Vertex Buffer Objects and Transformations
Week 4

David Breen
Department of Computer Science
Drexel University

Based on material from Ed Angel, University of New Mexico
Manolya Eyiyurekli, Drexel University
Tomas Lozano-Perez & Jovan Popovic, MIT
Jeff Trinkle, RPI
Vertex Buffer Objects (VBOs)

- enhance the performance of OpenGL by providing the benefits of vertex arrays and display lists, while avoiding downsides of their implementations
- allow vertex array data to be stored in high-performance graphics memory on the server side and promotes efficient data transfer
- GL_ARB_vertex_buffer_object extension should be supported by your graphics card.
Why not use vertex arrays?

- Advantage - Using a vertex array can reduce the number of function calls and redundant usage of the shared vertices.
- Disadvantage – Vertex array functions are in the client state and the data in the arrays must be resent to the server each time it is referenced.
Why not use display lists?

- **Advantage** - Display list is a server side function, so it does not suffer from overhead of data transfer
- **Disadvantage** – Once a display list is compiled, the data in the display list cannot be modified
Why use VBOs?

Only Advantages

• Creates "buffer objects" for vertex attributes in high-performance memory on the server side

• Provides same access functions to reference the arrays, which are used in vertex arrays

• Data in vertex buffer object can be read and updated by mapping the buffer into client's memory space
Why use VBOs?

Only Advantages

• The memory manager in vertex buffer object will put the buffer objects into the best place of memory based on user's hints.

• Memory manager can optimize the buffers by balancing between 3 kinds of memory:
  – system, AGP and video memory

• Shares the buffer objects with many clients, like display lists and textures. Since VBO is on the server's side, multiple clients will be able to access the same buffer with the corresponding identifier
Next

• How to
  – Create a VBO
  – Draw a VBO
  – Update a VBO
Creating VBOs

• Generate a new buffer object with glGenBuffersARB().
• Bind the buffer object with glBindBufferARB().
• Copy vertex data to the buffer object with glBufferObjectARB().
• `glGenBuffersARB()`
  – creates buffer objects and returns the identifiers of the buffer objects

```c
void glGenBuffersARB(GLsizei n, GLuint* ids)
```

• `n`: number of buffer objects to create
• `ids`: the address of a GLuint variable or array to store a single ID or multiple IDs
• `glBindBufferARB()`
  – Once the buffer object has been created, we need to hook the buffer object with the corresponding ID before using the buffer object.

```c
void glBindBufferARB(GLenum target, GLuint id)
```

– Target is either
  • `GL_ARRAY_BUFFER_ARB`: Any vertex attributes, such as vertex coordinates, texture coordinates, normals and color component arrays
  • `GL_ELEMENT_ARRAY_BUFFER_ARB`: Index array which is used for `glDraw[Range]Elements()`

– Once `glBindBufferARB()` is first called, VBO initializes the buffer with a zero-sized memory buffer and set the initial VBO states, such as usage and access properties.
• **glBufferDataARB()**
  - You can copy the data into the buffer object with `glBufferDataARB()` when the buffer has been initialized.

```c
void glBufferDataARB(GLenum target, GLsizei size, const void* data, GLenum usage)
```

- **target** is either `GL_ARRAY_BUFFER_ARB` or `GL_ELEMENT_ARRAY_BUFFER_ARB`.
- **size** is the number of bytes of data to transfer.
- The third parameter is the pointer to the array of source data.
- **"usage"** flag is a performance hint for VBO to provide how the buffer object is going to be used: static, dynamic or stream, and read, copy or draw.
• 9 enumerated values for usage flags
  – GL_STATIC_DRAW_ARB
  – GL_STATIC_READ_ARB
  – GL_STATIC_COPY_ARB
  – GL_DYNAMIC_DRAW_ARB
  – GL_DYNAMIC_READ_ARB
  – GL_DYNAMIC_COPY_ARB
  – GL_STREAM_DRAW_ARB
  – GL_STREAM_READ_ARB
  – GL_STREAM_COPY_ARB
  – GL_STREAM_COPY_ARB

  • Static: data in VBO will not be changed
  • Dynamic: the data will be changed frequently
  • Stream: the data will be changed every frame
  • Draw: the data will be sent to GPU in order to draw
  • Read: the data will be read by the client's application
  • Copy: the data will be used both drawing and reading
• `glBufferSubDataARB()`

```c
void glBufferSubDataARB(GLenum target, GLint offset, GLsizei size, void* data)
```

– Like `glBufferDataARB()`,
  • used to copy data into VBO
– It only replaces a range of data into the existing buffer, starting from the given offset.
– The total size of the buffer must be set by `glBufferDataARB()` before using `glBufferSubDataARB()`.
• `glDeleteBuffersARB()`

```c
void glDeleteBuffersARB(GLsizei n, const GLuint* ids)
```

– You can delete a single VBO or multiple VBOs with `glDeleteBuffersARB()` if they are not used anymore. After a buffer object is deleted, its contents will be lost.
Drawing VBO

• VBO sits on top of the existing vertex array implementation
• Rendering VBO is almost same as using vertex array.
• The pointer to the vertex array is now an offset into a currently bound buffer object.
• No additional APIs are required to draw a VBO except glBindBufferARB().
• Binding the buffer object with 0 switches off VBO operation.
  – It is a good idea to turn VBO off after use, so normal vertex array operations with absolute pointers will be re-activated.
Example Code

// bind VBOs for vertex array and index array
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vbold1); // for vertex coords
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, vbold2); // for indices

// do same as vertex array except pointer
glEnableClientState(GL_VERTEX_ARRAY); // activate vertex coords array
glVertexPointer(3, GL_FLOAT, 0, 0); // last param is offset, not ptr

// draw 6 quads using offset of index array
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, 0);

// deactivate vertex array

// bind with 0, so, switch back to normal pointer operation
glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, 0);
Updating VBO

• Two ways to update
  – Copy new data into the bound VBO with `glBufferDataARB()` or `glBufferSubDataARB()`.
    • 2 copies of vertex data: one in your application and the other in VBO.
  – Map the buffer object into client's memory, and the client can update data with the pointer to the mapped buffer.
• **glMapBufferARB()**
  - map the buffer object into client's memory
  - returns pointer to buffer

```c
void* glMapBufferARB(GLenum target, GLenum access)
```

- target is either GL_ARRAY_BUFFER_ARB or GL_ELEMENT_ARRAY_BUFFER_ARB.
- The second parameter, access flag specifies what to do with the mapped data: read, write or both.
  - GL_READ_ONLY_ARB
  - GL_WRITE_ONLY_ARB
  - GL_READ_WRITE_ARB
- Causes a synchronizing issue.
  - If GPU is still working with the buffer object, `glMapBufferARB()` will not return until GPU finishes its job with the corresponding buffer object.
  - To avoid waiting (idle), you can call first `glBufferDataARB()` with NULL pointer, then call `glMapBufferARB()`.
  - Valid only if you want to update entire data set
• `glUnmapBufferARB()`

```c
GLboolean glUnmapBufferARB(GLenum target)
```

- After modifying the data of VBO, buffer object must be unmapped from the client's memory
- Returns GL_TRUE if no problems with update
- Returns GL_FALSE if contents were corrupted while buffer was mapped
Transformations
Objectives

• Introduce standard transformations
  – Rotations
  – Translation
  – Scaling
  – Shear

• Derive homogeneous coordinate transformation matrices

• Learn to build arbitrary transformation matrices from simple transformations
General Transformations

- A transformation maps points to other points and/or vectors to other vectors

Q = T(P)

v = T(u)
Affine Transformations

- Line preserving
- Characteristic of many physically important transformations
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- Importance in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints
Pipeline Implementation

T (from application program)

T(p)

T(q)

vertices

vertices

pixels

transformed vertices

transformed pixels

frame buffer

from application program

Angel: Interactive Computer Graphics 3E
© Addison-Wesley 2002
Notation

We will be working with both coordinate-free representations of transformations and representations within a particular frame.

\( P, Q, R: \) points in an affine space
\( u, v, w: \) vectors in an affine space
\( \alpha, \beta, \gamma: \) scalars

\( p, q, r: \) representations of points
- array of 4 scalars in homogeneous coordinates
- e.g., \([1.0, 2.3, -0.4, 1]\)

\( u, v, w: \) representations of vectors
- array of 4 scalars in homogeneous coordinates
- e.g., \([-2.8, 1.5, 0.9, 0]\)
Translation

• Move (translate, displace) a point to a new location

• Displacement determined by a vector $d$
  – Three degrees of freedom
  – $P'=P+d$
How many ways?

Although we can move a point to a new location in infinite ways, when we move many points there is usually only one way.

object

translation: every point displaced by same vector

$\vec{d}$
Translation Using Representations

Using the homogeneous coordinate representation in some frame

\[ \mathbf{p} = [x \, y \, z \, 1]^T \]
\[ \mathbf{p}' = [x' \, y' \, z' \, 1]^T \]
\[ \mathbf{d} = [dx \, dy \, dz \, 0]^T \]

Hence \( \mathbf{p}' = \mathbf{p} + \mathbf{d} \) or

\[ x' = x + d_x \]
\[ y' = y + d_y \]
\[ z' = z + d_z \]

Note that this expression is in four dimensions and expresses that point = vector + point.
Scaling

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

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

\[ p' = Sp \]

\[ S = S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Reflection corresponds to negative scale factors

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

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

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

We can also express translation using a 4 x 4 matrix $T$ in homogeneous coordinates

$$p' = Tp$$

where

$$T = T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together.
Rotation (2D)

• Consider rotation about the origin by $\theta$ degrees
  – radius stays the same, angle increases by $\theta$

\[
x = r \cos (\phi + \theta) \\
y = r \sin (\phi + \theta)
\]

\[
x' = x \cos \theta - y \sin \theta \\
y' = x \sin \theta + y \cos \theta
\]

\[
x = r \cos \phi \\
y = r \sin \phi
\]
Rotation about the $z$ axis

- Rotation about $z$ axis in three dimensions leaves all points with the same $z$
  - Equivalent to rotation in two dimensions in planes of constant $z$
    \[
    x' = x \cos \theta - y \sin \theta \\
    y' = x \sin \theta + y \cos \theta \\
    z' = z
    \]
  - or in homogeneous coordinates
    \[
    p' = R_z(\theta)p
    \]
Rotation Matrix

\[ R = R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Rotation about $x$ and $y$ axes

- Same argument as for rotation about $z$ axis
  - For rotation about $x$ axis, $x$ is unchanged
  - For rotation about $y$ axis, $y$ is unchanged

\[
\mathbf{R} = \mathbf{R}_x(\theta) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\mathbf{R} = \mathbf{R}_y(\theta) = \begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
General Rotation About the Origin

A rotation by $\theta$ about an arbitrary axis can be decomposed into the concatenation of rotations about the $x$, $y$, and $z$ axes

$$R(\theta) = R_z(\theta_z) R_y(\theta_y) R_x(\theta_x)$$

$\theta_x \theta_y \theta_z$ are called the Euler angles

Note that rotations do not commute. We can use rotations in another order but with different angles.
Rotation Around an Arbitrary Axis

- Rotate a point P around axis \( n (x,y,z) \) by angle \( \theta \)

\[
R = \begin{bmatrix}
    tx^2 + c & txy + sz & txz - sy & 0 \\
txy - sz & ty^2 + c & tyz + sx & 0 \\
txz + sy & tyz - sx & tz^2 + c & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- \( c = \cos(\theta) \)
- \( s = \sin(\theta) \)
- \( t = (1 - c) \)

Graphics Gems I, p. 466 & 498
Rotation Around an Arbitrary Axis

• Also can be expressed as the Rodrigues Formula

\[
P_{rot} = P \cos(\vartheta) + (n \times P) \sin(\vartheta) + n(n \cdot P)(1 - \cos(\vartheta))
\]
Rotation About a Fixed Point other than the Origin

Move fixed point to origin
Rotate
Move fixed point back

\[ p' = (ABC)p = A(B(Cp)) \]
\[ M = T(p_f) R(\theta) T(-p_f) \]
Improved Rotations

- Euler Angles have problems
  - How to interpolate keyframes?
  - Angles aren’t independent
  - Interpolation can create Gimble Lock, i.e.
    loss of a degree of freedom when axes align

- Solution: Quaternions!
Quaternions

Matrices are not the only (or best) way of representing rotations. For one thing, they are redundant (9 numbers instead of 3) and, for another, they are difficult to interpolate.

An alternative representation was developed by Hamilton in the early 19th century (and forgotten until relatively recently). The quaternion is a 4-tuple of reals with the operations of addition and multiplication defined. Just as complex numbers allow us to multiply and divide two-dimensional vectors, quaternions enable us to multiply and divide four dimensional vectors.

\[ q = q_0 + q_1 i + q_2 j + q_3 k \]
\[ i^2 = j^2 = k^2 = -1 \quad ij = k, jk = i, ki = j \]

A quaternion can also be interpreted as having a scalar part and a vector part. This will give us a more convenient notation.

\[ q = (s, \bar{a}) \quad \text{pure quaternion: } p = (0, \bar{a}) \]

Quaternion addition is just the usual vector addition, the quaternion product is defined as:

\[ q_1 q_2 = (s_1 s_2 - (\bar{a}_1 \cdot \bar{a}_2), s_1 \bar{a}_2 + s_2 \bar{a}_1 + \bar{a}_1 \times \bar{a}_2) \]
Quaternion Facts

conjugate: \( q^* = (s, -\vec{a}) \)
magnitude: \( |q| = \sqrt{qq^*} = \sqrt{s^2 + \vec{a} \cdot \vec{a}} \)
unit quaternion: \( |q| = 1 \)
inverse: \( q^{-1} = \frac{1}{|q|^2} q^* \)
\( q^{-1} = q^* \), for unit quaternions

It turns out that we will be able to represent rotations with a unit quaternion. Before looking at why this is so, there are a few important properties to keep in mind:

- The unit quaternions form a three-dimensional sphere in the 4-dimensional space of quaternions.
- Any quaternion can be interpreted as a rotation simply by normalizing it (dividing it by its length).
- Both \( q \) and \( -q \) represent the same rotation (corresponding to angles of \( q \) and \( 2p - q \)
Rotation by Quaternion

\[ R_\mathbf{q}(\mathbf{p}) = \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \quad p = (0, \mathbf{x}) \]

\[ \mathbf{q} = (\cos(\theta/2), \sin(\theta/2)\mathbf{\alpha}), \quad \text{where } |\mathbf{\alpha}| = 1 \]

\[ R_\mathbf{q}(\mathbf{p}) = (0, \quad (s^2 - \mathbf{\alpha} \cdot \mathbf{\alpha})\mathbf{x} \\
+ 2\mathbf{\alpha}(\mathbf{\alpha} \cdot \mathbf{x}) \\
+ 2s(\mathbf{\alpha} \times \mathbf{x}) \ ) \]

\[ R_\mathbf{q}(\mathbf{p}) = (0, \quad (\cos^2(\theta/2) - \sin^2(\theta/2))\mathbf{x} \\
+ (2\sin^2(\theta/2))\mathbf{\alpha}(\mathbf{\alpha} \cdot \mathbf{x}) \\
+ (2\cos(\theta/2)\sin(\theta/2))(\mathbf{\alpha} \times \mathbf{x}) \ ) \]

\[ R_\mathbf{q}(\mathbf{p}) = (0, \quad (\cos \theta)\mathbf{x} \\
+ (1 - \cos \theta)\mathbf{\alpha}(\mathbf{\alpha} \cdot \mathbf{x}) \\
+ (\sin \theta)(\mathbf{\alpha} \times \mathbf{x}) \ ) \]

Recognize this? It is the Rodrigues formula!
QuatERNION COMPOSITION

Since a quaternion basically stores the axis vector and angle of rotation, it is not surprising that we can write the components of a rotation matrix given the quaternion components.

\[ q = (\cos(\theta/2), \sin(\theta/2) \hat{a}) = (w, (x, y, z)) \]

\[
R_q = \\
\begin{pmatrix}
1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy & 0 \\
2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx & 0 \\
2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

Crucially, the composition of two rotations given by quaternions is simply their quaternion product.

\[ R_{q'}(R_q(p)) = R_{q''}(p) \quad \text{where} \quad q'' = q'q \]

- Note that the product of two unit quaternions is another unit quaternion.
- Note that quaternion multiplication, like matrix multiplication, is not commutative.
From rotation matrix to quaternion

Given $R = (r_{ij})$, solve expression on previous page for quaternion elements $q_i$

Linear combinations of diagonal elements seem to solve the problem:

$$q_0^2 = \frac{1}{4} (1 + r_{11} + r_{22} + r_{33})$$

$$q_1^2 = \frac{1}{4} (1 + r_{11} - r_{22} - r_{33})$$

$$q_2^2 = \frac{1}{4} (1 - r_{11} + r_{22} - r_{33})$$

$$q_3^2 = \frac{1}{4} (1 - r_{11} - r_{22} + r_{33})$$

so take four square roots and you’re done? You have to figure the signs out. There is a better way ...
Look at the off-diagonal elements

\[ q_0 q_1 = \frac{1}{4}(r_{32} - r_{23}) \]
\[ q_0 q_2 = \frac{1}{4}(r_{13} - r_{31}) \]
\[ q_0 q_3 = \frac{1}{4}(r_{21} - r_{12}) \]
\[ q_1 q_2 = \frac{1}{4}(r_{12} + r_{21}) \]
\[ q_1 q_3 = \frac{1}{4}(r_{13} + r_{31}) \]
\[ q_2 q_3 = \frac{1}{4}(r_{23} + r_{32}) \]

Given any one \( q_i \), could solve the above for the other three.
The procedure

1. Use first four equations to find the largest $q_i^2$. Take its square root.

2. Use the last six equations (well, three of them anyway) to solve for the other $q_i$.

That way, only have to worry about getting one sign right.

Actually $q$ and $-q$ represent the same rotation, so no worries about signs.

Taking the largest square root avoids division by small numbers.

Quaternion Interpolation

One of the main motivations for using quaternions in Graphics is the ease with which we can define interpolation between two orientations. Think, for example, about moving a camera smoothly between two views.

\[ \cos \Omega = A \cdot B \]

\[
C(t) = \text{slerp}(A, B, t) = A \frac{\sin(\Omega(1-t))}{\sin \Omega} + B \frac{\sin(\Omega t)}{\sin \Omega}
\]

slerp – Spherical linear interpolation

Need to take equals steps on the sphere
What about interpolating multiple keyframes?

• Shoemake suggests using Bezier curves on the sphere
• Offers a variation of the De Casteljau algorithm using slerp and quaternion control points
Shear

- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions
Shear Matrix

Consider simple shear along $x$ axis

\[
x' = x + y \cot \theta \\
y' = y \\
z' = z
\]

\[
H(\theta) = \begin{bmatrix}
1 & \cot \theta & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - Translation: $T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z)$
  - Rotation: $R^{-1}(\theta) = R(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$
      $R^{-1}(\theta) = R^T(\theta)$
  - Scaling: $S^{-1}(s_x, s_y, s_z) = S(1/s_x, 1/s_y, 1/s_z)$
Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices.
- Because the same transformation is applied to many vertices, the cost of forming a matrix $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ is not significant compared to the cost of computing $Mp$ for many vertices $p$.
- The difficult part is how to form a desired transformation from the specifications in the application.
Order of Transformations

- Note that matrix on the right is the first applied.
- Mathematically, the following are equivalent:
  \[ p' = (ABC)p = A(B(Cp)) \]
- Note many references use column matrices to present points. In terms of column matrices:
  \[ p'^T = p^TC^TB^TA^T \]
Instancing

• In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size

• We apply an instance transformation to its vertices to
  
  Scale
  Orient
  Locate
OpenGL Transformations
Objectives

• Learn how to carry out transformations in OpenGL
  – Rotation
  – Translation
  – Scaling

• Introduce OpenGL matrix modes
  – Model-view
  – Projection
OpenGL Matrices

• In OpenGL matrices are part of the state
• Three types
  – Model-View (GL_MODELVIEW)
  – Projection (GL_PROJECTION)
  – Texture (GL_TEXTURE) (ignore for now)
• Single set of functions for manipulation
• Select which to manipulated by
  – glMatrixMode(GL_MODELVIEW);
  – glMatrixMode(GL_PROJECTION);
Current Transformation Matrix (CTM)

- Conceptually there is a 4 x 4 homogeneous coordinate matrix, the *current transformation matrix* (CTM) that is part of the state and is applied to all vertices that pass down the pipeline.
- The CTM is defined in the user program and loaded into a transformation unit.
CTM in OpenGL

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM.
- Can manipulate each by first setting the matrix mode.
CTM operations

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

Load an identity matrix: \( C \leftarrow I \)
Load an 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: \( C \leftarrow CS \)
Rotation about a Fixed Point

Start with identity matrix: \( C \leftarrow I \)
Move fixed point to origin: \( C \leftarrow C T^{-1} \)
Rotate: \( C \leftarrow CR \)
Move fixed point back: \( C \leftarrow CT \)

Result: \( C = T^{-1}R T \)
Each operation corresponds to one function call in program.

Recall! \( p' = (ABC)p = A(B(Cp)) \)

Note: The last operation specified is the first applied to point!

So transformation should be composed “backwards”,
\( C = TRT^{-1} \)
Rotation, Translation, Scaling

Load an identity matrix:

\[ \text{glLoadIdentity()} \]

Multiply CTM on right with associated matrix:

\[ \text{glRotatef}(\theta, vx, vy, vz) \]

\( \theta \) in degrees, \((vx, vy, vz)\) define axis of rotation

\[ \text{glTranslatef}(dx, dy, dz) \]

\[ \text{glScalef}(sx, sy, sz) \]

Each has a float (f) and double (d) format (\textit{glScaled})
Example

• Two examples give the same result

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -2.0);
glutWireTetrahedron();
glLoadIdentity();
glTranslatef(0.0, 0.0, -3.0);
glutWireCube();

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -2.0);
glutWireTetrahedron();
glTranslatef(0.0, 0.0, -1.0);
glutWireCube();
```

• Type-o in Angel Primer!
Example

• Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(1.0, 2.0, 3.0);
        glRotatef(30.0, 0.0, 0.0, .10);
        glTranslatef(-1.0, -2.0, -3.0);

• Remember that last matrix specified in the program is the first applied
Arbitrary Matrices

• Can load and multiply by matrices defined in the application program
  
  \[
  \text{glLoadMatrix}\{\text{fd}\}(m) \\
  \text{glMultMatrix}\{\text{fd}\}(m)
  \]

• The matrix \(m\) is a one dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by \text{columns}.

• In \text{glMultMatrix}\{\text{fd}\}, \(m\) multiplies the existing matrix on the right.
Matrix Stacks

• In many situations we want to save transformation matrices for use later
  – Traversing hierarchical data structures
  – Avoiding state changes when executing display lists

• OpenGL maintains stacks for each type of matrix
  – Save current matrix (as set by `glMatrixMode`) by
    
    `glPushMatrix()`

    • Does not change current matrix value!
  – Restore saved matrix to CTM by
    
    `glPopMatrix()`
Reading Back Matrices

• Can also access matrices (and other parts of the state) by *enquiry* (*query*) functions

  ```
  GLfloat m[16];
  glGetFloatv(GL_MODELVIEW, m);
  ```

• For matrices, we use as

  ```
  glGetIntegerv
  glGetFloatv
  glGetBooleanv
  glGetDoublev
  glGetBooleanv
  glEnable
  ```
Using Transformations

• Example: use idle function to rotate a cube and mouse function to change direction of rotation

• Start with a program that draws a cube (colorcube.c) in a standard way
  – Centered at origin
  – Sides aligned with axes
  – Will discuss modeling in next lecture
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();
}
Idle and Mouse callbacks

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

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

void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube();
    glutSwapBuffers();
}

Note that because of fixed form of callbacks, variables such as theta and axis must be defined as globals

Camera information is in standard reshape callback
Using the Model-View Matrix

• In OpenGL the model-view matrix is used to
  – Position the camera
    • Can be done by rotations and translations but is often easier to use \texttt{gluLookAt}
  – Build models of objects

• The projection matrix is used to define the view volume and to select a camera lens

• Although both are manipulated by the same functions, we have to be careful because incremental changes are always made by postmultiplication
  – For example, rotating model-view and projection matrices by the same matrix are not equivalent operations.
Quaternions

- Extension of imaginary numbers from two to three dimensions
- Requires one real and three imaginary components $i$, $j$, $k$

\[ q = q_0 + q_1 i + q_2 j + q_3 k \]

- Quaternions can express rotations on sphere smoothly and efficiently. Process:
  - Model-view matrix $\rightarrow$ quaternion
  - Carry out operations with quaternions
  - Quaternion $\rightarrow$ Model-view matrix
Hierarchical Modeling
Objectives

• Examine the limitations of linear modeling
  – Symbols and instances
• Introduce hierarchical models
  – Articulated models
  – Robots
• Introduce Tree and DAG models
Instance Transformation

• Start with a prototype object (a *symbol*)
• Each appearance of the object in the model is an *instance*
  – Must scale, orient, position
  – Defines instance transformation
Symbol-Instance Table

Can store a model by assigning a number to each symbol and storing the parameters for the instance transformation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Scale</th>
<th>Rotate</th>
<th>Translate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(s_x, s_y, s_z)</td>
<td>(\theta_x, \theta_y, \theta_z)</td>
<td>(d_x, d_y, d_z)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Relationships in Car Model

- Symbol-instance table does not show relationships between parts of model
- Consider model of car
  - Chassis + 4 identical wheels
  - Two symbols

- Rate of forward motion determined by rotational speed of wheels
Structure Through Function Calls

car(speed)
{
    chassis()
    wheel(right_front);
    wheel(left_front);
    wheel(right_rear);
    wheel(left_rear);
}

• Fails to show relationships well
• Look at problem using a graph
Graphs

• Set of nodes and edges (links)
• Edge connects a pair of nodes
  – Directed or undirected
• Cycle: directed path that is a loop
Tree

- Graph in which each node (except the root) has exactly one parent node
  - May have multiple children
  - Leaf or terminal node: no children
Tree Model of Car

- Chassis
  - Right-front wheel
  - Left-front wheel
  - Right-rear wheel
  - Left-rear wheel
DAG Model

- If we use the fact that all the wheels are identical, we get a directed acyclic graph.
  – Not much different than dealing with a tree.
Modeling with Trees

• Must decide what information to place in nodes and what to put in edges

• Nodes
  – What to draw
  – Pointers to children

• Edges
  – May have information on incremental changes to transformation matrices (can also store in nodes)
Robot Arm

robot arm

parts in their own coordinate systems
Articulated Models

- Robot arm is an example of an *articulated model*
  - Parts connected at joints
  - Can specify state of model by giving all joint angles
Relationships in Robot Arm

• Base rotates independently
  – Single angle determines position

• Lower arm attached to base
  – Its position depends on rotation of base
  – Must also translate relative to base and rotate about connecting joint

• Upper arm attached to lower arm
  – Its position depends on both base and lower arm
  – Must translate relative to lower arm and rotate about joint connecting to lower arm
Required Matrices

• Rotation of base: $R_b$
  – Apply $M = R_b$ to base

• Translate lower arm relative to base: $T_{lu}$

• Rotate lower arm around joint: $R_{lu}$
  – Apply $M = R_b T_{lu} R_{lu}$ to lower arm

• Translate upper arm relative to lower arm: $T_{uu}$

• Rotate upper arm around joint: $R_{uu}$
  – Apply $M = R_b T_{lu} R_{lu} T_{uu} R_{uu}$ to upper arm
OpenGL Code for Robot

robot_arm()
{
    glRotate(theta, 0.0, 1.0, 0.0);
    base();
    glTranslate(0.0, h1, 0.0);
    glRotate(phi, 0.0, 0.0, 1.0);
    lower_arm();
    glTranslate(0.0, h2, 0.0);
    glRotate(psi, 0.0, 0.0, 1.0);
    upper_arm();
}
Robot Arm

robot arm

parts in their own coordinate systems
OpenGL Code for base()

GLUquadricObj *p;

void base()
{
    glPushMatrix();
    glRotate(-90.0, 1.0, 0.0, 0.0);
    gluCylinder(p, BASE_RADIUS, BASE_RADIUS, BASE_HEIGHT, 5, 5);
    glPopMatrix();
}
OpenGL Code for lower_arm()

```c
void lower_arm()
{
    glPushMatrix();
    glTranslatef(0.0, 0.5*LOWER_ARM_HEIGHT, 0.0);
    glScalef(LOWER_ARM_WIDTH, LOWER_ARM_HEIGHT, LOWER_ARM_WIDTH);
    glutWireCube(1.0);
    glPopMatrix();
}
```
Tree Model of Robot

- Note code shows relationships between parts of model
  - Can change “look” of parts easily without altering relationships
- Simple example of tree model
- Want a general node structure for nodes
Possible Node Structure

- Code for drawing part or pointer to drawing function
- Linked list of pointers to children
- Matrix relating node to parent
Generalizations

• Need to deal with multiple children
  – How do we represent a more general tree?
  – How do we traverse such a data structure?

• Animation
  – How to use dynamically?
  – Can we create and delete nodes during execution?
Objectives

- Build a tree-structured model of a humanoid figure
- Examine various traversal strategies
- Build a generalized tree-model structure that is independent of the particular model
Humanoid Figure
Building the Model

• Can build a simple implementation using quadrics: ellipsoids and cylinders
• Access parts through functions
  – torso()
  – left_upper_arm()
• Matrices describe position of node with respect to its parent
  – $\mathbf{M}_{lla}$ positions left lower arm with respect to left upper arm
Tree with Matrices
Display and Traversal

- The position of the figure is determined by 11 joint angles (two for the head and one for each other part)
- Display of the tree requires a graph traversal
  - Visit each node once
  - Display function at each node that describes the part associated with the node, applying the correct transformation matrix for position and orientation
Transformation Matrices

• There are 10 relevant matrices
  – $M$ positions and orients entire figure through the torso which is the root node
  – $M_h$ positions head with respect to torso
  – $M_{lua}$, $M_{rua}$, $M_{lul}$, $M_{rul}$ position arms and legs with respect to torso
  – $M_{lla}$, $M_{rla}$, $M_{lll}$, $M_{rll}$ position lower parts of limbs with respect to corresponding upper limbs
Stack-based Traversal

• Set model-view matrix to $M$ and draw torso
• Set model-view matrix to $MM_h$ and draw head
• For left-upper arm need $MM_{lua}$ and so on
• Rather than recomputing $MM_{lua}$ from scratch or using an inverse matrix, we can use the matrix stack to store $M$ and other matrices as we traverse the tree
Traversal Code

```c
figure() {
    glPushMatrix()
    torso();
glRotate3f(...);
    head();
glPopMatrix();
    glPushMatrix();
    save present model-view matrix
    update model-view matrix for head
    recover original model-view matrix
    save it again
    update model-view matrix
    for left upper arm
    recover and save original
    model-view matrix again
    glPushMatrix();
    left_upper_arm();
glPopMatrix();
glPushMatrix();
glTranslate3f(...);
glRotate3f(...);

rest of code
```
Analysis

• The code describes a particular tree and a particular traversal strategy
  – Can we develop a more general approach?
• Note that the sample code does not include state changes, such as changes to colors
  – May also want to use glPushAttrib and glPopAttrib to protect against unexpected state changes affecting later parts of the code
General Tree Data Structure

• Need a data structure to represent tree and an algorithm to traverse the tree
• We will use a left-child right sibling structure
  – Uses linked lists
  – Each node in data structure has two pointers
  – Left: linked list of children next node
  – Right: next node (siblings)
Left-Child Right-Sibling Tree

Child   Sibling

Root
Child  Siblings
Tree node Structure

• At each node we need to store
  – Pointer to sibling
  – Pointer to child
  – Pointer to a function that draws the object represented by the node
  – Homogeneous coordinate matrix to multiply on the right of the current model-view matrix
    • Represents changes going from parent to node
    • In OpenGL this matrix is a 1D array storing matrix by columns
C Definition of treenode

typedef struct treenode
{
    GLfloat m[16];
    void (*f)();
    struct treenode *child;
    struct treenode *sibling;
} treenode;
treenode torso_node, head_node, lua_node, ... ;
    /* use OpenGL functions to form matrix */
glLoadIdentity();
    glRotatef(theta[0], 0.0, 1.0, 0.0);
    /* move model-view matrix to m */
glGetFloatv(GL_MODELVIEW_MATRIX, torso_node.m)

    torso_node.f = torso; /* torso() draws torso */
    Torso_node.sibling = NULL;
    Torso_node.child = &head_node;
Notes

• The position of figure is determined by 11 joint angles stored in \texttt{theta[11]}
• Animate by changing the angles and redisplaying
• We form the required matrices using \texttt{glRotate} and \texttt{glTranslate}
  – More efficient than software
  – Because the matrix is formed in model-view matrix, we should first push original model-view matrix on matrix stack
Preorder Traversal

```c
void traverse(treenode *node) {
    if(node == NULL) return;
    glPushMatrix();
    glMultMatrix(node->m);
    node->f();
    if(node->child != NULL)
        traverse(node->child);
    glPopMatrix();
    if(node->sibling != NULL)
        traverse(node->sibling);
}
```
Notes

• We must save modelview matrix before multiplying it by node matrix
  – Updated matrix applies to children of node but not to siblings which contain their own matrices
• The traversal program applies to any left-child right-sibling tree
  – The particular tree is encoded in the definition of the individual nodes
• The order of traversal matters because of possible state changes in the functions
Dynamic Trees

• If we use pointers, the structure can be dynamic

```c
typedef treenode *tree_ptr;
tree_ptr torso_ptr;
torso_ptr = malloc(sizeof(treenode));
```

• Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution