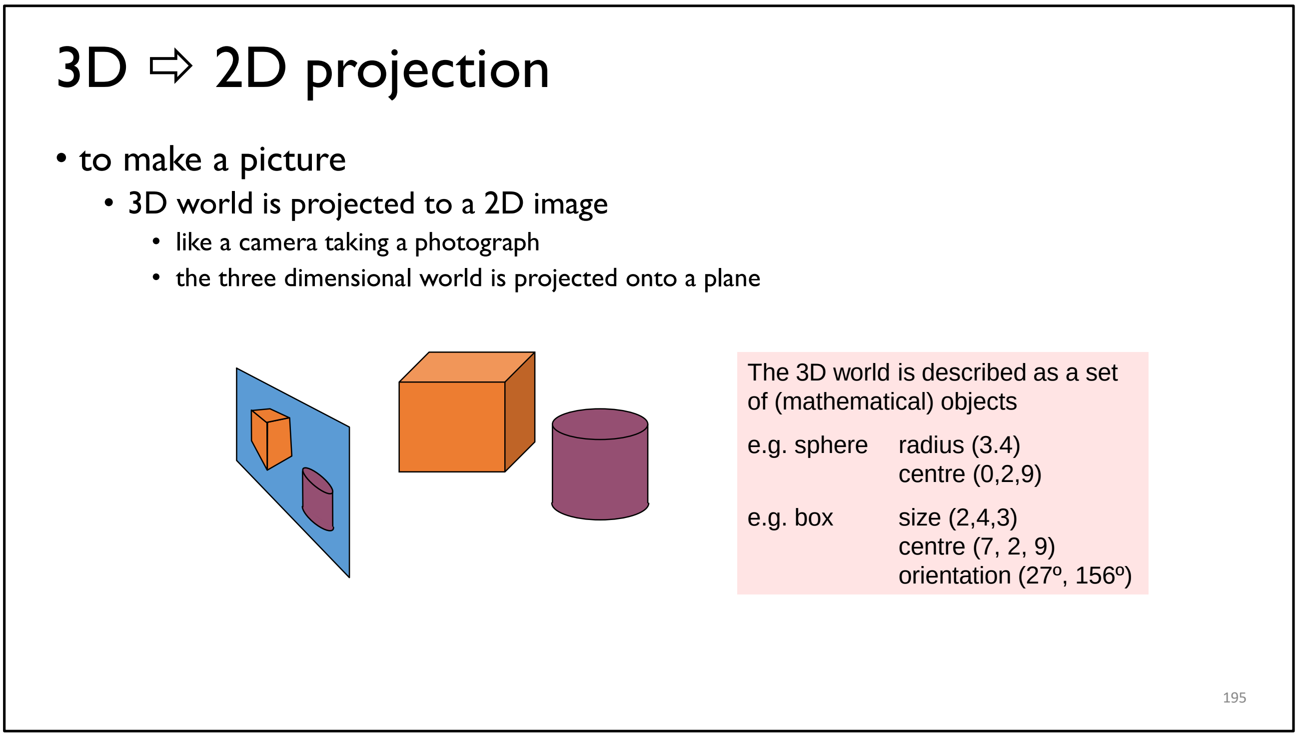
A transformation example 6

- the overall transformation is:
  - first scale
  - then take the inverse of the rotation we just calculated
  - finally translate to the correct position

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{T} \times \mathbf{R}_1^{-1} \times \mathbf{R}_2^{-1} \times \mathbf{S} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Application: display multiple instances

- transformations allow you to define an object at one location and then place multiple instances in your scene



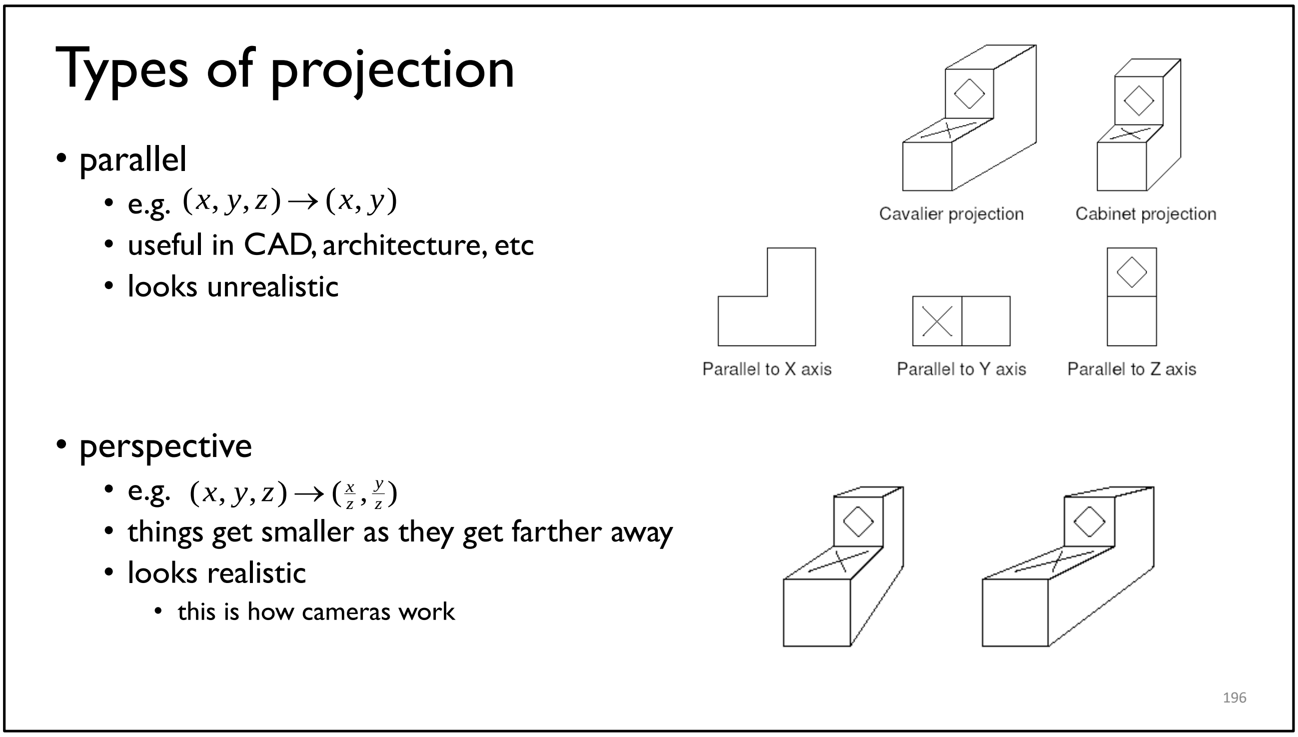
3D ⇨ 2D projection

- to make a picture
  - 3D world is projected to a 2D image
    - like a camera taking a photograph
    - the three dimensional world is projected onto a plane

The 3D world is described as a set of (mathematical) objects

e.g. sphere	radius (3.4) centre (0,2,9)
e.g. box	size (2,4,3) centre (7, 2, 9) orientation (27º, 156º)

Line of sight intersect with a point on an object



Types of projection

- parallel
  - e.g. $(x, y, z) \to (x, y)$
  - useful in CAD, architecture, etc
  - looks unrealistic

- perspective
  - e.g. $(x, y, z) \to (\frac{x}{z}, \frac{y}{z})$
  - things get smaller as they get farther away
  - looks realistic
    - this is how cameras work

Normally, when we talk about projection, we mean projecting to a plane which is parallel to x-y plane, perpendicular to z axis

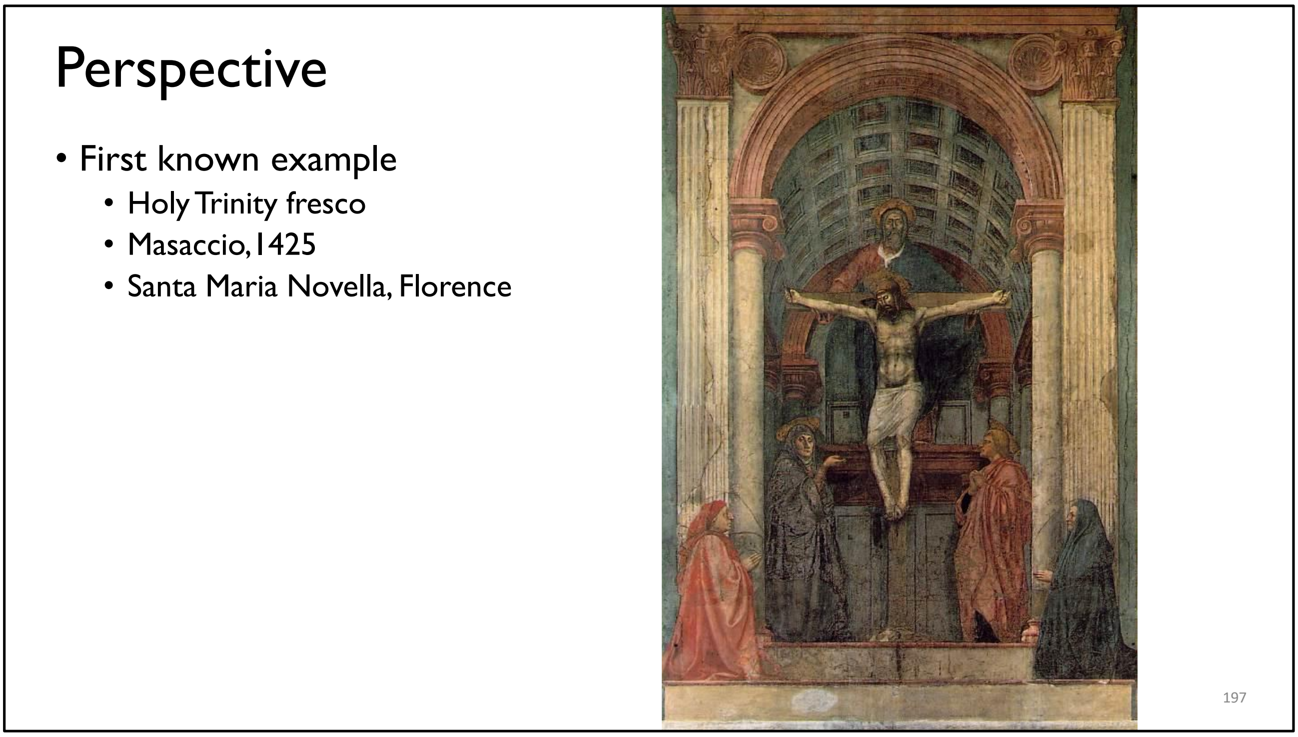
Parallel projection include both oblique projection and orthographic projection, parallel lines of the source object produce parallel lines in the projected image.

Parallel projection is the way we draw stereo geometry in high school. It is also used in modern CAD designing software. Because they need to check the accurate distance from the different views.

Oblique projection is a type of parallel projection:
it projects an image by intersecting parallel rays (projectors), try to depict the 3 dimensional information.
from the three-dimensional source object with the drawing surface (projection plane).
Oblique projection is commonly used in technical drawing. The cavalier

projection was used by French military artists in the 18th century to depict fortifications.

Like **cavalier** perspective, one face of the **projected** object is parallel to the viewing plane, and the third axis is **projected** as going off at an angle (typically 63.4°). Unlike **cavalier projection**, where the third axis keeps its length, with cabinet **projection** the length of the receding lines is cut in half.



Perspective

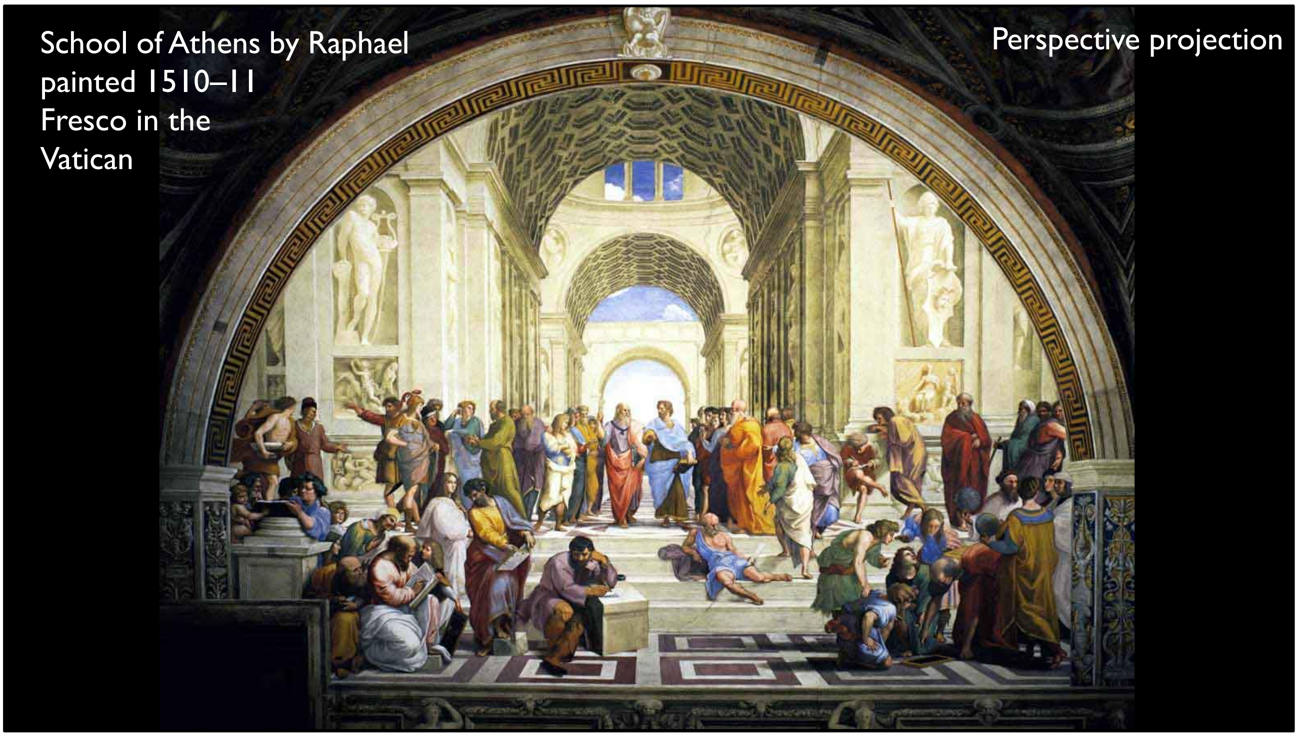
- First known example
  - Holy Trinity fresco
  - Masaccio, 1425
  - Santa Maria Novella, Florence

God’s gift of Christ on the cross.

Lines converge to point at infinity level with the viewer’s eye.

When it was executed, no actual coffered barrel vault had been constructed since the Romans.

Plus Mary, St John and kneeling donors outside the frame of the picture.



School of Athens by Raphael
painted 1510–11
Fresco in the
Vatican

Perspective projection

Perspective projection examples

Gates Building – the rounded version
(Stanford University)

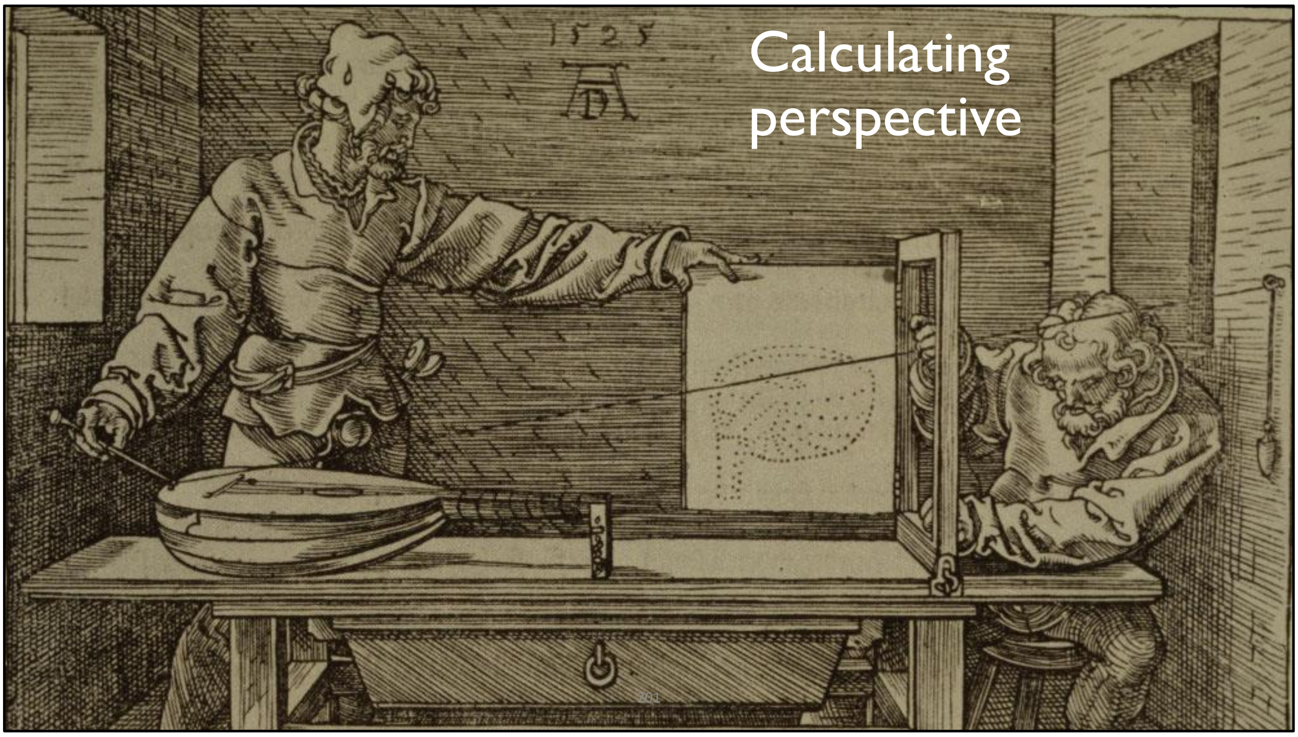
Gates Building – the rectilinear version
(University of Cambridge)

False perspective

MC Escher’s 1961 lithograph Waterfall
Escher Haus in the Hague

Ames Room in the City of Sciences in Paris

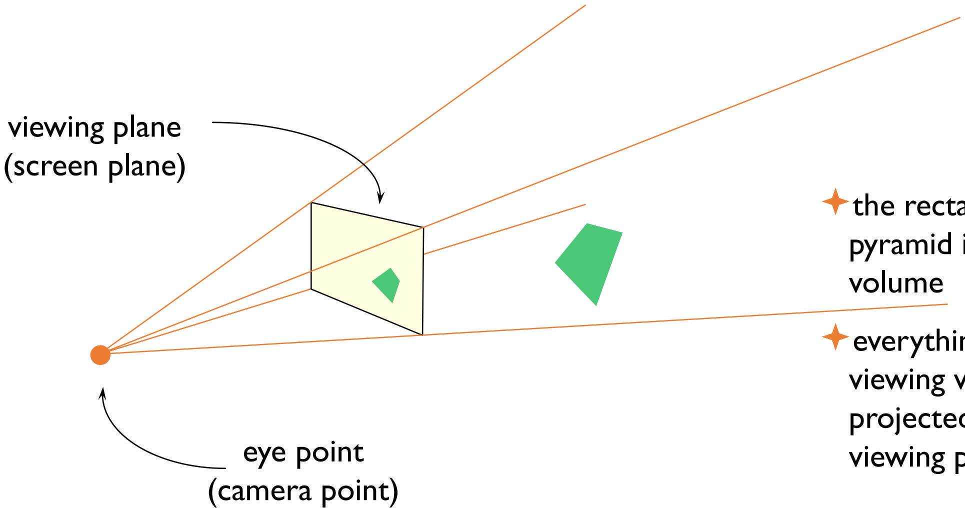
Calculating perspective



Albrecht Dürer’s 1525 woodcut ‘Man drawing a Lute’

A **lute** is any plucked string instrument with a neck (either fretted or unfretted)
Metropolitan Museum of Art in New York

Viewing volume

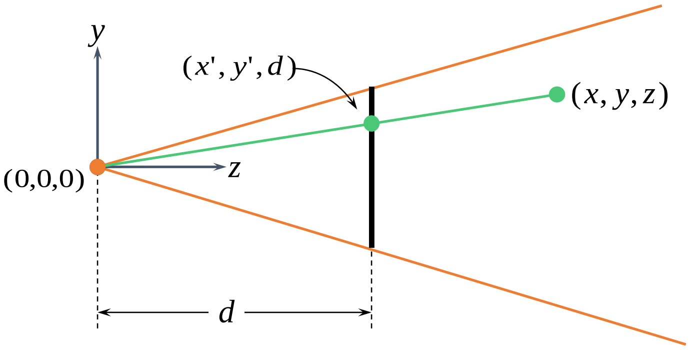


viewing plane
(screen plane)

eye point
(camera point)

- the rectangular pyramid is the viewing volume
- everything within the viewing volume is projected onto the viewing plane

Geometry of perspective projection



$$x' = x\frac{d}{z}$$

$$y' = y\frac{d}{z}$$

Projection as a matrix operation

$$\begin{bmatrix} x \\ y \\ 1/d \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$x' = x\frac{d}{z}$$

$$y' = y\frac{d}{z}$$

remember $\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$

This is useful in the z-buffer algorithm where we need to interpolate $1/z$ values rather than z values.

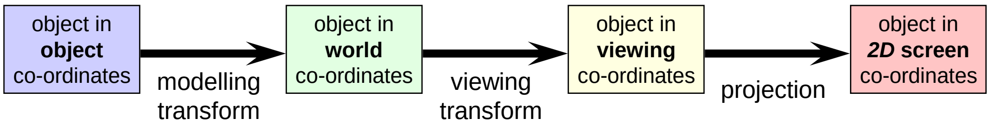
$$z' = \frac{1}{z}$$

Perspective projection with an arbitrary camera

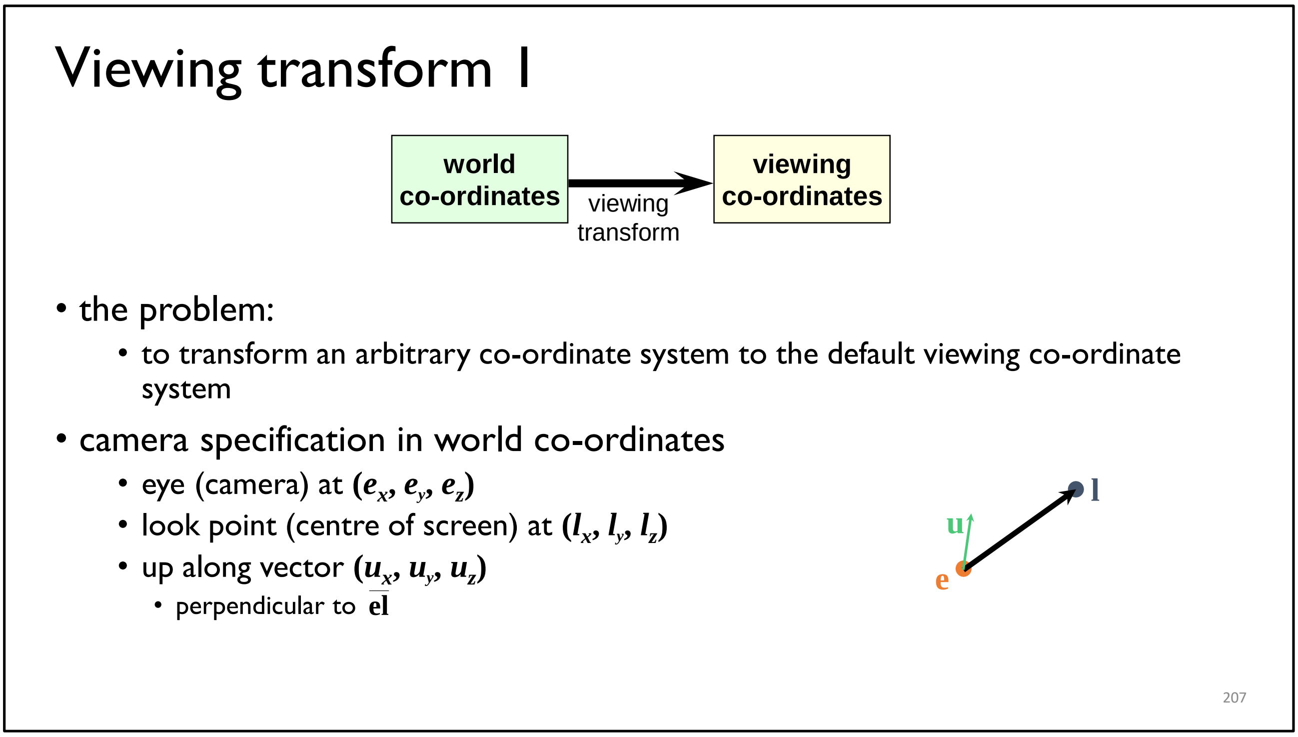
- we have assumed that:
  - screen centre at **(0,0,d)**
  - screen parallel to ***xy***-plane
  - ***z***-axis into screen
  - ***y***-axis up and ***x***-axis to the right
  - eye (camera) at origin **(0,0,0)**

- for an arbitrary camera we can either:
  - work out equations for projecting objects about an arbitrary point onto an arbitrary plane
  - transform all objects into our standard co-ordinate system (viewing co-ordinates) and use the above assumptions

A variety of transformations



- the modelling transform and viewing transform can be multiplied together to produce a single matrix taking an object directly from object co-ordinates into viewing co-ordinates
- either or both of the modelling transform and viewing transform matrices can be the identity matrix
  - e.g. objects can be specified directly in viewing co-ordinates, or directly in world co-ordinates
- this is a useful set of transforms, not a hard and fast model of how things should be done



Viewing transform 1

- the problem:
  - to transform an arbitrary co-ordinate system to the default viewing co-ordinate system
- camera specification in world co-ordinates
  - eye (camera) at (e_x, e_y, e_z)
  - look point (centre of screen) at (l_x, l_y, l_z)
  - up along vector (u_x, u_y, u_z)
    - perpendicular to $\overline{\mathbf{el}}$

Before, we talked about how to transform some vector into another vector, or to the new positions. But now we are talking about transform a whole coordinate system, which means that these transformations can be performed to change the coordinates of any given points from one system to another.
A lucky thing is that these transformations are the same for any position if you find them.
We have to find something we have already know what's the coordinates after transformations.

Viewing transform 2

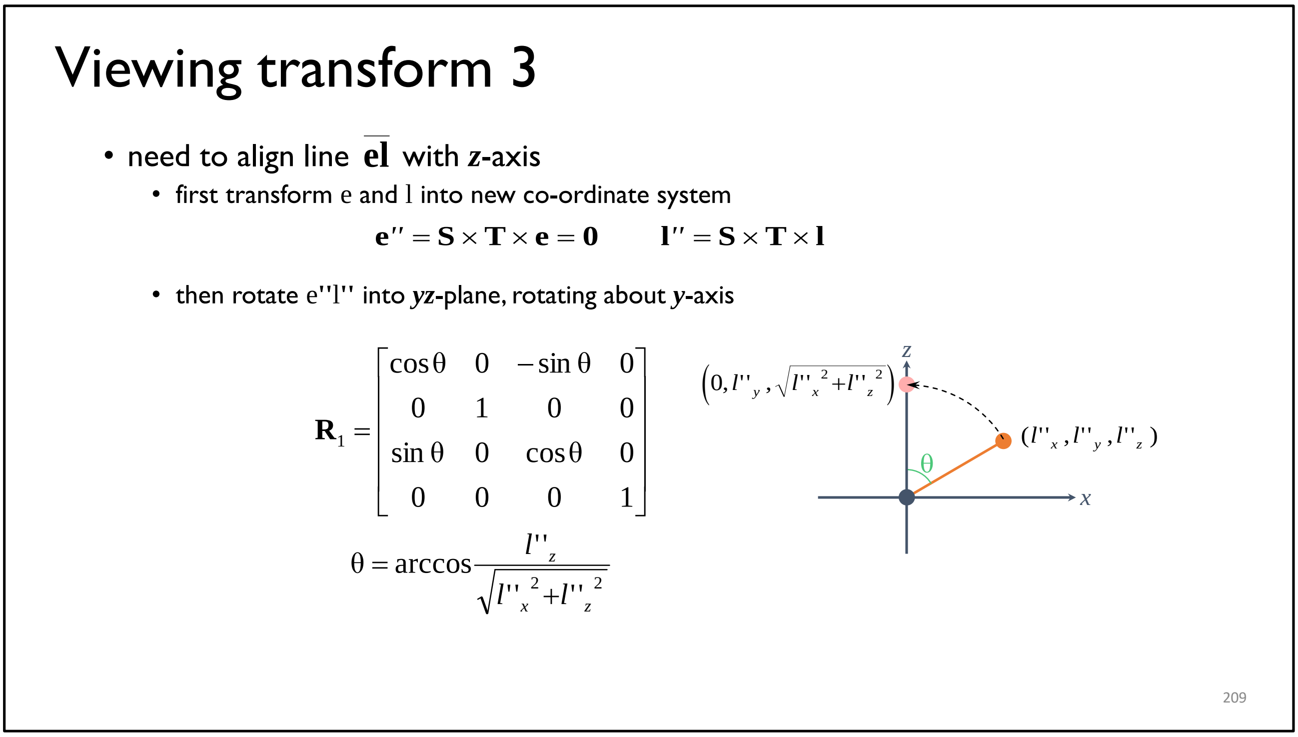
- translate eye point, $(\boldsymbol{e_x},\boldsymbol{e_y},\boldsymbol{e_z})$, to origin, $\mathbf{(0,0,0)}$

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- scale so that eye point to look point distance, $\left|\overline{\mathbf{el}}\right|$, is distance from origin to screen centre, $\boldsymbol{d}$

$$\left|\overline{\mathbf{el}}\right| = \sqrt{(l_x - e_x)^2 + (l_y - e_y)^2 + (l_z - e_z)^2}$$

$$\mathbf{S} = \begin{bmatrix} d/\left|\overline{\mathbf{el}}\right| & 0 & 0 & 0 \\ 0 & d/\left|\overline{\mathbf{el}}\right| & 0 & 0 \\ 0 & 0 & d/\left|\overline{\mathbf{el}}\right| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Viewing transform 3

- need to align line $\overline{\mathbf{el}}$ with $\boldsymbol{z}$-axis
  - first transform e and l into new co-ordinate system

$$\mathbf{e}'' = \mathbf{S} \times \mathbf{T} \times \mathbf{e} = \mathbf{0} \qquad \mathbf{l}'' = \mathbf{S} \times \mathbf{T} \times \mathbf{l}$$

  - then rotate $\mathrm{e''l''}$ into $\boldsymbol{yz}$-plane, rotating about $\boldsymbol{y}$-axis

$$\mathbf{R}_1 = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\theta = \arccos\frac{l''_z}{\sqrt{l''^{\,2}_x + l''^{\,2}_z}}$$

Crooked

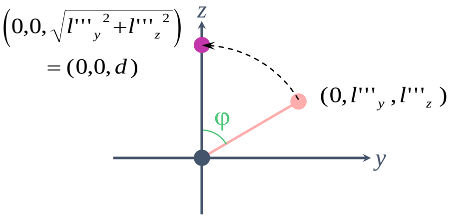
Viewing transform 4

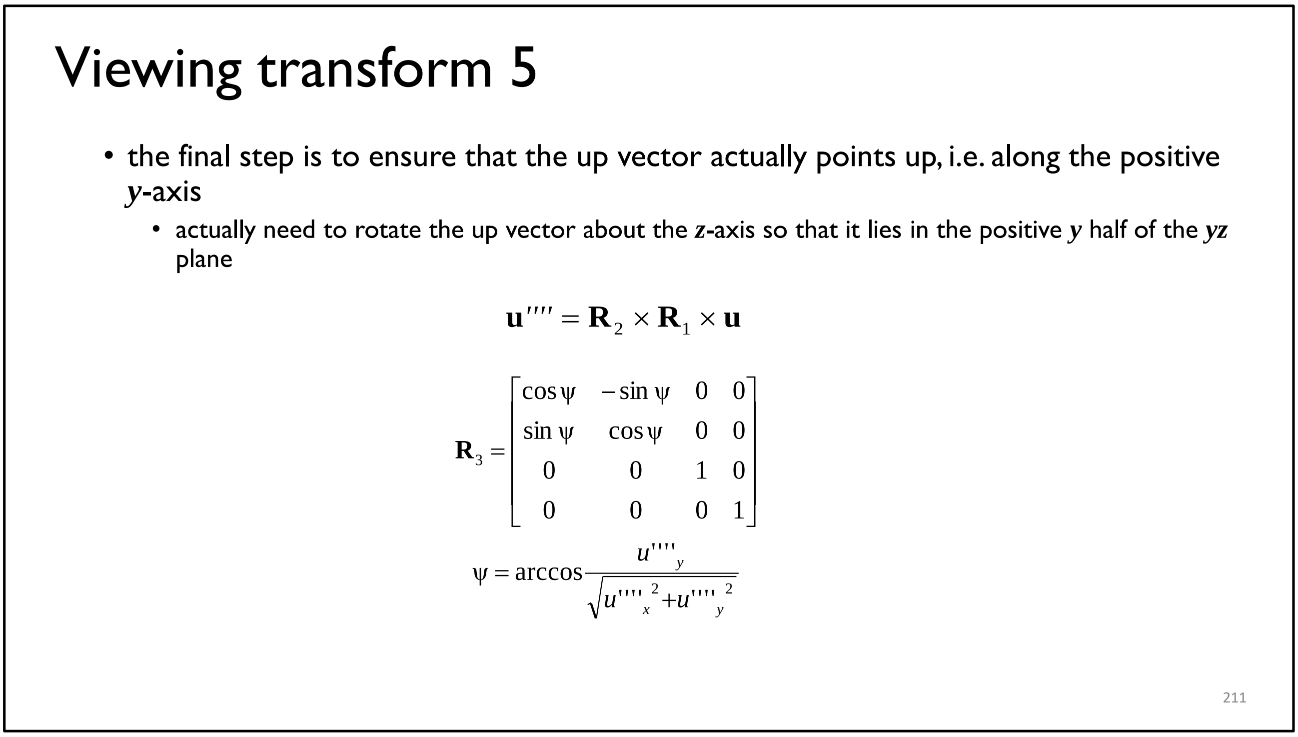
- having rotated the viewing vector onto the *yz* plane, rotate it about the *x*-axis so that it aligns with the *z*-axis

$$\mathbf{l}''' = \mathbf{R}_1 \times \mathbf{l}''$$

$$\mathbf{R}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\varphi = \arccos \frac{l'''_z}{\sqrt{l'''_y{}^2 + l'''_z{}^2}}$$





Viewing transform 5

- the final step is to ensure that the up vector actually points up, i.e. along the positive ***y***-axis
  - actually need to rotate the up vector about the ***z***-axis so that it lies in the positive ***y*** half of the ***yz*** plane

$$\mathbf{u}'''' = \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{u}$$

$$\mathbf{R}_3 = \begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\psi = \arccos \frac{u''''_y}{\sqrt{u''''^2_x + u''''^2_y}}$$

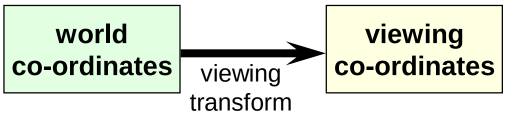
why don't we need to multiply $\mathbf{u}$ by $\mathbf{S}$ or $\mathbf{T}$?

$\mathbf{u}$ is a vector rather than a point, all we care about is its direction

Translating a vector makes no difference to its direction

Scaling makes no difference to its direction, so long as the scaling is the same in all dimensions

Viewing transform 6



- we can now transform any point in world co-ordinates to the equivalent point in viewing co-ordinate

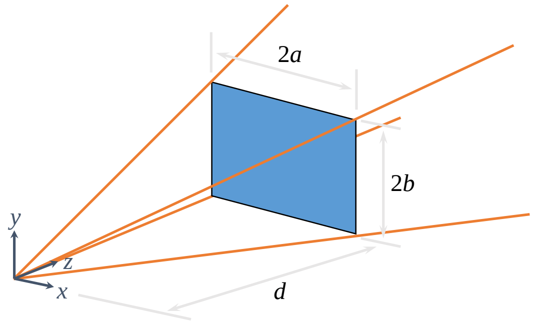
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- in particular: $\mathbf{e} \to (0,0,0) \quad \mathbf{l} \to (0,0,d)$
- the matrices depend only on e, l, and u, so they can be pre-multiplied together

$$\mathbf{M} = \mathbf{R}_3 \times \mathbf{R}_2 \times \mathbf{R}_1 \times \mathbf{S} \times \mathbf{T}$$

Clipping in 3D

- clipping against a volume in viewing co-ordinates



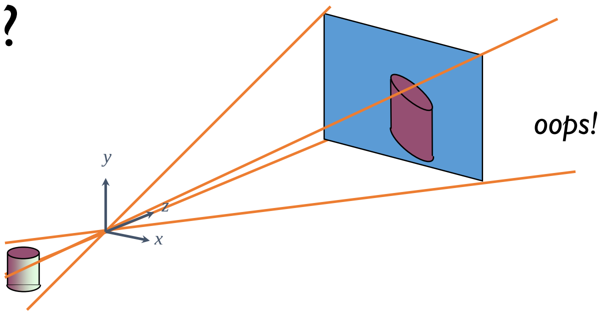
a point (x,y,z) can be clipped against the pyramid by checking it against four planes:

$$x > -z\frac{a}{d} \qquad x < z\frac{a}{d}$$

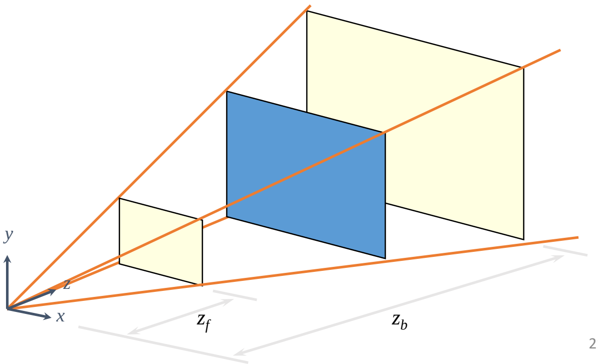
$$y > -z\frac{b}{d} \qquad y < z\frac{b}{d}$$

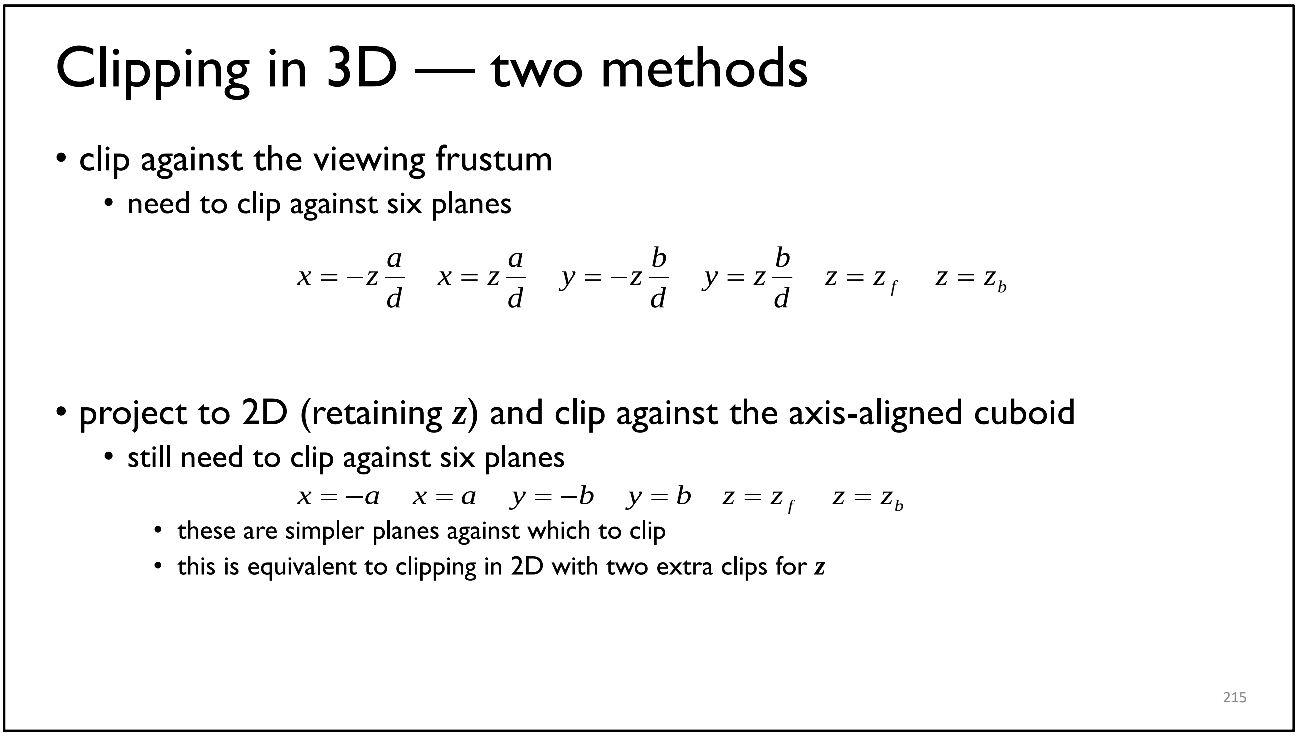
What about clipping in z?

- need to at least check for $z < 0$ to stop things behind the camera from projecting onto the screen



- can also have front and back clipping planes:
  $z > z_f$ and $z < z_b$
  - resulting clipping volume is called the *viewing frustum*





Clipping in 3D — two methods

- clip against the viewing frustum
  - need to clip against six planes

$$x = -z\frac{a}{d} \quad x = z\frac{a}{d} \quad y = -z\frac{b}{d} \quad y = z\frac{b}{d} \quad z = z_f \quad z = z_b$$

- project to 2D (retaining z) and clip against the axis-aligned cuboid
  - still need to clip against six planes

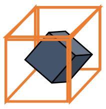
$$x = -a \quad x = a \quad y = -b \quad y = b \quad z = z_f \quad z = z_b$$

    - these are simpler planes against which to clip
    - this is equivalent to clipping in 2D with two extra clips for z

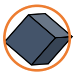
Which is best? It depends on how you implement things. The top version requires those multiplications. The bottom version requires that you do the projection first.

Bounding volumes & clipping

- can be very useful for reducing the amount of work involved in clipping
- what kind of bounding volume?
  - axis aligned box



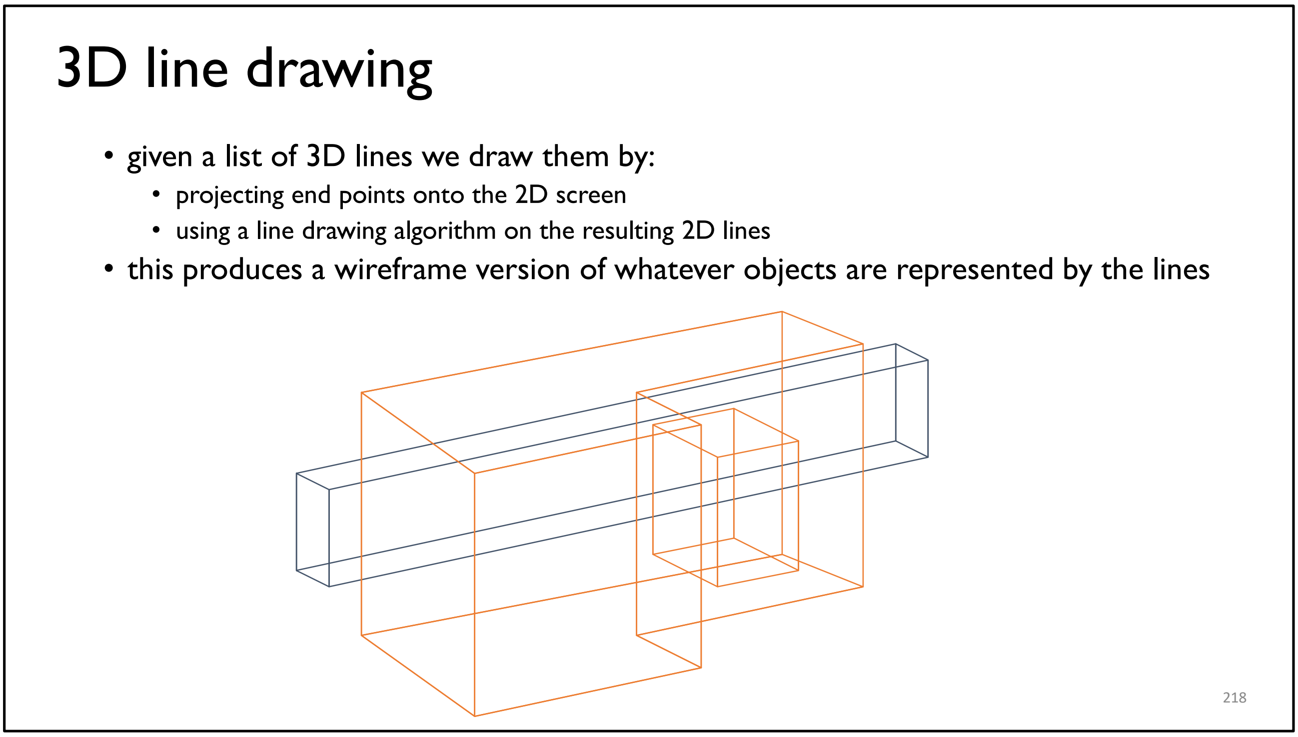
  - sphere



- can have multiple levels of bounding volume

3D scan conversion

- lines
- polygons
  - depth sort
  - Binary space partition tree
  - z-buffer



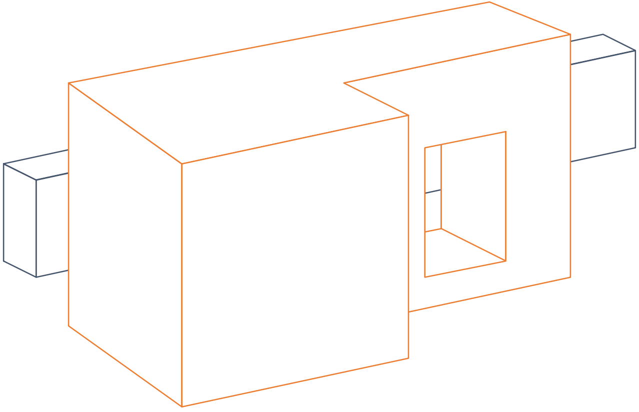
3D line drawing

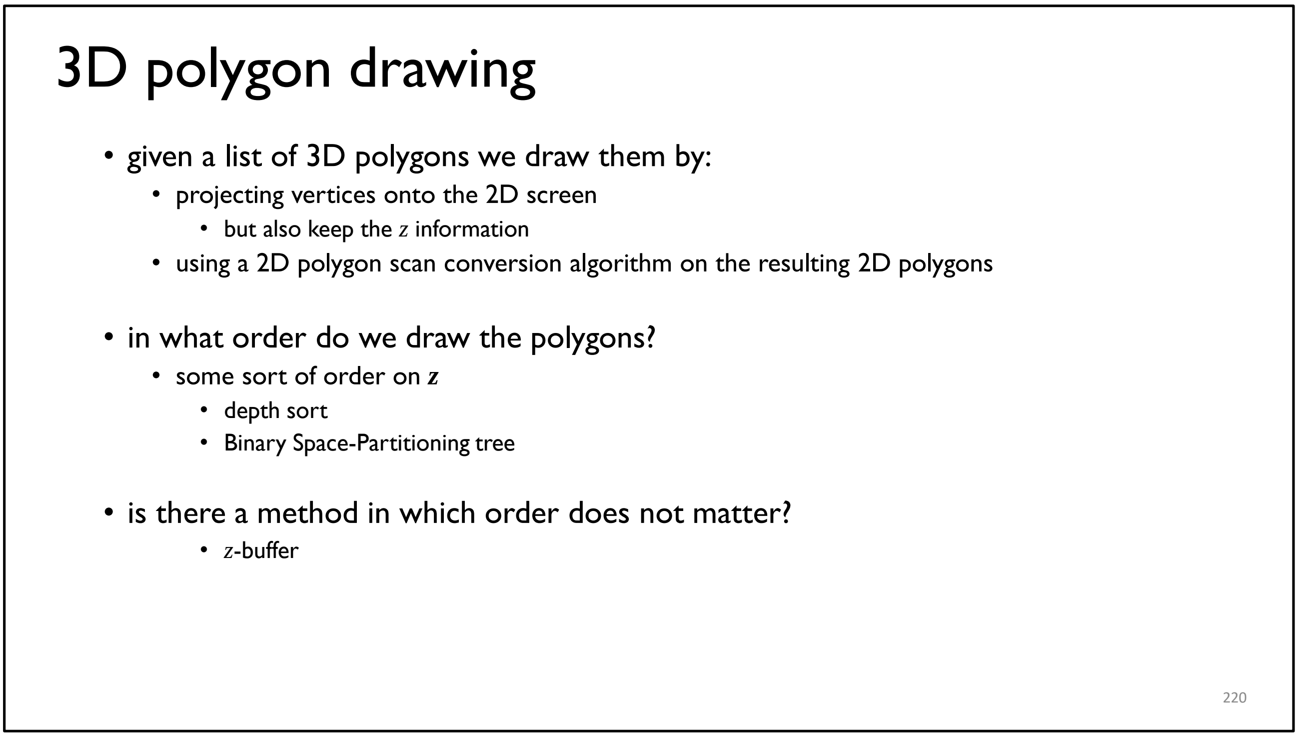
- given a list of 3D lines we draw them by:
  - projecting end points onto the 2D screen
  - using a line drawing algorithm on the resulting 2D lines
- this produces a wireframe version of whatever objects are represented by the lines

So your 2D line drawing algorithms from earlier in the course work perfectly in 3D.

Hidden line removal

- by careful use of cunning algorithms, lines that are hidden by surfaces can be carefully removed from the projected version of the objects
    - still just a line drawing
    - extraordinarily important in the 1960s and 1970s
    - will not be covered further in this course





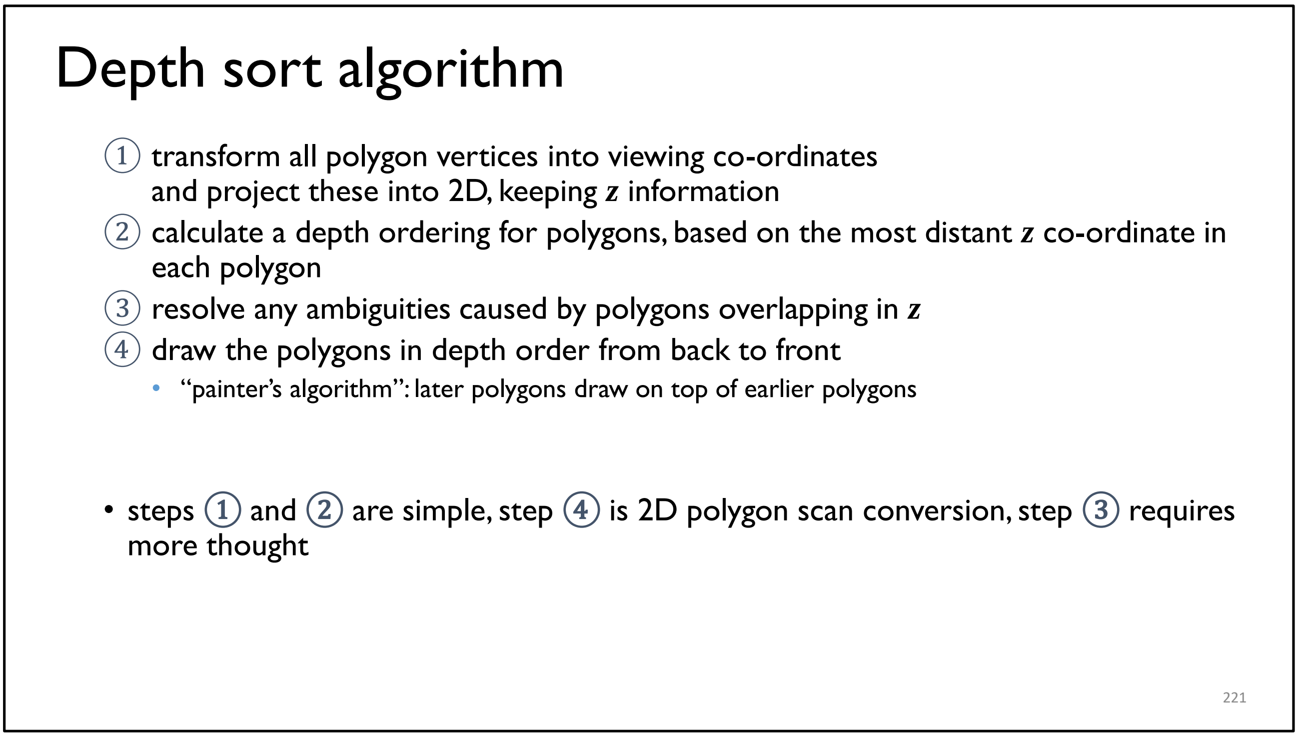
3D polygon drawing

- given a list of 3D polygons we draw them by:
  - projecting vertices onto the 2D screen
    - but also keep the z information
  - using a 2D polygon scan conversion algorithm on the resulting 2D polygons

- in what order do we draw the polygons?
  - some sort of order on $\boldsymbol{z}$
    - depth sort
    - Binary Space-Partitioning tree

- is there a method in which order does not matter?
    - z-buffer

It is not straight forward to get depth order according to the z coordinates. They are not polygons all parallel to image plane.



Depth sort algorithm

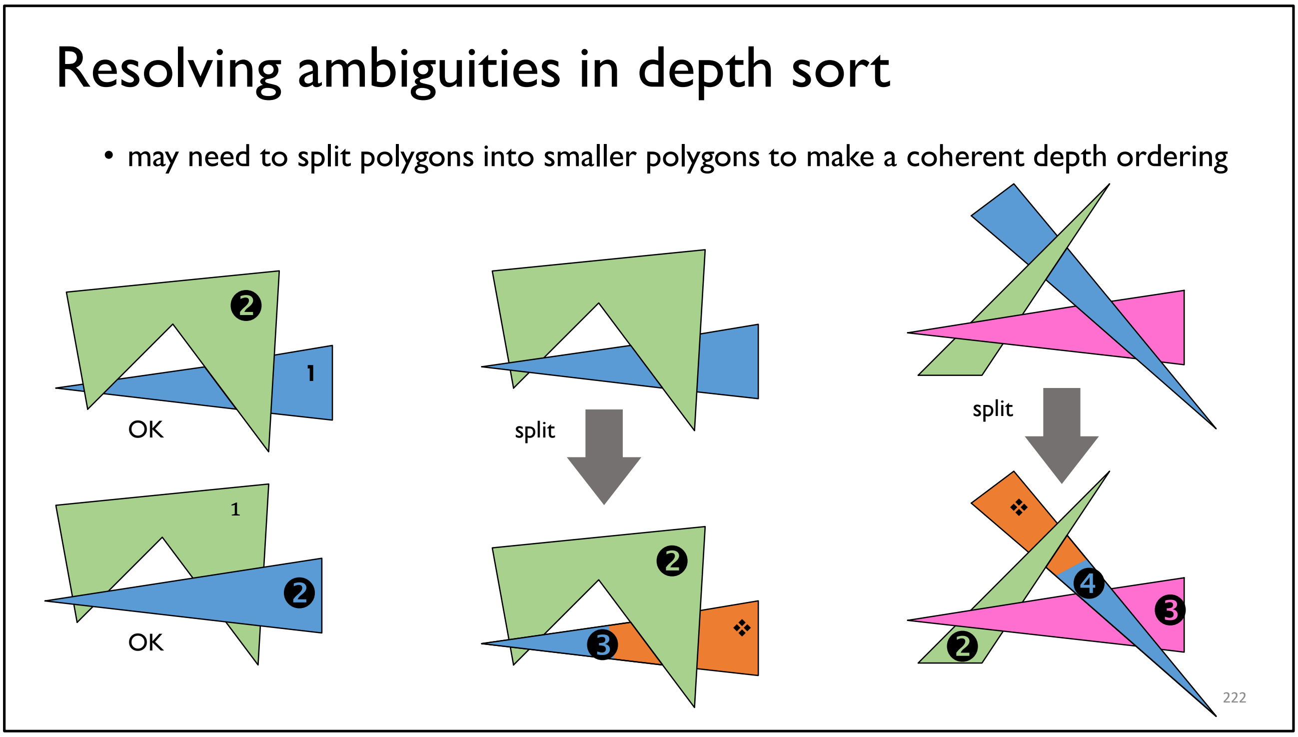
① transform all polygon vertices into viewing co-ordinates
and project these into 2D, keeping z information

② calculate a depth ordering for polygons, based on the most distant z co-ordinate in each polygon

③ resolve any ambiguities caused by polygons overlapping in z

④ draw the polygons in depth order from back to front
  - "painter's algorithm": later polygons draw on top of earlier polygons

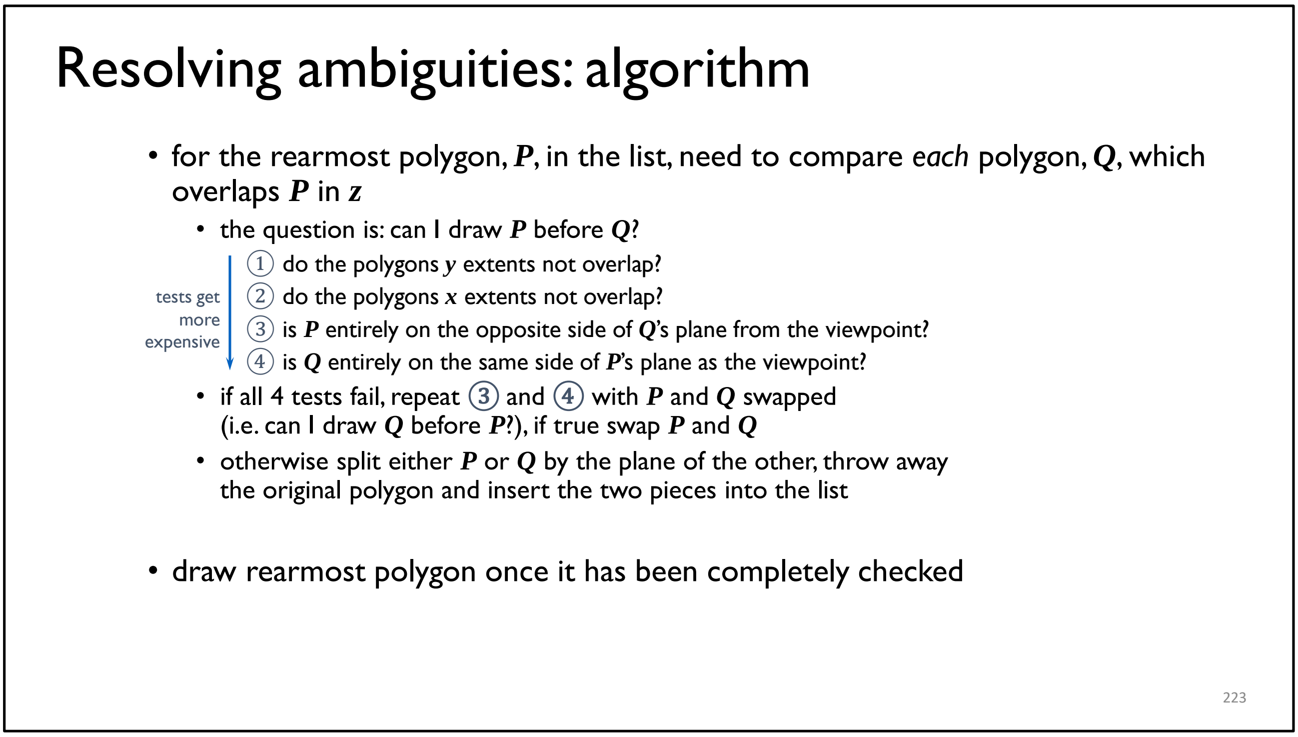
- steps ① and ② are simple, step ④ is 2D polygon scan conversion, step ③ requires more thought



Resolving ambiguities in depth sort

- may need to split polygons into smaller polygons to make a coherent depth ordering

Even when every pair of polygons are OK to get a depth sort, we may also get some conflict when we want to put them in a coherent list concave



Resolving ambiguities: algorithm

- for the rearmost polygon, P, in the list, need to compare *each* polygon, Q, which overlaps P in z
  - the question is: can I draw P before Q?
    - ① do the polygons y extents not overlap?
    - ② do the polygons x extents not overlap?
    - ③ is P entirely on the opposite side of Q's plane from the viewpoint?
    - ④ is Q entirely on the same side of P's plane as the viewpoint?

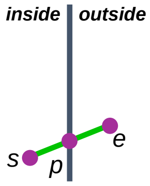
  tests get more expensive

  - if all 4 tests fail, repeat ③ and ④ with P and Q swapped (i.e. can I draw Q before P?), if true swap P and Q
  - otherwise split either P or Q by the plane of the other, throw away the original polygon and insert the two pieces into the list
- draw rearmost polygon once it has been completely checked

This algorithm gets awfully complicated awfully quickly. Is it really the best way to do things?
For small numbers of polygons with few overlaps it is very good. For a modern system with lots of polygons and lots of overlaps it is very expensive.

Robot with the backmost rectangle can fail 3 but pass 4

Split a polygon by a plane



- remember the Sutherland-Hodgman algorithm
  - splits a 2D polygon against a 2D line
- do the same in 3D: split a (planar) polygon by a plane

- line segment defined by (x_s,y_s,z_s) and (x_e,y_e,z_e)
- clipping plane defined by $ax+by+cz+d=0$

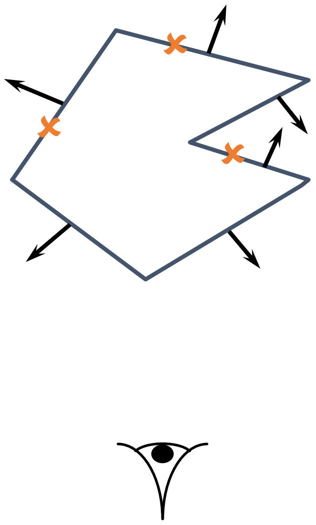
- test to see which side of plane a point is on:
  - $k=ax+by+cz+d$
  - k negative: inside, k positive: outside, $k=0$: on edge
  - apply this test to all vertices of a polygon; if all have the same sign then the polygon is entirely on one side of the plane

Depth sort: comments

- the depth sort algorithm produces a list of polygons which can be scan-converted in 2D, backmost to frontmost, to produce the correct image
- it is cheap for small number of polygons, but becomes rapidly more expensive for large numbers of polygons
- the ordering is only valid from one particular viewpoint

Back face culling: a time-saving trick

- if a polygon is a face of a closed polyhedron *and* faces backwards with respect to the viewpoint *then* it need not be drawn at all because front facing faces would later obscure it anyway
  - saves drawing time at the the cost of one extra test per polygon
  - assumes that we know which way a polygon is oriented
- back face culling can be used in combination with any 3D scan-conversion algorithm



Binary Space-Partitioning trees

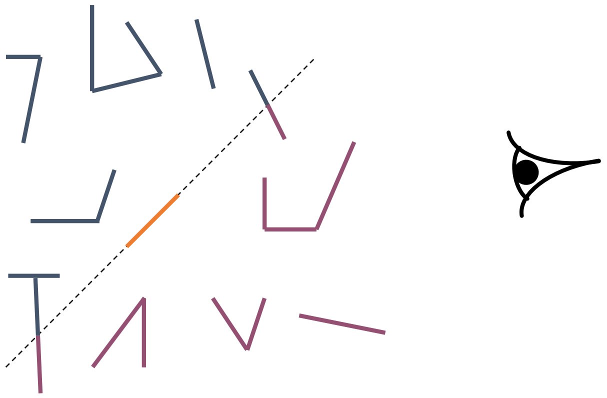
- BSP trees provide a way of quickly calculating the correct depth order:
  - for a collection of static polygons
  - from an arbitrary viewpoint
- the BSP tree trades off an initial time- and space-intensive pre-processing step against a linear display algorithm ($O(N)$) which is executed whenever a new viewpoint is specified
- the BSP tree allows you to easily determine the correct order in which to draw polygons by traversing the tree in a simple way

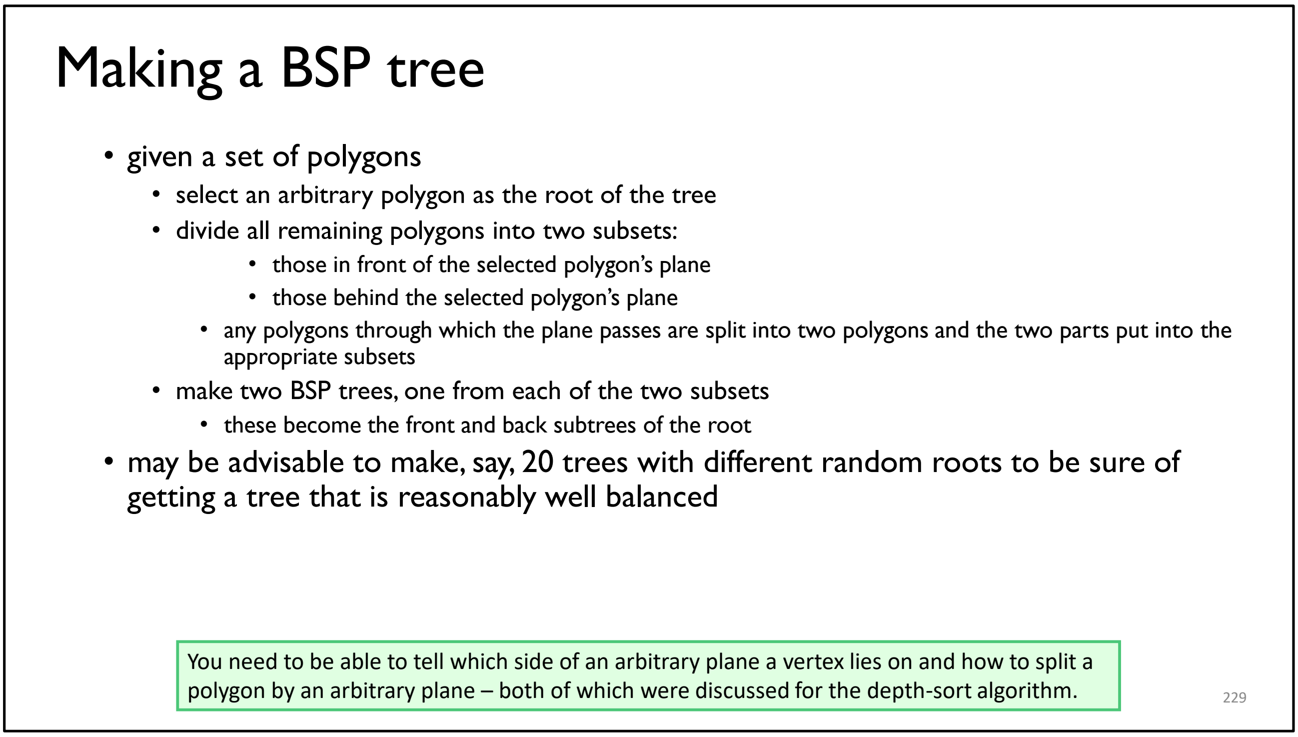
The BSP tree was used in the 1993 game *Doom*

proof by contradiction

BSP tree: basic idea

- a given polygon will be correctly scan-converted if:
  - all polygons on the far side of it from the viewer are scan-converted first
  - then it is scan-converted
  - then all the polygons on the near side of it are scan-converted



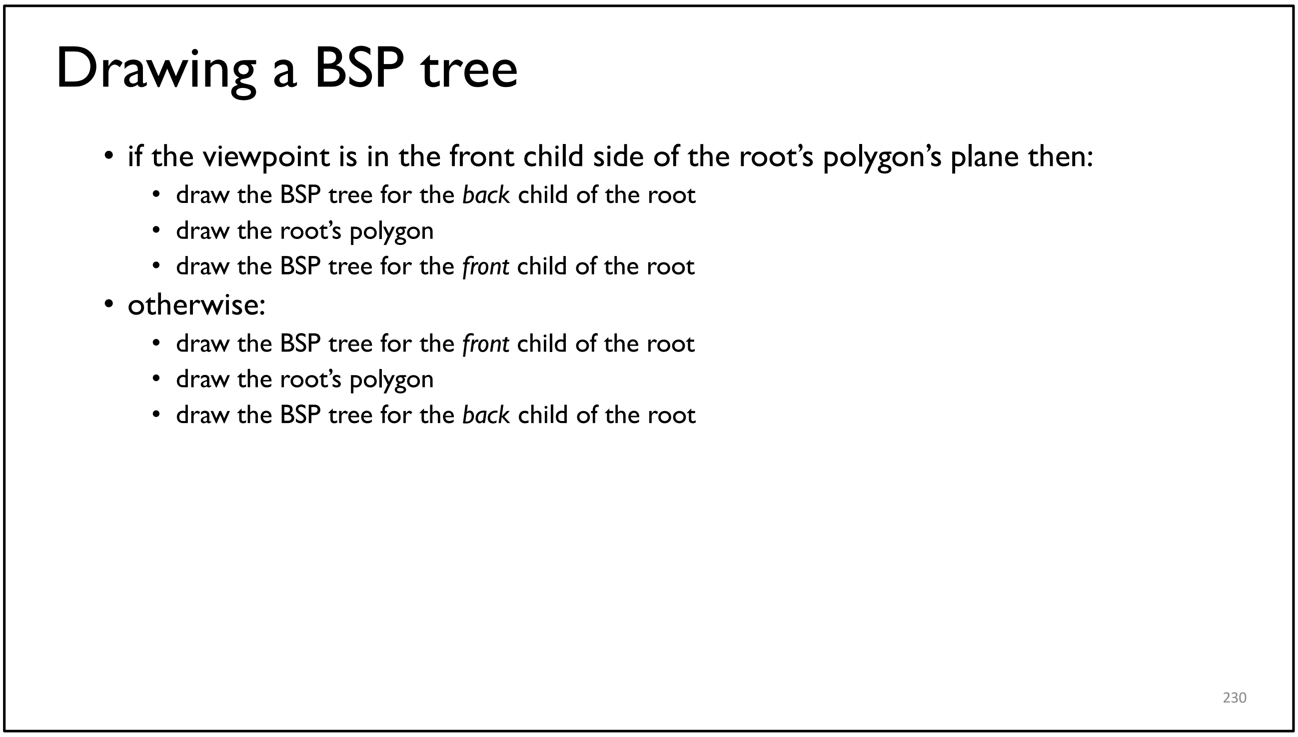


Making a BSP tree

- given a set of polygons
  - select an arbitrary polygon as the root of the tree
  - divide all remaining polygons into two subsets:
      - those in front of the selected polygon's plane
      - those behind the selected polygon's plane
    - any polygons through which the plane passes are split into two polygons and the two parts put into the appropriate subsets
  - make two BSP trees, one from each of the two subsets
    - these become the front and back subtrees of the root
- may be advisable to make, say, 20 trees with different random roots to be sure of getting a tree that is reasonably well balanced

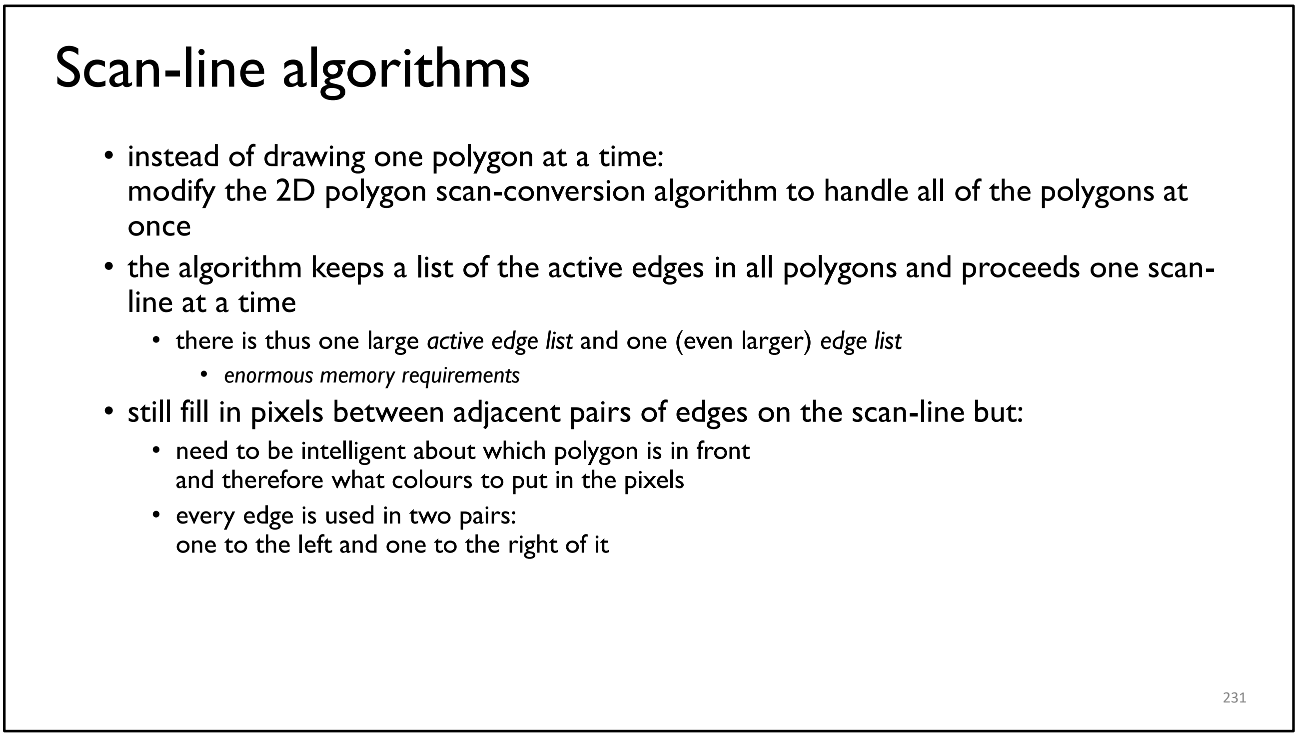
You need to be able to tell which side of an arbitrary plane a vertex lies on and how to split a polygon by an arbitrary plane – both of which were discussed for the depth-sort algorithm.

See the example on Wikipedia:
https://en.wikipedia.org/wiki/Binary_space_partitioning



Drawing a BSP tree

- if the viewpoint is in the front child side of the root’s polygon’s plane then:
  - draw the BSP tree for the *back* child of the root
  - draw the root’s polygon
  - draw the BSP tree for the *front* child of the root
- otherwise:
  - draw the BSP tree for the *front* child of the root
  - draw the root’s polygon
  - draw the BSP tree for the *back* child of the root

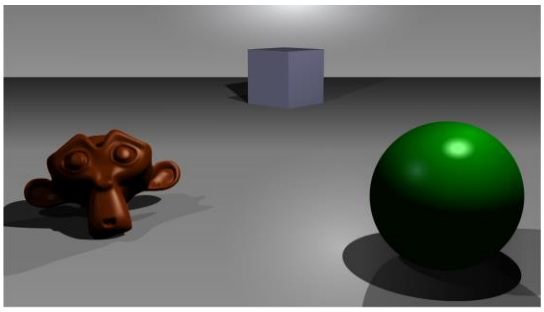


Scan-line algorithms

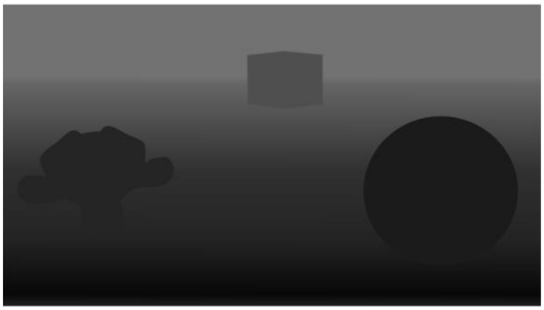
- instead of drawing one polygon at a time:
  modify the 2D polygon scan-conversion algorithm to handle all of the polygons at once
- the algorithm keeps a list of the active edges in all polygons and proceeds one scan-line at a time
  - there is thus one large *active edge list* and one (even larger) *edge list*
    - *enormous memory requirements*
- still fill in pixels between adjacent pairs of edges on the scan-line but:
  - need to be intelligent about which polygon is in front
    and therefore what colours to put in the pixels
  - every edge is used in two pairs:
    one to the left and one to the right of it

z-buffer polygon drawing

- depth sort & BSP-tree methods involve clever sorting algorithms followed by the invocation of the standard 2D polygon scan conversion algorithm
- by modifying the 2D scan conversion algorithm we can remove the need to sort the polygons
  - makes hardware implementation easier
  - this is the algorithm used on graphics cards



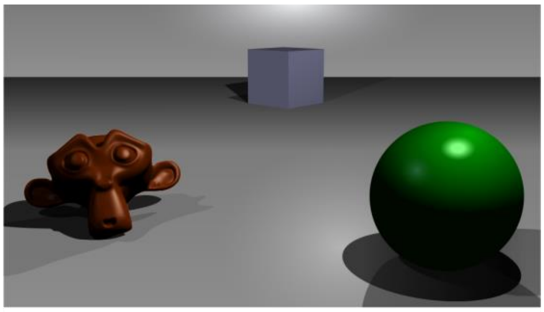
A simple three-dimensional scene



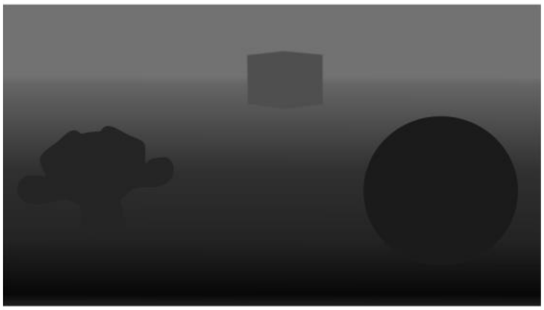
Z-buffer representation

z-buffer basics

- store both *colour* and *depth* at each pixel
- scan convert one polygon at a time in any order
- when scan converting a polygon:
  - calculate the polygon’s depth at each pixel
  - if the polygon is closer than the current depth stored at that pixel
    - then store both the polygon’s colour and depth at that pixel
    - otherwise do nothing



A simple three-dimensional scene



Z-buffer representation

z-buffer algorithm

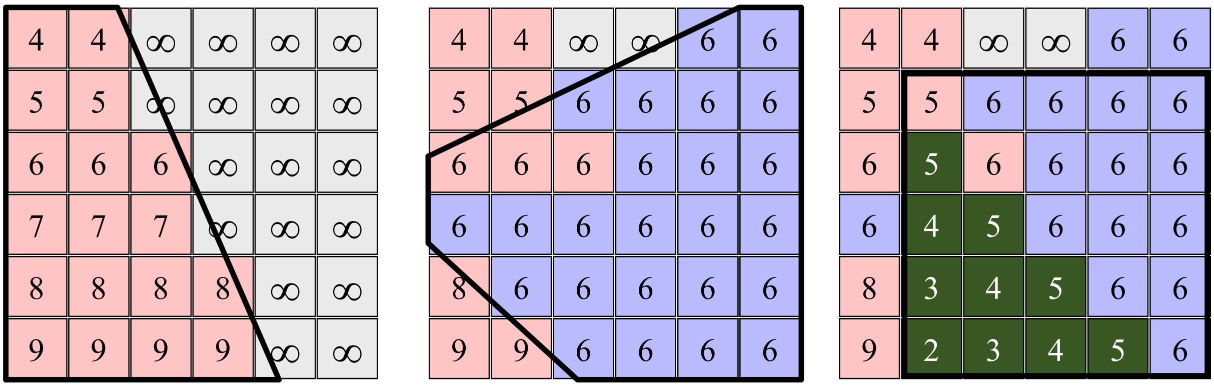
```
FOR every pixel (x,y)
    Colour[x,y] = background colour ;
    Depth[x,y] = infinity ;
END FOR ;

FOR each polygon
    FOR every pixel (x,y) in the polygon's projection
        z = polygon's z-value at pixel (x,y) ;
        IF z < Depth[x,y] THEN
            Depth[x,y] = z ;
            Colour[x,y] = polygon's colour at (x,y) ;
        END IF ;
    END FOR ;
END FOR ;
```

this requires you to project the polygon's vertices to 2D and run the 2D polygon scan-conversion algorithm

this requires you to modify the 2D algorithm so that it can compute the z-value at each pixel

z-buffer example



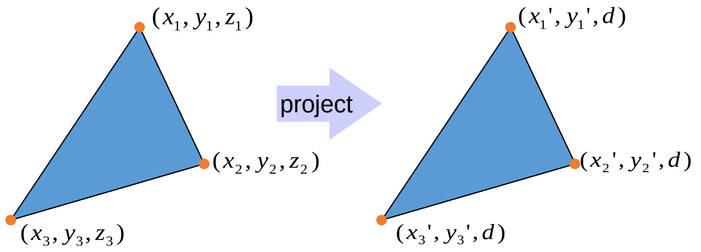
4	4	∞	∞	∞	∞
5	5	∞	∞	∞	∞
6	6	6	∞	∞	∞
7	7	7	∞	∞	∞
8	8	8	8	∞	∞
9	9	9	9	∞	∞

4	4	∞	∞	6	6
5	5	6	6	6	6
6	6	6	6	6	6
6	6	6	6	6	6
8	6	6	6	6	6
9	9	6	6	6	6

4	4	∞	∞	6	6
5	5	6	6	6	6
6	5	6	6	6	6
6	4	5	6	6	6
8	3	4	5	6	6
9	2	3	4	5	6

Interpolating depth values I

- just as we incrementally interpolate x as we move along each edge of the polygon, we can incrementally interpolate z:
  - as we move along the edge of the polygon
  - as we move across the polygon's projection



$$x_a' = x_a \frac{d}{z_a}$$

$$y_a' = y_a \frac{d}{z_a}$$

Interpolating depth values 2

- we thus have 2D vertices, with added depth information

$$[(x_a', y_a'), z_a]$$

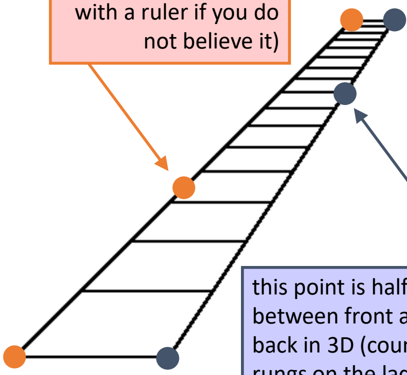
- we can interpolate *x* and *y* in 2D

$$x' = (1-t)x_1' + (t)x_2'$$

$$y' = (1-t)y_1' + (t)y_2'$$

- but *z* must be interpolated in 3D

$$\frac{1}{z} = (1-t)\frac{1}{z_1} + (t)\frac{1}{z_2}$$



this point is halfway between front and back in 2D (measure with a ruler if you do not believe it)

this point is halfway between front and back in 3D (count the rungs on the ladder)

Interpolating depth values 3

(the gory details: only for those who really want to know)

$$x = az + bd$$

consider the projection onto the plane $y=0$

$$x' = x\frac{d}{z} = ad + \frac{bd}{z}$$

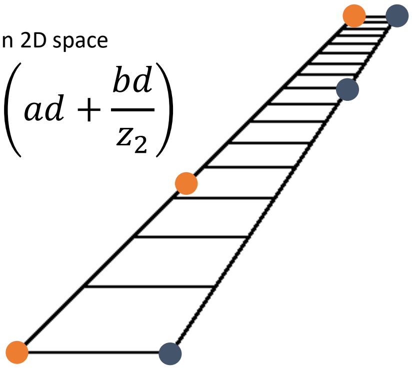
now project to $z=d$

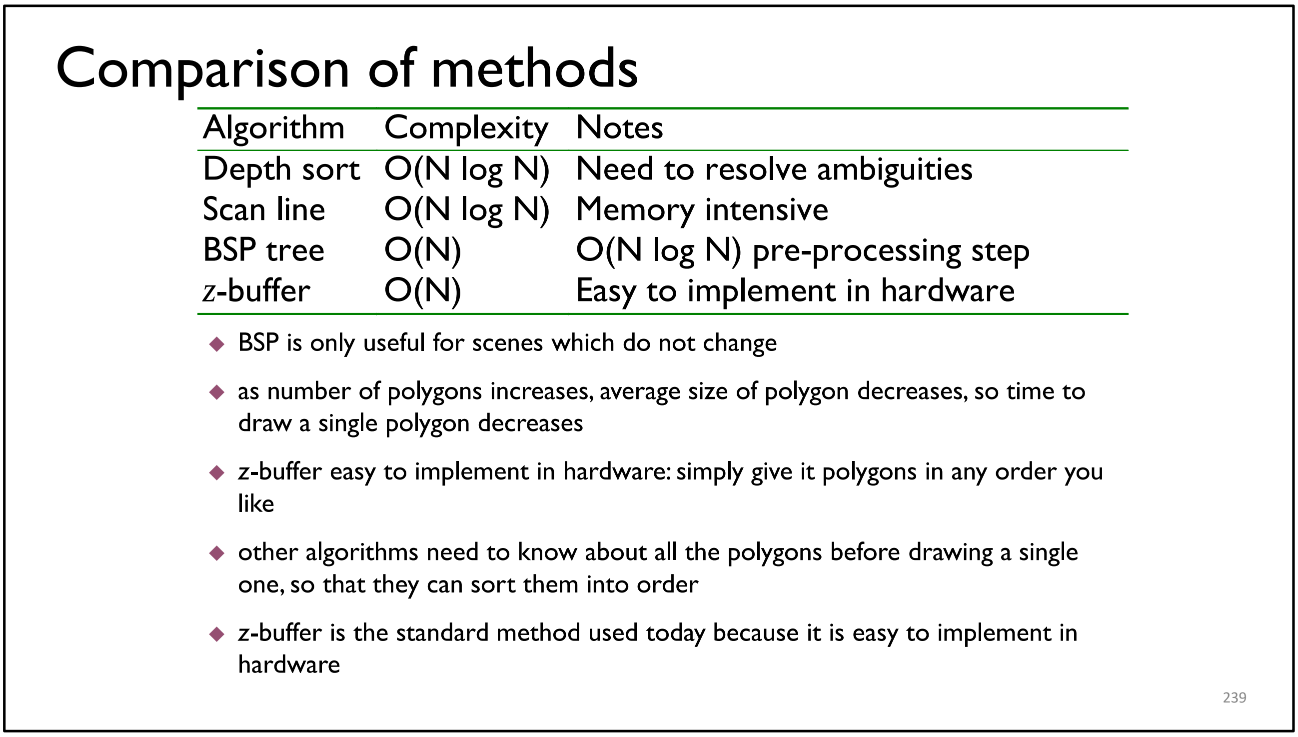
$$x' = (1-t){x_1}' + t{x_2}'$$

interpolate x' in 2D space

$$ad + \frac{bd}{z} = (1-t)\left(ad + \frac{bd}{z_1}\right) + t\left(ad + \frac{bd}{z_2}\right)$$

$$\frac{1}{z} = (1-t)\left(\frac{1}{z_1}\right) + t\left(\frac{1}{z_2}\right)$$





Comparison of methods

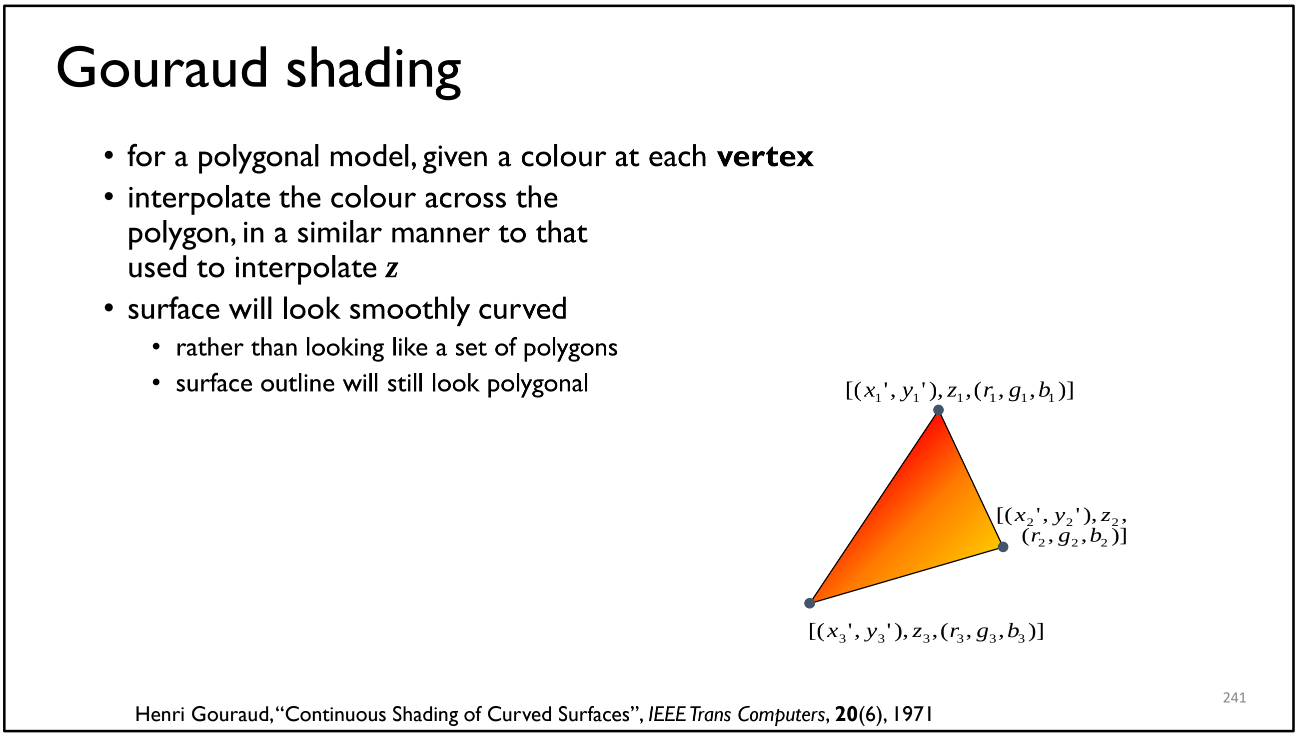
Algorithm	Complexity	Notes
Depth sort	O(N log N)	Need to resolve ambiguities
Scan line	O(N log N)	Memory intensive
BSP tree	O(N)	O(N log N) pre-processing step
z-buffer	O(N)	Easy to implement in hardware

- BSP is only useful for scenes which do not change
- as number of polygons increases, average size of polygon decreases, so time to draw a single polygon decreases
- z-buffer easy to implement in hardware: simply give it polygons in any order you like
- other algorithms need to know about all the polygons before drawing a single one, so that they can sort them into order
- z-buffer is the standard method used today because it is easy to implement in hardware

Putting it all together - a summary

- a 3D polygon scan conversion algorithm needs to include:
  - a 2D polygon scan conversion algorithm
  - 2D or 3D polygon clipping
  - projection from 3D to 2D
  - either:
    - ordering the polygons so that they are drawn in the correct order

    or:
    - calculating the z value at each pixel and using a depth-buffer



Gouraud shading

- for a polygonal model, given a colour at each **vertex**
- interpolate the colour across the polygon, in a similar manner to that used to interpolate z
- surface will look smoothly curved
  - rather than looking like a set of polygons
  - surface outline will still look polygonal

$[(x_1', y_1'), z_1, (r_1, g_1, b_1)]$

$[(x_2', y_2'), z_2, (r_2, g_2, b_2)]$

$[(x_3', y_3'), z_3, (r_3, g_3, b_3)]$

Henri Gouraud, "Continuous Shading of Curved Surfaces", *IEEE Trans Computers*, **20**(6), 1971

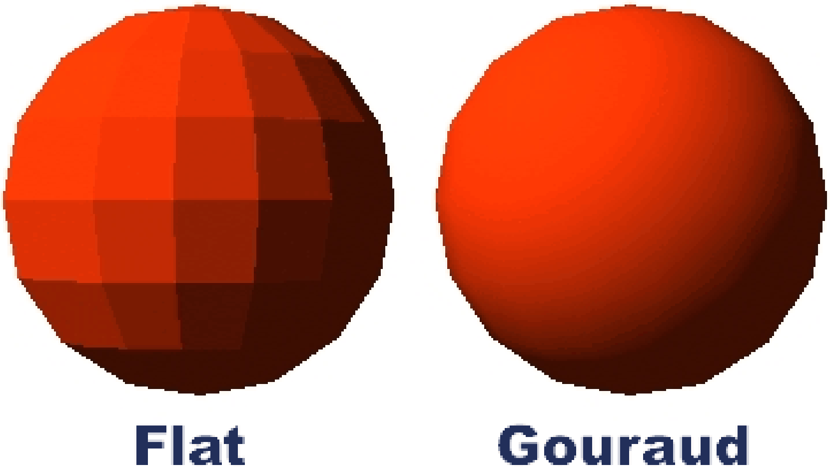
Shading is used in drawing for depicting levels of darkness on paper by applying media more densely or with a darker shade for darker areas, and less densely or with a lighter shade for lighter areas. The appearance of the surface of objects you draw will look natural

In computer graphics, shading refers to the process of altering the color of an object/surface/polygon in the 3D scene, based on things like the surface's angle to lights, its distance from lights, its angle to the camera and material properties (e.g. bidirectional reflectance distribution function) to create a photorealistic effect.

Barycentric

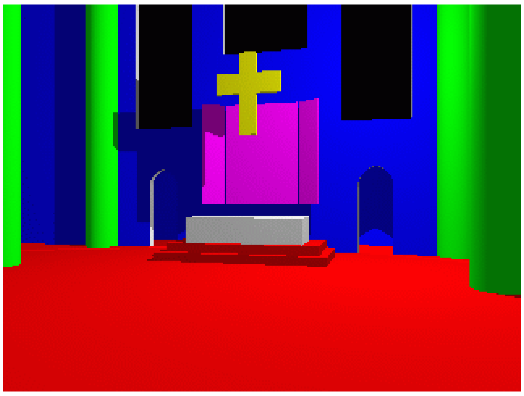
Flat *vs* Gouraud shading

- note how the interior is smoothly shaded but the outline remains polygonal



Texture mapping

without



all surfaces are smooth and of uniform colour

with

most surfaces are textured with
2D texture maps
the pillars are textured with a solid texture

Basic texture mapping

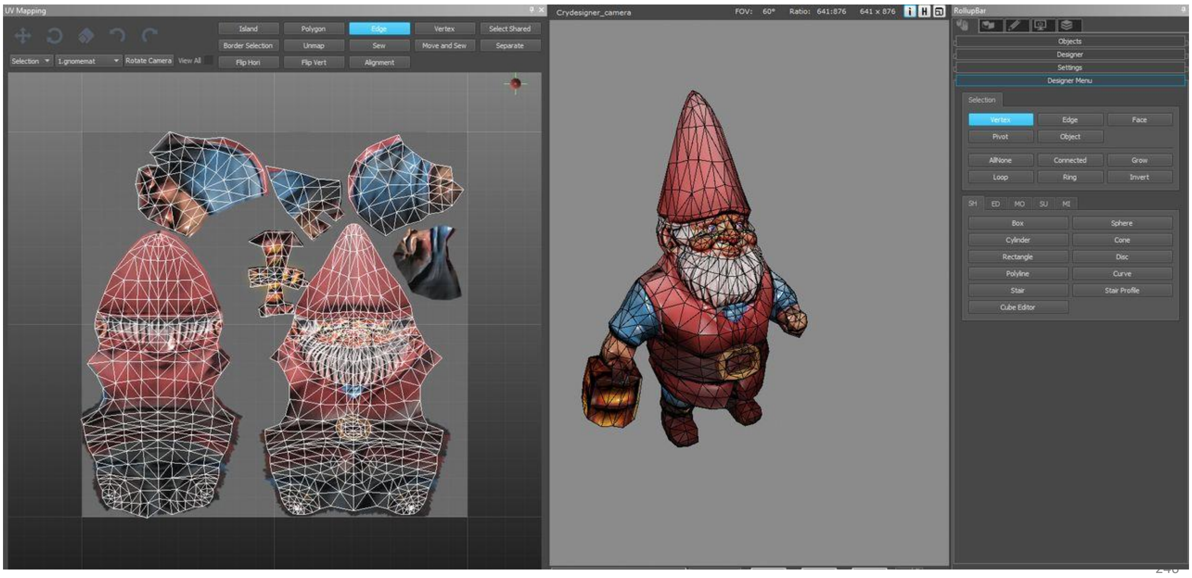
- a texture is simply an image, with a 2D coordinate system (u,v)

- each 3D object is parameterised in (u,v) space
- each pixel maps to some part of the surface
- that part of the surface maps to part of the texture

Paramaterising a primitive

- polygon: give (u,v) coordinates for three vertices, or treat as part of a plane
- plane: give u-axis and v-axis directions in the plane
- cylinder: one axis goes up the cylinder, the other around the cylinder

UV mapping

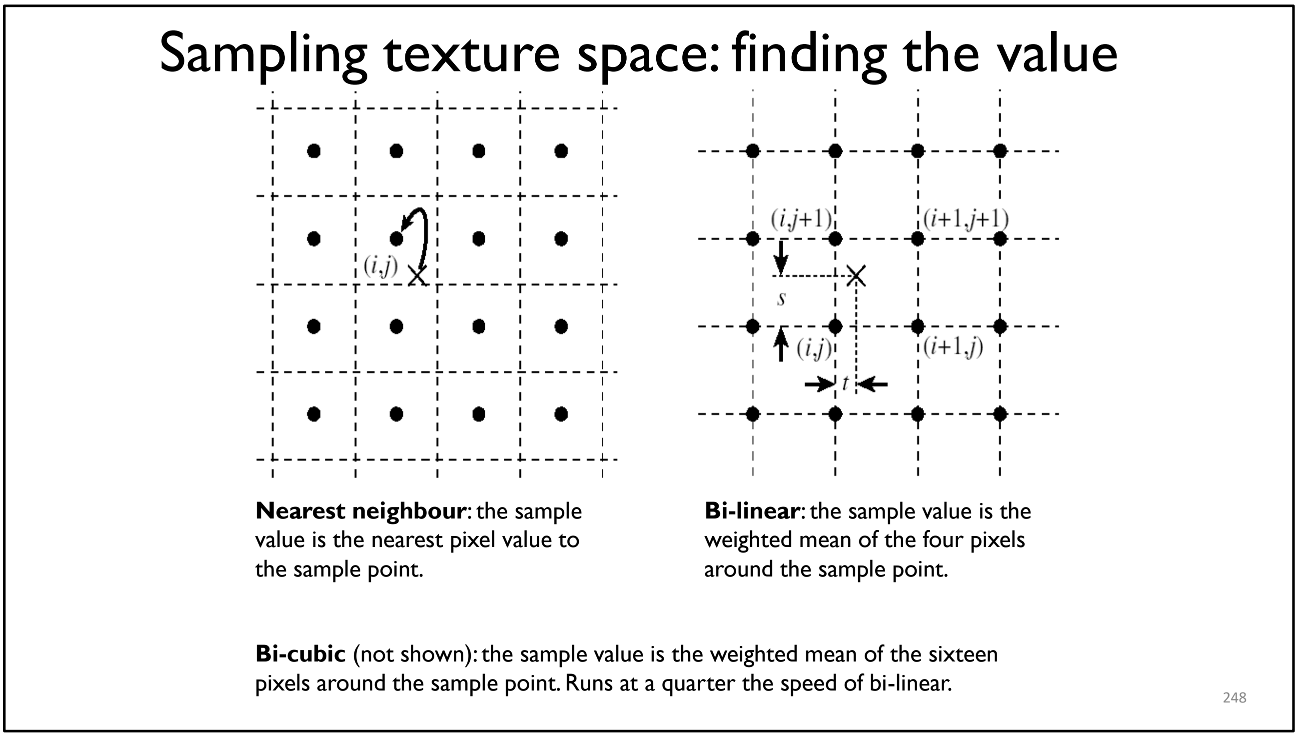

UV Mapping
Island
Polygon
Edge
Vertex
Select Shared
Border Selection
Unmap
Sew
Move and Sew
Separate
Selection
1.gnomemat
Rotate Camera
View All
Flip Hori
Flip Vert
Alignment
Crydesigner_camera
FOV: 60°
Ratio: 641:876
641 x 876
RollupBar
Objects
Designer
Settings
Designer Menu
Selection
Vertex
Edge
Face
Pivot
Object
AllNone
Connected
Grow
Loop
Ring
Invert
SH
ED
MO
SU
MI
Box
Sphere
Cylinder
Cone
Rectangle
Disc
Polyline
Curve
Stair
Stair Profile
Cube Editor

Sampling texture space

“Texture space” Object (on screen)

Find (*u*,*v*) coordinate of the sample point on the object and map this into texture space

Sample texture space to determine the pixel’s colour



Sampling texture space: finding the value

Nearest neighbour: the sample value is the nearest pixel value to the sample point.

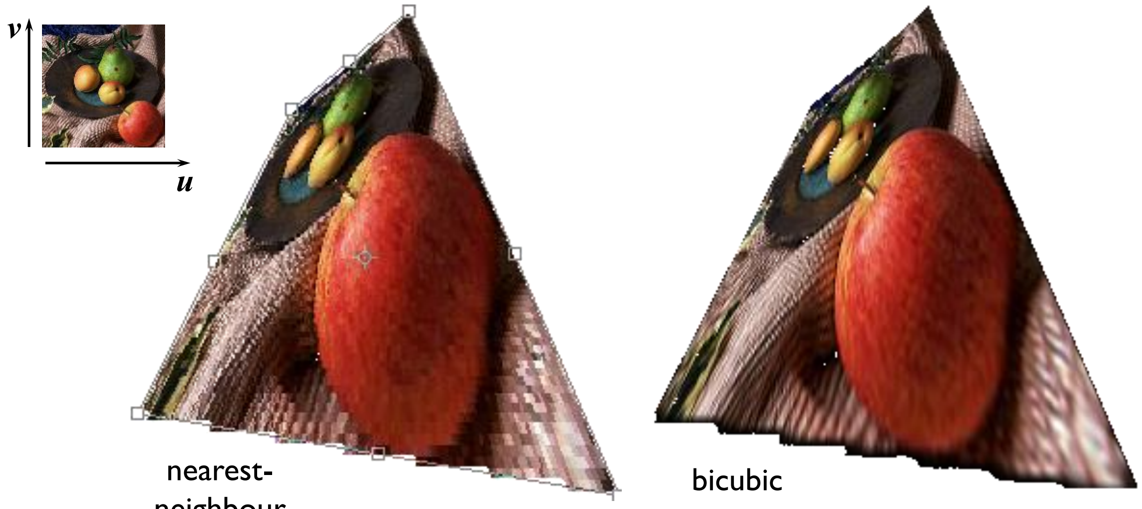
Bi-linear: the sample value is the weighted mean of the four pixels around the sample point.

Bi-cubic (not shown): the sample value is the weighted mean of the sixteen pixels around the sample point. Runs at a quarter the speed of bi-linear.

Bilinear calculation:

$$P(i+t, j+s) = (1-t)(1-s)\ P_{i,j} + (t)(1-s)\ P_{i+1,j} + (1-t)(s)\ P_{i,j+1} + (t)(s)\ P_{i+1,j+1}$$

Texture mapping examples

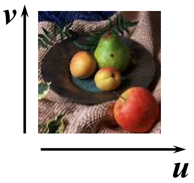


nearest-neighbour

bicubic

look at the bottom right hand corner of the distorted image to compare the two interpolation methods

Up-sampling



nearest-neighbour

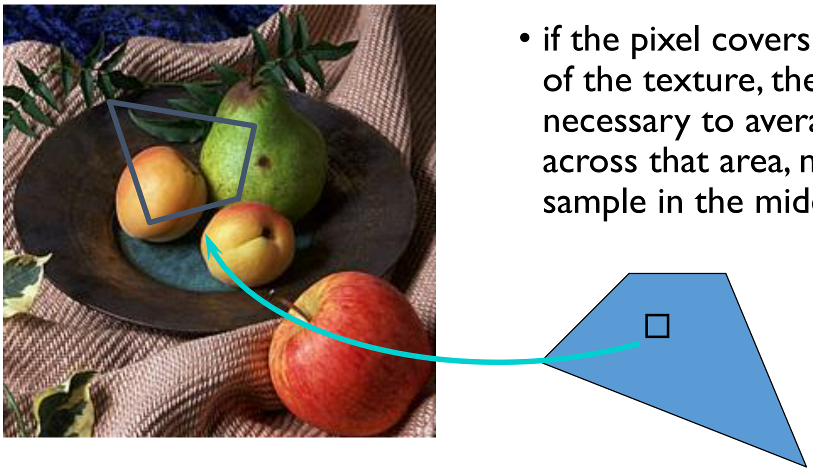
blocky artefacts

bicubic

blurry artefacts

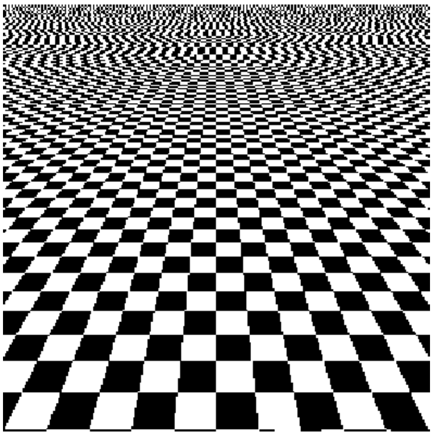
- if one pixel in the texture map covers several pixels in the final image, you get visible artefacts
- only practical way to prevent this is to ensure that texture map is of sufficiently high resolution that it does not happen

Down-sampling

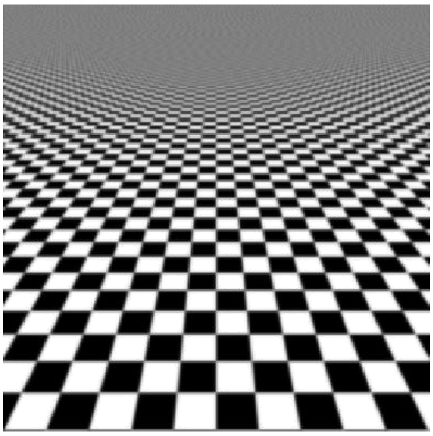


- if the pixel covers quite a large area of the texture, then it will be necessary to average the texture across that area, not just take a sample in the middle of the area

Down-sampling



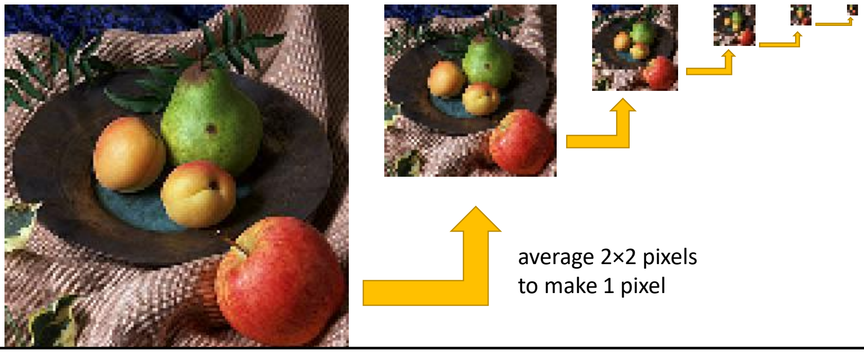
without area averaging



with area averaging

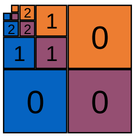
Multi-resolution texture

- To make this fast, pre-calculate multiple versions of the texture at different resolutions and pick the appropriate resolution to sample from…
- Use tri-linear interpolation to a better result: use bi-linear interpolation on each of the two nearest levels and then linearly interpolate between the two interpolated values



The MIP (Latin: Multum In Parvo) map

- an efficient memory arrangement for a multi-resolution colour image
- pixel (x,y) is a bottom level pixel location (level 0); for an image of size (m,n), it is stored at these locations in level k:



Red

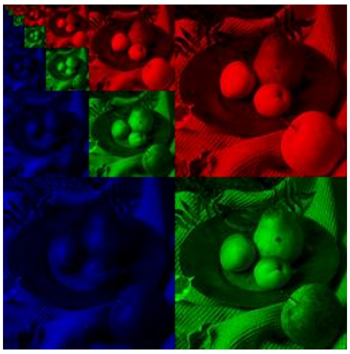
$$\left(\left\lfloor \frac{m+x}{2^k} \right\rfloor, \left\lfloor \frac{y}{2^k} \right\rfloor\right)$$

$$\left(\left\lfloor \frac{x}{2^k} \right\rfloor, \left\lfloor \frac{n+y}{2^k} \right\rfloor\right)$$

Blue

$$\left(\left\lfloor \frac{m+x}{2^k} \right\rfloor, \left\lfloor \frac{n+y}{2^k} \right\rfloor\right)$$

Green



The origin of the term **mipmap** is an initialism of the Latin phrase Multum In Parvo ("much in a **small** space")
A large block memory vs small blocks