CS 432 – Interactive Computer Graphics

Lecture 4 – Part 2

3D Transformations

Transformations

- We had already mentioned how to do transformations in 2D
- This included concepts of
  - Homogenous coordinates
  - Translation, Rotation, Scale, and Skew Matrices
  - Concatenation of matrices
  - The model matrix
- Now let’s look at this for 3D
- First thing’s first: homogenous 3D coordinates
  - Add a 4^{th} value:

$$p = (x, y, z, 1)$$

Translation Matrix

- We can express translation using a 4x4 matrix T in homogenous coordinates $p' = Tp$ where

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

Rotation Matrix

- The 2D rotation we discussed is really just rotation about the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- We can then decompose a rotation by θ about an arbitrary axis to be a 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: The rotations do not commute!

Rotation about the x and y axes

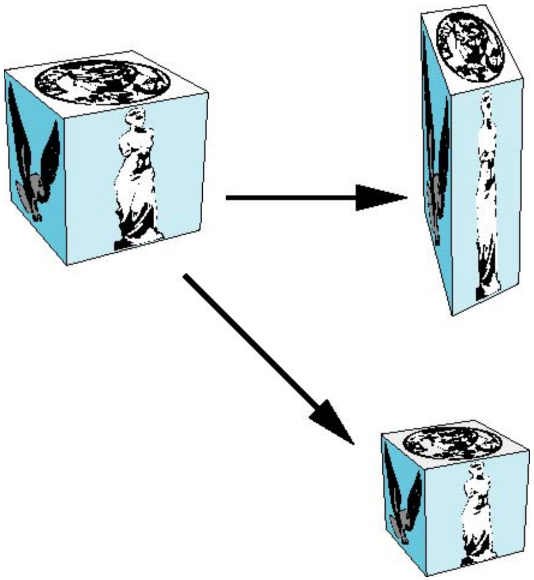
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

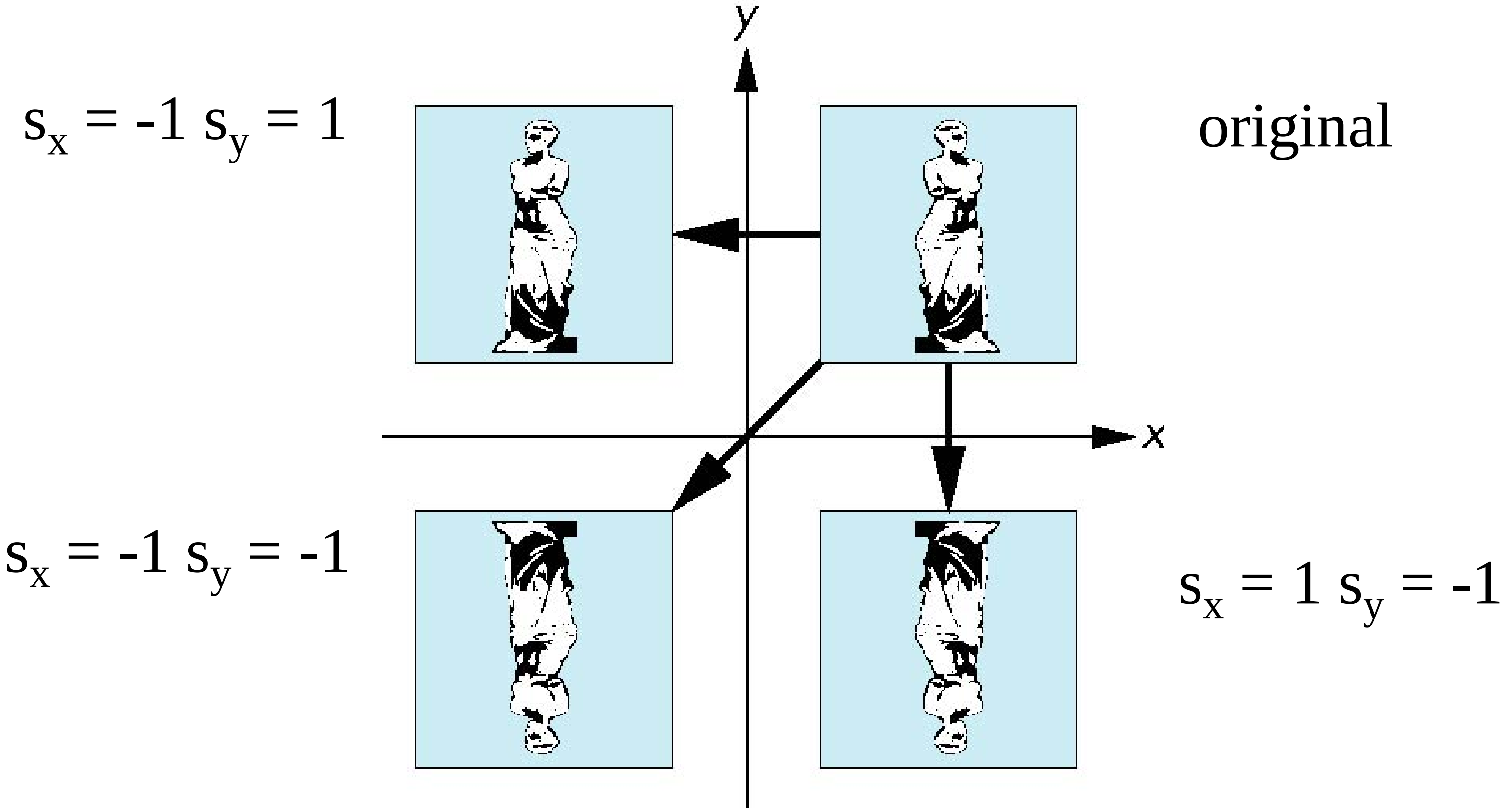
- In 3D we can now scale different amounts about different axis:

$$S = S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Reflection

- We can also do reflections easily.
- This corresponds to negative scale factors



Inverse

- We can use general matrix mathematics to compute inverses.
- But this can often be expensive and in some cases the inverses are straight forward
  - Translation: $\boldsymbol{T}^{-\mathbf{1}}(dx, d_y, d_z) = \boldsymbol{T}(-dx, -d_y, -d_z)$
  - Rotation: $\boldsymbol{R}^{-1}(\theta) = \boldsymbol{R}(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$, $\boldsymbol{R}^{-1}(\theta) = \boldsymbol{R}^{T}(\theta)$
  - Scaling: $\boldsymbol{S}^{-1}(s_x, s_y, s_z) = \boldsymbol{S}\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right)$

Transformations in OpenGL

- OpenGL gives us an easy way to generate transformation matrices using their built-in functions!
- Get a rotation matrix, theta **degrees** around the axis (vx, vy, vz):
  ```
  mat4 r = Rotate(theta,vx,vy,vz);
  ```
- Get a rotation matrix around x axis
  ```
  mat4 rx = RotateX(theta);
  ```
- Get a translation matrix
  ```
  mat4 t = Translate(dx,dy,dz);
  ```
- Get a scaling matrix
  ```
  mat4 s = Scale(sx,sy,sz);
  ```
- **REMEMBER!**
  - OpenGL/mat.h/vec.h give back matrices in column major instead of row major which GLSL uses.
  - So we need to transpose the matrices when we send them to the shaders.

Example

- Rotate about z-axis by 30 degrees with a fixed point of (1.0,2.0,3.0);
- ```
  mat4 m = Translate(1.0,2.0,3.0)*
      Rotate(30.0,0.0,0.0,1.0)*
      Translate(-1.0,-2.0,-3.0);
  ```
- Remember, the last matrix specified is the first applied
  ```

Example: Cube Rotation

- Just like in 2D we can do this either in the client application or in the shader program
- Lets let the user change the axis of rotation by clicking
    - Each time the user clicks the left button it changes the axis of rotation as $x \rightarrow y \rightarrow z \rightarrow x \ldots$
- Every 1/10 of a second rotate by 5 degrees

Example: Cube Rotation

- Lets create variable
  - `vec3 angle = vec3(0.0);`
- And initialize our axis of choice to be the x-axis
  - `GLubyte axis = 0;`
- Then each time we click the left button we change the axis of choice

```
void mouse(int button, int state, int x, int y){
    if(button == GLUT_LEFT_BUTTON
        && state==GLUT_DOWN)
        axis = (axis+1)%3;
}
```

All-Client Rotation

- Initialize our rotation matrix to the identity matrix
  - `mat4 rot = mat4(1.0);`
- In the timer function
  - Based on the current axis choice, generate a rotation
    - ```
      mat4 rot2;
      if(axis==0)
          rot2 = RotateX(5.0);
      //etc..
      ```
  - Update the rotation matrix
    - `rot = rot2*rot;`
  - Multiply all vertices by this
  - Send new vertices to GPU
      ```

GPU/Server Rotation

- Doing it that way is probably a waste of resources.
- Instead, let's compute the model matrix (to go from model→world coordinates) and pass that to vertex shader where it will apply it to each vertex.
- After all it's nice to do as much on the GPU as possible
  - Highly parallelize so can do per-vertex operations really quickly

Passing the Model Matrix

Client

- Initialize:

```
mat4 m = mat4(1.0);
```

- On timer

```
if(axis==0)
     m = RotateX(5.0)*m;
//etc..
```

- In Display

```
glUniformMatrix4fv(model_matrix,
     1, GL_TRUE,m);
```

Vertex Shader

```
in  vec4 vPosition;
in  vec4 vColor;
out vec4 color;
uniform mat4 model_matrix;
void main()
{
    gl_Position = model_matrix*vPosition;
    color = vColor;
}
```

TRANSPOSE!