TRANSFORMATIONS

Slides modified from Angel book 6e
Objectives

• Introduce standard transformations
  • Rotation
  • 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
- Why?
Pipeline Implementation

T (from application program)

vertices → transformation → T(u) → rasterizer → T(v) → frame buffer

• u
• v

vertices

pixels

T(u)

T(v)
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
- $u, v, w$: representations of vectors
  - array of 4 scalars in homogeneous coordinates
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
Translation Using Representations

Using the homogeneous coordinate representation in some frame

\[ p = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T \]
\[ p' = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix}^T \]
\[ d = \begin{bmatrix} dx & dy & dz & 0 \end{bmatrix}^T \]

Hence \( p' = p + d \) or

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

note that this expression is in four dimensions and expresses point = vector + point
Translation – 2D

Before Translation

Translation by (3, -4)

Translation

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

\[ P = \begin{bmatrix} x \\ y \end{bmatrix} \]
\[ P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \]
\[ T = \begin{bmatrix} d_x \\ d_y \end{bmatrix} \]
\[ P' = P + T \]

Homogeneous Form

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & d_x \\
  0 & 1 & d_y \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix}
\]
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.

What if we used a row vector instead of a column vector?
Translation – 3D

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

\[
\begin{bmatrix}
1 & 0 & 0 & d_x \\
0 & 1 & 0 & d_y \\
0 & 0 & 1 & d_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
= 
\begin{bmatrix}
x + d_x \\
y + d_y \\
z + d_z \\
1
\end{bmatrix}
\]

\[ T(d_x,d_y,d_z) \ast P = P' \]
Scaling – 2D

**Types of Scaling:**

- **Differential** \(( s_x \neq s_y )\)
- **Uniform** \(( s_x = s_y )\)

**Equations:**

\[
\begin{align*}
x' &= s_x \cdot x \\
y' &= s_y \cdot y
\end{align*}
\]

**Homogeneous Form:**

\[
\begin{bmatrix}
x' \\
y' \\
1
\end{bmatrix} =
\begin{bmatrix}
s_x & 0 & 0 \\
0 & s_x & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]
Scaling

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

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

\[
p' = Sp
\]

\[
S = S(s_x, s_y, s_z) = \\
\begin{bmatrix}
  s_x & 0 & 0 & 0 \\
  0 & s_y & 0 & 0 \\
  0 & 0 & s_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]
Scaling – 3D

Original

scale Y axis

scale all axes

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

$S(s_x, s_y, s_z) \cdot P = P'$

$$\begin{bmatrix}
s_x & 0 & 0 & 0 \\
0 & s_y & 0 & 0 \\
0 & 0 & s_z & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
1 \\
\end{bmatrix} = \begin{bmatrix}
x \cdot s_x \\
y \cdot s_y \\
z \cdot s_z \\
1 \\
\end{bmatrix}$$
Rotation (2D):

Consider rotation about the origin by $\theta$ degrees:

- radius stays the same, angle increases by $\theta$

\[
\begin{align*}
    x &= r \cos (\phi + \theta) \\
    y &= r \sin (\phi + \theta)
\end{align*}
\]
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 \( z \)
    \[
    \begin{align*}
    x' &= x \cos \theta - y \sin \theta \\
    y' &= x \sin \theta + y \cos \theta \\
    z' &= z
    \end{align*}
    \]
  - 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 – 3D

For 3D-Rotation 2 parameters are needed

✿ Angle of rotation
✿ Axis of rotation

Rotation about z-axis:

\[
R_{\theta,k} \cdot P = P'
\]

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} = \begin{bmatrix}
x \cdot \cos \theta - y \cdot \sin \theta \\
x \cdot \sin \theta + y \cdot \cos \theta \\
z \\
1
\end{bmatrix}
\]
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

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

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

\[
R_{\theta,j} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cos \theta + z \sin \theta \\ y \\ -x \sin \theta + z \cos \theta \\ 1 \end{bmatrix}
\]
Reflection

corresponds to negative scale factors

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

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

\[ s_x = 1 \quad s_y = -1 \]
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 $M = ABCD$ 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
  \[ p' = ABCp = A(B(Cp)) \]
• From an efficiency standpoint, how many operations are done with \((ABC)p\) vs \(A(B(Cp))\)?
  • Is there a situation where this would differ?
• Note many references use column matrices to represent points. In terms of column matrices
  \[ p'^T = p^T C^T B^T A^T \]
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 About a Fixed Point other than the Origin

Move fixed point to origin
Rotate
Move fixed point back

\[ M = T(p_f) \ R(\theta) \ T(-p_f) \]
Rotation of $\theta$ about $P(h,k)$: $R_{\theta,P}$

**Step 1:** Translate $P(h,k)$ to origin

**Step 2:** Rotate $\theta$ w.r.t to origin

**Step 3:** Translate $(0,0)$ to $P(h,k)$

$$R_{\theta,P} = T(h,k) \ast R_{\theta} \ast T(-h,-k)$$
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

\[
M = TRS
\]
Shear

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

Consider simple shear along $x$ axis

$$x' = x + y \cot \theta$$
$$y' = y$$
$$z' = z$$

$$H(\theta) = \begin{bmatrix} 1 & \cot \theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
OPENGL
TRANSFORMATIONS
Objectives

- Learn how to carry out transformations in OpenGL
  - Rotation
  - Translation
  - Scaling
- Introduce mat.h and vec.h transformations
  - Model-view
  - Projection
The OpenGL Pipeline

Modelview Transform → Lighting → Perspective Transform → Rasterization → Texturing
The OpenGL Pipeline

Vertices with:
  - Colors
  - Texture coords
  - Normals

Modelview Transform

Lighting

Perspective Transform

Rasterization

Texturing
The OpenGL Pipeline

Vertices in eye coords
with:
  Colors
  Texture coords
  Normals in eye coords

Modelview Transform

Lighting

Perspective Transform

Rasterization

Texturing
The OpenGL Pipeline

Vertices in eye coords
with:
  Lit colors
  Texture coords

Modelview Transform

Lighting

Perspective Transform

Rasterization

Texturing
The OpenGL Pipeline

1. Vertices in clip coords with:
   - Lit colors
   - Texture coords
2. Modelview Transform
3. Lighting
4. Perspective Transform
5. Rasterization
6. Texturing
The OpenGL Pipeline

- Modelview Transform
- Lighting
- Perspective Transform
- Rasterization
- Texturing

Fragments with:
- Interpolated colors
- Interpolated texture coords
The OpenGL Pipeline

Modelview Transform

Lighting

Perspective Transform

Rasterize + Interpolate

Texturing

Fragments with:
  Colors

Texture Memory
The OpenGL Pipeline with Shaders

- Vertex Shader
  - Rasterize + Interpolate
  - Fragment Shader
The OpenGL Pipeline with Shaders

Vertex Shader

Small programs that run on the graphics card

Rasterize + Interpolate

Fragment Shader
The OpenGL Pipeline with Shaders

Vertices with:
- Colors
- Texture coords
- Normals

Vertex Shader

Rasterize + Interpolate

Fragment Shader
The OpenGL Pipeline with Shaders

- **Vertex Shader**
  - Transformed Vertices with:
    - (Anything you want here, e.g., normals, colors, texture coords)
  - Rasterize + Interpolate
  - Texture Memory

- **Fragment Shader**
The OpenGL Pipeline with Shaders

Vertex Shader

Rasterize + Interpolate

Fragments with:
(Interpolated values from previous stage)

Fragment Shader
The OpenGL Pipeline with Shaders

- **Vertex Shader**: Rasterize + Interpolate
- **Fragment Shader**: Fragments with: Colors, Texture, Memory
Viewport
\texttt{glViewport(x1,x2,1,x2-x1,y2-y1)};

Clipping Volume
\texttt{glOrtho(x1,x2,y1,y2,z1,z2)};

Viewports Origin

Window

Global Coordinates
Pre 3.1 OpenGL Matrices

- In OpenGL matrices were part of the state
- Multiple types
  - Model-View (GL_MODELVIEW)
  - Projection (GL_PROJECTION)
  - Texture (GL_TEXTURE)
  - Color(GL_COLOR)
- 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.

\[ p' = Cp \]
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: \( C \leftarrow CT \)
Rotate: \( C \leftarrow CR \)
Move fixed point back: \( C \leftarrow CT^{-1} \)

Result: \( C = TR T^{-1} \) which is **backwards**.

This result is a consequence of doing postmultiplications.
Let’s try again.
Reversing the Order

We want \( C = T^{-1} R T \)
so we must do the operations in the following order

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

Each operation corresponds to one function call in the program.

Note that the last operation specified is the first executed in the program.
CTM in old OpenGL

- OpenGL had 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 correct matrix mode
Create an identity matrix:

\[
\text{mat4 } m = \text{Identity}();
\]

Multiply on right by rotation matrix of theta in degrees where (vx, vy, vz) define axis of rotation

\[
\text{mat4 } r = \text{Rotate}(\text{theta, vx, vy, vz})
\]
\[
m = m*r;
\]

Do same with translation and scaling:

\[
\text{mat4 } s = \text{Scale}(sx, sy, sz)
\]
\[
\text{mat4 } t = \text{Transalate}(dx, dy, dz);
\]
\[
m = m*s*t;
\]
Example

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

```c
mat 4 m = Identity();
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 that last matrix specified in the program is the first applied
Arbitrary Matrices

- Can load and multiply by matrices defined in the application program
- Matrices are stored as one dimensional array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns
- OpenGL functions that have matrices as parameters allow the application to send the matrix or its transpose
Matrix Stacks

• In many situations we want to save transformation matrices for use later
  • Traversing hierarchical data structures (Chapter 8)
  • Avoiding state changes when executing display lists
• Pre 3.1 OpenGL maintained stacks for each type of matrix
• Straightforward to create the same functionality with a simple stack class
Reading Back State

- Can also access OpenGL variables (and other parts of the state) by *query* functions

  ```
  glGetIntegerv
  glGetFloatv
  glGetBooleanv
  glGetDoublev
  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
main.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();
}
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

We can form matrix in application and send to shader and let shader do the rotation or we can send the angle and axis to the shader and let the shader form the transformation matrix and then do the rotation

More efficient than transforming data in application and resending the data

```c
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glUniform(...); // or glUniformMatrix
    glDrawArrays(...);
    glutSwapBuffers();
}
```
Display (transforms in CPU)

```c
void display( void )
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ctm = RotateX(theta[0])*RotateY(theta[1])*RotateZ(theta[2]);
    points[i++] = ctm*vertices[a];
    points[i++] = ctm*vertices[b];

    glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );

    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glutSwapBuffers();
}
```
Display (transforms in GPU)

```c
GLuint ctm_loc = glGetUniformLocation(program, "CTM");
void display( void ) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ctm = RotateX(theta[0]) * RotateY(theta[1]) * RotateZ(theta[2]);
    glUniformMatrix4fv(ctm_loc, 1, GL_TRUE, ctm);
    glDrawArrays(GL_TRIANGLES, 0, N);
    glutSwapBuffers();
}

in vec4 vPosition;
uniform mat4 CTM;
void main() {
    gl_Position = CTM * vPosition;
}
```
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 the \texttt{LookAt} function
  - Build models of objects

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

- Although these matrices are no longer part of the OpenGL state, it is usually a good strategy to create them in our own applications
uniform Transformation {
    mat4 projection_matrix;
    mat4 modelview_matrix;
};

in vec3 vertex;

void main() {
    gl_Position = projection_matrix * modelview_matrix * vec4(vertex, 1.0);
}
Another Example – 3 Matrices

in vec4 vPosition;
in vec4 vColor;
out vec4 oColor;
uniform mat4 ModelMat;
uniform mat4 ViewMat;
uniform mat4 ProjectionMat;
void main(void) {
    gl_Position = (ProjectionMat*ViewMat*ModelMat)*vPosition;
oColor = inColor;
}
- Create an identity matrix:

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

- Multiply on right by rotation matrix of `theta` in degrees
- where `(vx, vy, vz)` define axis of rotation

```cpp
mat4 r = Rotate(theta, vx, vy, vz);
mat4 m = m*r;
```

- Do same with translation and scaling:

```cpp
mat4 s = Scale(sx, sy, sz)
mat4 t = Translate(dx, dy, dz);
mat4 m = m*s*t;
```
Using Angel Headers

```cpp
model = glGetUniformLocation( program, "ModelMat" );
view = glGetUniformLocation( program, "ViewMat" );
proj = glGetUniformLocation( program, "ProjectionMat" );

mat4 m = Identity(); // Display or update function
... // Transform to world.
Mat4 v = LookAt(eye, at, up); // Camera may not move
Mat4 p = Perspective(fovy, aspect, zNear, zFar); // often?

// In Display()
glUniformMatrix4fv(model, 1, GL_TRUE, m);
glUniformMatrix4fv(view, 1, GL_TRUE, v);
glUniformMatrix4fv(proj, 1, GL_TRUE, p);
glDrawArrays();
```
GLM

- Potential library to use
- Probably better than the functions from Angel.h/mat.h/vec.h
- Certinally appears to be very powerful
- C++
- Header only
- A little bit harder to use
- Seems to have good functionality
- At least one person had issues with it crashing their OS.
  - Seems to work on my laptop.
An Example with GLM

```cpp
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
{
    glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.f);
    glm::mat4 ViewTranslate = glm::translate( glm::mat4(1.0f),
        glm::vec3(0.0f, 0.0f, -Translate));
    glm::mat4 ViewRotateX = glm::rotate( ViewTranslate, Rotate.y,
        glm::vec3(-1.0f, 0.0f, 0.0f));
    glm::mat4 View = glm::rotate( ViewRotateX, Rotate.x,
        glm::vec3(0.0f, 1.0f, 0.0f));
    glm::mat4 Model = glm::scale( glm::mat4(1.0f), glm::vec3(0.5f));
    glm::mat4 MVP = Projection * View * Model;
    glUniformMatrix4fv(LocationMVP, 1, GL_FALSE, glm::value_ptr(MVP));
}
```

Should Transpose?
Another GLM example: set it up

// Projection matrix : 45° Field of View, 4:3 ratio, display range : 0.1 unit <-> 100 units
glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);

// Camera matrix
glm::mat4 View = glm::lookAt(
    glm::vec3(4,3,3), // Camera is at (4,3,3), in World Space
    glm::vec3(0,0,0), // and looks at the origin
    glm::vec3(0,1,0)  // Head is up (set to 0,-1,0 to look upside-down)
);

// Model matrix : an identity matrix (model will be at the origin)
glm::mat4 Model = glm::mat4(1.0f); // Changes for each model !

// Our ModelViewProjection : multiplication of our 3 matrices
glm::mat4 MVP = Projection * View * Model;

// Remember, matrix multiplication is the other way around

• Tutorial 3 from opengl-tutorial.org
Pass to GLSL

// Get a handle for our "MVP" uniform.
// Only at initialisation time.
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

// Send our transformation to the currently bound shader,
// in the "MVP" uniform
// For each model you render, since the MVP will be different
// (at least the M part)
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
Use in shader

in vec3 vertexPosition_modelspace;
uniform mat4 MVP;

void main(){

    // Output position of the vertex, in clip space : MVP * position
    vec4 v = vec4(vertexPosition_modelspace,1);

    // Transform an homogeneous 4D vector, remember ?
    gl_Position = MVP * v;
}

What About GLSL?

uniform float timer;
attribute vec4 position;
varying vec2 texcoord;
varying float fade_factor;
void main() {
    mat3 rotation = mat3(
        vec3( cos(timer), sin(timer), 0.0),
        vec3(-sin(timer), cos(timer), 0.0),
        vec3( 0.0, 0.0, 1.0 )
    );
    gl_Position = vec4(rotation * position.xyz, 1.0);
}
Smooth Rotation

• From a practical standpoint, we are often want to use transformations to move and reorient an object smoothly
  • Problem: find a sequence of model-view matrices $M_0, M_1, \ldots, M_n$ so that when they are applied successively to one or more objects we see a smooth transition

• For orientating an object, we can use the fact that every rotation corresponds to part of a great circle on a sphere
  • Find the axis of rotation and angle
  • Virtual trackball (see text)
Incremental Rotation

- Consider the two approaches
  - For a sequence of rotation matrices $R_0, R_1, \ldots, R_n$, find the Euler angles for each and use $R_i = R_{iz} R_{iy} R_{ix}$
  - Not very efficient
  - Use the final positions to determine the axis and angle of rotation, then increment only the angle
- Quaternions can be more efficient than either
Quaternions

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

\[ q = q_0 + q_1i + q_2j + q_3k \]

- 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
Interfaces

• One of the major problems in interactive computer graphics is how to use two-dimensional devices such as a mouse to interface with three-dimensional objects.

• Example: how to form an instance matrix?

• Some alternatives
  • Virtual trackball
  • 3D input devices such as the spaceball
  • Use areas of the screen
    • Distance from center controls angle, position, scale depending on mouse button depressed