Geometric Transformation

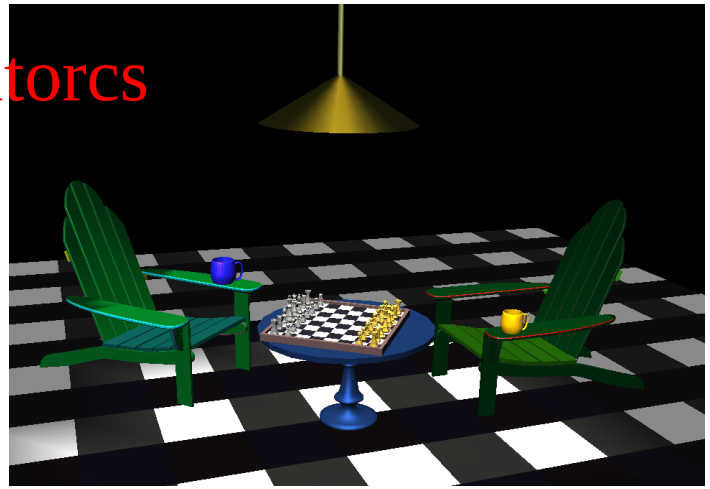
Frederick Li

Durham University

This Lesson

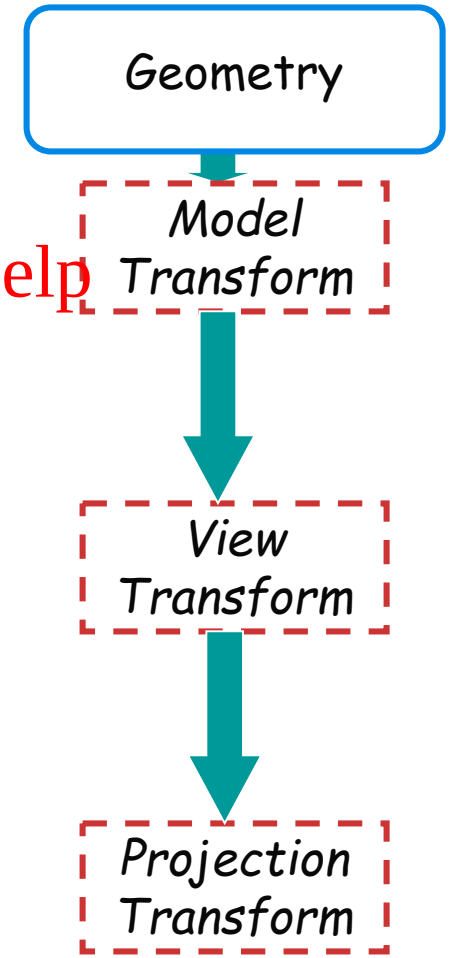
- Mathematics of geometric transformations
  - Translate, rotate, scale, etc.
  - Homogenous coordinates
- Incorporation of geometric transformations in WebGL programs

- Note: This lesson provides a mathematical foundation of 3D object and scene construction, which is important to 3D modelling and graphics programming.



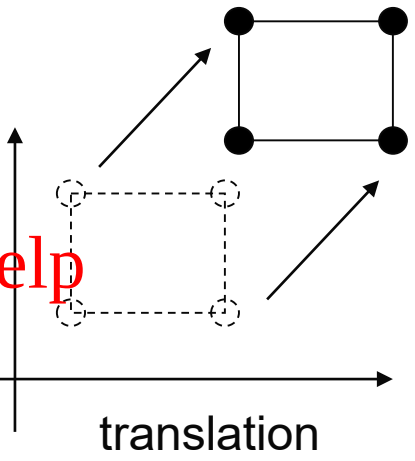
Transformations in Rendering Pipeline

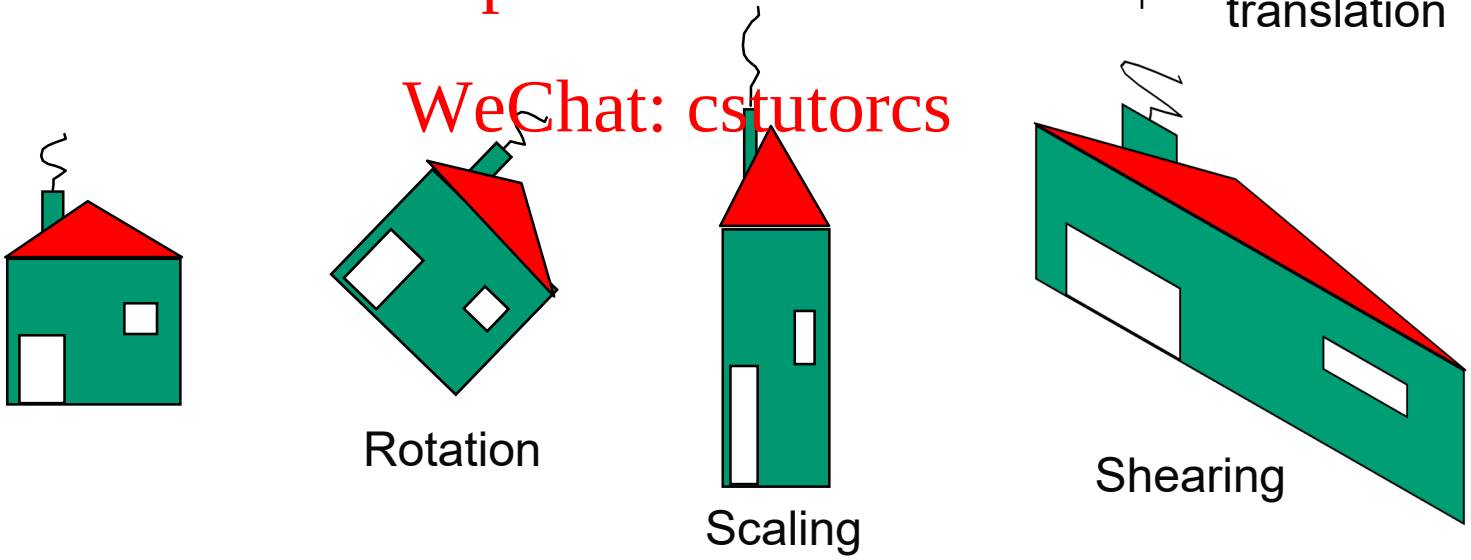
- Three parts
  - Model transform
    - Place objects within a 3D scene
    - Apply transformations: translate, rotate, scale, …, etc. to place objects
  - View transform
    - Place a virtual camera
    - Define from where you would like to look at the 3D scene
  - Projection transform
    - Change the type of virtual frustum
    - Apply orthogonal / perspective projection



Geometric Transformation

- Types
  - Translation – change position
  - Scaling – change size
  - Rotation – change orientation
  - Shear – change shape
- Implemented by transformation matrix

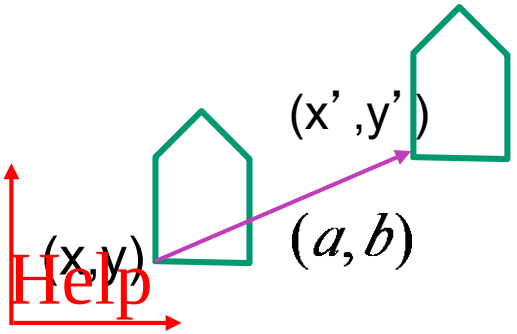




2D Translation and Scaling

Translation: by vector addition

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$



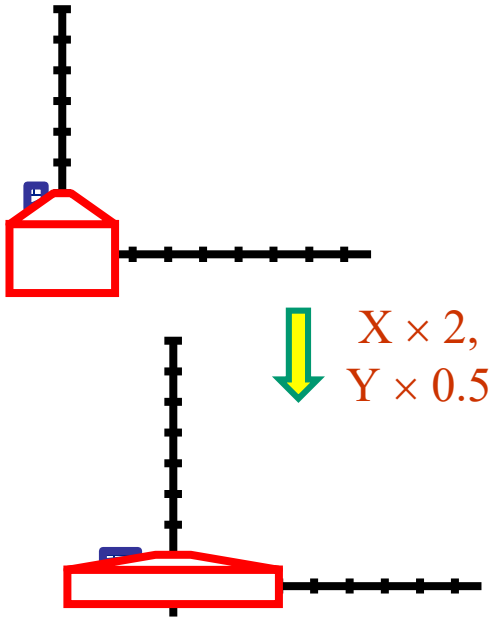
- Scaling operation: (vector form)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ax \\ by \end{bmatrix}$$

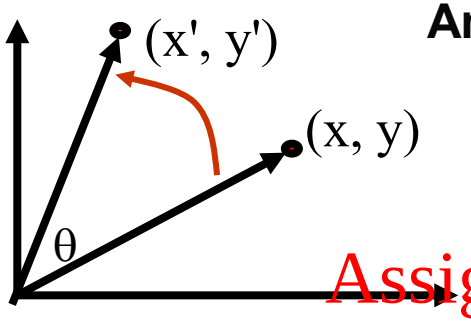
- or, in matrix form:

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

scaling matrix



2D Rotation and Shear

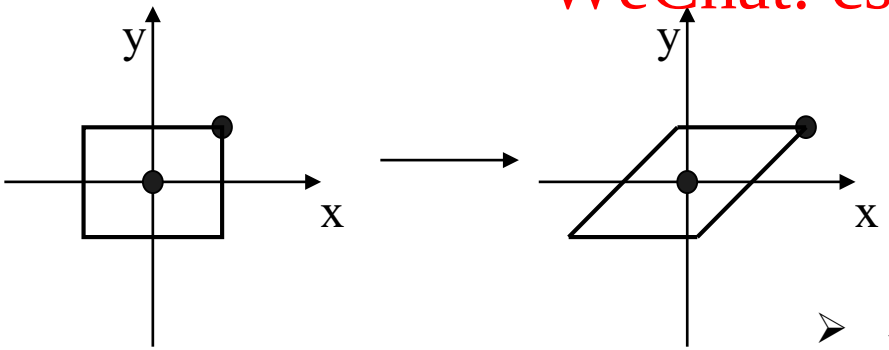


Anti-clockwise Rotation and its Matrix Form:

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

Rotation matrix

$$x' = x\ \mathbf{cos}(\theta) - y\ \mathbf{sin}(\theta)$$
$$y' = x\ \mathbf{sin}(\theta) + y\ \mathbf{cos}(\theta)$$



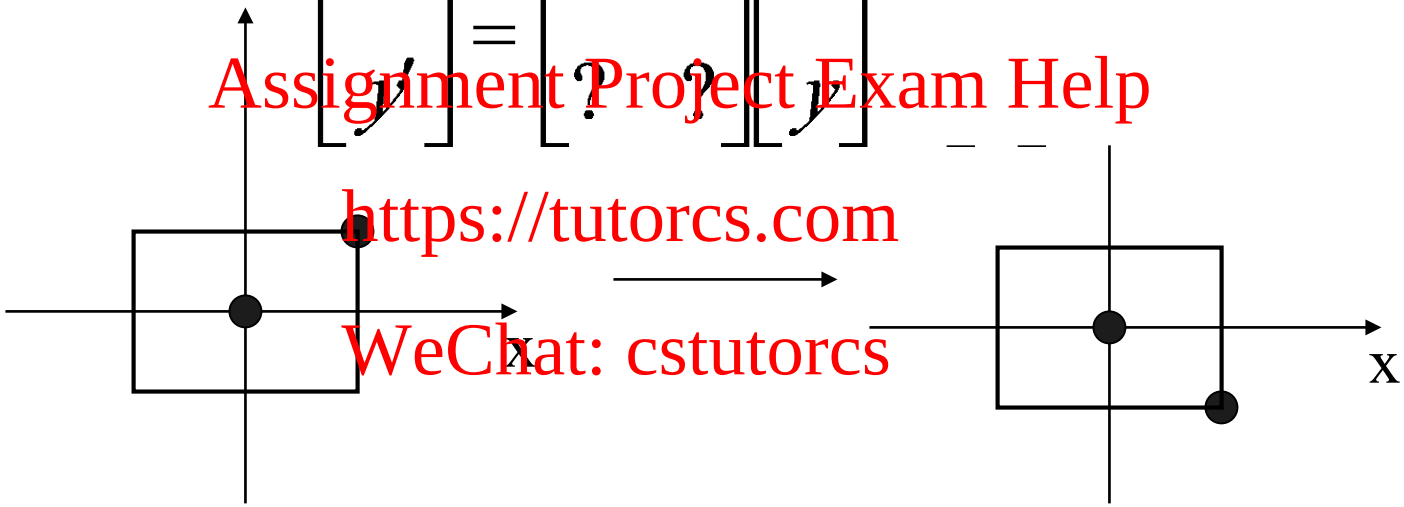
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & sh_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- **shear along x axis → push points to right in proportion to height**

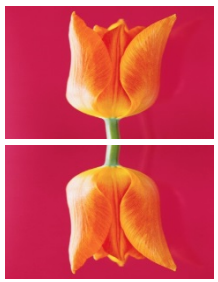
2D Reflection

- reflect across x axis → mirror

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

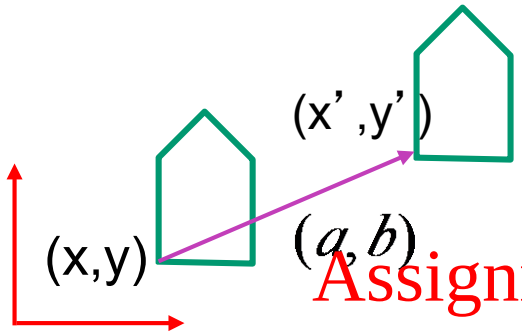


Example:



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

2D Translation



vector addition

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

matrix multiplication

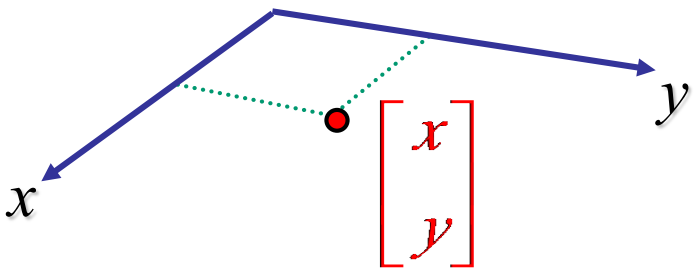
matrix multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\text{scaling matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}}_{\text{rotation matrix}} \begin{bmatrix} x \\ y \end{bmatrix}$$

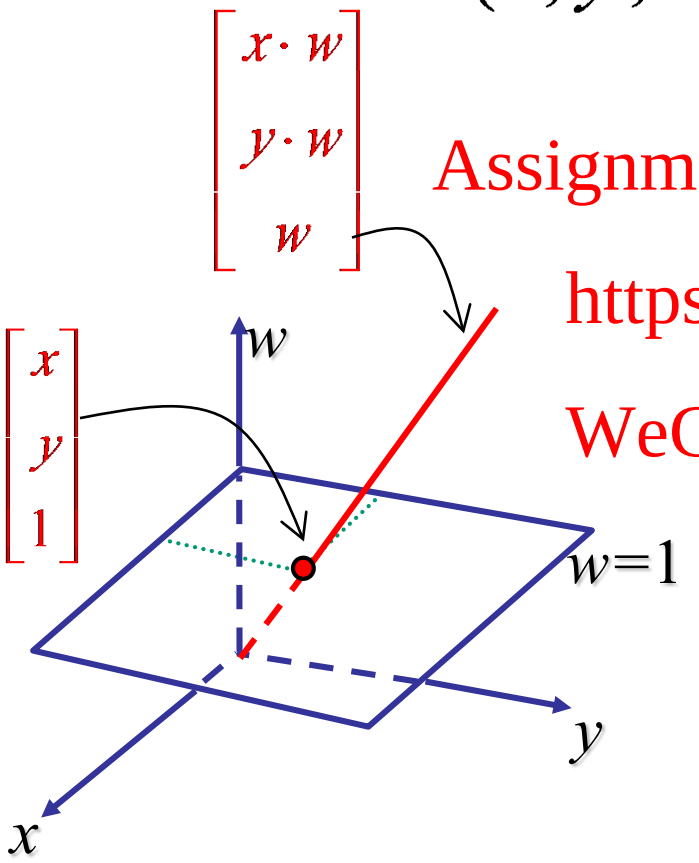
Homogeneous Coordinates

- Suppose we have a point (x,y) in the Euclidean plane. We can represent this same point in a projective plane, and the characteristics of its geometry and transformations are all preserved.
  - point in 2D cartesian



homogeneous **cartesian**

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



- point in 2D cartesian + weight w = point P in 3D homogenous coords
- multiples of (x,y,w)
- form a line L in 3D
- all homogeneous points on L represent same 2D cartesian point
- example:
  (2,2,1) = (4,4,2) = (1,1,0.5)

Transformation matrix using Homogeneous Coordinates

- 2D transformation matrices now become 3x3:

$$\mathbf{R}otation = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{S}cale = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}rans\,lation = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x*1+a*1 \\ y*1+b*1 \\ 1 \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ 1 \end{bmatrix}$$

3D Transformation

- Rotation about z-axis

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

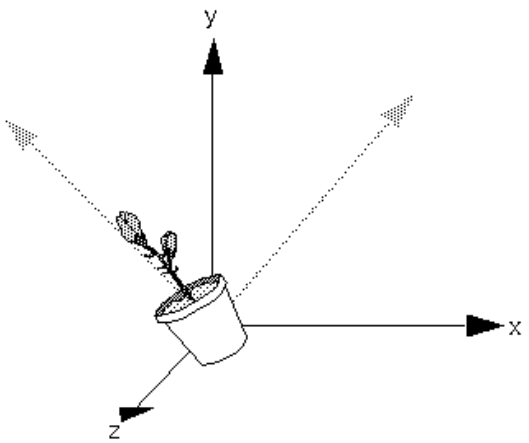
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  - WebGL command

    Matrix4.setRotate(angle,x,y,z);

  - rotate around z-axis

    Matrix4.setRotate(angle,0,0,1);



3D Rotation in X, Y

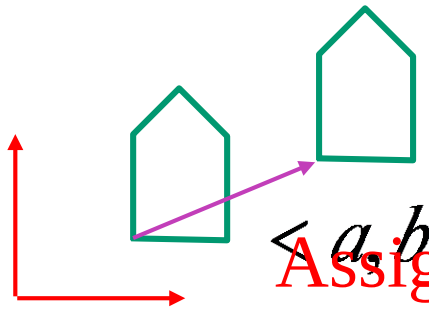
around x-axis: **Matrix4.setRotate(angle,1,0,0);**

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

around y-axis: **Matrix4.setRotate(angle,0,1,0);**

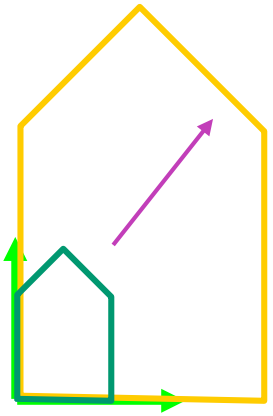
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3D Translation and Scaling



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

Matrix4.translate(a,b,c);



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

Matrix4.scale(a,b,c);

- **General shear:**

$$shear(hxy, hxz, hyx, hyz, hzx, hzy) = \begin{bmatrix} 1 & hyx & hzx & 0 \\ hxy & 1 & hzy & 0 \\ hxz & hyz & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- to avoid ambiguity, always say "shear along <axis> in direction of <axis>"

$$shearAlongXinDirectionOfY(h) = \begin{bmatrix} 1 & h & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad shearAlongXinDirectionOfZ(h) = \begin{bmatrix} 1 & 0 & h & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$shearAlongYinDirectionOfX(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ h & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad shearAlongYinDirectionOfZ(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & h & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$shearAlongZinDirectionOfX(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ h & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad shearAlongZinDirectionOfY(h) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Table 4.1 The Methods and Properties Supported by Matrix4

Methods and Properties	Description
`Matrix4.setIdentity()`	Initialize a matrix (to the identity matrix*).
`Matrix4.setTranslate(x, y, z)`	Set `Matrix4` to the translation matrix, which translates *x* units in the direction of the x-axis, *y* units in the direction of the y-axis, and *z* units in the direction of the z-axis.
`Matrix4.setRotate(angle, x, y, z)`	Set `Matrix4` to the rotation matrix, which rotates *angle* degrees around the rotation axis (*x*, *y*, *z*). The (*x*, *y*, *z*) coordinates do not need to be **normalized**. (See Chapter 8, "Lighting Objects.")
`Matrix4.setScale(x, y, z)`	Set `Matrix4` to the scaling matrix with scaling factors *x*, *y*, and *z*.
`Matrix4.translate(x, y, z)`	Multiply the matrix stored in `Matrix4` by the translation matrix, which translates *x* units in the direction of the x-axis, *y* units in the direction of the y-axis, and *z* units in the direction of the z-axis, storing the result back into `Matrix4`.
`Matrix4.rotate(angle, x, y, z)`	Multiply the matrix stored in `Matrix4` by the rotation matrix, which rotates *angle* degrees around the rotation axis (*x*, *y*, *z*), storing the results back into `Matrix4`. The (*x*, *y*, *z*) coordinates do not need to be normalized. (See Chapter 8.)
`Matrix4.scale(x, y, z)`	Multiply the matrix stored in `Matrix4` by the scaling matrix, with scaling factors *x*, *y*, and *z*, storing the results back into `Matrix4`.

Methods and Properties	Description
`Matrix4.set(m)`	Set the matrix *m* to `Matrix4`. *m* must be a `Matrix4` object.
`Matrix4.elements`	The typed array (`Float32Array`) containing the elements of the matrix stored in `Matrix4`.

Model Transform – Example

- Translate first, then rotate:
  i.e. < rotation matrix > × < translation matrix >
- **WebGL Code**


```
Javascript main()
var modelMatrix = new Matrix4();
modelMatrix.setRotate(ANGLE, 0, 0, 1);
modelMatrix.translate(Tx, 0, 0);

var u_ModelMatrix =
    gl.getUniformLocation(gl.program, ' u_ModelMatrix');
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
```


-

```
Vertex Shader Program
'void main() {\n' +
' gl_Position = u_ModelMatrix * a_Position;\n' +
'}\n';
```

Transformation Composition

- e.g. Rotate line segment by 45 degrees about endpoint

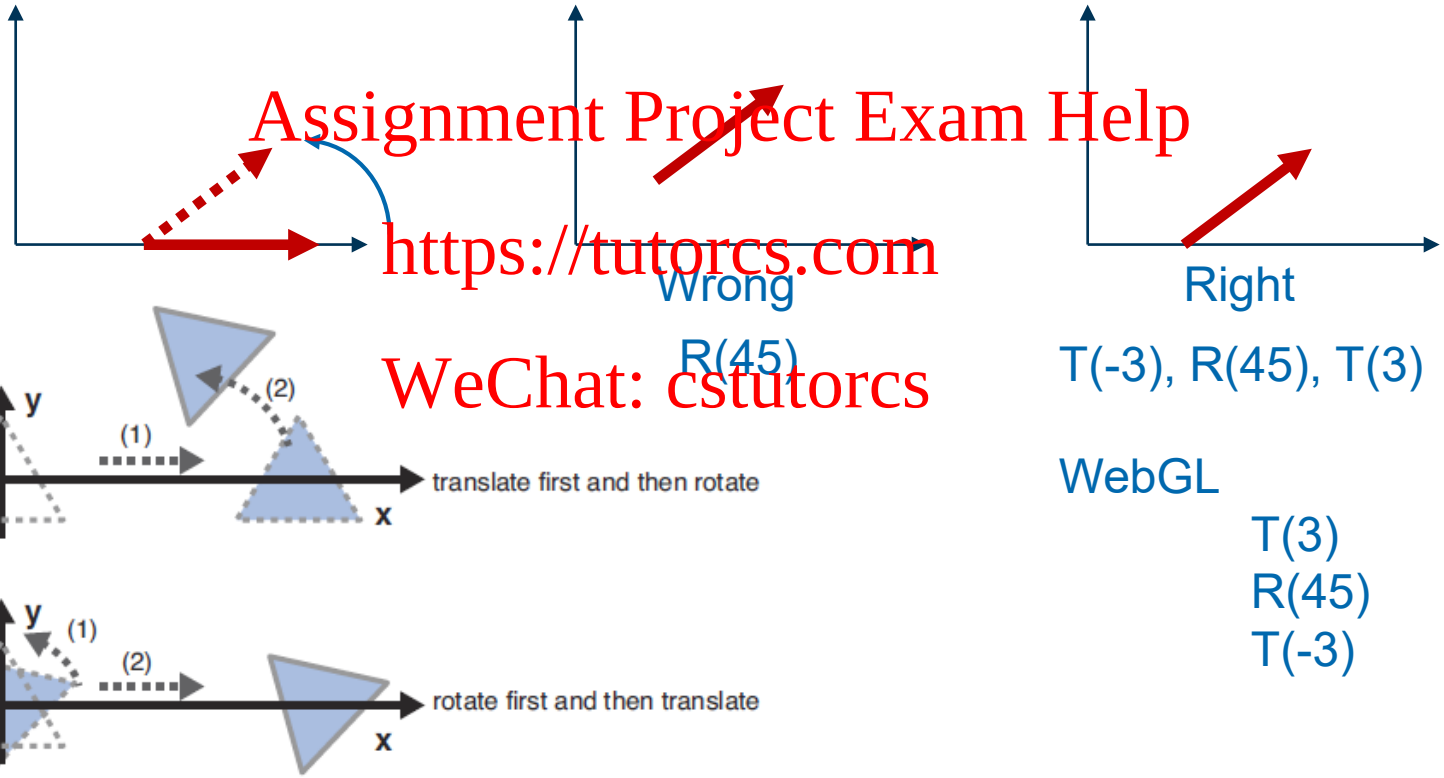


Figure 4.5 The order of transformations will show different results

View Transform (Virtual Camera)

Figure 7.2 LookAtTriangles (left), and the color and z coordinate of each triangle (right)

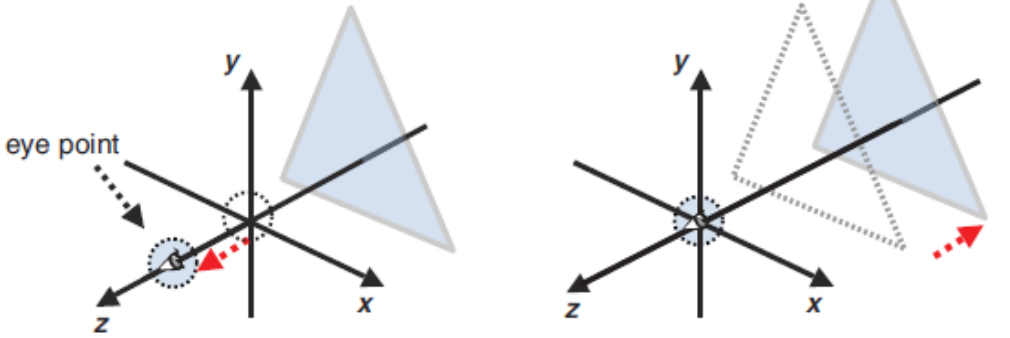
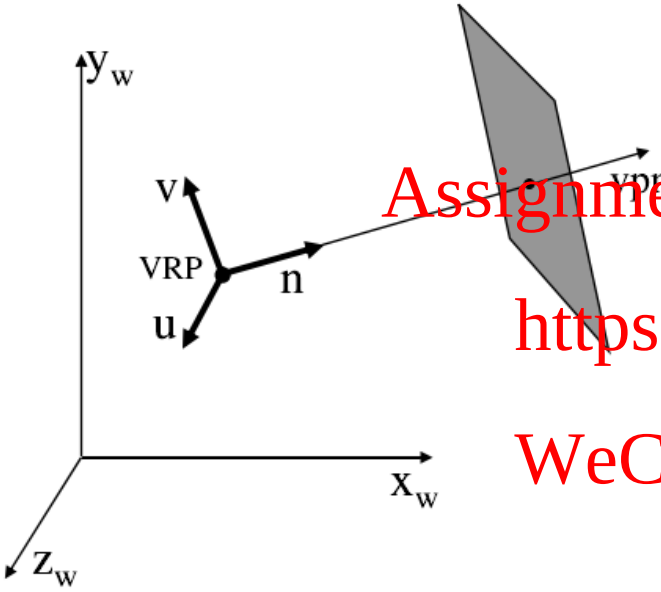


Figure 7.6 Movement of the eye point is identical to that of objects in the scene

Figure 7.18 LookAtTrianglesWithKeys_ViewVolume

View Transform Matrix (Virtual Camera)



$$\begin{pmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix} = \begin{pmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \\ 1 \end{pmatrix}$$

This slide is for reference only.
Mathematics details are not examinable.

Example – LookAtTriangles

- Virtual Camera: setLookAt()
  - Parameters: Eye Position, Look-at Position, Camera Orientation

$(x_{eye}, y_{eye}, z_{eye})$ (x_{at}, y_{at}, z_{at}) (dir_x, dir_y, dir_z)


```
// Set the matrix to be used for to set the camera view
var viewMatrix = new Matrix4();
viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);

// Set the view matrix
gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);

// Draw the rectangle
gl.drawArrays(gl.TRIANGLES, 0, n);
```

(main program)

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute vec4 a_Color;\n' +
  'uniform mat4 u_ViewMatrix;\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_Position = u_ViewMatrix * a_Position;\n' +
  '  v_Color = a_Color;\n' +
  '}\n';
```

Example – LookAtRotatedTriangles

```
// Set the matrix to be used for to set the camera view
var viewMatrix = new Matrix4();
viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);

// Calculate matrix for rotate
var modelMatrix = new Matrix4();
modelMatrix.setRotate(-10, 0, 0, 1); // Rotate around z-axis

// Pass the view projection matrix and model matrix
gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

// Draw the rectangle
gl.drawArrays(gl.TRIANGLES, 0, n);
```

Define a rotation matrix to manipulate 3D objects

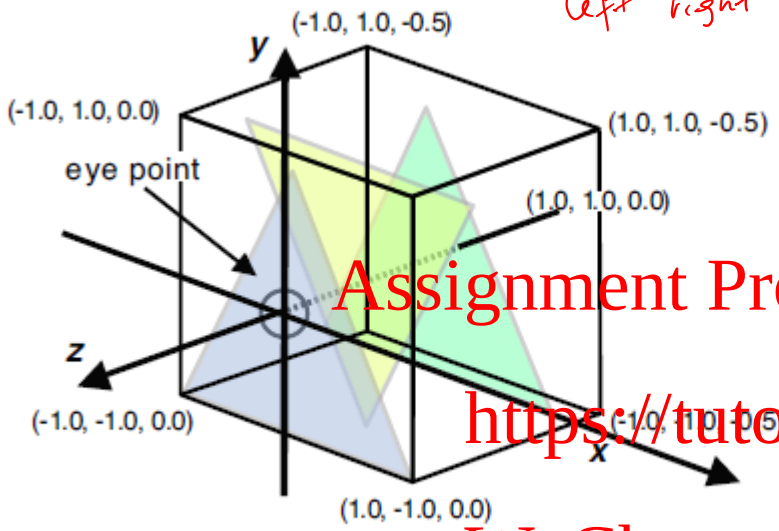
Modified part of vertex shader

```
'void main() {\n' +
'  gl_Position = u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
'  v_Color = a_Color;\n' +
'}\n';
```

Projection Transform – Orthogonal Projection

projMatrix.setOrtho(-1.0, 1.0, -1.0, 1.0, g_near, g_far);

left right bottom up



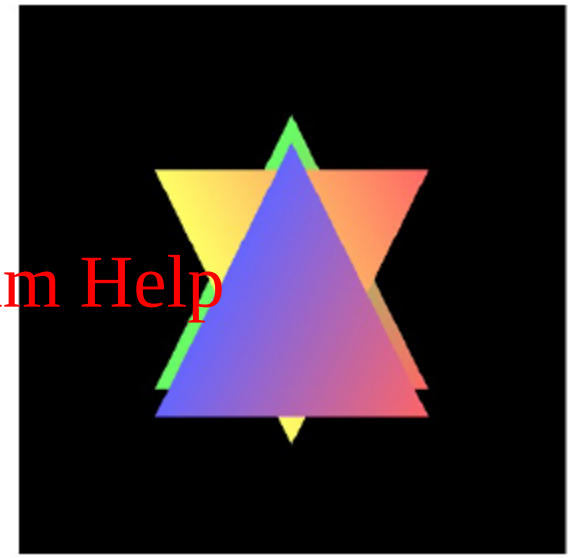
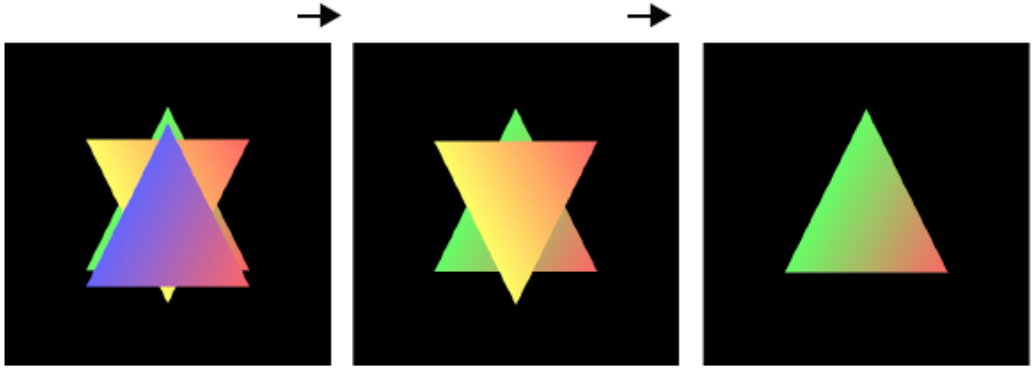


Figure 7.12 The box-shaped viewing volume used in OrthoView



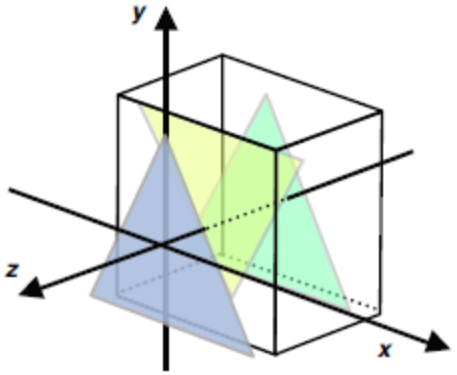


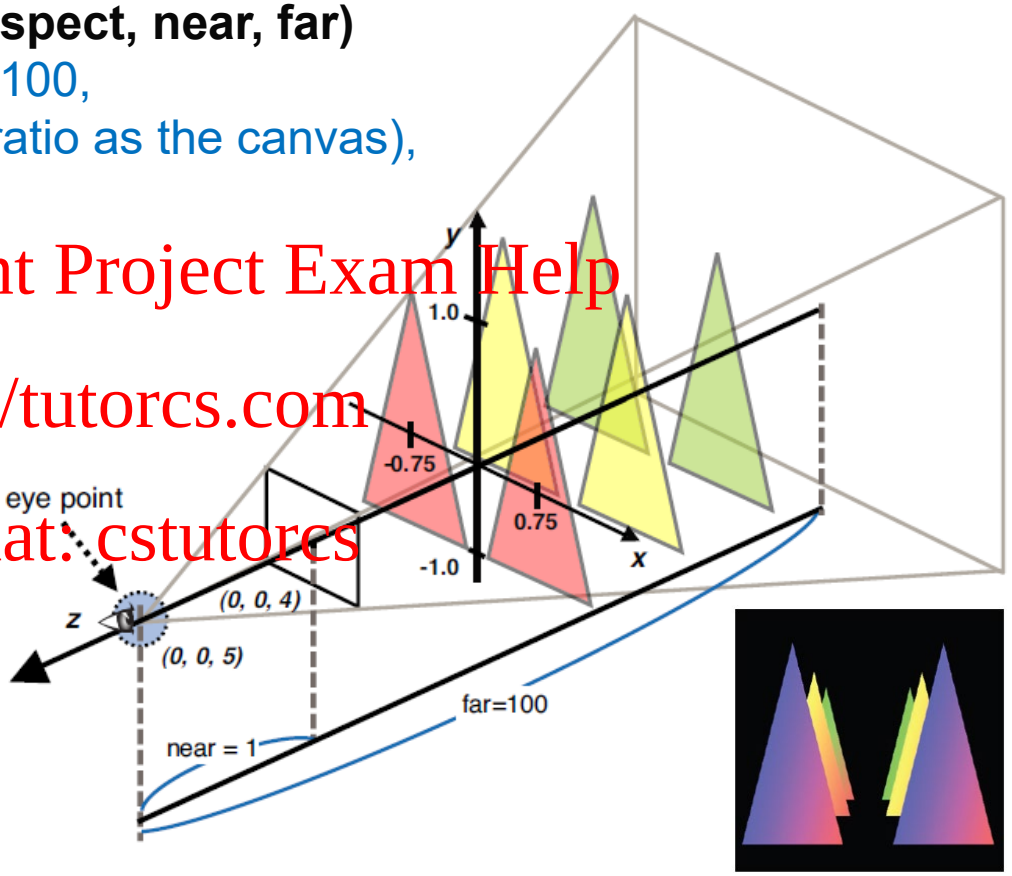
Figure 7.14 Increase the near value using the right arrow key

Projection Transform – Perspective Projection

Matrix4.setPerspective(fov, aspect, near, far)
It is specified by near=1.0, far=100,
aspect =1.0 (the same aspect ratio as the canvas),
and fov=30.0.

Perspective Projection matrix
(not for exam):

$$P = \begin{bmatrix} \frac{\cot\frac{fovy}{2}}{aspect} & 0 & 0 & 0 \\ 0 & \cot\frac{fovy}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2*n*f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$




```
V.shader: gl_Position = u_ProjMatrix * u_ViewMatrix * a_Position;
```

Example – PerspectiveView_mvp

- Support perspective projection

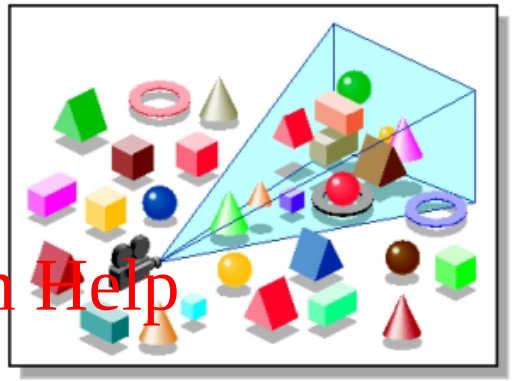
```
// Calculate the view matrix and the projection matrix
modelMatrix.setTranslate(0.75, 0, 0);  // Translate 0.75 units along the positive x-axis
viewMatrix.setLookAt(0, 0, 5, 0, 0, -100, 0, 1, 0);
projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
// Pass the model, view, and projection matrix to the uniform variable respectively
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);

gl.drawArrays(gl.TRIANGLES, 0, n); // Draw the triangles
```

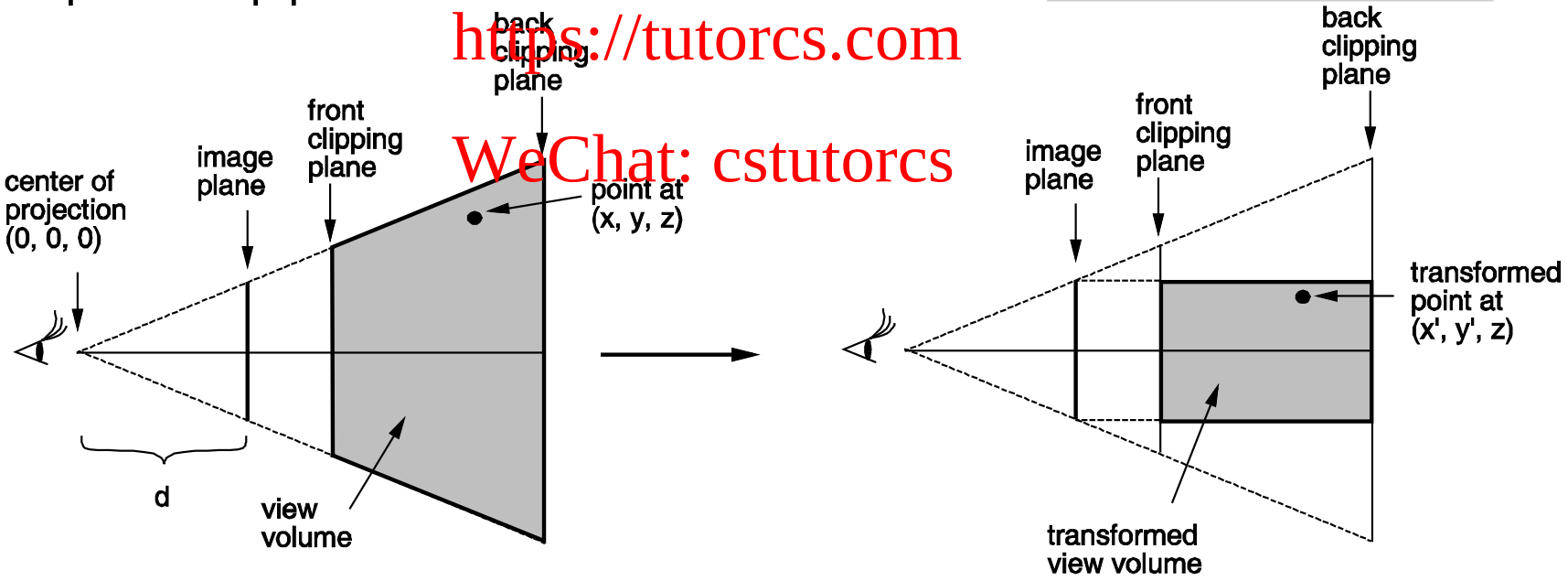

Modified part of vertex shader

```
'void main() {\n' +
'  gl_Position = u_ProjMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
'  v_Color = a_Color;\n' +
'}\n';
```

How Perspective Transformation Works?

- **Perspective Transformation:**
  transforms each object such that distant object will appear smaller. After the transformation, the view volume in the shape of a frustum will become a regular parallelepiped





How Perspective Transformation Works?

The transformation equations are as follows:

$$x' = d * x / z$$

$$y' = d * y / z$$

$$z' = z$$

where (x, y, z) is the original position of a vertex,
(x', y', z') is the transformed position of the vertex, and
d is the distance of the image plane from the center of projection.

In image generation, we usually apply **perspective projection**.
Perspective transformation is different from **perspective projection** in that perspective transformation retains the depth value of each transformed vertex, while perspective projection does not.

Reference

- WebGL Programming Guide [Ch. 3, 4, 7]