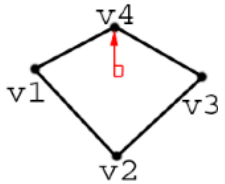
Object Representation
Affine Transforms

Polygonal Representation of Objects

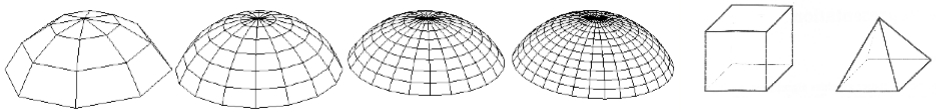
- Although perceivable the simplest form of representation they can also be the most problematic.
- To represent an object polygonally, a mesh of polygonal facets is generated.
- **Vertices** are points in 3D space.
- An **edge** is a line segment generated between 2 vertices.
- A **face** is a planar shape which consists of a closed sequence of edges.

Polygonal Representation



- In a polyhedron mesh, edges must be shared by 1 or 2 adjacent faces
- And similarly a vertex must be shared by at least 2 edges

Polygonal Representation

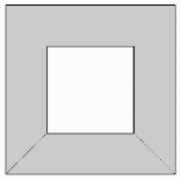


- The detail of the approximation used to represent an object can be increased by the utilisation of more polygon faces.
- Thus, smoother surfaces can be generated by the utilisation of more polygon faces.
- A polyhedron is a an entirely enclosed polygon mesh - a solid polygon mesh(?)

Polygon Representation

- Polygon faces are generated by the specification of vertices in an anti-clockwise manner
  - thus determining the direction of the vector perpendicular to the plane,
  - the face normal and effectively the front and back of the face itself.
- Vertices also contain normal information
  - it is used in shading models
  - but polygon faces are usually planar by nature so the face itself has a normal too.

Polygonal Representation

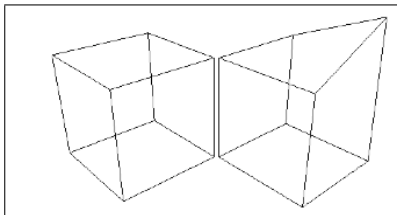


- The ability to have holes in polygon faces is easily clarified by this direction specified
- vertices that generate holes are done so in a clockwise order.
- Most CG applications however simply this notion further and require at least an edge
- joining the space between the hole and outer side of the polygon.

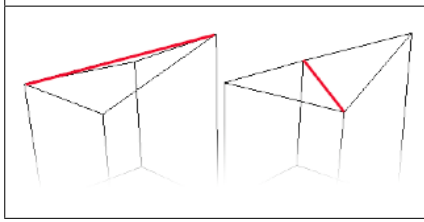
Planar Polygons

- A plane is defined by 3 points in space - the minimum amount of data required.
- Polygon faces that contain more than 3 vertices/ edges have the possibility to be non-planar.
- Polygon faces that contain four or more vertices therefore possibly cannot describe a plane.

Planar Polygons

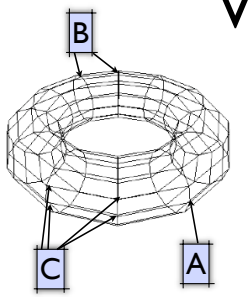


The rendering process of software prefers planar polygons, and as such each object is not only converted to polygons,



but also triangulated to ensure planar faces.

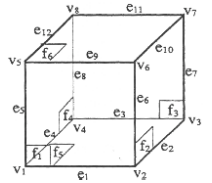
Visualisation of Object



A	Vertex List
B	Edge List
C	Face List

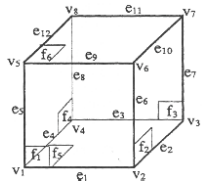
- There are many ways to visualise an object wire-frame ranging from wire-frame to fully rendered.
- We require at least a list of edges to draw (in the case of wire-frame).
- Such a list would require a list of vertices.
- Advanced image synthesis procedures require the face information as well.
- Therefore a polygonally represented object requires the elements in the table opposite.

Example A Cube



- With the example above we would therefore require:
- A. Vertex List
  - A list of vertices (v_1,v_2,v_3 ..... v_8) with Cartesian coordinates in the form $v_i=(v_{ix},v_{iy},v_{iz})$
- B Edge List
  - A list of each edge (e_1,e_2 .... e_{12}) Each edge is define by 2 vertices: $e_1=(v_1,v_2)$

Example A Cube

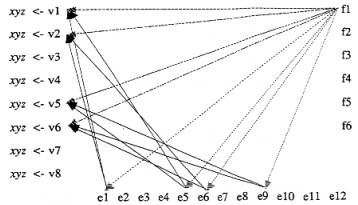


- C. Face List
  - A list of faces (f_1,f_2 .... f_6) Each face is defined as a determined sequence of edges or vertices.
- In both cases the number of edges / vertices used must be greater or equal to 3
- • e.g. $f_1=(e_1,e_6,e_9,e_5)$ or (v_1,v_2,v_6,v_5)

Example A Cube

- The preferred method is to store the positions of the vertices in memory and use pointers in the edge list to refer to the memory holding the vertex position.
- Like wise, these edge lists can be stored in memory and the face list can use pointers to refer to the memory with the correct edge data.

Example A Cube

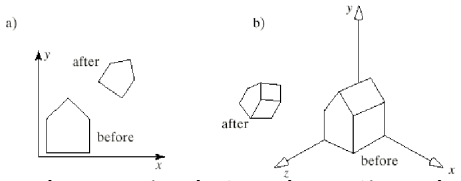


- Vertex information is only stored once in memory, and it is subsequently references many times.

Introduction to Affine Transforms

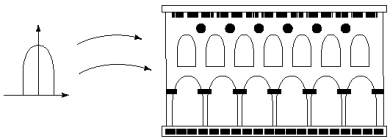
- Affine transforms are a fundamental computer graphics operation and are central to most graphics operations
- They also cause problems as it is difficult to get them right
- This is due to the difference between points and vectors and the fact that they do not transform in the same way.
- To overcome these problems we use homogeneous coordinates and an appropriate coordinate frame.

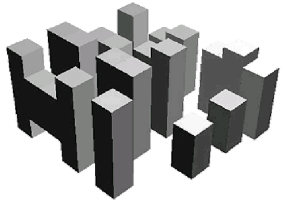
Transformations



- We have already seen simple transformation when we looked at window to viewport mapping
- The affine transforms are a more complex extension of these
- The following figure shows some transformations in 2D and 3D
- The Figure shows a house before and after all the points have been transformed

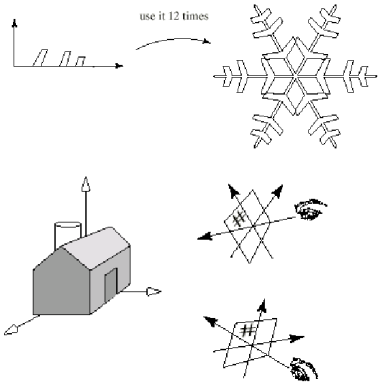
Why use transformations?





- We can compose a scene from a number of objects by placing instances of an object in the scene as shown in the first figure
- Or we can use simple primitives like cubes to build a city

Why use transforms

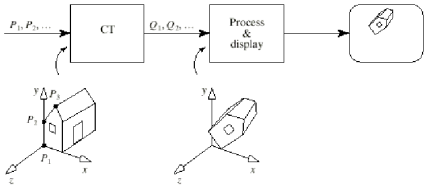


- Some objects may be symmetrical and can be created by using a "motif"
- We also use transforms to change how we look at objects from a camera

Why use transforms

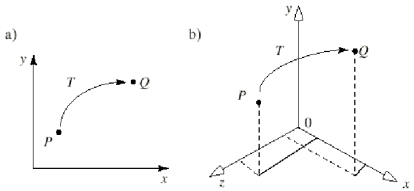
- We can transform complex object to provide animation, in the following case we use local transformation to move the limbs

Transformation in the Graphics Pipeline



- Affine transforms fit into the graphics pipeline as shown
- First points are sent down the pipeline (p_1,p_2 ..)
- Next the points encounters the current transform (CT) which change them to a new position
- After this they are displayed in their new position

Transforming Points and Objects



- A transformation alters each point **P** in space (2D or 3D) into a new point **Q** by means of a specific formula or algorithm
- In the above example the point **P** in the plane is mapped to another point **Q**
- We say that **Q** is the image of **P** under mapping **T**

Coordinate Frames and Transformations

- To keep things straight we use an explicit coordinate frame when performing transformations
- So we now have a particular point ϑ called the origin and some mutually perpendicular vectors *i,j,k* which serve as axis of the coordinate frame.
- Using this coordinate frame (in 2D for this example) we can represent P as

$$P = \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix} \quad \text{and} \quad Q = \begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix}$$

Coordinate Frames and Transformations

- we know that P "is at" location $P=P_x\mathbf{i}+P_y\mathbf{j}+\vartheta$ and the same for Q
- The values P_x and P_y are called the coordinates of P
- The transformation operates on the representation P and produces the representation Q according to some function T() or

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = T \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$$

- or more succinctly Q=T(P)

The affine transformations

- Affine transforms are the most common transformations in computer graphics
- They make it easy to scale, translate and rotate a figure
- A succession of affine transformations can easily be combined into a simple overall affine transformation
- This overall transformation can be put into a compact matrix form

The affine transformations

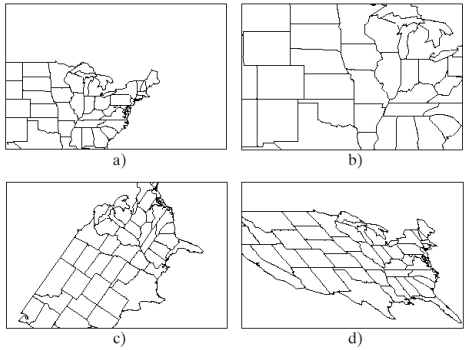
- The coordinates Q are linear combinations of those of P that is,

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{12}P_x + m_{22}P_y + m_{23} \\ 1 \end{bmatrix}$$

- For some six given constants m_{11},m_{12} etc. Q_x consists of portions both of P_x and P_y and so does Q_y
- This "cross fertilization" between x and y components gives rise to rotations and shears.
- To extend this we now represent the above equation as

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$$

Geometric Effects of 2D affine transforms



A) Translation

B) Scale

C) Rotation

D) Shear

Translation

- Translation is the movement of an object or image into a new position
- The translation part of the affine transform arises from the third column of the matrix

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & m_{13} \\ 0 & 1 & m_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$$

- or simply

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} P_x + m_{13} \\ P_y + m_{23} \\ 1 \end{bmatrix}$$

- So in ordinary coordinates Q=P+**d** where the "offset vector" **d** is [m_{13},m_{23}] so if the offset vector is [2,3] every point will be altered two units to the right and 3 units above.

Scaling

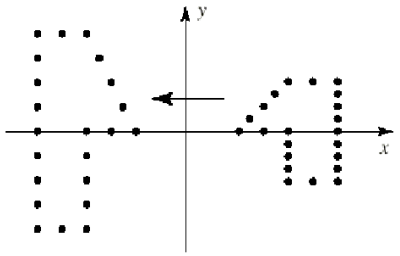
- Scaling changes the size of the picture and involves two scale factors S_x and S_y for the x and y coordinates

$[Q_x, Q_y] = [S_x P_x, S_y P_y]$ thus the matrix for scaling by S_x and S_y is $\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$

- Scaling in this fashion is more accurately called scaling about the origin because each point P is moved P_x times farther from the origin in the x-direction and S_y times farther from the origin in the y-direction.

Reflection as a special case of Scaling

- If a scale factor is negative then there is also a reflection about a coordinate axis.
- The figure below shows a scaling [S_x,S_y]=[-1,2] applied to a collection of points. Each point is reflected about the y axis and scaled by 2 in the y direction

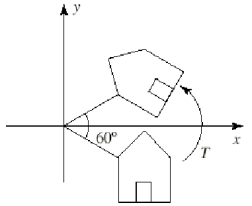


Pure reflections

- A "pure" reflection is where the scale factor is +1 or -1
- For example $T[P_x,P_y]=[-P_x,-P_y]$ will produce a mirror image of an image by "flipping" horizontally about the y-axis
- If the two scale factors are the same $[S_x=S_y=S]$ the transformation is a uniform scaling of a magnification about the origin with magnification factor |S|
- If |S|<1 the points will result in a reduction of size (demagnification)
- This is know as differential scaling

Rotation

- The following image shows a set of points rotated through an angle of θ=60°



- When T() is a rotation about the origin, the offset vector **d** is zero and Q=T(P) has the form

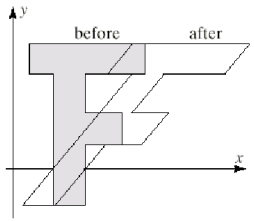
$$Q_x = P_x cos(\theta) - P_y sin(\theta)$$
$$Q_y = P_x sin(\theta) + P_y cos(\theta)$$

Rotation

- This causes positive values of θ to perform a counterclockwise (CCW) rotation and may be expressed in matrix form as

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

- Some applications (OpenGL, some XSI Functions etc) uses degrees for the measurement of angles for rotations, most C++ functions use radians, therefore care must be taken when expressing angles especially in scripting languages (Python maths functions for example)



Shearing

- The image shows a shearing in the x direction
- In this case the y coordinate of each point is unaffected whereas each x coordinate is translated by an amount that increases linearly with y

Shearing in the X direction

- A shear in the x direction is given by

$$\begin{aligned} \mathrm{Q}_x &= P_x + hP_y \\ \mathrm{Q}_y &= P_y \end{aligned}$$

- Where the coefficient h specifies what fraction of the y coordinate of P is to be added to the x coordinate. h can be positive or negative and expressed as a matrix gives us

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

Shearing in the Y direction

- We can also shear along the y axis using

$$\begin{aligned} \mathrm{Q}_x &= P_x \\ \mathrm{Q}_y &= gP_x + P_y \end{aligned}$$

- giving us the matrix

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

3D Affine Transforms

- The same ideas apply to 3D affine transforms as apply to 2D affine transforms
- However the expressions are more complex as we now have one extra coordinate
- Also the transforms are harder to visualise
- We still use coordinate frames with an origin ϑ and three mutually perpendicular axis in the direction **i**,**j** and **k**
- The point P in this frame is given by P=ϑ+P_x**i**+P_y**j**+P_z**k**

3D affine Transforms

- Any point can be expressed in the coordinate frame as $\mathrm{P}=\begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$
- Now using T() as a function to transform the points P to Q we use the matrix M as follows

$$\mathrm{M}=\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D affine Transforms

- Now using T() as a function to transform the points P to Q we use the matrix M as follows

$$\begin{bmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{bmatrix} = M \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

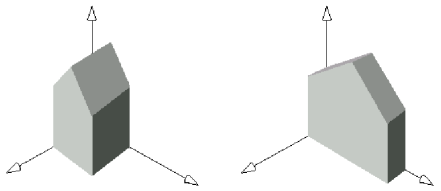
Translation

- For a pure translation the matrix has the following form

$$\begin{bmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where Q=MP is a shift in Q by the vector m=$[m_{14}, m_{24}, m_{34}]$

Scaling



- Scaling in 3D is a direct extension of the 2D case with a matrix

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- where S_x,S_y and S_z causes scaling of the corresponding coordinates.
- Scaling is about the origin as in the 2D as shown in the figure

Shearing

- 3D shears appear in greater variety compared to 2D versions
- The simplest shears are obtained by the identity matrix with one zero term replaced by some value as in

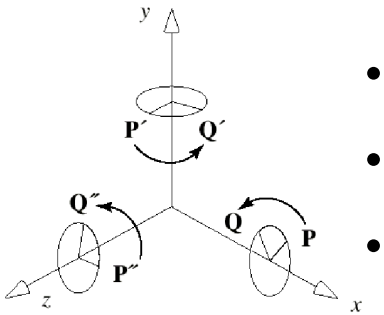
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- which produces Q=(P_x,fP_x+P_y,P_z) this gives P_y offset by some amount proportional to P_x and the other components are unchanged.
- For interesting applications for shears see the paper by Barr in the accompanying papers.

Rotations

- Rotations in 3D are common in graphics (to rotate objects, cameras etc)
- In 3D we must specify an axis about which the rotations occurs, rather than just a single point
- One helpful approach is to decompose a rotation into a combination of simpler ones.

Elementary rotations about a coordinate axis



- • The simplest rotation is a rotation about one of the coordinate axis.
- • We call a rotation about the x-axis an "x-roll" about the y a "y-roll" and z a "z-roll"
- • The figure shows the different "rolls" around the different axis

Elementary rotations about a coordinate axis

- The following three matrices represent transformations that rotate points through and angle β about a coordinate axis
- The angle is represented in radians

x-roll

$$\mathrm{R}_x(\beta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\beta) & -sin(\beta) & 0 \\ 0 & sin(\beta) & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

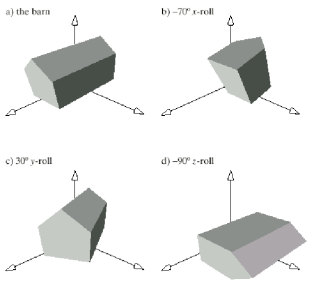
y-roll

$$\mathrm{R}_y(\beta) = \begin{bmatrix} cos(\beta) & 0 & sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

z-roll

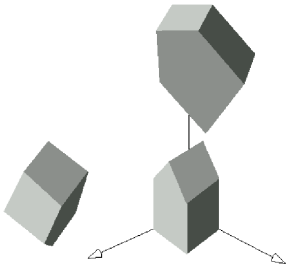
$$\mathrm{R}_z(\beta) = \begin{bmatrix} cos(\beta) & -sin(\beta) & 0 & 0 \\ sin(\beta) & cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Elementary rotations about a coordinate axis



- Note that 12 of the terms in each matrix are the zeros and ones of the identity matrix
- They occur in the row and column that correspond to the axis about which the rotation is being made
- These terms guarantee that the corresponding coordinate of the point being transformed will not be altered.
- An example of the different rolls are shown below

Composing 3D Affine Transforms



- Composing 3D affine transforms works the same way as in 2D
- We take the individual matrices for each rotation (M_1 and M_2) and then combine them by pre multiplying M_2 with M_1 to give $M=M_2M_1$
- Any number of affine transforms can be composed in this way, and a single matrix gives us the desired rotation.
- This is shown in the figure

Combining Rotations

- One of the most important distinctions between 2D and 3D transformations is the manner in which rotations combine
- In 2D two rotations R(β_1) and R(β_2) combine to produce R($\beta_1 + \beta_2$) and the order in which they combine make no difference
- In 3D the situation is more complex because rotations can be about different axes
- The order in which two rotations about different axes are performed does matter.
- 3D rotation matrices do not commute

3D rotations

- It is common to build a rotation in 3D by composing three elementary rotations
- An x-roll followed by a y-roll and then a z-roll
- Using the previous equations for the rolls we get

$$\mathrm{M}=\mathrm{R}_z(\beta_3)R_y(\beta_2)R_x(\beta_1)$$

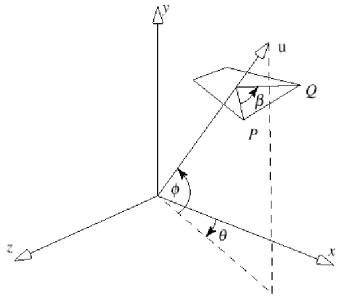
- In this context the angles β_1, β_2 and β_3 are called the **Euler** angles
- Euler's Theorem asserts that any 3D rotation can be obtained by three rolls about the x-, y- and z-axes
- so any rotation can be written as a particular product of 3 matrices for the appropriate choice of Euler angles.

Rotations about an Arbitrary Axis

- When using Euler angles we perform a sequence of x-,y- and z-rolls (rotations about coordinate axis)
- But it is much easier to work with rotations if we have a way to rotate about an axis that points in an arbitrary direction.
- Euler's Theorem states that every rotation can be represented as

Euler's Theorem : Any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point

Euler

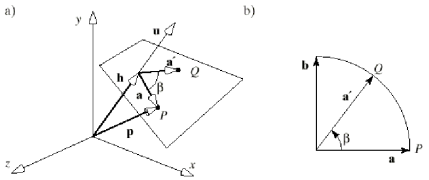


- The figure shows an axis represented by a vector **u** and an arbitrary point P that is to be rotated through angle β about **u** to produce the point Q
- Because **u** can have any direction it is difficult to find a single matrix that represents the rotation
- However we can do the rotation in one of two ways

The classic way

- We decompose the rotation into a sequence of known steps
  1. Perform two rotations so that **u** becomes aligned with the x-axis
  2. Do a z-roll through the angle β
  3. Undo the two alignment rotations to restore **u** to its original direction
- This method is similar to a rotation about a point in 2D The first step moves the point into the correct 2D location, we then do the rotation and finally replace into the original position (in the other axis no affected)
- The resultant equation is $\mathrm{R}_u(\beta) = R_y(-\theta)R_z(\phi)R_x(\beta)R_z(-\phi)R_y(\theta)$

The constructive way



- The above figure shows the axis of rotation **u** and the point P that we wish to rotate by β to make point Q
- As seen in figure b) the point Q is the linear combinations of the two vectors **a** and **b**
- Now we use cross products and dot products of the vectors to produce a final matrix as shown below

$$\mathrm{R}_u(\beta) = \begin{bmatrix} c+(1-c)u_x^2 & (1-c)u_yu_x - su_z & (1-c)u_zu_x + su_y & 0 \\ (1-c)u_xu_y + su_z & c+(1-c)u_y^2 & (1-c)u_zu_y - su_x & 0 \\ (1-c)u_xu_z - su_y & (1-c)u_yu_z + su_x & c+(1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 VinceSpringer-Verlag London
- "Geometry for Computer Graphics: Formulae, Examples and Proofs" John Vince Springer-Verlag London 2004