

Computer Graphics (CS 543)

Lecture 4c: Implementing Transformations

Prof Emmanuel Agu

*Computer Science Dept.
Worcester Polytechnic Institute (WPI)*





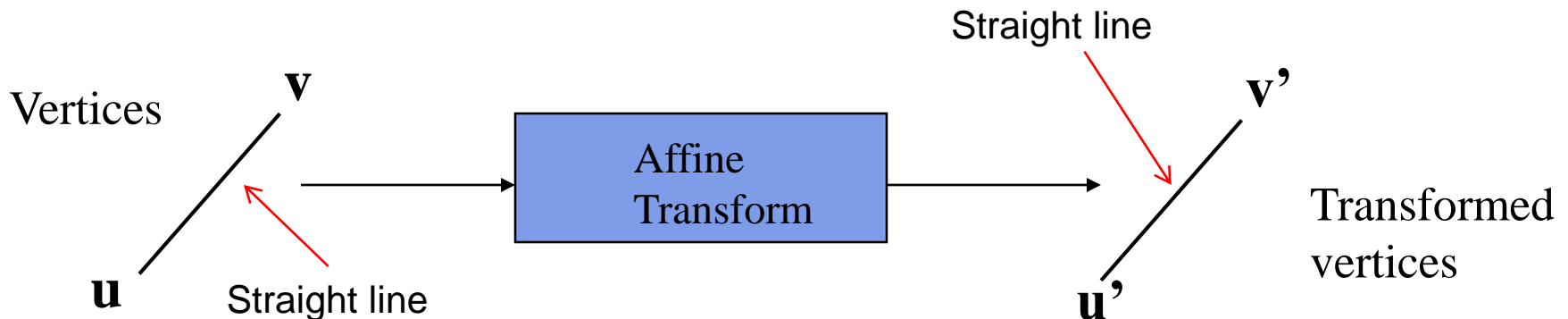
Objectives

- Learn how to implement transformations in OpenGL
 - Rotation
 - Translation
 - Scaling
- Introduce mat.h and vec.h header files for transformations
 - Model-view
 - Projection



Affine Transformations

- Translate, Scale, Rotate, Shearing, are affine transforms
- **Rigid body transformations:** rotation, translation, scaling, shear
- **Line preserving:** important in graphics since we can
 1. Transform endpoints of line segments
 2. Draw line segment between the transformed endpoints



Previously: Transformations in OpenGL



- Pre 3.0 OpenGL had a set of transformation functions
 - glTranslate
 - glRotate()
 - glScale()
- Previously, OpenGL would
 - Receive transform commands (glTranslate, glRotate, glScale)
 - Multiply transform matrices together and maintain transform matrix stack known as **modelview matrix**



Previously: Modelview Matrix Formed?

```
glMatrixMode (GL_MODELVIEW)
glLoadIdentity ();
glScale (1, 2, 3);
glTranslate (3, 6, 4);
```

Specify transforms
In OpenGL Program (.cpp file)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 12 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**Identity
Matrix**

**glScale
Matrix**

**glTranslate
Matrix**

**Modelview
Matrix**

OpenGL implementations
(glScale, glTranslate, etc)
in Hardware (Graphics card)

OpenGL multiplies transforms together
To form modelview matrix
Applies final matrix to vertices of objects



Previously: OpenGL Matrices

- OpenGL maintained 4 matrix stacks maintained as part of OpenGL state
 - Model-View (**GL_MODELVIEW**)
 - Projection (**GL_PROJECTION**)
 - Texture (**GL_TEXTURE**)
 - Color(**GL_COLOR**)



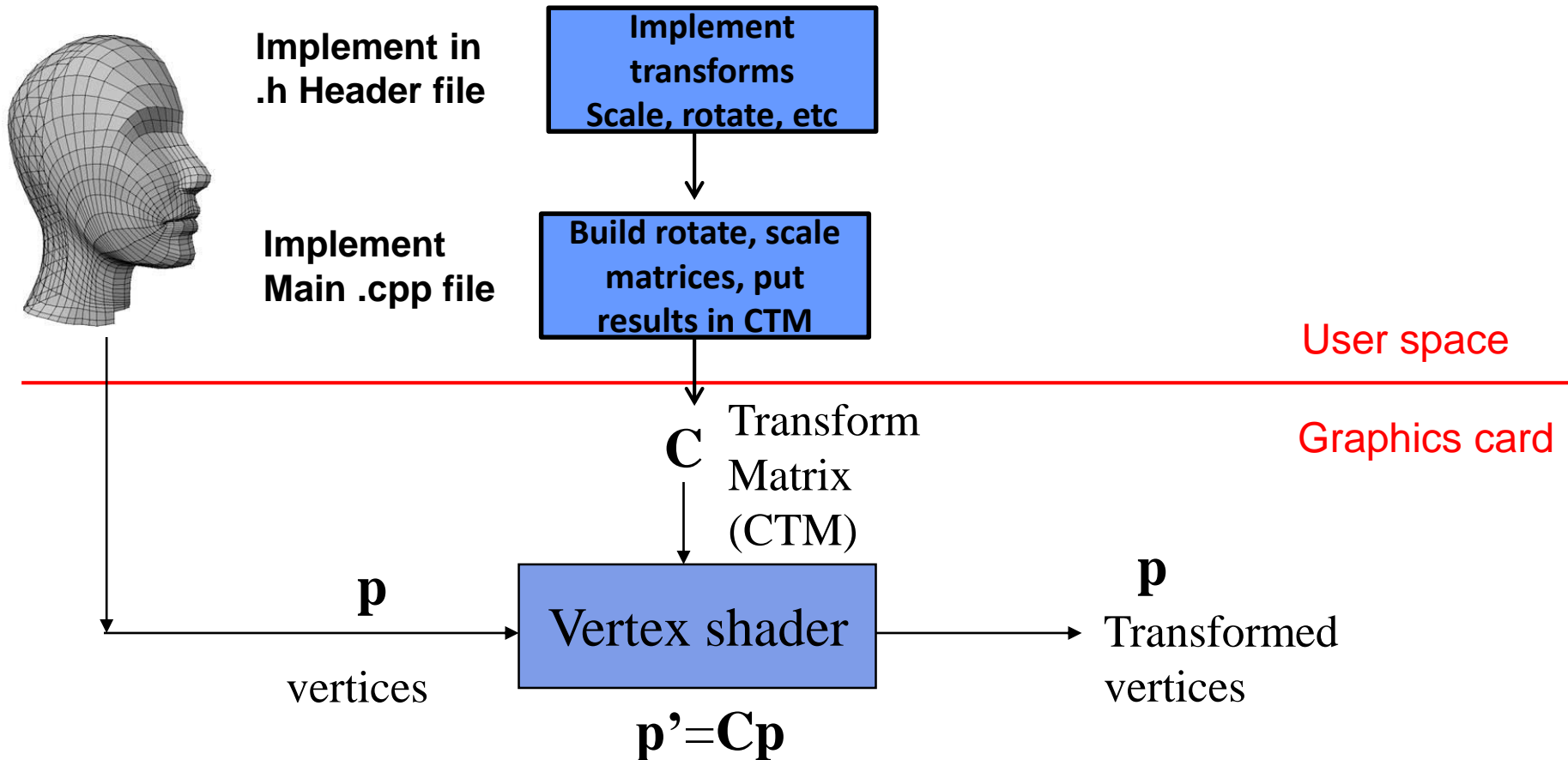
Now: Transformations in OpenGL

- **From OpenGL 3.0:** No transform commands (scale, rotate, etc), matrices maintained by OpenGL!!
- glTranslate, glScale, glRotate, OpenGL modelview matrix all deprecated!!
- If programmer needs transforms, matrices implement it!
- **Optional:** Programmer ***may*** now choose to maintain transform matrices **or NOT!**



Current Transformation Matrix (CTM)

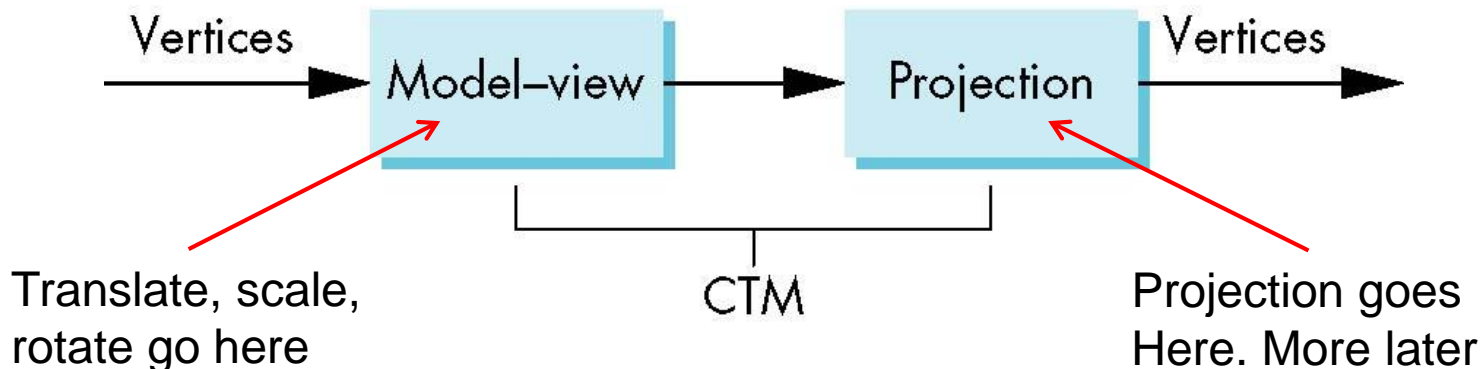
- Conceptually user can implement a 4 x 4 homogeneous coordinate matrix, the *Current Transformation Matrix (CTM)*
- The **CTM** defined and updated in user program





Homegrown CTM Matrices

- **CTM = modelview + projection**
 - Model-View (**GL_MODELVIEW**)
 - Projection (**GL_PROJECTION**) } **CTM**
 - Texture (**GL_TEXTURE**)
 - Color(**GL_COLOR**)





CTM Functionality

```
LoadIdentity();  
Scale(1,2,3);  
Translate(3,6,4);
```

1. We need to implement our own transforms (in header file)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 12 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Identity Matrix **Scale Matrix** **Translate Matrix** **CTM Matrix**

2. Multiply our transforms together to form **CTM matrix**
3. Apply final matrix to vertices of objects



Implementing Transforms and CTM

- Where to implement transforms and CTM?
- We implement CTM in 3 parts
 1. mat.h (Header file)
 - Implementations of `translate()`, `scale()`, etc
 2. Application code (.cpp file)
 - Multiply together `translate()`, `scale()` = final CTM matrix
 3. GLSL functions (vertex and fragment shader)
 - Apply final CTM matrix to vertices

Implementing Transforms and CTM



- We just have to include `mat.h` (`#include "mat.h"`), use it
- **Uniformity:** `mat.h` syntax resembles GLSL language in shaders
- **Matrix Types:** `mat4` (4x4 matrix), `mat3` (3x3 matrix).

```
class mat4 {  
    vec4  _m[4];  
    .....  
}
```

- Can declare CTM as `mat4` type

```
mat4 ctm = Translate(3,6,4);
```

CTM ←
$$\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
 Translation Matrix

- **mat.h also has transform functions:** Translate, Scale, Rotate, etc.

```
mat4 Translate(const GLfloat x, const GLfloat y, const GLfloat z )  
mat4 Scale( const GLfloat x, const GLfloat y, const GLfloat z )
```



CTM operations

- The CTM can be altered either by loading a new CTM or by postmultiplication

Load identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$

Load arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{M}$

Load a translation matrix: $\mathbf{C} \leftarrow \mathbf{T}$

Load a rotation matrix: $\mathbf{C} \leftarrow \mathbf{R}$

Load a scaling matrix: $\mathbf{C} \leftarrow \mathbf{S}$

Postmultiply by an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{M}$

Postmultiply by a translation matrix: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

Postmultiply by a rotation matrix: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$

Postmultiply by a scaling matrix: $\mathbf{C} \leftarrow \mathbf{C}\mathbf{S}$



Example: Creating Identity Matrix

- All transforms (translate, scale, rotate) converted to 4x4 matrix
- We put 4x4 transform matrix into **CTM**
- Example: Create an identity matrix

```
mat4 m = Identity();
```



mat4 type stores 4x4 matrix
Defined in mat.h



CTM Matrix

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



Transformation matrices Formed?

```
mat4 m = Identity();  
mat4 t = Translate(3,6,4);  
m = m*t;
```

Identity Matrix	Translation Matrix	CTM Matrix
$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	\times $\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$=$ $\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix}$



Transformation matrices Formed?

- Consider following code snippet

```
mat4 m = Identity();  
mat4 s = Scale(1,2,3);  
m = m*s;
```

$$\begin{array}{ccc} \text{Identity} & \text{Scaling} & \\ \text{Matrix} & \text{Matrix} & \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \times & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & = & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ & & \text{CTM Matrix} & & \end{array}$$



Transformation matrices Formed?

- What of translate, then scale, then
- Just multiply them together. Evaluated in *reverse order*!! E.g:

```
mat4 m = Identity();  
mat4 s = Scale(1,2,3);  
mat4 t = Translate(3,6,4);  
m = m*s*t;
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 12 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**Identity
Matrix**

**Scale
Matrix**

**Translate
Matrix**

Final CTM Matrix

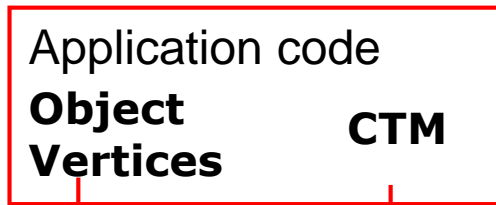
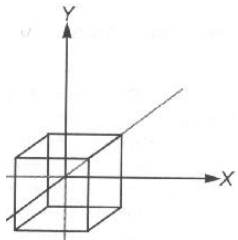


How are Transform matrices Applied?

```
mat4 m = Identity();  
mat4 s = Scale(1,2,3);  
mat4 t = Translate(3,6,4);  
m = m*s*t;  
colorcube( );
```

1. In application:

Load object vertices into points[] array -> VBO
Call glDrawArrays



CTM Matrix

$$\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 12 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2. CTM built in application, passed to vertex shader

$$\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 12 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 14 \\ 15 \\ 1 \end{pmatrix} \quad \text{Transformed vertex}$$

3. In vertex shader: Each vertex of object (cube) is multiplied by CTM to get transformed vertex position

```
gl_Position = model_view*vPosition;
```



Passing CTM to Vertex Shader

- Build CTM (modelview) matrix in application program
- Pass matrix to shader

```
void display( ){
```

```
.....  
mat4 m = Identity();  
mat4 s = Scale(1,2,3);  
mat4 t = Translate(3,6,4);  
m = m*s*t;
```

Build CTM
in application

CTM matrix **m** in application
is same as **model_view** in shader

```
// find location of matrix variable "model_view" in shader  
// then pass matrix to shader
```

```
matrix_loc = glGetUniformLocation(program, "model_view");  
glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, m);
```

```
.....
```

```
}
```



Implementation: Vertex Shader

- On `glDrawArrays()`, vertex shader invoked with different `vPosition` per shader
- E.g. If `colorcube()` generates 8 vertices, each vertex shader receives a vertex stored in `vPosition`
- Shader calculates modified vertex position, stored in `gl_Position`

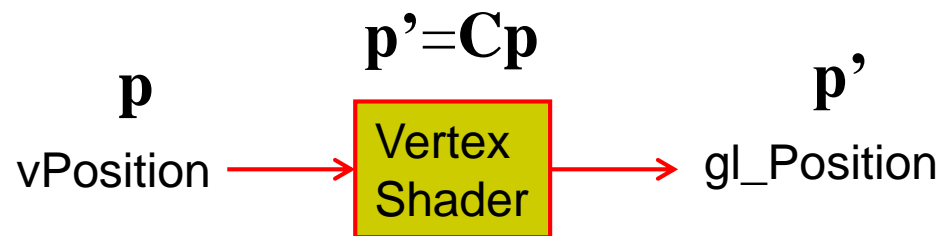
```
in vec4 vPosition;  
uniform mat4 model_view;
```

```
void main( )  
{  
    gl_Position = model_view*vPosition;  
}
```

Transformed
vertex **position**

Contains **CTM**

Original vertex
position





What Really Happens to Vertex Position Attributes?

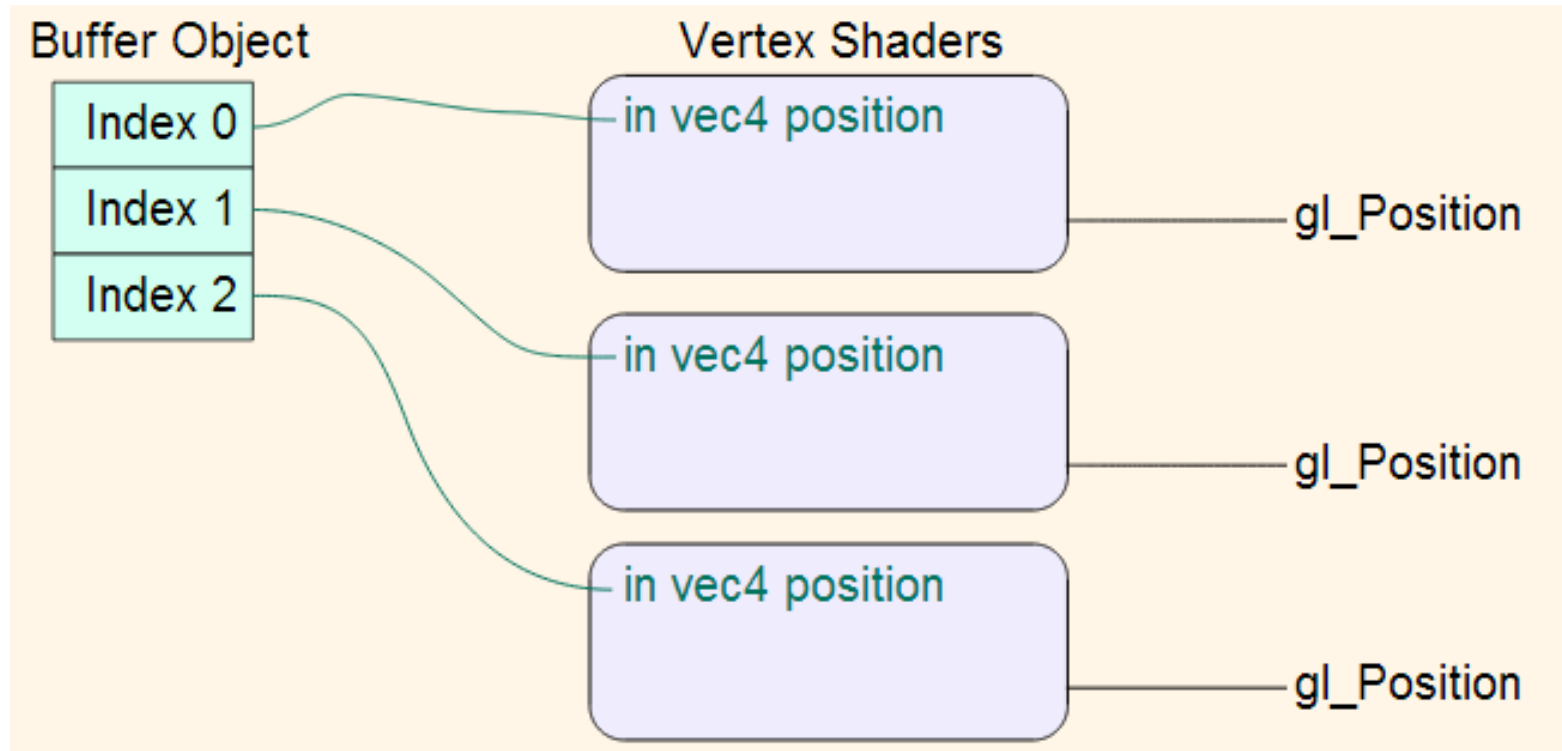


Image credit: Arcsynthesis tutorials



What About Multiple Vertex Attributes?

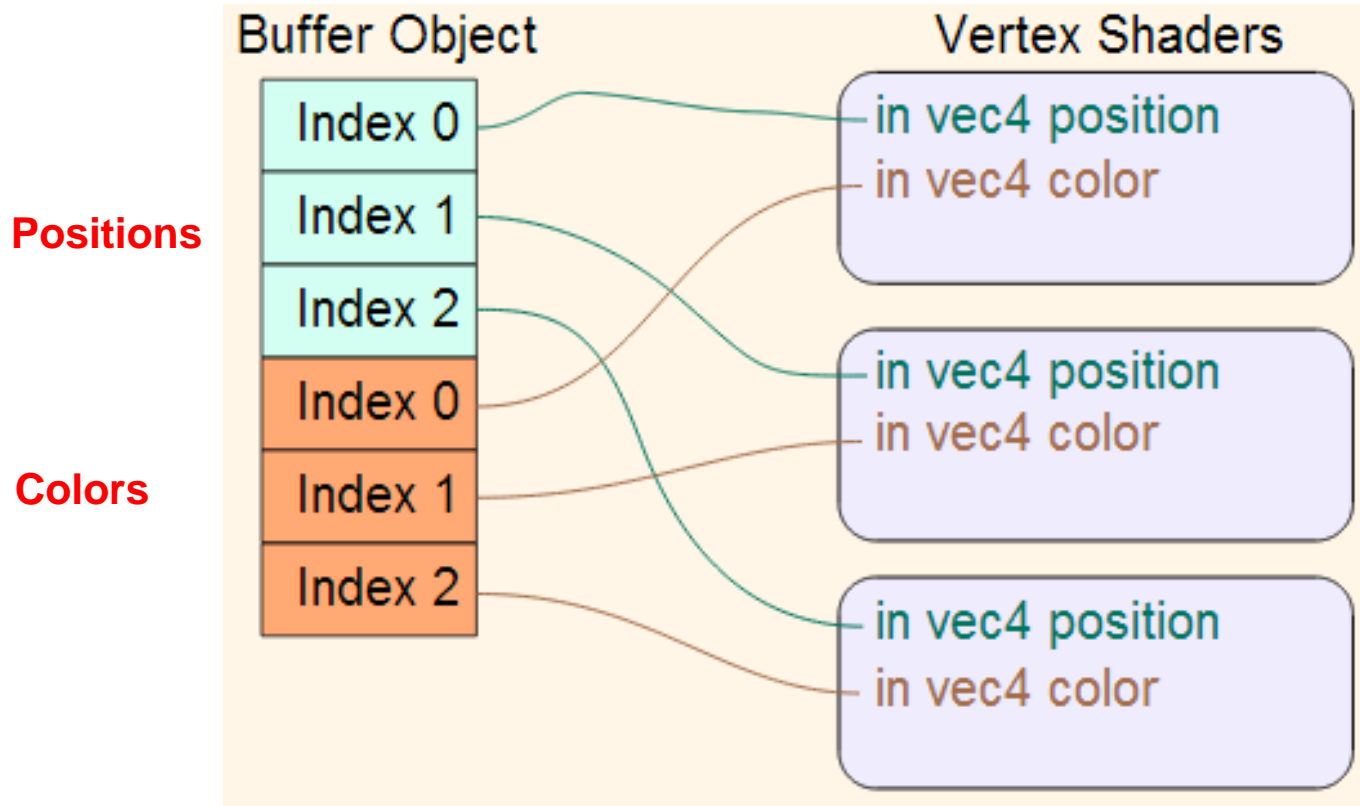


Image credit: Arcsynthesis tutorials



Transformation matrices Formed?

- Example: Vertex (1, 1, 1) is one of 8 vertices of cube

In application

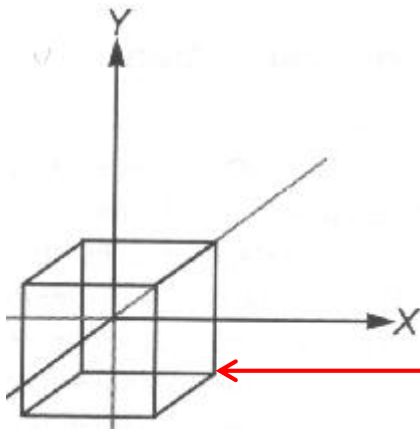
```
mat4 m = Identity();  
mat4 s = Scale(1,2,3);  
m = m*s;  
colorcube( );
```

In vertex shader

$$\begin{matrix} \text{CTM (m)} & & \mathbf{p} & = & \mathbf{p}' \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & * & \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} & = & \begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \end{pmatrix} \end{matrix}$$

Original
vertex

Transformed
vertex



Each vertex of cube is multiplied by modelview matrix to get scaled vertex position

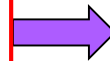


Transformation matrices Formed?

- **Another example:** Vertex (1, 1, 1) is one of 8 vertices of cube

In application

```
mat4 m = Identity();  
mat4 s = Scale(1,2,3);  
mat4 t = Translate(3,6,4);  
m = m*s*t;  
colorcube( );
```



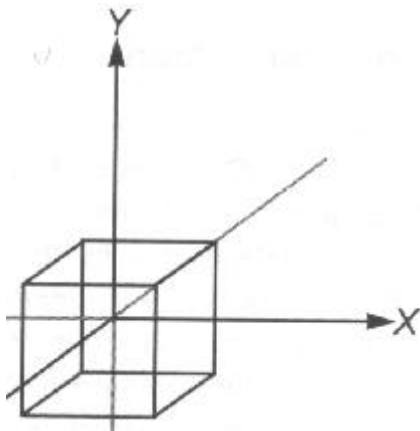
In vertex shader

$$\begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 0 & 12 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 14 \\ 15 \\ 1 \end{pmatrix}$$

CTM Matrix

**Original
vertex**

**Transformed
vertex**



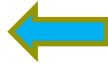
Each vertex of cube is multiplied by modelview matrix to get scaled vertex position



Arbitrary Matrices

- Can multiply by matrices from transformation commands (Translate, Rotate, Scale) into CTM
- Can also load arbitrary 4x4 matrices into CTM

Load into
CTM Matrix



$$\begin{pmatrix} 1 & 0 & 15 & 3 \\ 0 & 2 & 0 & 12 \\ 34 & 0 & 3 & 12 \\ 0 & 24 & 0 & 1 \end{pmatrix}$$



Example: Rotation about a Fixed Point

- We want $\mathbf{C} = \mathbf{T} \mathbf{R} \mathbf{T}^{-1}$
- Be careful with order. Do operations in following order

$$\mathbf{C} \leftarrow \mathbf{I}$$

$$\mathbf{C} \leftarrow \mathbf{C} \mathbf{T}$$

$$\mathbf{C} \leftarrow \mathbf{C} \mathbf{R}$$

$$\mathbf{C} \leftarrow \mathbf{C} \mathbf{T}^{-1}$$

- Each operation corresponds to one function call in the program.
- **Note:** last operation specified is first executed



Matrix Stacks

- CTM is actually not just 1 matrix but a matrix **STACK**
 - Multiple matrices in stack, “current” matrix at top
 - Can save transformation matrices for use later (push, pop)
- E.g: Traversing hierarchical data structures (Ch. 8)
- Pre 3.1 OpenGL also maintained matrix stacks
- Right now just implement 1-level CTM
- Matrix stack later for hierarchical transforms



Reading Back State

- Can also access OpenGL variables (and other parts of the state) by *query* functions

```
glGetIntegerv  
glGetFloatv  
glGetBooleanv  
glGetDoublev  
glIsEnabled
```

- Example: to find out max. number texture units on GPU

```
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &MaxTextureUnits);
```



Using Transformations

- **Example:** use idle function to rotate a cube and mouse function to change direction of rotation
- Start with program that draws cube as before
 - Centered at origin
 - Sides aligned with axes



Recall: main.c

```
void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
        GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube); ← Calls spinCube continuously
    glutMouseFunc(mouse);      Whenever OpenGL program is idle
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```



Recall: Idle and Mouse callbacks

```
void spinCube()
{
    theta[axis] += 2.0;
    if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
    glutPostRedisplay();
}

void mouse(int button, int state, int x, int y)
{
    if(button==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        axis = 0;
    if(button==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
        axis = 1;
    if(button==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        axis = 2;
}
```

Display callback



```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ctm = RotateX(theta[0])*RotateY(theta[1])
        *RotateZ(theta[2]);
    glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, ctm);
    glDrawArrays(GL_TRIANGLES, 0, N);
    glutSwapBuffers();
}
```

Update CTM Matrix

*RotateZ(theta[2]);

glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, ctm);

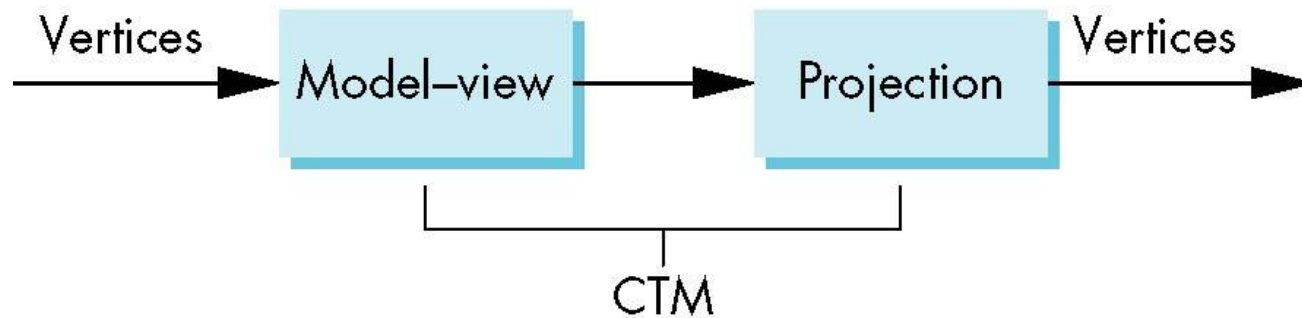
glDrawArrays(GL_TRIANGLES, 0, N);

glutSwapBuffers();

Pass CTM to vertex shader

- Alternatively, we can
 - send rotation angle + axis to vertex shader,
 - Let shader form CTM then do rotation
- Inefficient: if mesh has 10,000 vertices each one forms CTM, redundant!!!!

Using the Model-view Matrix



- In OpenGL the model-view matrix used to
 - Transform 3D models (translate, scale, rotate)
 - Position camera (using LookAt function) (next)
- The projection matrix used to define view volume and select a camera lens (later)
- Although these matrices no longer part of OpenGL, good to create them in our applications (as CTM)



References

- Angel and Shreiner, Interactive Computer Graphics (6th edition), Chapter 3