Computer Graphics (CS 543)
Lecture 5: 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

![Diagram showing affine transformation of straight lines and vertices](image)
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?

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

Specify transforms in OpenGL Program (.cpp file)

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

OpenGL multiplies transforms together to form modelview matrix:

- Identity Matrix
- `glScale` Matrix
- `glTranslate` Matrix
- 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, the user can implement a 4 x 4 homogeneous coordinate matrix, the *Current Transformation Matrix (CTM)*.
- The CTM is defined and updated in the user program.

\[ \mathbf{p}' = \mathbf{Cp} \]

Where \( \mathbf{p} \) represents the vertices, \( \mathbf{C} \) is the transformation matrix, and \( \mathbf{p}' \) is the transformed vertex.

Implementation steps:
- Implement in the .h Header file.
- Implement in the Main .cpp file.
- Implement transforms such as scale, rotate, etc.
- Build rotate, scale matrices, put results in the CTM.

Diagram:
- From user space to vertex shader.
- Transformation of vertices through the CTM.
- Transformed vertices sent to graphics card.

User space

Graphics card
Homegrown CTM Matrices

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

CTM = Translation, scaling, rotation → Projection goes here. More later
CTM Functionality

1. We need to implement our own transforms i.e. math functions to transform points

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

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);
  ```
- `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: $C \leftarrow I$
  - Load arbitrary matrix: $C \leftarrow M$

  - Load a translation matrix: $C \leftarrow T$
  - Load a rotation matrix: $C \leftarrow R$
  - Load a scaling matrix: $C \leftarrow S$

  - Postmultiply by an arbitrary matrix: $C \leftarrow CM$
  - Postmultiply by a translation matrix: $C \leftarrow CT$
  - Postmultiply by a rotation matrix: $C \leftarrow CR$
  - Postmultiply by a scaling matrix: $C \leftarrow CS$
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

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

**CTM Matrix**

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```
Transformation matrices Formed?

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

<table>
<thead>
<tr>
<th>Identity Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 0 0 0)</td>
</tr>
<tr>
<td>(0 1 0 0)</td>
</tr>
<tr>
<td>(0 0 1 0)</td>
</tr>
<tr>
<td>(0 0 0 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Translation Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 0 0 3)</td>
</tr>
<tr>
<td>(0 1 0 6)</td>
</tr>
<tr>
<td>(0 0 1 4)</td>
</tr>
<tr>
<td>(0 0 0 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CTM Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 0 0 3)</td>
</tr>
<tr>
<td>(0 1 0 6)</td>
</tr>
<tr>
<td>(0 0 1 4)</td>
</tr>
<tr>
<td>(0 0 0 1)</td>
</tr>
</tbody>
</table>
Transformation matrices Formed?

- Consider following code snippet

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

<table>
<thead>
<tr>
<th>Identity Matrix</th>
<th>Scaling Matrix</th>
<th>CTM Matrix</th>
</tr>
</thead>
</table>
| \[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\] | \[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\] | \[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\] |
Transformation matrices Formed?

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

```plaintext
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;
colorcubed();

1. In application:
Load object vertices into points[] array -> VBO
Call glDrawArrays

2. CTM built in application, passed to vertex shader

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

CTM Matrix
\[
\begin{pmatrix}
1 & 0 & 0 & 3 \\
0 & 2 & 0 & 12 \\
0 & 0 & 3 & 12 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 \\
0 \\
2 \\
0
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 1 \\
4 & 14 \\
15 & 1
\end{pmatrix}
\]

gl_Position = model_view*vPosition;
Passing CTM to Vertex Shader

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

```c
void display()
{
    // Build CTM in application
    mat4 m = Identity();
    mat4 s = Scale(1,2,3);
    mat4 t = Translate(3,6,4);
    m = m*s*t;

    // 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;
}
```

Original vertex position \( p \)  \( \rightarrow \)  Modified vertex position \( p' \) = \( Cp \)  \( \rightarrow \)  `gl_Position`

Transformed vertex position  \( \rightarrow \)  `CTM`
What Really Happens to Vertex Position Attributes?

Buffer Object:
- Index 0
- Index 1
- Index 2

Vertex Shaders:
- in vec4 position
  - gl_Position
- in vec4 position
  - gl_Position
- in vec4 position
  - gl_Position

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**

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

**In vertex shader**

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
1 \\
1 \\
1
\end{bmatrix}
= \begin{bmatrix}
1 \\
2 \\
3 \\
1
\end{bmatrix}
\]

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

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

```
ctm
```

CTM Matrix

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

In application

In vertex shader
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 \( C = T R T^{-1} \)
- Be careful with order. Do operations in following order:

\[
\begin{align*}
C & \leftarrow I \\
C & \leftarrow CT \\
C & \leftarrow CR \\
C & \leftarrow CT^{-1}
\end{align*}
\]

- 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
  
  ```c
  glGetIntegerv
  glGetFloatv
  glGetBooleanv
  glGetDoublev
  glEnable
  ```

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

  ```c
  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

```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);
    glutMouseFunc(mouse);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

Calls spinCube continuously whenever OpenGL program is idle
Recall: Idle and Mouse callbacks

```c
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;
}
```
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();
}

• 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