Transitions

- In OpenGL, transformation are performed in the opposite order they are called:

  4. `translate(1.0, 1.0, 0.0);`
  3. `rotateZ(45.0);`
  2. `scale(2.0, 2.0, 0.0);`
  1. `DrawSquare(0.0, 0.0, 1.0);`

Rotation and Scaling

- Rotation and Scaling is done about origin:
  - You always get what you expect
  - Correct on all parts of model:

  4. `rotateZ(45.0);`
  3. `scale(2.0, 2.0, 0.0);`
  2. `translate(-0.5, -0.5, 0.0);`
  1. `DrawSquare(0.0, 0.0, 1.0);`
Load and Mult Matrices in MV.js

- $\text{Mat4}(m)$
- $\text{Mat4}(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)$
  - Sets the sixteen values of the current matrix to those specified by $m$.
- $\text{CTM} = \text{mult}(\text{CTM}, \text{xformMatrix});$
  - Multiplies the matrix $\text{CTM}$, by $\text{xformMatrix}$ and stores the result as the current matrix, $\text{CTM}$.

- OpenGL uses column instead of row vectors
- However, MV.js treats things in row-major order
  - Flatten does the transpose
  - Matrices are defined like this (use float $m[16]$):

$$
M = \begin{bmatrix}
1 & 5 & 9 & 13 \\
2 & 6 & 10 & 14 \\
3 & 7 & 11 & 15 \\
4 & 8 & 12 & 16
\end{bmatrix}
$$
Object Coordinate System

- Used to place objects in scene
  - Draw at origin of WCS
  - Scale and Rotate
  - Translate to final position
- Use the MODELVIEW matrix as the CTM
  - scale(x, y, z)
  - rotate(\text{XYZ})(\text{angle})
  - translate(x, y, z)
  - lookAt(\text{eyeX}, \text{eyeY}, \text{eyeZ}, \text{atX, atY, atZ, upX, upY, upZ})

lookAt

\text{LookAt(eye, at, up)}
The lookAt Function

- The GLU library contained the function gluLookAt to form the required modelview matrix through a simple interface
- Note the need for setting an up direction
- Replaced by lookAt() in MV.js
  - Can concatenate with modeling transformations
- Example: isometric view (45 deg) of cube aligned with axes

```javascript
var eye = vec3(1.0, 1.0, 1.0);
var at = vec3(0.0, 0.0, 0.0);
var up = vec3(0.0, 1.0, 0.0);
var mv = LookAt(eye, at, up);
```

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015

The lookAt Function: change from WORLD space to EYE space

**View Matrix**

We want to compute the view matrix that aligns the orthonormal basis at the origin and pointing down either the +Z (right-handed) or -Z (left-handed). Here's the picture:
\[ M_{sys} = M_{screen} \times M_{perspective} \times M_{view} \]

**Perspective Transformations**

Viewing system matrix \( M_{sys} \) transform is obtained by combining the view matrix with the perspective projection with the viewport to screen matrix. These are defined as:

\[ M_{sys} = M_{screen} M_{perspective} M_{view} \]

We will look at \( M_{screen} M_{perspective} \) later.

---

**Right Hand System**

\[
\begin{align*}
W &= \frac{\text{eye} - \text{at}}{\|\text{eye} - \text{at}\|} \\
U &= \frac{\text{up} \times W}{\|\text{up} \times W\|} \\
V &= \frac{W \times U}{\|W \times U\|}
\end{align*}
\]

**Left Hand System**

\[
\begin{align*}
W &= \frac{\text{at} - \text{eye}}{\|\text{at} - \text{eye}\|} \\
U &= \frac{\text{up} \times W}{\|\text{up} \times W\|} \\
V &= \frac{\text{u} \times W}{\|\text{u} \times W\|}
\end{align*}
\]
Some Examples $M_{view}$

Orthonormal Rotation about origin | Translation to origin | $W = \frac{\text{eye} - \text{at}}{\|\text{eye} - \text{at}\|}$
---|---|---
$U_x \ U_y \ U_z \ 0 \quad 1 \ 0 \ 0$ | $-\text{eye}_x$ | $\text{VERTEX}_x$
$V_x \ V_y \ V_z \ 0 \quad 0 \ 1 \ 0$ | $-\text{eye}_y$ | $\text{VERTEX}_y$
$W_x \ W_y \ W_z \ 0 \quad 0 \ 0 \ 1$ | $-\text{eye}_z$ | $\text{VERTEX}_z$
$0 \ 0 \ 0 \ 1$ | $0 \ 0 \ 0 \ 1$ | $1$

$M_{view}$

Orthonormal Rotation about origin | Translation to origin | $W = \frac{\text{eye} - \text{at}}{\|\text{eye} - \text{at}\|}$
---|---|---
$U_x \ U_y \ U_z \ 0 \quad 1 \ 0 \ 0$ | $-\text{eye}_x$ | $\text{VERTEX}_x$
$V_x \ V_y \ V_z \ 0 \quad 0 \ 1 \ 0$ | $-\text{eye}_y$ | $\text{VERTEX}_y$
$W_x \ W_y \ W_z \ 0 \quad 0 \ 0 \ 1$ | $-\text{eye}_z$ | $\text{VERTEX}_z$
$0 \ 0 \ 0 \ 1$ | $0 \ 0 \ 0 \ 1$ | $1$
\[ M_{\text{sys}} = M_{\text{screen}} \times M_{\text{perspective}} \times M_{\text{view}} \]

**Perspective Transformations**

Viewing system matrix $M_{\text{sys}}$ transform is obtained by combining the view matrix with the perspective projection with the viewport to screen matrix. These are defined as:

\[ M_{\text{sys}} = M_{\text{screen}} \times M_{\text{perspective}} \times M_{\text{view}} \]
\[ M_{sys} = M_{screen} \times M_{perspective} \times M_{view} \]

**Perspective Transformations**

Viewing system matrix \( M_{sys} \) transform is obtained by combining the view matrix with the perspective projection with the viewport to screen matrix. These are defined as:

\[ M_{sys} = M_{screen} \times M_{perspective} \times M_{view} \]

Now Map Rectangles

\[ (u_{max}, v_{max}) \]

\[ (u_{min}, v_{min}) \]

\[ (x_{min}, y_{min}) \]

\[ (x_{max}, y_{max}) \]
Transformation in $x$ and $y$

\[
\begin{bmatrix}
1 & 0 & u_{\text{min}} \\
0 & 1 & v_{\text{min}} \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\lambda_x & 0 & 0 \\
0 & \lambda_y & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -x_{\text{min}} \\
0 & 1 & -y_{\text{min}} \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]

where, \( \lambda_x = \frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}} \), \( \lambda_y = \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}} \)

This isViewport Transformation

- Good for mapping objects from one coordinate system to another
- This is what we do with windows and viewports
- \( M_{\text{window}} = M_{\text{screen}} \)
Canonical to Window

- Canonical Viewing Volume (what is it? (NDC))
- To Window (where Nx = number of pixels)
- \( M_{\text{window}} = M_{\text{screen}} \)

\[
M_{\text{window}} = \begin{bmatrix}
\frac{-n_x}{2} & 0 & 0 & \frac{n_x - 1}{2} \\
0 & \frac{-n_y}{2} & 0 & \frac{n_y - 1}{2} \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
M_{\text{sys}} = M_{\text{window}} M_{\text{persp}} M_{\text{view}}
\]
Other Viewing APIs

- The LookAt function is only one possible API for positioning the camera (but a really nice one)
- Others include
  - View reference point, view plane normal, view up (PHIGS, GKS-3D)
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles

General Transformation Commands

- Deprecated:
  - `glMatrixMode()`
    - `Modelview`
    - `Projection`
    - `Texture`
    - `Which matrix will be modified`
    - Subsequent transformation commands affect the specified matrix.
  - `void glLoadIdentity(void);`
    - Sets the currently modifiable matrix to the 4 × 4 identity matrix.
    - Usually done when you first switch matrix mode
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>
</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,direction,time)
{
  chassis(speed,direction,time)
  wheel(right_front,speed,direction,time);
  wheel(left_front,speed,direction,time);
  wheel(right_rear,speed,direction,time);
  wheel(left_rear,speed,direction,time);
}

- 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
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
  - But dealing with a tree is good

Modeling with Trees

- Must decide what information to place in nodes and what to put in edges
- Nodes
  - What to draw
  - Pointers to children
  - Transformation matrices (see below)
- Edges
  - May have information on incremental changes to transformation matrices (can also store in nodes)
Tree Model of Car

Stack Operations

- `mvStack.push(M)`
- `M = mvStack.pop()`
Transformations

- Two ways to specify transformations
  - (1) Each part of the object is transformed independently relative to the world space origin
    Not the best way!
    Translate the base by (5,0,0);
    Translate the lower arm by (5,0);
    Translate the upper arm by (5,0);
  ...

Relative Transformation

A better (and easier) way:
(2) Relative transformation: Specify the transformation for each object relative to its parent
Object Dependency

- A graphical scene often consists of many small objects
- The attributes of an object (positions, orientations) can depend on others

Hierarchical Representation - Scene Graph

- We can describe the object dependency using a tree structure

The position and orientation of an object can be affected by its parent, grand-parent, grand-grand-parent ... nodes

This hierarchical representation is sometimes referred to as Scene Graph
Relative Transformation

Relative transformation: Specify the transformation for each object relative to its parent

Step 1: Translate base and its descendants by (5,0,0);

Relative Transformation (2)

Step 2: Rotate the lower arm and all its descendants relative to its local y axis by -90 degree
Relative Transformation (3)

- Represent relative transformations using scene graph

Do it in WebGL

- Translate base and all its descendants by (5,0,0)
- Rotate the lower arm and its descendants by -90 degree about the locally defined frame

```
// LoadIdentity
modelView = mat4();

... // setup your camera

translatef(5,0,0);

Draw_base();
rotateY(-90);

Draw_lower_arm();
Draw_upper_arm();
Draw_hammer();
```
A more complicated example

- How about this model?

Do this …

- Base and everything – translate (5,0,0)
- Left hammer – rotate 75 degree about the local y
- Right hammer – rotate -75 degree about the local y
Depth-first traversal

- Program this transformation by depth-first traversal

Do transformation(s)
Draw base
Do transformation(s)
Draw left arm
Do transformation(s)
Draw right arm

How about this?

Translate(5,0,0)
Draw base
RotateY(75)
Draw left hammer
RotateY(-75)
Draw right hammer

What's wrong?!
Something is wrong ...

- What’s wrong? - We want to transform the right hammer relative to the base, not to the left hammer.

How about this?

- Do **Translate(5,0,0)**
  - Draw base

- Do **RotateY(75)**
  - Draw left hammer

- Do **RotateY(-75)**
  - Draw right hammer

What’s wrong?!

We should **undo the left hammer transformation** before we transform the right hammer.

Need to undo this first.

Undo the previous transformation(s)

- Need to save the modelview matrix right after we draw base.

Initial modelview **M**

- **Translate(5,0,0) - > M = M x T**
  - Draw base
- **RotateY(75)**
  - Draw left hammer
- **RotateY(-75)**
  - Draw right hammer

Undo the previous transformation means we want to restore the Modelview Matrix **M** to what it was here.

i.e., save **M** right here

... And then restore the saved Modelview Matrix.
We can use OpenGL Matrix Stack to perform matrix save and restore.

<table>
<thead>
<tr>
<th>Initial modelView M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do Translate(5,0,0) -&gt; M = M x T</td>
</tr>
<tr>
<td>Draw base</td>
</tr>
<tr>
<td>Do RotateY(75)</td>
</tr>
<tr>
<td>Draw left hammer</td>
</tr>
<tr>
<td>Do RotateY(-75)</td>
</tr>
<tr>
<td>Draw right hammer</td>
</tr>
</tbody>
</table>

* Store the current modelview matrix
  - Make a copy of the current matrix and **push** into Matrix Stack:
    - Call mvStack.push(modelView)
  - continue to modify the current matrix
* Restore the saved Matrix
  - **Pop** the top of the Matrix and copy it back to the current ModelView Matrix:
    - Call modeView = mvStack.pop()

**Push and Pop Matrix Stack**

A simple OpenGL routine:

```
push base
Lower arm
Upper arm
Hammer
(left hammer)

pop
Lower arm
Upper arm
Hammer
(right hammer)

Depth First Traversal
```

translate(5,0,0)
Draw_base();
mvStack.push(modelView)

rotateY(75);
Draw_left_hammer();

modelView = mvStack.pop();
rotateY(-75);
Draw_right_hammer();

```
Push and Pop Matrix Stack

- Nested push and pop operations

```cpp
// LoadIdentity
modelView = mat4();
...
// Transform using M1;
// Transform using M2;
mvStack.push(modelView);
...
// Transform using M3
mvStack.push(modelView);
...
// Transform using M4
// Transform using M5
...
modelView = mvStack.pop();
...
modelView = mvStack.pop();
```

<table>
<thead>
<tr>
<th>Modelview matrix (M)</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>M = I</td>
<td></td>
</tr>
<tr>
<td>M = M1</td>
<td>M1xM2</td>
</tr>
<tr>
<td>M = M1 x M2</td>
<td>M1xM2xM3</td>
</tr>
<tr>
<td>M = M1 x M2 x M3</td>
<td>M1 x M2</td>
</tr>
<tr>
<td>M = M1 x M2 x M3 x M4</td>
<td></td>
</tr>
<tr>
<td>M = M1 x M2 x M3 x M5</td>
<td></td>
</tr>
<tr>
<td>M = M1 x M2</td>
<td></td>
</tr>
<tr>
<td>M = M1</td>
<td></td>
</tr>
</tbody>
</table>

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
Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders
- Access parts through functions
  - torso()
  - leftUpperArm()
- Matrices describe position of node with respect to its parent
  - $M_{l_{ll}}$ positions left lower leg with respect to left upper arm
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
  - $M_{lua}$, $M_{lua}$, $M_{lul}$, $M_{rul}$ 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
Traversals Code

```c
figure() {
    torso();
    PushMatrix();
    Rotate(...);
    head();
    PopMatrix();
    PushMatrix();
    Translate(...);
    Rotate(...);
    left_upper_arm();
    PushMatrix();
    Translate(...);
    Rotate(...);
    left_lower_arm();
    PopMatrix();
    PopMatrix();
}
```

- 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
- save left upper arm model-view matrix again
- update model-view matrix for left lower arm
- recover upper arm model-view matrix
- recover original model-view matrix
- rest of code

Tree with Matrices

```
```

```
```
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 push and pop other attributes 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 is two pointers
  - Left: next node
  - Right: linked list of children
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 WebGL this matrix is a 1D array storing matrix by columns
Creating a treenode

function createNode(transform, render, sibling, child) {
    var node = {
        transform: transform,
        render: render,
        sibling: sibling,
        child: child,
    }
    return node;
}

Initializing Nodes

function initNodes(Id) {
    var m = mat4();
    switch(Id) {
        case torsoId:
            m = rotate(theta[torsoId], 0, 1, 0);
            figure[torsoId] = createNode( m, torso, null, headId );
            break;
        case head1Id:
        case head2Id:
            m = translate(0.0, torsoHeight+0.5*headHeight, 0.0);
            m = mult(m, rotate(theta[head1Id], 1, 0, 0))m = mult(m,
                rotate(theta[head2Id], 0, 1, 0));
            m = mult(m, translate(0.0, -0.5*headHeight, 0.0));
            figure[headId] = createNode( m, head, leftUpperArmId, null);
            break;
    }
}

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015
Notes

- The position of figure is determined by 11 joint angles stored in \( \theta[11] \)
- Animate by changing the angles and redisplaying
- We form the required matrices using \texttt{rotate} and \texttt{translate}
- Because the matrix is formed using the model-view matrix, we may want to first push original model-view matrix on matrix stack

Preorder Traversal

```javascript
function traverse(Id) {
  if(Id == null) return;
  stack.push(modelViewMatrix);
  modelViewMatrix = mult(modelViewMatrix, figure[Id].transform);
  figure[Id].render();
  if(figure[Id].child != null) traverse(figure[Id].child);
  modelViewMatrix = stack.pop();
  if(figure[Id].sibling != null) traverse(figure[Id].sibling);
}
```

```javascript
var render = function() {
  gl.clear( gl.COLOR_BUFFER_BIT );
  traverse(torsoId);
  requestAnimFrame(render);
}
```
Notes

- We must save model-view 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

- Because we are using JS, the nodes and the node structure can be changed during execution
- Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution
- In desktop OpenGL, if we use pointers, the structure can be dynamic
Animation

- Kinematics/dynamics
- Inverse Kinematics/dynamics
- Keyframing
Hierarchy vs Scene Graph

- Hierarchy just involves object transformations
- Scene Graph involves objects, appearance, lighting, etc.