Recall: Introduction to Transformations

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

- Move each vertex by **same** distance $\mathbf{d} = (d_x, d_y, d_z)$

object

translation: every point displaced by same vector
Recall: Scaling

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

\[ x' = s_x x \]
\[ y' = s_y y \]
\[ z' = s_z z \]
\[ p' = Sp \]

where

\[ S = S(s_x, s_y, s_z) \]
Introduction to Transformations

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

\[
\begin{pmatrix}
P_x' \\
P_y' \\
P_z' \\
1
\end{pmatrix} =
\begin{pmatrix}
m_{11} & m_{12} & m_{13} & m_{14} \\
m_{21} & m_{22} & m_{23} & m_{24} \\
m_{31} & m_{32} & m_{33} & m_{34} \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
P_x \\
P_y \\
P_z \\
1
\end{pmatrix}
\]

- 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
- One final resulting matrix applied (efficient!)
- For example:
  
  transform 1
  
  transform 2 ....

$$\begin{pmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{pmatrix} = \begin{pmatrix}
m_{11} & m_{12} & m_{13} & m_{14} \\
m_{21} & m_{22} & m_{23} & m_{24} \\
m_{31} & m_{32} & m_{33} & m_{34} \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
m_{11} & m_{12} & m_{13} & m_{14} \\
m_{21} & m_{22} & m_{23} & m_{24} \\
m_{31} & m_{32} & m_{33} & m_{34} \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
P_x \\
P_y \\
P_z \\
1
\end{pmatrix}$$
3D Translation Example

- **Example**: If we translate a point \((2,2,2)\) by displacement \((2,4,6)\), new location of point is \((4,6,8)\)

\[
\begin{pmatrix}
4 \\
6 \\
8 \\
1
\end{pmatrix}
= \begin{pmatrix}
1 & 0 & 0 & 2 \\
0 & 1 & 0 & 4 \\
0 & 0 & 1 & 6 \\
0 & 0 & 0 & 1
\end{pmatrix}
\times
\begin{pmatrix}
2 \\
2 \\
2 \\
1
\end{pmatrix}
\]

Translated point
Translation Matrix
Original point
3D Translation

- Translate object = Move each vertex by same distance $d = (d_x, d_y, d_z)$

Translate($dx, dy, dz$)

Where:
- $x' = x + dx$
- $y' = y + dy$
- $z' = z + dz$

Translation Matrix:

$$\begin{pmatrix}
x' \\
y' \\
z' \\
1
\end{pmatrix} = \begin{pmatrix}
1 & 0 & 0 & d_x \\
0 & 1 & 0 & d_y \\
0 & 0 & 1 & d_z \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}$$
Scaling

Scale object = Move each object vertex by scale factor $S = (S_x, S_y, S_z)$
Expand or contract along each axis (relative to origin)

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

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

Scale Matrix

$\text{Scale}(S_x, S_y, S_z)$
Scaling Example

If we scale a point (2,4,6) by scaling factor (0.5,0.5,0.5)
Scaled point position = (1, 2, 3)

- Scaled x: 2 x 0.5 = 1
- Scaled y: 4 x 0.5 = 2
- Scaled z: 6 x 0.5 = 3

\[
\begin{pmatrix}
1 \\
2 \\
3 \\
1 \\
\end{pmatrix}
= \begin{pmatrix}
0.5 & 0 & 0 & 0 \\
0 & 0.5 & 0 & 0 \\
0 & 0 & 0.5 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\times \begin{pmatrix}
2 \\
4 \\
6 \\
1 \\
\end{pmatrix}
\]

Scale Matrix for Scale(0.5, 0.5, 0.5)
Shearing

- Y coordinates are unaffected, but x coordinates are translated linearly with y.
- That is:
  - $y' = y$
  - $x' = x + y \times h$

$h$ is fraction of $y$ to be added to $x$.
3D Shear
Reflection

- corresponds to negative scale factors

\[ s_x = -1 \quad s_y = 1 \]

\[ s_x = -1 \quad s_y = -1 \]

\[ s_x = 1 \quad s_y = -1 \]
Computer Graphics (CS 4731)  
Lecture 9: 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 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 (Translate, Rotate, Scale)
  - 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);
glTranslatef(3,6,4);
```

Specify transforms in OpenGL Program

<table>
<thead>
<tr>
<th>Identity Matrix</th>
<th>glScale Matrix</th>
<th>glTranslate Matrix</th>
<th>Modelview Matrix</th>
</tr>
</thead>
</table>
| \[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\] | \[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\] | \[
\begin{pmatrix}
1 & 0 & 0 & 3 \\
0 & 1 & 0 & 6 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\] | \[
\begin{pmatrix}
1 & 0 & 0 & 3 \\
0 & 2 & 0 & 12 \\
0 & 0 & 3 & 12 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\] |

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 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.

\[ p' = Cp \]

User space

Graphics card
CTM in OpenGL Matrices

CTM = modelview + projection

- Model-View (GL_MODELVIEW)
- Projection (GL_PROJECTION)
- Texture (GL_TEXTURE)
- Color (GL_COLOR)

Vertices -> Model-view -> Projection -> Vertices

Translate, scale, rotate go here

CTM

Projection goes Here. More later
CTM Functionality

1. We need to implement our own transforms
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).
  ```cpp
class mat4 {
    vec4 _m[4];
    ........
};
```
- Can declare CTM as mat4 type
  ```cpp
  mat4 ctm = Translate(3,6,4);
  ```
- **mat.h also has transform functions:** Translate, Scale, Rotate, etc.

```cpp
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: Rotation, Translation, Scaling

Create an identity matrix:

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

Form Translate and Scale matrices, multiply together

```cpp
mat4 s = Scale(sx, sy, sz);
mat4 t = Transalate(dx, dy, dz);
m = m*s*t;
```
Example: Rotation about a Fixed Point

- We want $C = TRT^{-1}$
- Be careful with order. Do operations in following order

$C \leftarrow I$
$C \leftarrow CT$
$C \leftarrow CR$
$C \leftarrow CT^{-1}$

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

- Converts all transforms (translate, scale, rotate) to 4x4 matrix
- We put 4x4 transform matrix into CTM
- Example

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

CTM Matrix

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

mat4 type stores 4x4 matrix
Defined in mat.h
Transformation matrices Formed?

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

Identity Matrix
\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

Translation Matrix
\[
\begin{pmatrix}
1 & 0 & 0 & 3 \\
0 & 1 & 0 & 6 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]

CTM Matrix
\[
\begin{pmatrix}
1 & 0 & 0 & 3 \\
0 & 1 & 0 & 6 \\
0 & 0 & 1 & 4 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}
\]
Transformation matrices Formed?

- Consider following code snippet

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

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

Transformation matrices Formed?

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

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

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

- 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

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

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

gl_Position = model_view*vPosition;
Passing CTM to Vertex Shader

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

```c
void display( ){

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

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

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

    ......  
}
```

Build CTM in application

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

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

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

\[ p' = Cp \]

- 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

```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.

In application:

```cpp
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.
References

- Angel and Shreiner, Chapter 3
- Hill and Kelley, appendix 4