Introduction to Computer Graphics

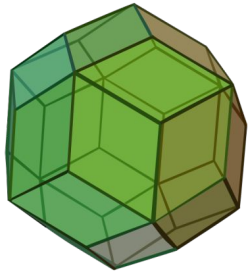
- Background
- Rendering
- **Graphics pipeline (real-time rendering pipeline)**
  - Polygonal mesh models
  - Transformations using matrices in 2D and 3D
  - Homogeneous coordinates
  - Projection: orthographic and perspective
- Rasterization
- Graphics hardware and modern OpenGL
- Human vision, colour and tone mapping

Unfortunately…

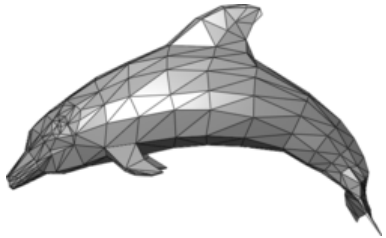
- Ray tracing is computationally expensive
  - used for super-high visual quality
- Video games and user interfaces need something faster
- Most real-time applications rely on **rasterization**
  - Model surfaces as polyhedra – meshes of polygons
  - Use composition to build scenes
  - Apply perspective transformation and project into plane of screen
  - Work out which surface was closest
  - Fill pixels with colour of nearest visible polygon
- Graphics cards have hardware to support this
- Ray tracing starts to appear in real-time rendering
  - The new generations of GPUs offer accelerated ray-tracing
  - But it still not as efficient as rasterization

Three-dimensional objects

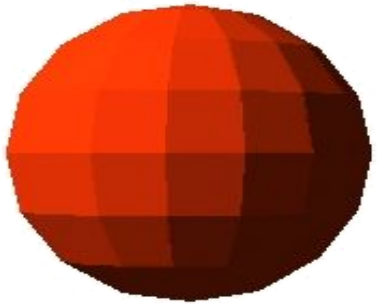
- Polyhedral surfaces are made up from meshes of multiple connected polygons



- Polygonal meshes
  - open or closed

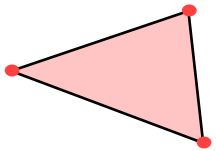


- Curved surfaces
  - must be converted to polygons to be drawn

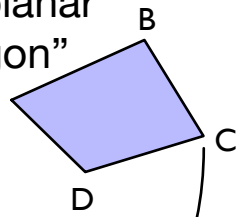


Surfaces in 3D: polygons

- Easier to consider planar polygons
  - 3 vertices (triangle) must be planar
  - > 3 vertices, not necessarily planar



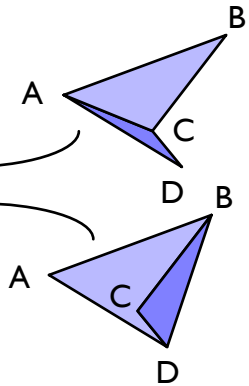
a non-planar "polygon"



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

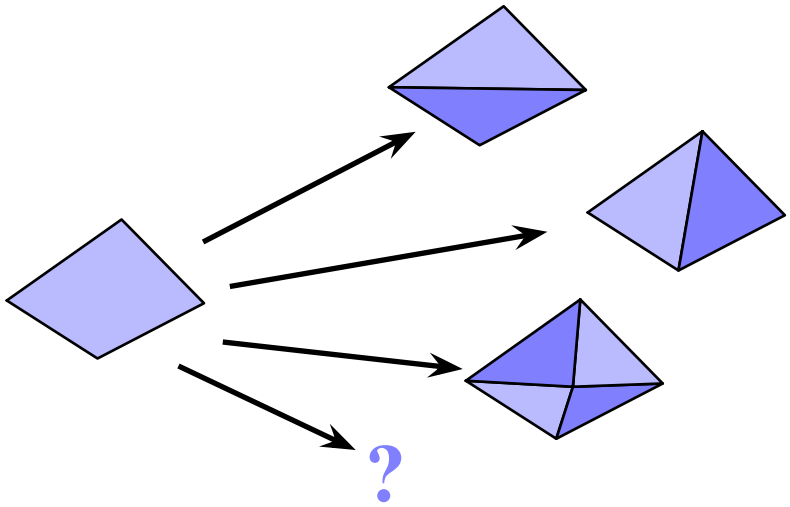
rotate the polygon about the vertical axis

should the result be this or this?



Splitting polygons into triangles

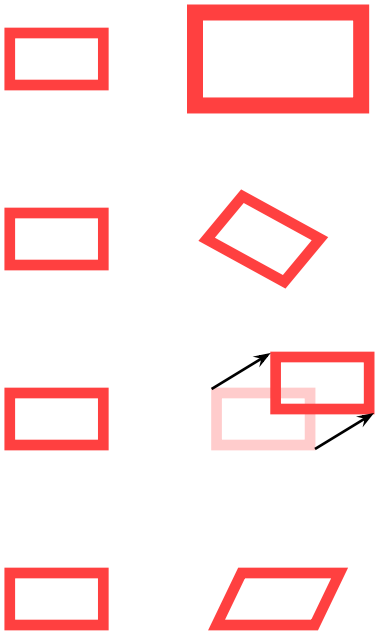
- Most Graphics Processing Units (GPUs) are optimised to draw triangles
- Split polygons with more than three vertices into triangles



which is preferable?

2D transformations

- scale
- rotate
- translate
- (shear)



- why?
  - it is extremely useful to be able to transform predefined objects to an arbitrary location, orientation, and size
  - any reasonable graphics package will include transforms
    - 2D Postscript
    - 3D OpenGL

Basic 2D transformations

- scale
  - about origin
  - by factor m

$$x' = mx$$
$$y' = my$$

- rotate
  - about origin
  - by angle θ

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

- translate
  - along vector (x_o, y_o)

$$x' = x + x_0$$
$$y' = y + y_0$$

- shear
  - parallel to x axis
  - by factor a

$$x' = x + ay$$
$$y' = y$$

Matrix representation of transformations

- scale
  - about origin, factor ***m***

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- rotate
  - about origin, angle **θ**

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- do nothing
  - identity

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- shear
  - parallel to ***x*** axis, factor ***a***

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Homogeneous 2D coordinates

- translations cannot be represented using simple 2D matrix multiplication on 2D vectors, so we switch to homogeneous co-ordinates

$$(x, y, w) \equiv \left(\frac{x}{w}, \frac{y}{w}\right)$$

- an infinite number of homogeneous co-ordinates map to every 2D point
- $w=0$ represents a point at infinity
- usually take the inverse transform to be:

$$(x, y) \equiv (x, y, 1)$$

normalised form

- The symbol $\equiv$ means equivalent

[FCG 6.3/7.3]

Matrices in homogeneous coordinates

scale

- about origin, factor m

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

rotate

- about origin, angle θ

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

do nothing

- identity

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

shear

- parallel to x axis, factor a

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Translation by matrix algebra

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

In homogeneous coordinates

$$x' = x + wx_o \qquad y' = y + wy_o \qquad w' = w$$

In conventional coordinates

$$\frac{x'}{w'} = \frac{x}{w} + x_0 \qquad \frac{y'}{w'} = \frac{y}{w} + y_0$$

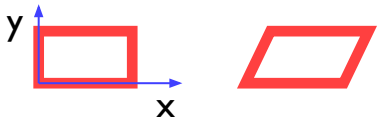
Okay, math checks out, but what is the intuition?

How come the additional coordinate linearizes the 2D translation

Translation by matrix algebra

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

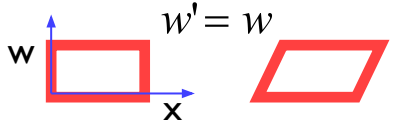
$$x' = x + ay$$
$$y' = y$$



- Shearing in 2D
  - Tilt x by a factor of a
  - Translation by a constant along x when y=1

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$x' = x + wx_o$$
$$y' = y + wy_o$$
$$w' = w$$



- Shearing in 3D
  - Tilt x,y by a factor of x0,y0
  - Translation by a constant along x,y when w=1

Concatenating transformations

- often necessary to perform more than one transformation on the same object
- can concatenate transformations by multiplying their matrices
  e.g. a shear followed by a scaling:

scale

$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$$

shear

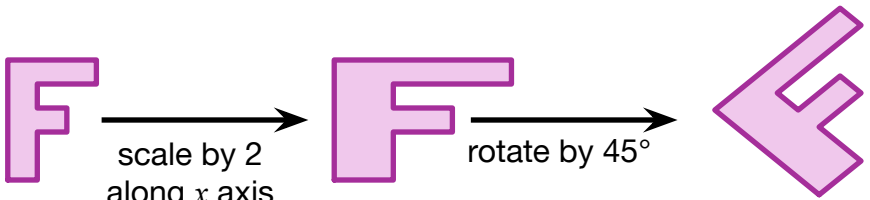
$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

scale shear both

$$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} m & ma & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Transformation are not commutative

- be careful of the order in which you concatenate transformations



rotate then scale

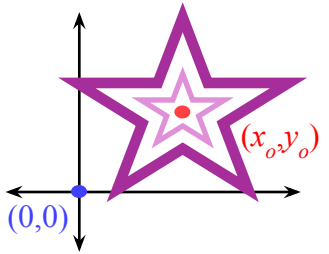
$$\begin{bmatrix} 2/\sqrt{2} & -2/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ scale

$$\begin{bmatrix} 2/\sqrt{2} & -1/\sqrt{2} & 0 \\ 2/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ rotate

scale then rotate

Scaling about an arbitrary point

- scale by a factor m about point (x_o, y_o)
  - (1) translate point (x_o, y_o) to the origin
  - (2) scale by a factor m about the origin
  - (3) translate the origin to (x_o, y_o)



(1) $$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

(2) $$\begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$$

(3) $$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x'' \\ y'' \\ w'' \end{bmatrix}$$

$$\begin{bmatrix} x''' \\ y''' \\ w''' \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_o \\ 0 & 1 & y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_o \\ 0 & 1 & -y_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Exercise: show how to perform rotation about an arbitrary point

3D transformations

- 3D homogeneous co-ordinates

$$(x, y, z, w) \rightarrow \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

- 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

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

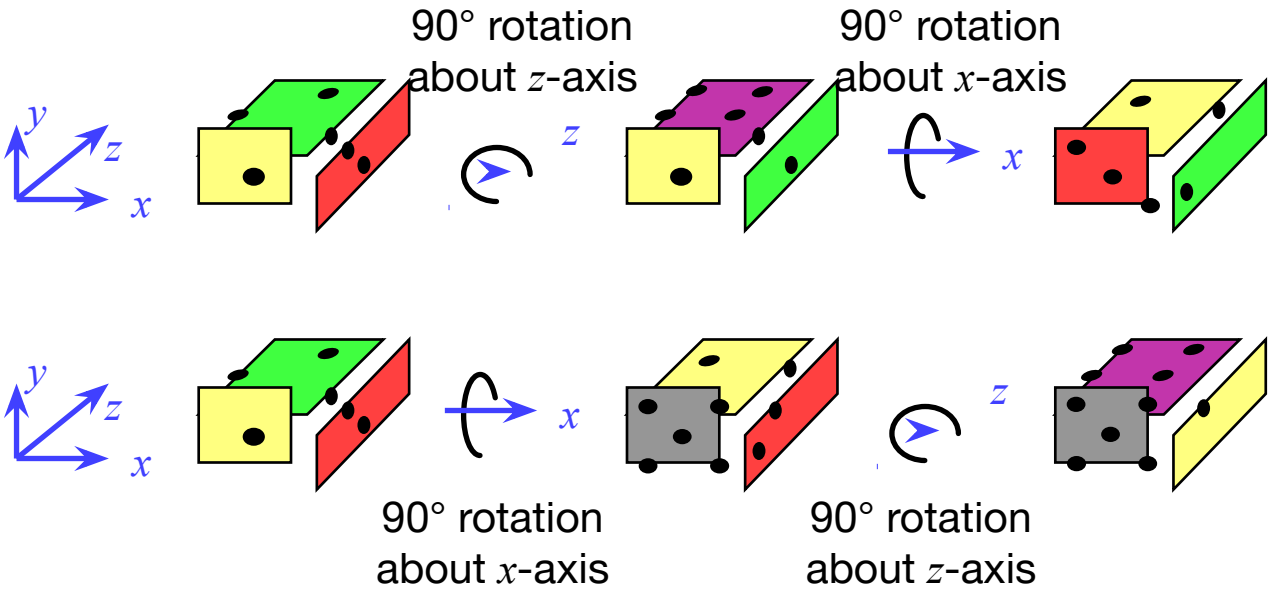
rotation about *z*-axis

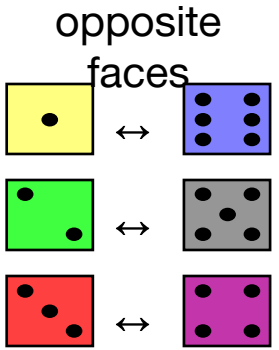
$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

rotation about *y*-axis

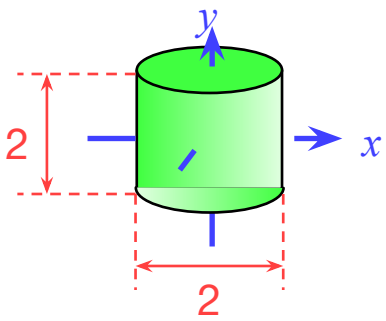
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D transformations are not commutative

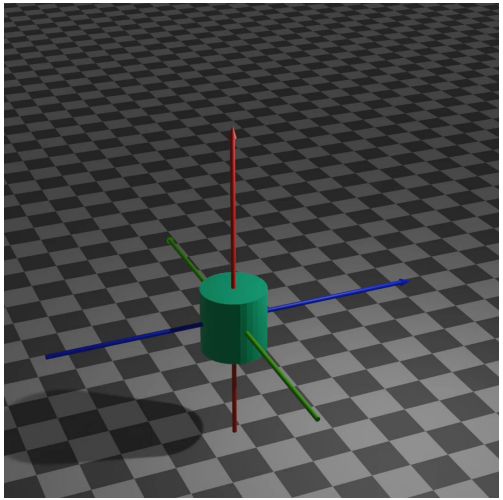




Model transformation 1



- 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?



Model transformation 2

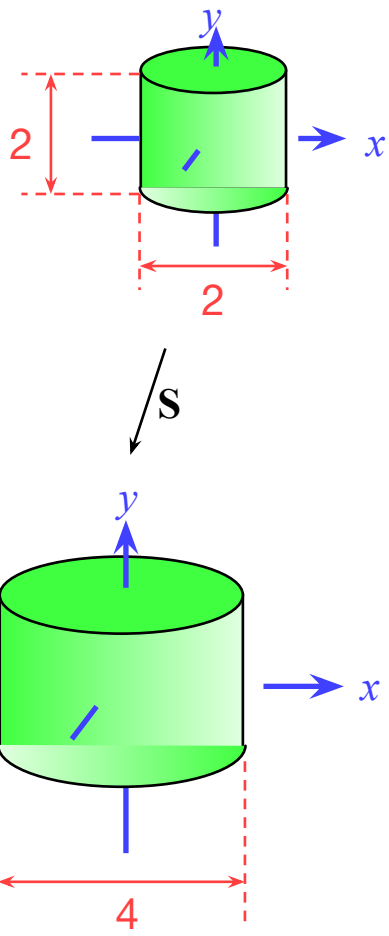
- order is important:
  - scale first
  - rotate
  - translate last
- scaling and translation are straightforward

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

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)



Model transformation 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)

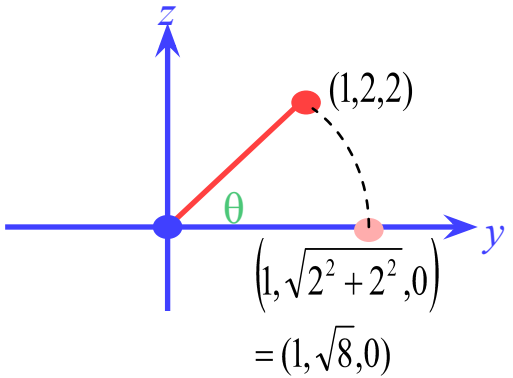
Model transformation 4

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

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

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

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

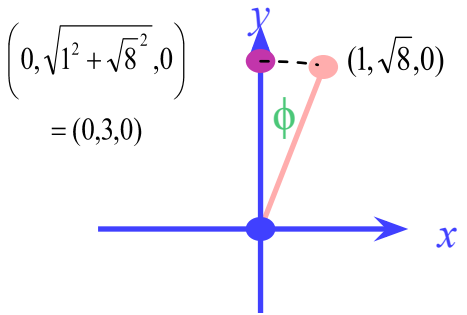


Model transformation 5

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

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

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



Model transformation 6

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

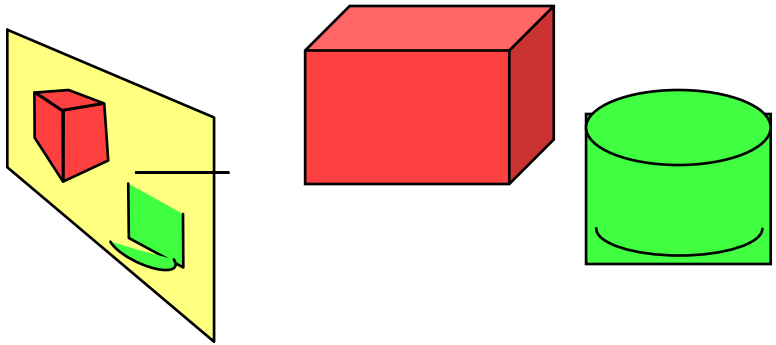
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{T} \times \mathbf{R}_1^{-1} \times \mathbf{R}_2^{-1} \times \mathbf{S} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

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



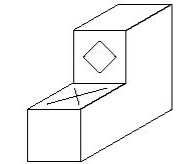
The 3D world is described as a set of (mathematical) objects

e.g. sphere radius (3.4)
centre (0,2,9)

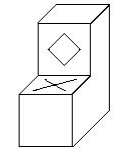
e.g. box size (2,4,3)
centre (7, 2, 9)
orientation (27°, 156°)

Types of projection

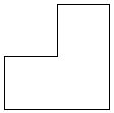
- parallel
  - e.g. $(x, y, z) \rightarrow (x, y)$
  - useful in CAD, architecture, etc
  - looks unrealistic



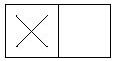
Cavalier projection



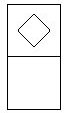
Cabinet projection



Parallel to X axis

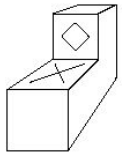


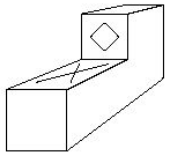
Parallel to Y axis



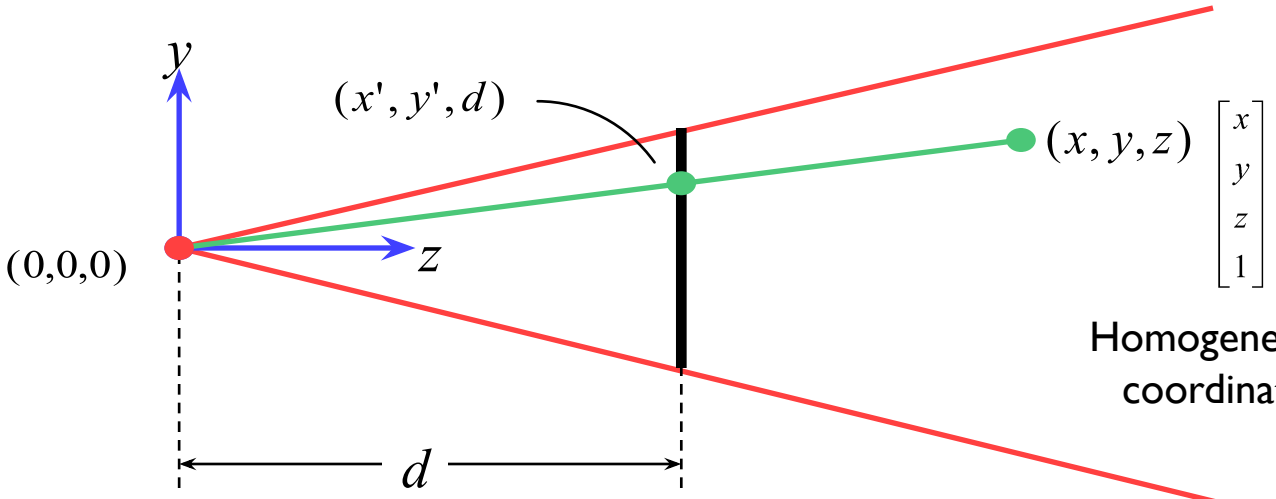
Parallel to Z axis

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





Geometry of perspective projection



$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Homogeneous coordinate

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

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

- Division of (x, y) by a scalar corresponds to the same ray — equivalent under homogeneous coordinates
- Want to construct a projection matrix that converts w from 1 to z/d

Projection as a matrix operation

invertible

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

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

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

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

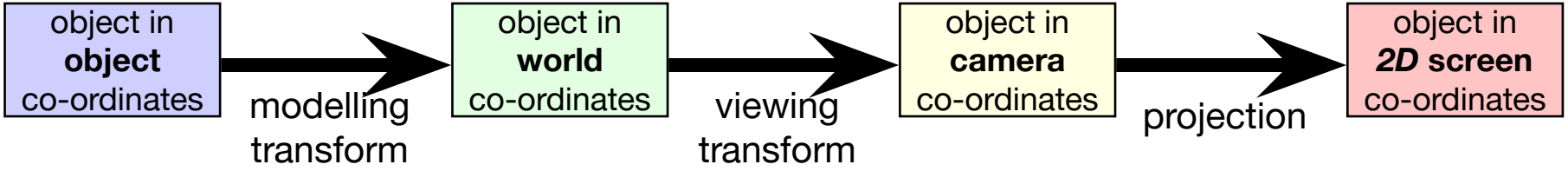
This is useful in the z-buffer algorithm where we need to interpolate $1/z$ values rather than z values.

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

Perspective projection with an arbitrary camera

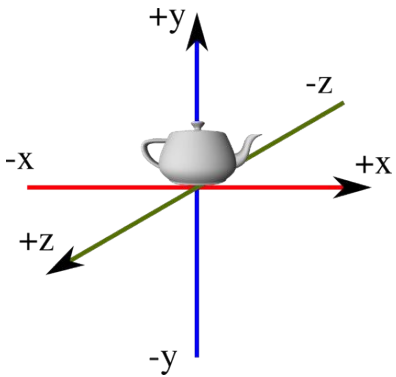
- we have assumed that:
  - screen centre at $(0,0,d)$
  - screen parallel to xy-plane
  - z-axis into screen
  - y-axis up and x-axis to the right
  - eye (camera) at origin $(0,0,0)$
- for an arbitrary camera we can either:
  - work out equations for projecting objects about an arbitrary point onto an arbitrary plane
  - **Change of co-ordinates** — 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 camera (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

Model, View, Projection matrices

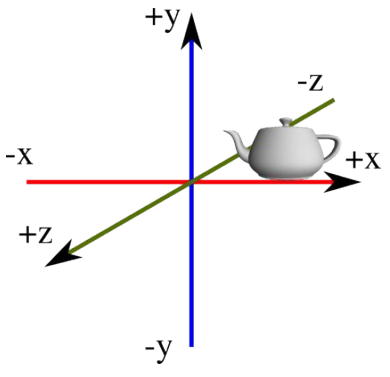


Object coordinates

Object centred at the origin

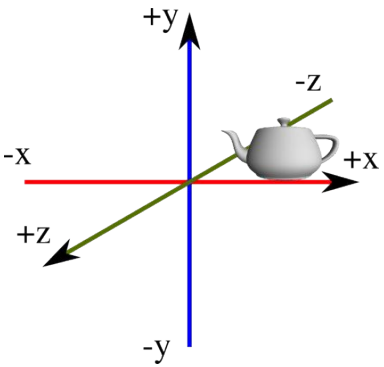
Model matrix

To position each object in the scene. Could be different for each object.



World coordinates

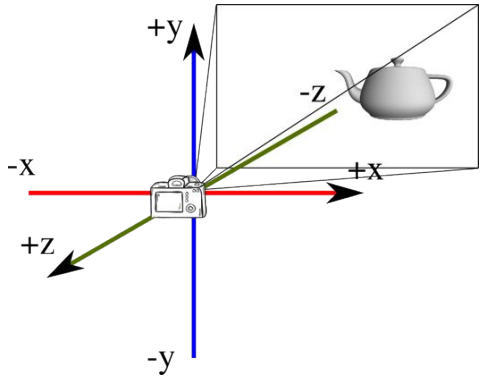
Model, View, Projection matrices



World coordinates

View matrix

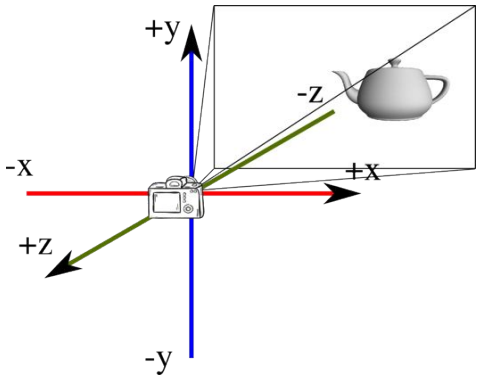
To position all objects relative to the camera



View (camera) coordinates

Camera at the origin, pointing at -z

Model, View, Projection matrices

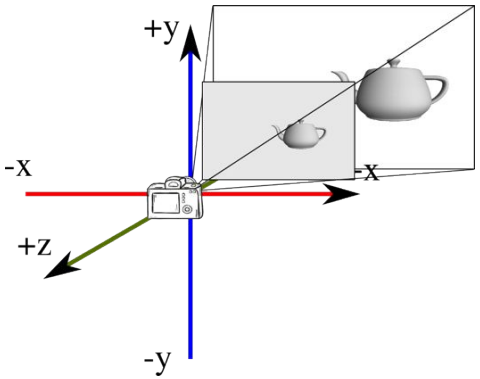


View (camera) coordinates

Projection matrix

To project 3D coordinates to a 2D plane. Note that *z* coordinate is retained for depth testing.

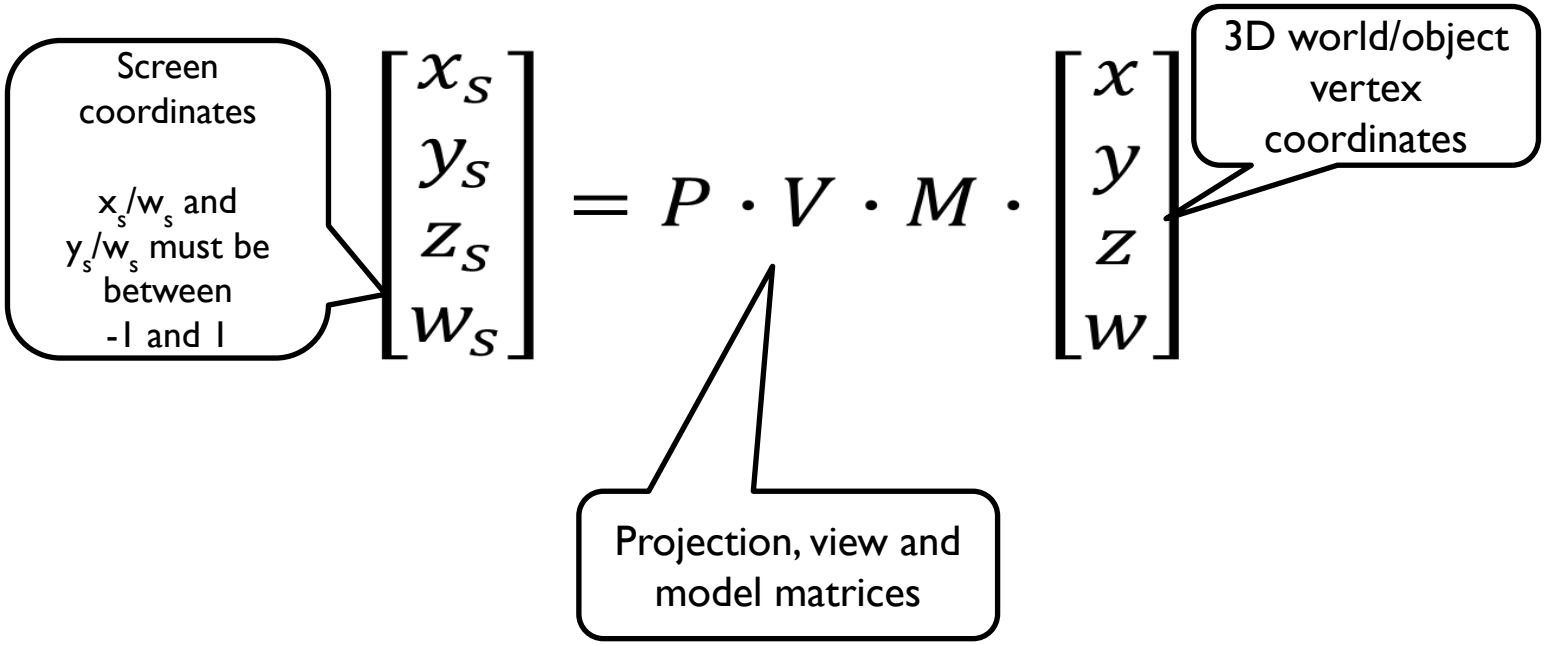
The default OpenGL coordinate system is right-handed



Screen coordinates

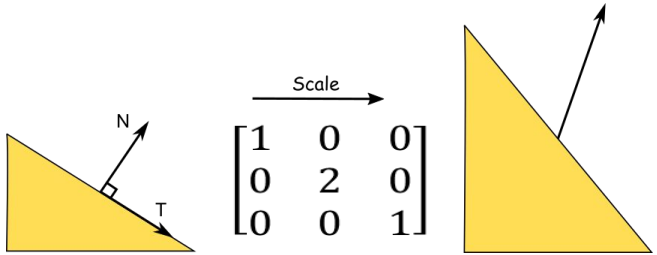
x and y must be in the range -1 and 1

All together



Transforming normal vectors

- Transformation by a nonorthogonal matrix does not preserve angles



- Since:

$$N \cdot T = 0$$

Normal transform

$$N' \cdot T' = (GN) \cdot (\mathrm{MT}) = 0$$

Transformed normal and tangent vector

Vertex position transform

- We can find that: $G = (M^{-1})^T$
  - Derivation shown in the lecture

[FCG 6.2.2/7.2.2]

Scene construction

- We will build a robot from basic parts
  - Body transformation

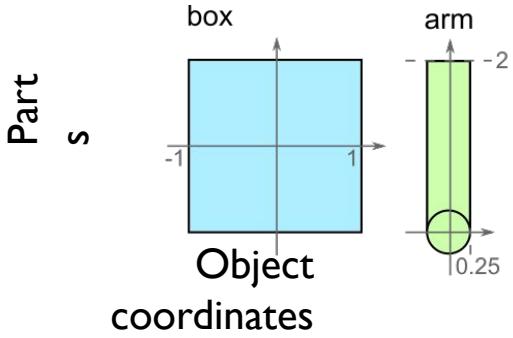
$M_{body} =$

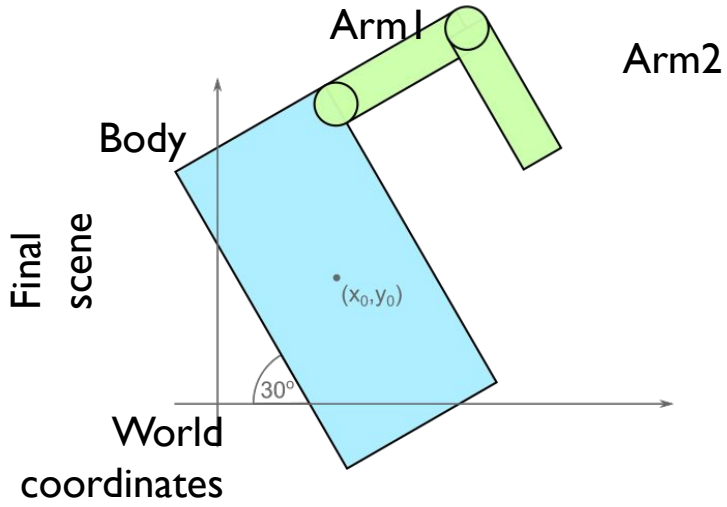
  - Arm1 transformation

$M_{arm1} =$

  - Arm2 transformation

$M_{arm2} =$





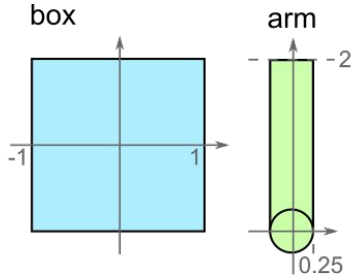
Scene construction

Body transformation

$$E_{body} = scale\begin{bmatrix}1\\2\end{bmatrix}$$

$$T_{body} = translate\begin{bmatrix}x_0\\y_0\end{bmatrix} \cdot rotate(30^o)$$

$$M_{body} = T_{body}E_{body}$$



Arm1 transformation

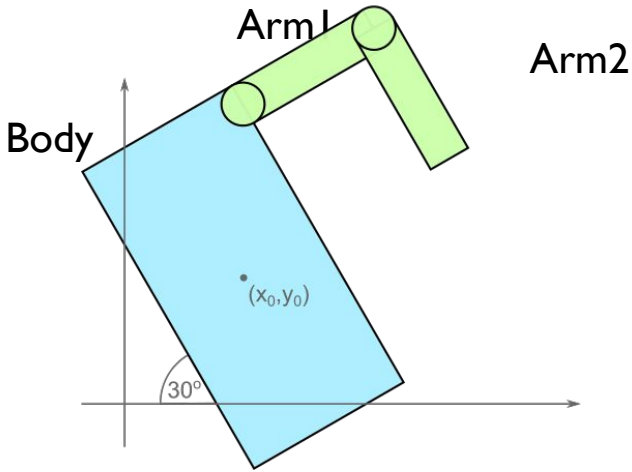
$$T_{arm1} = translate\begin{bmatrix}1\\1.75\end{bmatrix} \cdot rotate(-90^o)$$

$$M_{arm1} = T_{body}T_{arm1}$$

Arm2 transformation

$$T_{arm2} = translate\begin{bmatrix}0\\2\end{bmatrix} \cdot rotate(-90^o)$$

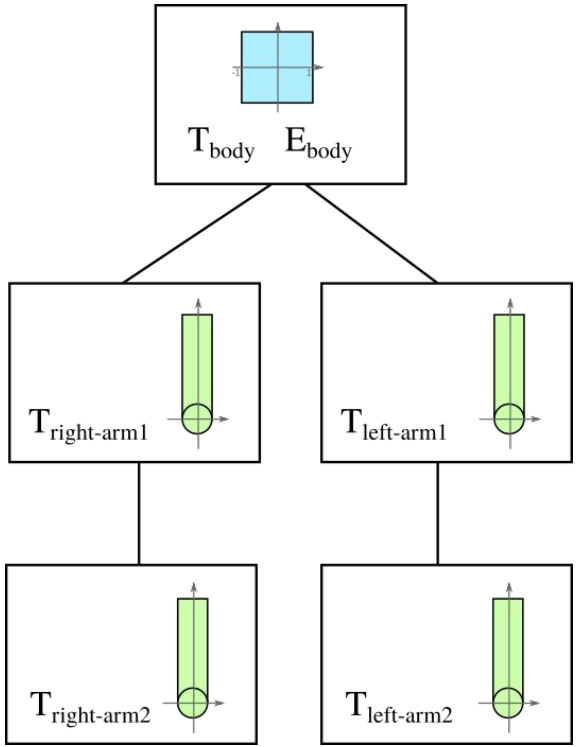
$$M_{arm2} = T_{body}T_{arm1}T_{arm2}$$



Scene Graph

- A scene can be drawn by traversing a scene graph:

```
traverse( node, T_parent ) {
    M = T_parent * node.T * node.E
    node.draw(M)
    for each child {
        traverse( child, T_parent * node.T )
    }
}
```



[FCG 12.2/12.2]

Introduction to Computer Graphics

- Background
- Rendering
- Graphics pipeline
- **Rasterization**
- Graphics hardware and OpenGL
- Human vision and colour & tone mapping

Rasterization algorithm(*)

Set model, view and projection (MVP) transformations

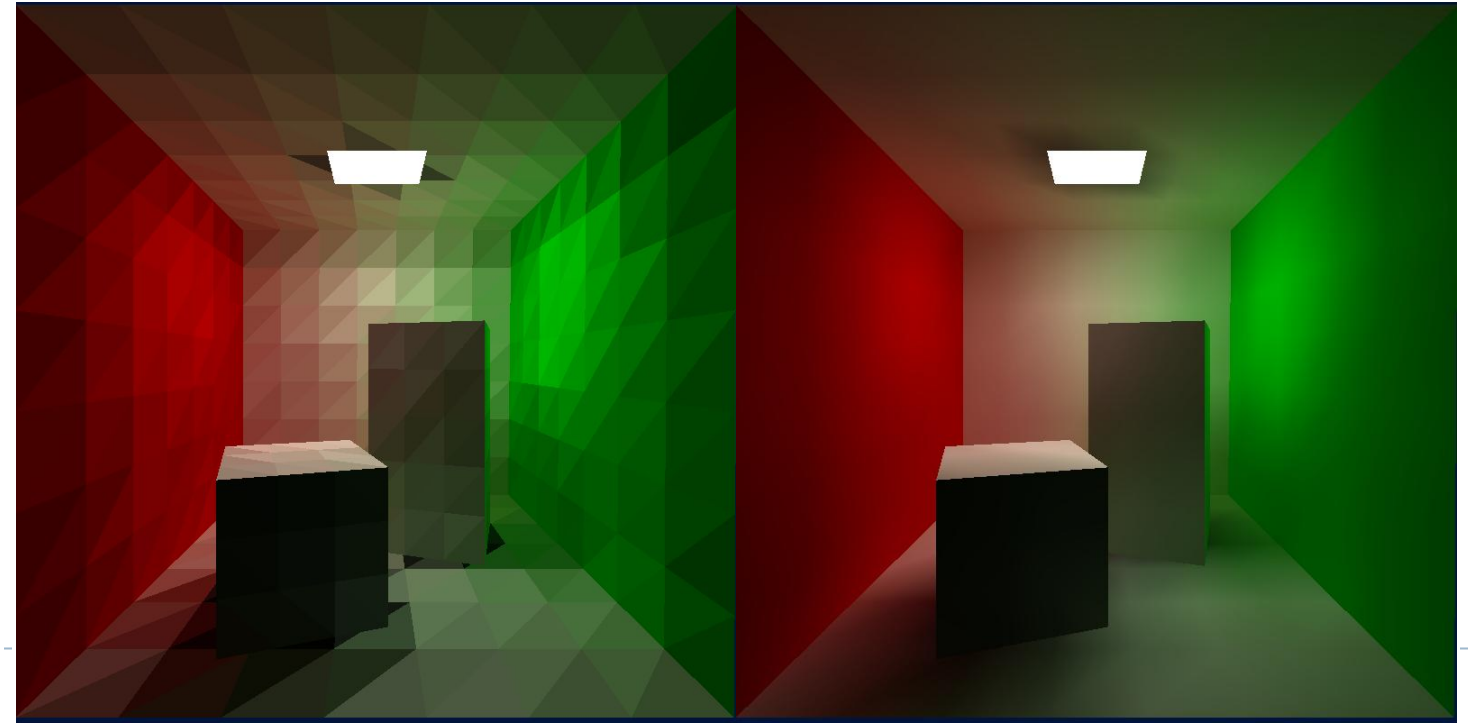
FOR every **triangle** in the scene
    transform its vertices using MVP matrices
    IF the **triangle** is within a view frustum
        *clip the **triangle** to the screen border*
        FOR each **fragment** in the triangle
            interpolate **fragment** *position and attributes between vertices*
            compute **fragment** *colour*
            IF the **fragment** is closer to the camera than any pixel drawn so far
                update the screen **pixel** with the **fragment** colour
            END IF ;
        END FOR ;
    END IF ;
END FOR ;

fragment – a candidate pixel in the triangle

(*) simplified

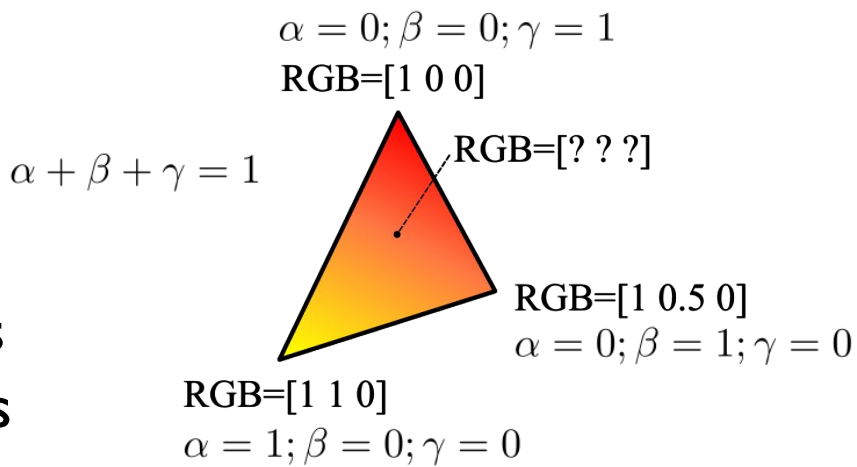
Illumination & shading

- Drawing polygons with uniform colours gives poor results
- Interpolate colours across polygons



Rasterization

- Efficiently draw (thousands of) triangles
  - Interpolate vertex attributes inside the triangle
- **Homogenous barycentric coordinates** are used to interpolate colours, normals, texture coordinates and other attributes inside the triangle



[FCG 2.7/2.9]

Homogeneous barycentric coordinates

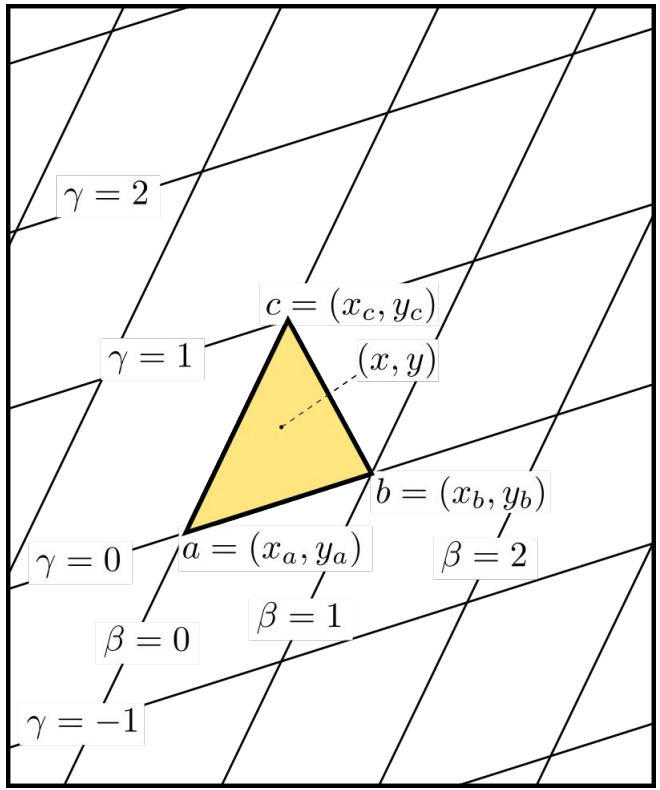
- Find barycentric coordinates of the point (x,y)
  - Given the coordinates of the vertices
  - Derivation in the lecture

$$\alpha = \frac{f_{cb}(x,y)}{f_{cb}(x_a,y_a)} \quad \beta = \frac{f_{ac}(x,y)}{f_{ac}(x_b,y_b)}$$

$f_{ab}(x,y)$ is the implicit line equation:

$$f_{ab}(x,y) = (y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a$$

Distance between point (x,y) and edge ab



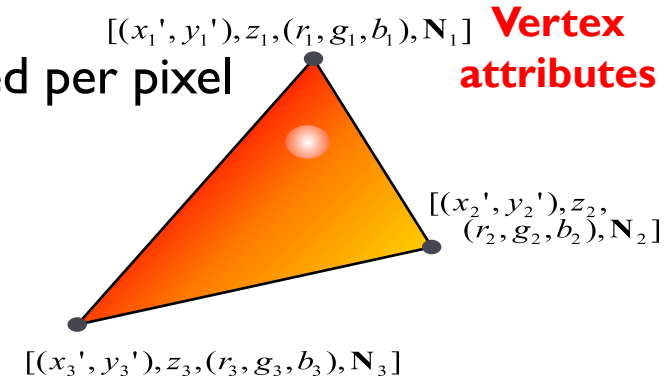
Triangle rasterization

for $y=y_{min}$ to y_{max} **do**
  for $x=x_{min}$ to x_{max} **do**
    $\alpha = f_{cb}(x,y)/f_{cb}(x_a,y_a)$
    $\beta = f_{ac}(x,y)/f_{ac}(x_b,y_b)$
    $\gamma = 1-\alpha-\beta$
    if ($\alpha > 0$ and $\beta > 0$ and $\gamma > 0$) **then**
      $c = \alpha c_a + \beta c_b + \gamma c_c$
      draw pixels (x,y) with colour c

- Optimization: the barycentric coordinates will change by the same amount when moving one pixel right (or one pixel down) regardless of the position
  - Precompute increments $\Delta\alpha, \Delta\beta, \Delta\gamma$ and use them instead of computing barycentric coordinates when drawing pixels sequentially

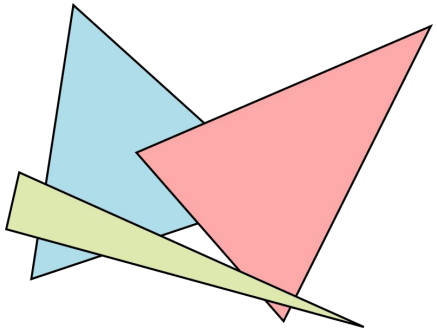
Surface normal vector interpolation

- for a polygonal model, interpolate normal vector between the vertices
  - Calculate colour (Phong reflection model) for each pixel
  - Diffuse component can be either interpolated or computed for each pixel
  - Specular component must be computed per pixel

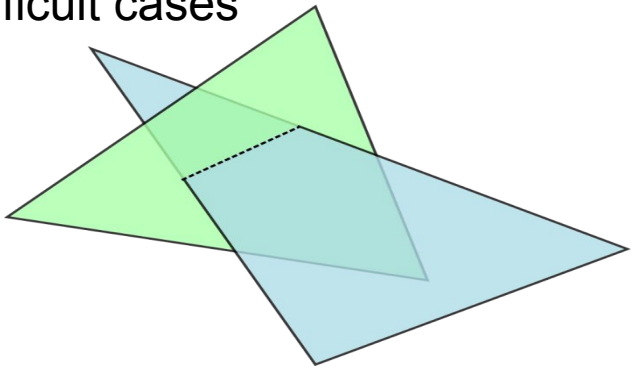


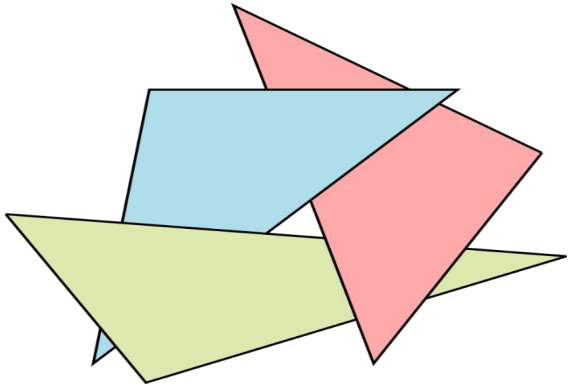
Occlusions (hidden surfaces)

Simple case



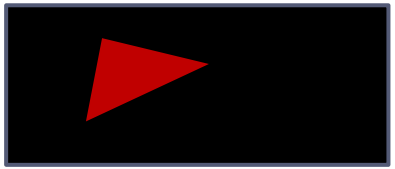
More difficult cases

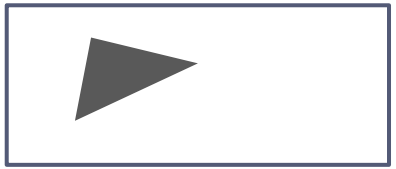




[FCG 8.2.3/9.2.3]

Z-Buffer - algorithm

Colour buffer 

Depth buffer 

- Initialize the depth buffer and image buffer for all pixels

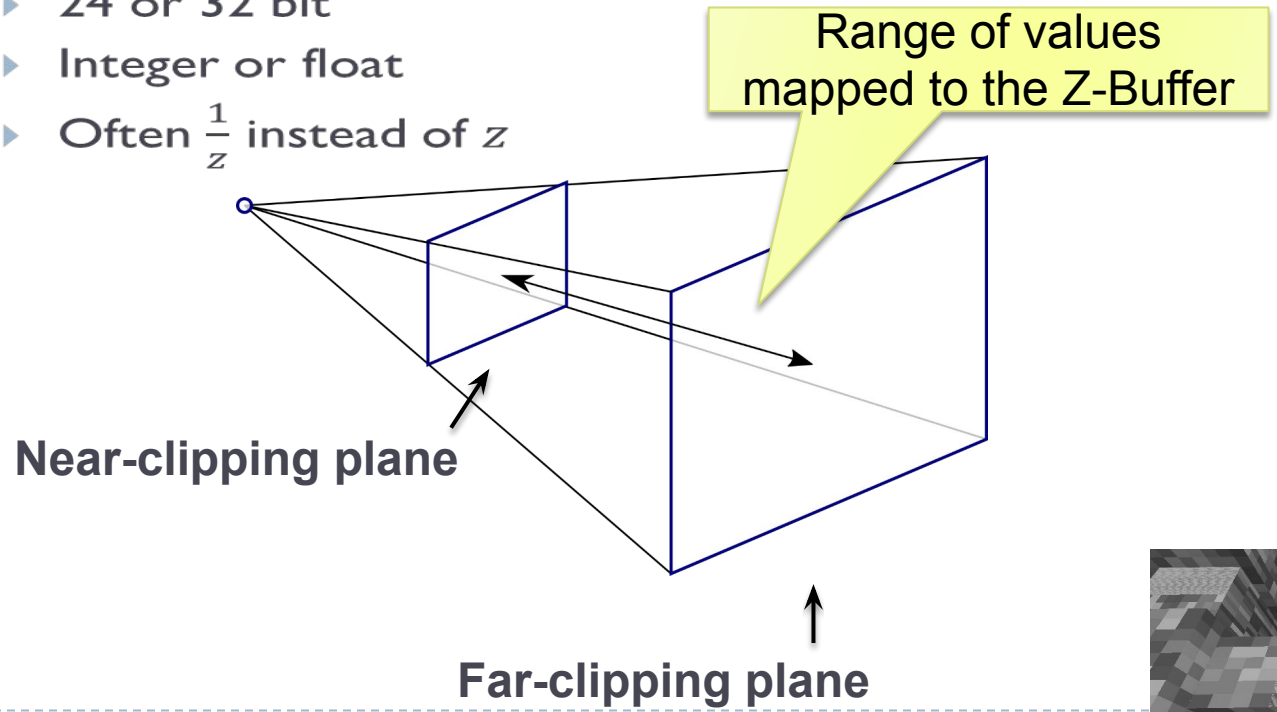
  colour(x, y) = Background_colour,

  depth(x, y) = z_{max} // position of the far clipping plane

- **For** every triangle in a scene **do**
  - **For** every fragment (x, y) in this triangle **do**
    - Calculate z for current (x, y)
    - **if** (z < depth(x, y)) and (z > z_{min}) **then**
      - depth(x, y) = z
      - colour(x, y) = fragment_colour(x, y)

View frustum and Z-buffer

- Z-buffer must store depth with sufficient precision
  - 24 or 32 bit
  - Integer or float
  - Often $\frac{1}{z}$ instead of z



Z-fighting