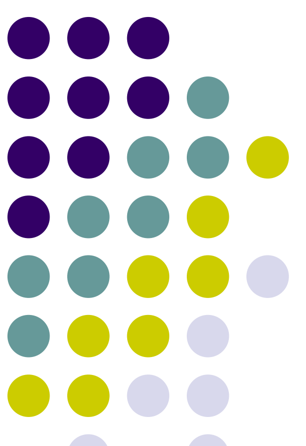
Computer Graphics (CS 543)
Lecture 3 (Part 2): Building 3D Models & Introduction to Transformations

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

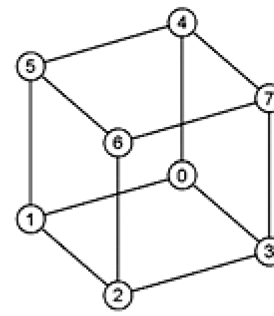


Full Example: Rotating Cube in 3D

- **Desired** Program behaviour:
  - Draw colored cube
  - Use 3-button mouse to change direction of rotation
  - Use idle function to increment angle of rotation
- **Note:** Default camera?
  - If we don't set camera, we get a default camera
  - Located at origin and points in the negative z direction

Cube Vertices

```
//  Declare array of vertex positions
//  (x,y,z,w) coordinates of the
//  vertices of a unit cube centered at origin
//  sides aligned with axes

point4 vertices[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```



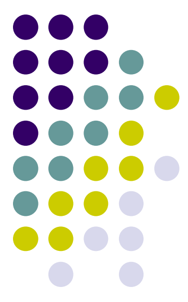
Colors

```
//  Declare array of vertex colors
// Unique set of RGBA colors that vertices can have

color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

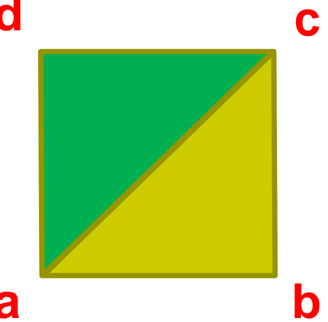
Color Cube

```
// generate 6 quads, sides of cube (12 triangles)

void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Function **quad** is
Passed vertex indices



```
// quad generates two triangles (a,b,c) and (a,c,d) for each face and
// assigns colors to the vertices

int Index = 0;  // Index goes from 0 to 5, one for each vertex of face

void quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++
    colors[Index] = vertex_colors[b]; points[Index] = vertices[b]; Index++
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++
    colors[Index] = vertex_colors[a]; points[Index] = vertices[a]; Index++
    colors[Index] = vertex_colors[c]; points[Index] = vertices[c]; Index++
    colors[Index] = vertex_colors[d]; points[Index] = vertices[d]; Index++
}
```



Initialization I

```
void init()
{
    colorcube(); // Generates cube data in application using quads


    // Create a vertex array object

    GLuint vao;
    glGenVertexArrays ( 1, &vao );
    glBindVertexArray ( vao );
```

Initialization II

```
// Create and initialize a buffer object
// and move data to GPU

    GLuint buffer;
    glGenBuffers( 1, &buffer );
    glBindBuffer( GL_ARRAY_BUFFER, buffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(points) +
                  sizeof(colors), NULL, GL_STATIC_DRAW );
```

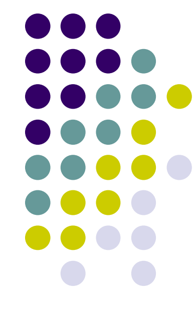
Initialization III

Specify points[ ] and colors[ ] data
Separately using **glBufferSubData**

```
glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
                                 sizeof(colors), colors );

// Load shaders and use the resulting shader program
    GLuint program = InitShader( "vshader36.glsl", "fshader36.glsl" );
    glUseProgram( program );
```

Initialize vertex and fragment shaders

Initialization IV



Specify vertex data

```
// set up vertex arrays

    GLuint vPosition = glGetAttribLocation( program, "vPosition" );
    glEnableVertexAttribArray( vPosition );
    glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
                           BUFFER_OFFSET(0) );
```

Specify color data

```
    GLuint vColor = glGetAttribLocation( program, "vColor" );
    glEnableVertexAttribArray( vColor );
    glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
                           BUFFER_OFFSET(sizeof(points)) );

    theta = glGetUniformLocation( program, "theta" );
```

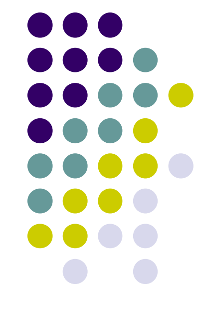
Connect variable theta in program
To variable in shader

Display Callback

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );

    glutSwapBuffers();
}
```

Draw series of triangles forming cube

Mouse Callback



```
void mouse( int button, int state, int x, int y )
{
    if ( state == GLUT_DOWN ) {
        switch( button ) {
            case GLUT_LEFT_BUTTON:    axis = Xaxis;  break;
            case GLUT_MIDDLE_BUTTON:  axis = Yaxis;  break;
            case GLUT_RIGHT_BUTTON:   axis = Zaxis;  break;
        }
    }
}
```

Select axis (x,y,z) to rotate around
Using mouse click

Idle Callback

```
void idle( void )
{
    theta[axis] += 0.01;

    if ( theta[axis] > 360.0 ) {
        theta[axis] -= 360.0;
    }

    glutPostRedisplay();
}
```

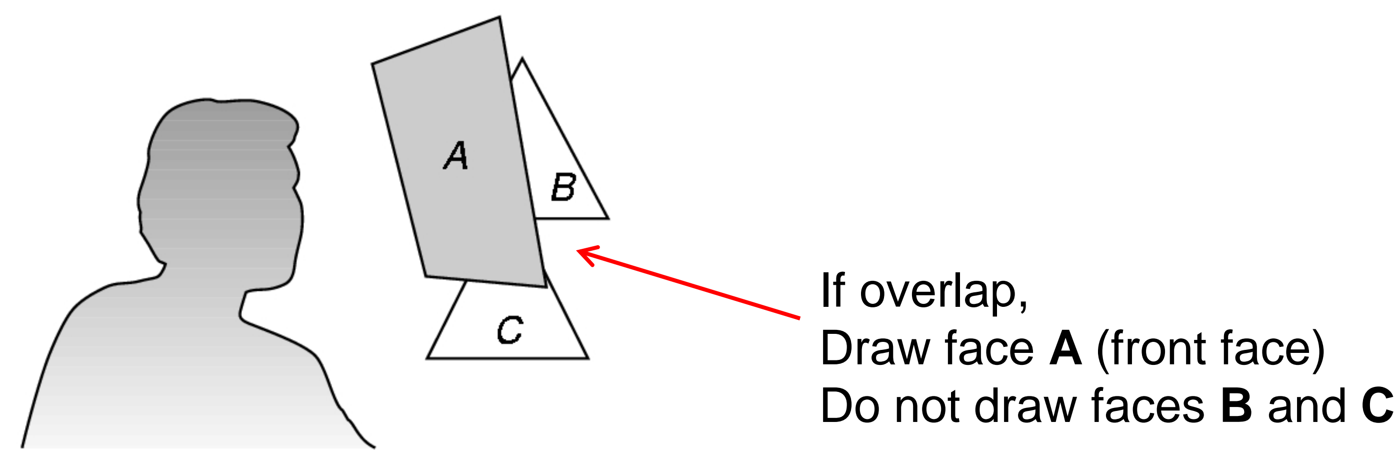
The idle( ) function is called Whenever nothing to do Rotate by theta = 0.01 around axes.

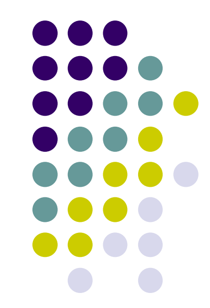
Note: still need to:

- Apply rotation by (theta) in shader

Hidden-Surface Removal

- We want to see only surfaces in front of other surfaces
- OpenGL uses *hidden-surface* technique called the ***z-buffer*** algorithm
- Z-buffer uses distance from viewer (depth) to determine closer objects
- Objects rendered so that only front objects appear in image





- Z-buffer uses an extra buffer, (the z-buffer), to store depth information as geometry travels down the pipeline
- 3 steps to set up Z-buffer:
  1. In **main.c**

     ```
     glutInitDisplayMode
        (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)
     ```

  2. Enabled in **init.c**

     ```
     glEnable(GL_DEPTH_TEST)
     ```

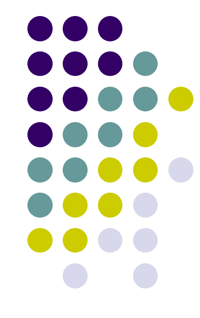
  3. Cleared in the display callback

     ```
     glClear(GL_COLOR_BUFFER_BIT | DEPTH_BUFFER_BIT)
     ```

3D Mesh file formats

- 3D meshes usually stored in 3D file format
- Format defines how vertices, edges, and faces are declared
- Over 400 different file format
- **Polygon File Format (PLY)** used a lot in graphics
- Originally PLY was used to store 3D files from 3D scanner
- We can get PLY models from web to work with
- We will use PLY files in this class

Sample PLY File



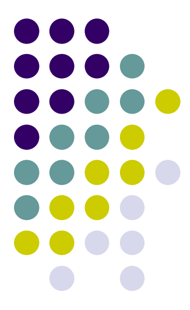
```
ply
format ascii 1.0
comment this is a simple file
obj_info any data, in one line of free form text
element vertex 3
property float x
property float y
property float z
element face 1
property list uchar int vertex_indices
end_header
-1 0 0
0 1 0
1 0 0
3 0 1 2
```

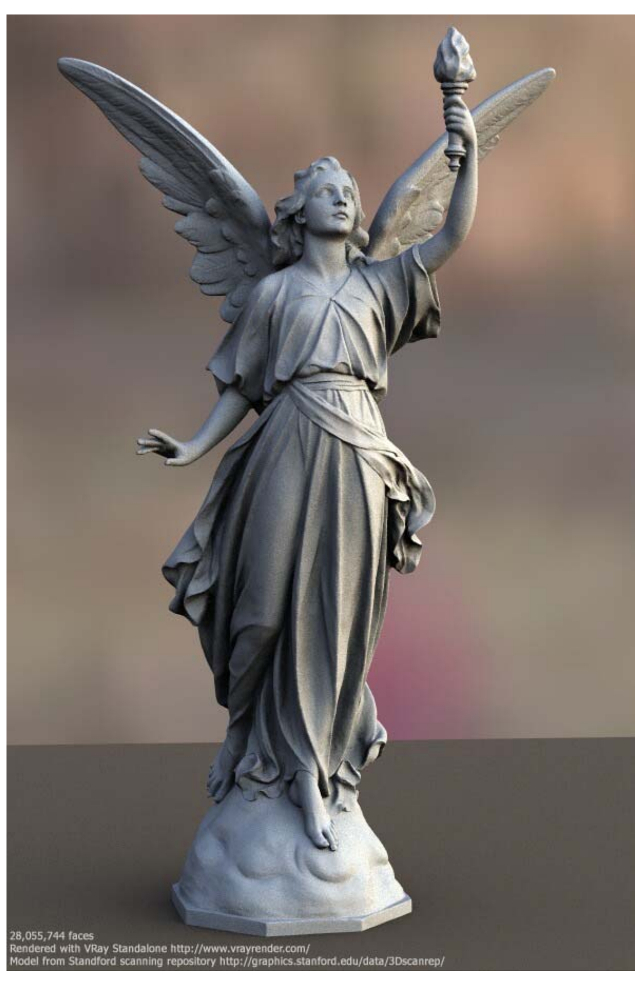
Georgia Tech Large Models Archive

Models

Stanford Bunny
Turbine Blade
Skeleton Hand
Dragon
Happy Buddha
Horse
Visible Man Skin
Visible Man Bone
Grand Canyon
Puget Sound
Angel

Stanford 3D Scanning Repository

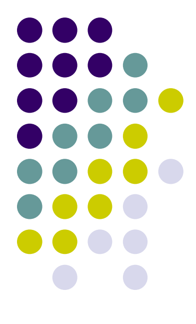




Lucy: 28 million faces

Happy Buddha: 9 million faces

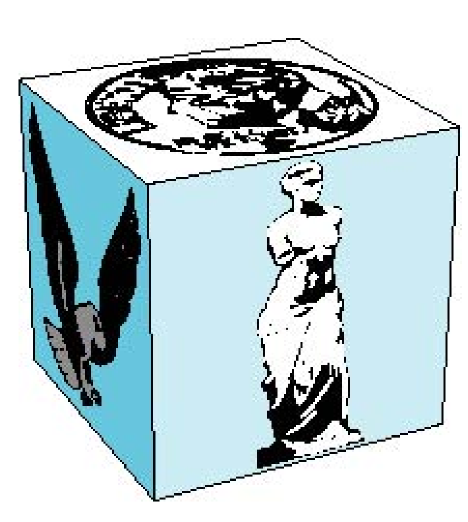
Introduction to Transformations



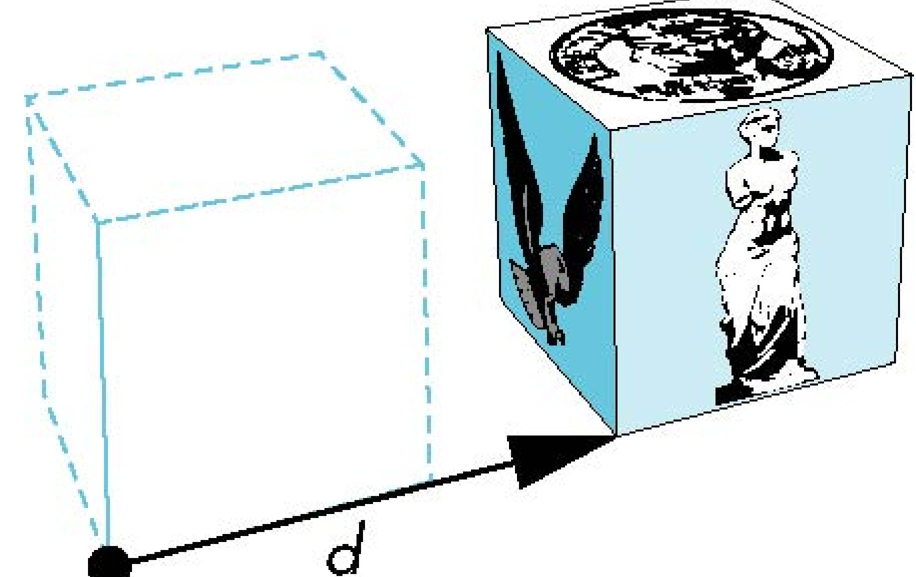
- May also want to transform objects by changing its:
  - Position (translation)
  - Size (scaling)
  - Orientation (rotation)
  - Shapes (shear)

Translation

- Move each vertex by **same** distance $\mathbf{d} = (\mathbf{d}_x, \mathbf{d}_y, \mathbf{d}_z)$

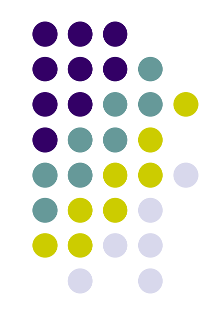


object



translation: every point displaced by same vector

Scaling



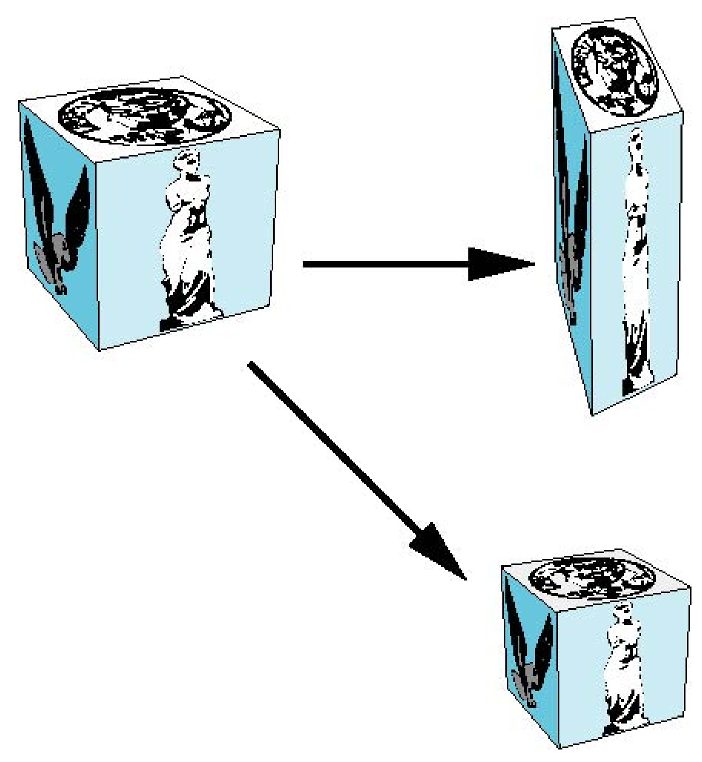
Expand or contract along each axis (fixed point of origin)

$$x' = s_x x$$
$$y' = s_y y$$
$$z' = s_z z$$

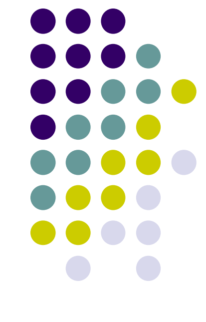
$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

where

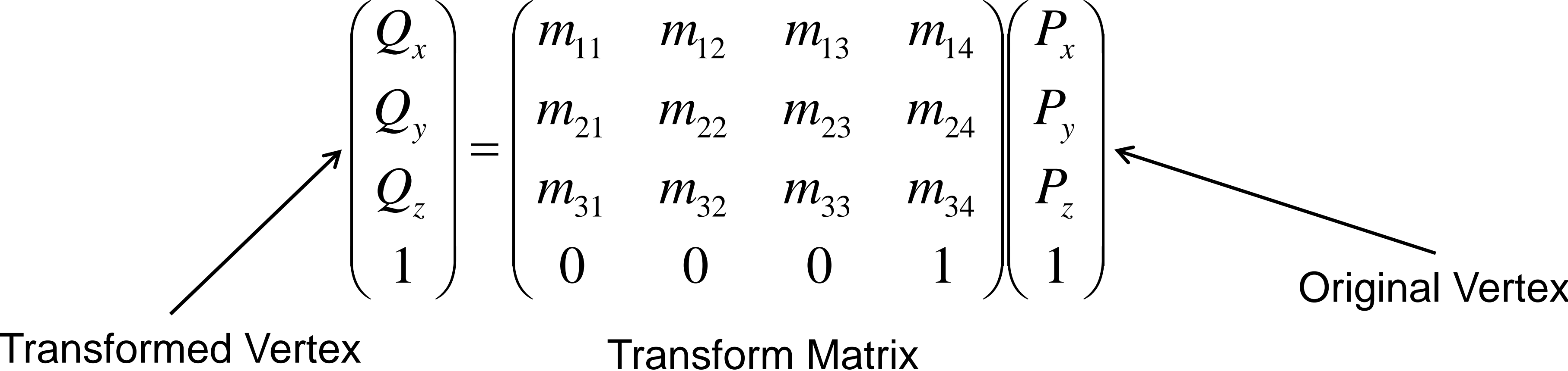
$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z)$$



Introduction to Transformations



- We can transform (translation, scaling, rotation, shearing, etc) object by applying matrix multiplications to object vertices



$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

- Note: point (x,y,z) needs to be represented as (x,y,z,1), also called **Homogeneous coordinates**

Why Matrices?

- Multiple transform matrices can be pre-multiplied
- For example:

  transform 1

  transform 2 ….

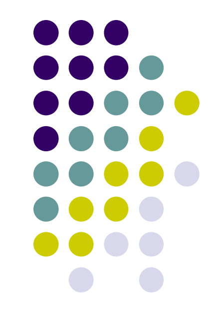
$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Transformed Point

Transform Matrices can
Be pre-multiplied

Original Point

Translation



- To reposition a point along a straight line
- Given point *(x,y)* and translation distance (t_x, t_y)
- The new point: *(x',y')*

$x' = x + t_x$

$y' = y + t_y$

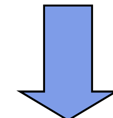
(x',y')

(x,y)

or

$$P' = P + T \quad \text{where} \quad P' = \begin{pmatrix} x' \\ y' \end{pmatrix} \quad P = \begin{pmatrix} x \\ y \end{pmatrix} \quad T = \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

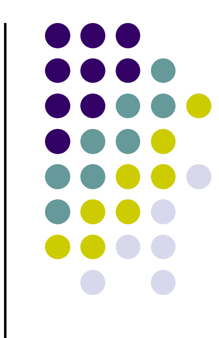


use 3x1 vector

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- Note: it becomes a matrix-vector multiplication

Translation of Objects



- How to translate an object with multiple vertices?

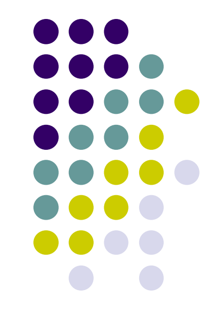
Translate individual vertices

$t_y = 3$

$t_x = 3$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 0.5 \\ 0.5 \\ 1 \end{pmatrix}$$

Repeat multiplication for all four vertices



- Now, 3D :

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

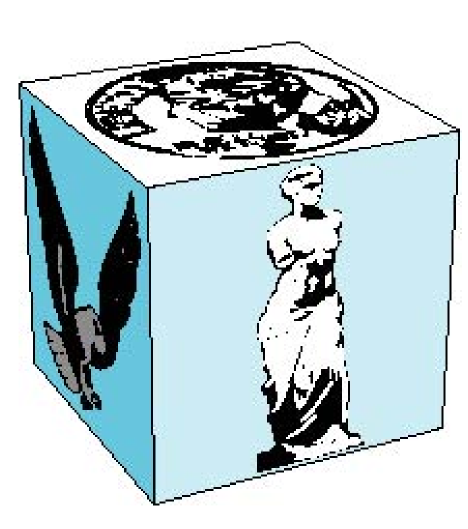
Translate(tx,ty,tz)

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

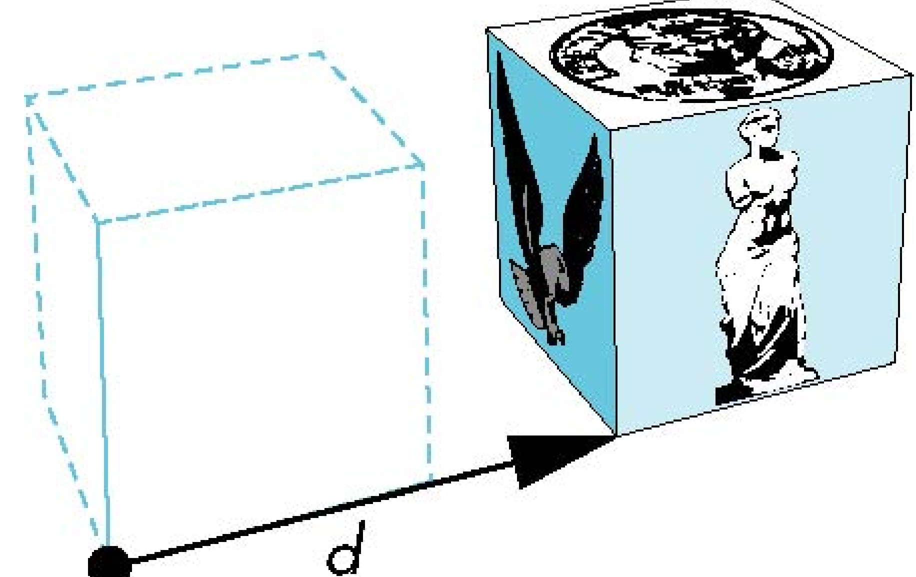
- Where: $x' = x.1 + y.0 + z.0 + tx.1 = x + tx$, … etc

3D Translation

- Move each vertex by same distance **d = (d_x, d_y, d_z)**



object



translation: every point displaced by same vector

- Scale: Alter object size by scaling factor (s_x, s_y). **about origin**

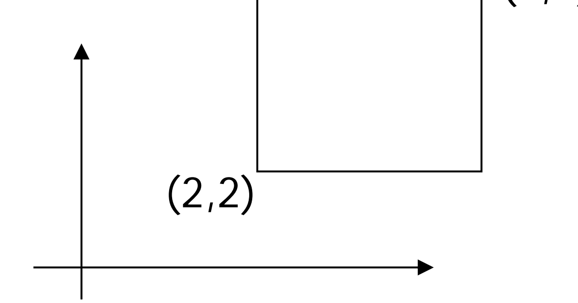
$$x' = x \cdot Sx$$
$$y' = y \cdot Sy$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} Sx & 0 \\ 0 & Sy \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

(2,2)

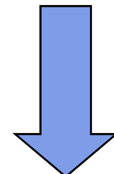
(1,1)

$Sx = 2$, $Sy = 2$



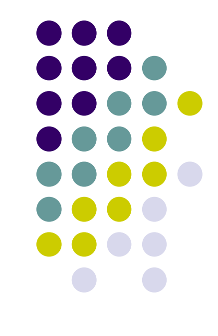
2D Scaling Matrix

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} Sx & 0 \\ 0 & Sy \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

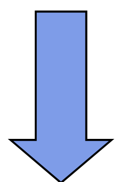


$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

4x4 3D Scaling Matrix



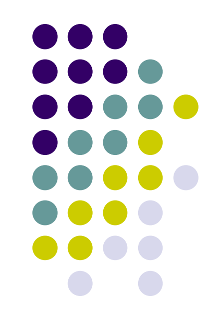
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$



$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- Example:
- If $Sx = Sy = Sz = 0.5$
- Can scale:
- big cube (sides = 1) to small cube (sides = 0.5)
- 2D: square, 3D cube

Scale(Sx,Sy,Sz)



Scaling

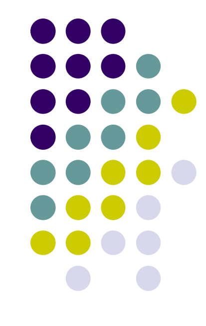
Expand or contract along each axis (fixed point of origin)

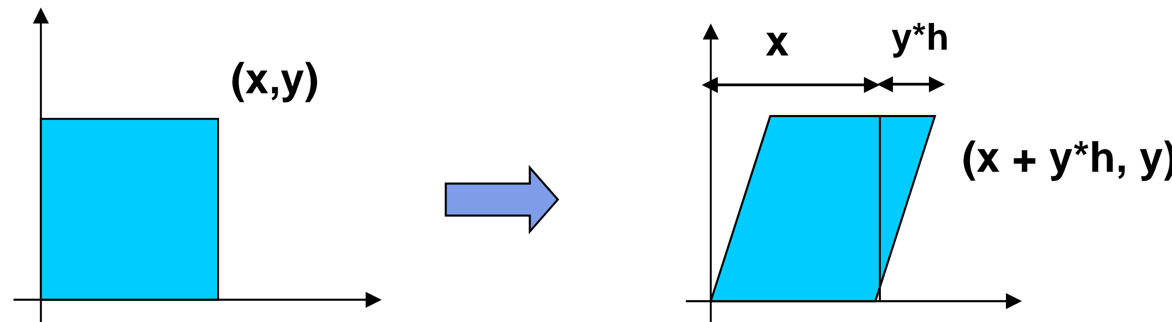
$$x' = s_x x$$
$$y' = s_y y$$
$$z' = s_z z$$

$$\mathbf{p}' = \mathbf{S}\mathbf{p}$$

$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shearing



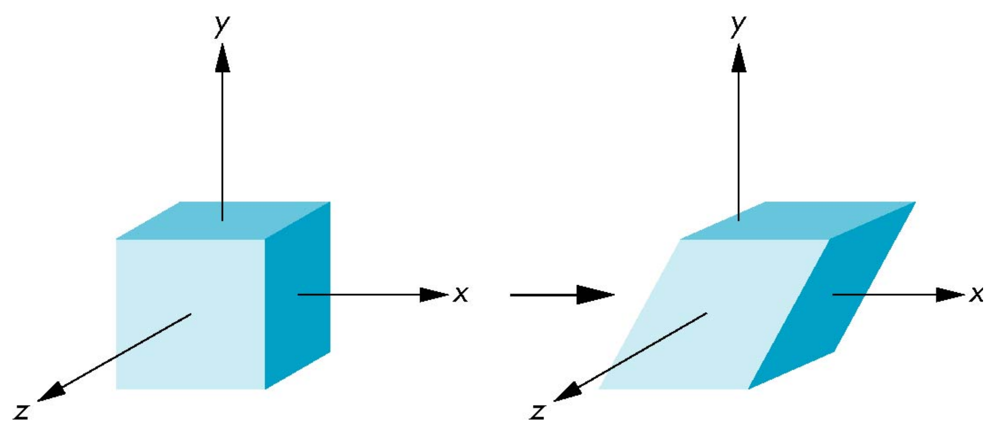


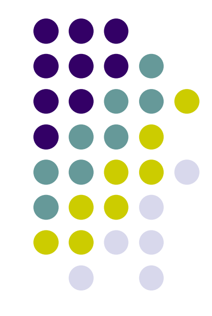
- Y coordinates are unaffected, but x cordinates are translated linearly with y
- That is:
  - **y' = y**
  - **x' = x + y * h**

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- h is fraction of y to be added to x

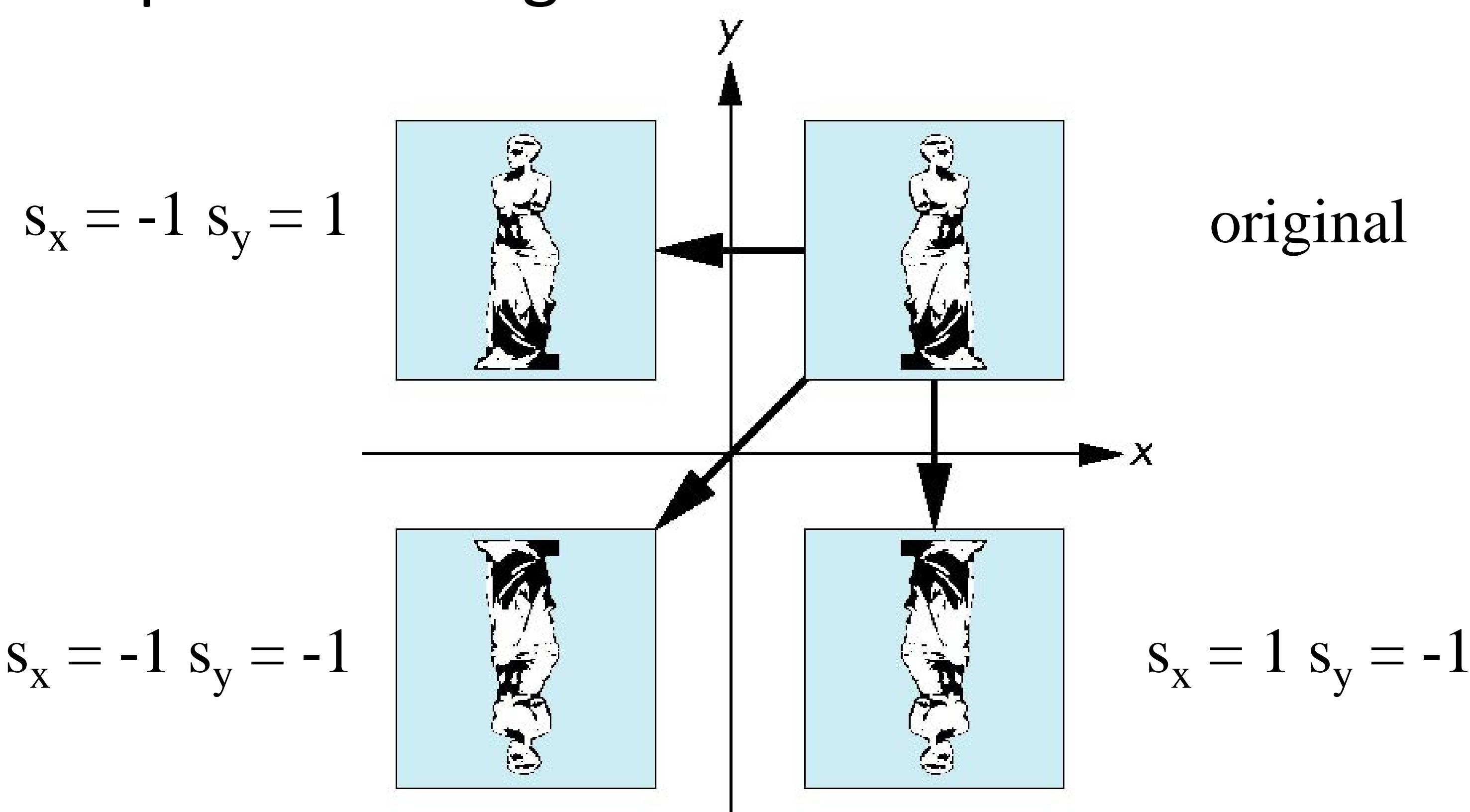
3D Shear



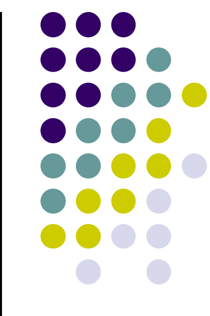


Reflection

corresponds to negative scale factors



References



- Angel and Shreiner, Interactive Computer Graphics, 6th edition, Chapter 3
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition