COSE436

Lecture 6: OpenGL Transformation

Sep 24, 2024

Won-Ki Jeong

(wkjeong@korea.ac.kr)

KOREA UNIVERSITY

Outline

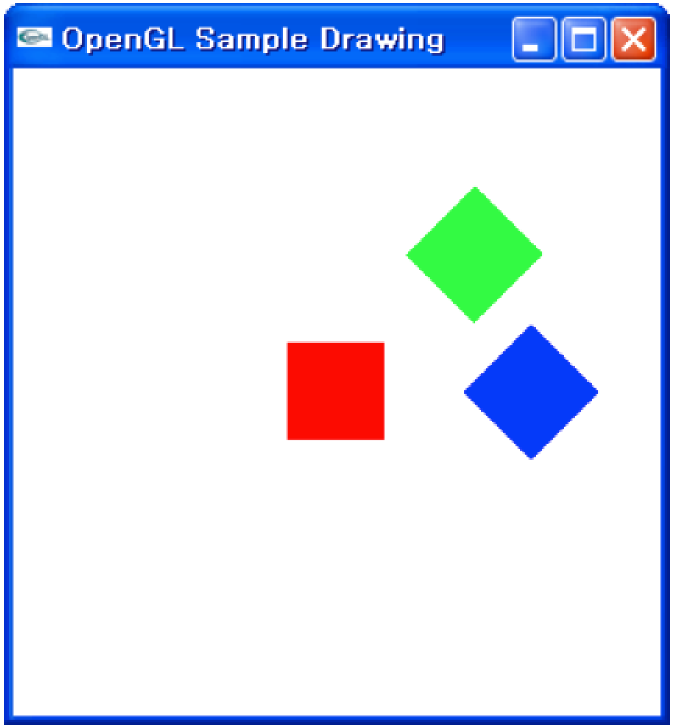
- OpenGL transformation
- Virtual trackball

Outline

- OpenGL transformation
- Virtual trackball

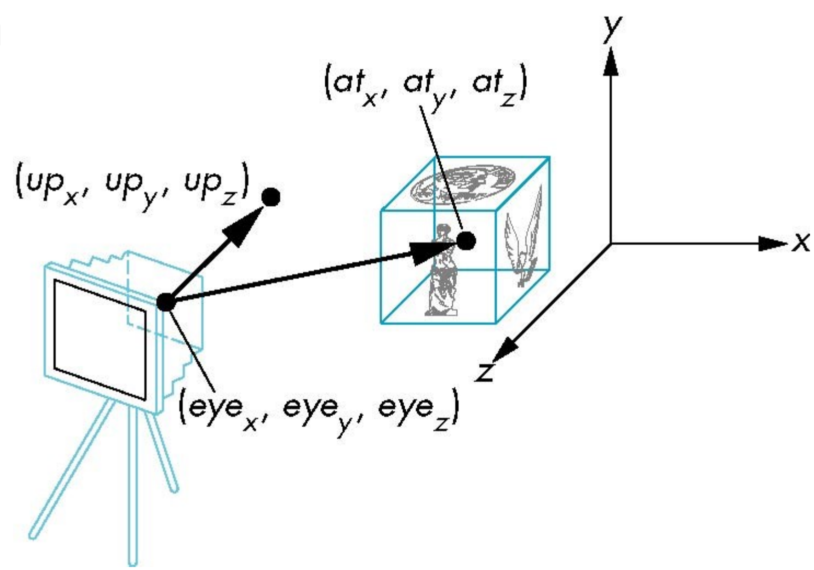
Modeling Transformation

- Local to global coordinate transform
- Rotate, Translate, Scale



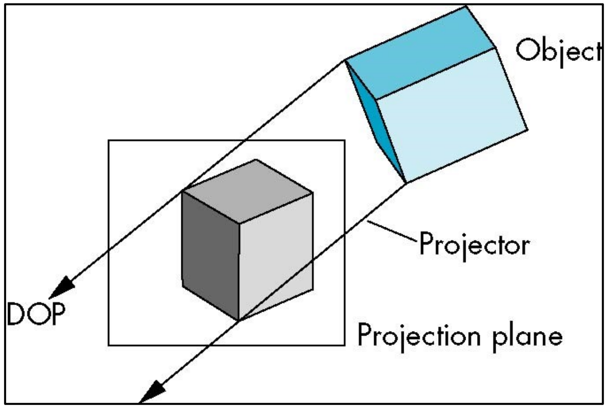
Viewing Transformation

- Global to 3D viewing coordinate transform
- Set eye position and viewing direction
- LookAt(eyex/y/z, atx/y/z, upx/y/z)
  - eye x/y/z : eye position (x,y,z)
  - at x/y/z : viewing direction
  - up x/y/z : up vector

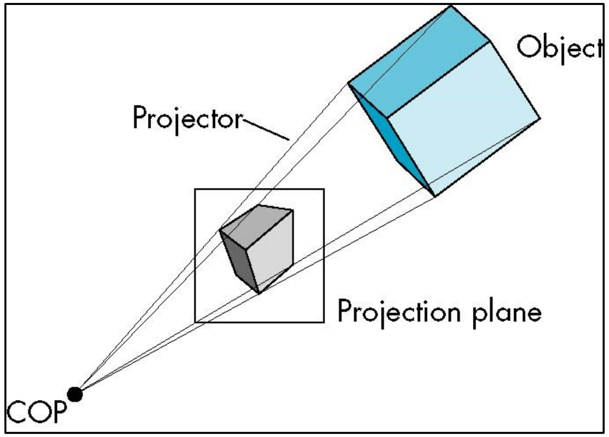


Projection Transformation

- 3D to 2D viewing coordinate transform
- Define clipping volume
- Projection types
  - Orthogonal
  - Perspective



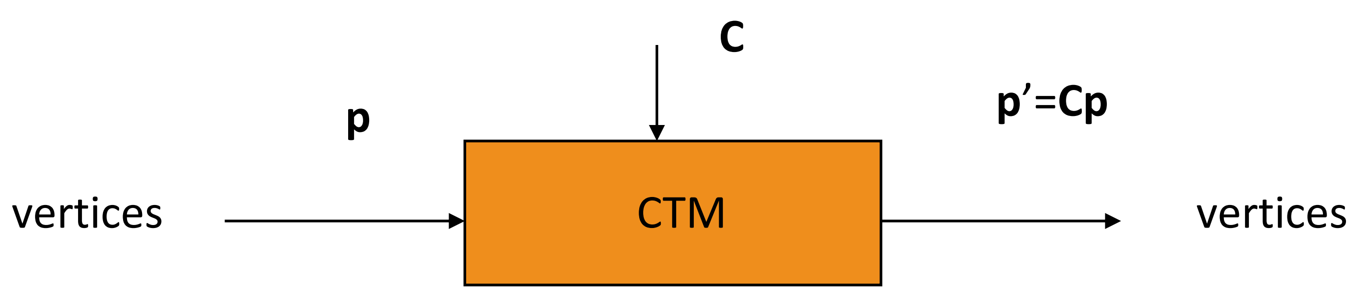
Orthogonal projection



Perspective projection

Current Transformation Matrix

- 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 Operations

Load an identity matrix: **C** = **I**
Load an arbitrary matrix: **C** = **M**

Load a translation matrix: **C** = **T**
Load a rotation matrix: **C** = **R**
Load a scaling matrix: **C** = **S**

Postmultiply by an arbitrary matrix: **C** = **CM**
Postmultiply by a translation matrix: **C** = **CT**
Postmultiply by a rotation matrix: **C** = **C R**
Postmultiply by a scaling matrix: **C** = **C S**

Matrix Order is Reversed

- Example: Rotation about a fixed point
  - Start with identity matrix: $\mathbf{C} = \mathbf{I}$
  - Move fixed point to origin: $\mathbf{C} = \mathbf{CT}$
  - Rotate: $\mathbf{C} = \mathbf{CR}$
  - Move fixed point back: $\mathbf{C} = \mathbf{CT}^{-1}$
  - Result: $\mathbf{C} = \mathbf{TR}\,\mathbf{T}^{-1}$ which is **backwards!**

Correct Matrix Order

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

$\mathbf{C} = \mathbf{I}$
$\mathbf{C} = \mathbf{C}\mathbf{T}^{-1}$
$\mathbf{C} = \mathbf{C}\mathbf{R}$
$\mathbf{C} = \mathbf{C}\mathbf{T}$

Each operation corresponds to one function call in the program

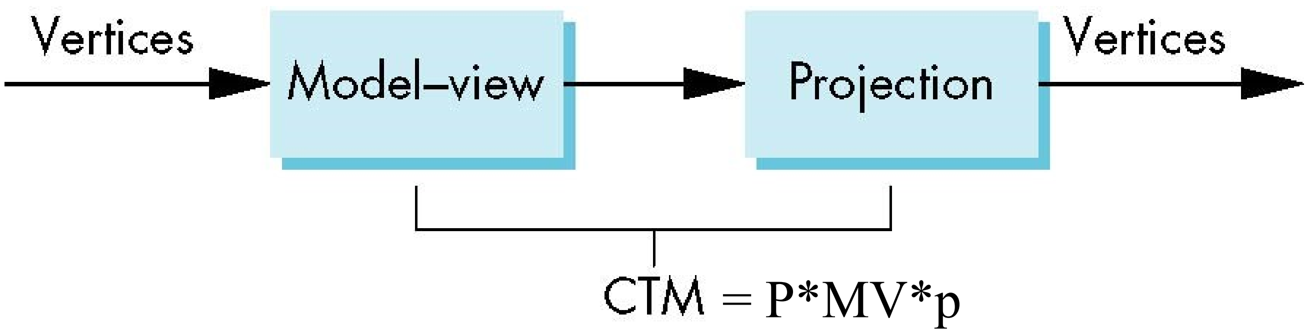
Note that the last operation specified is the first executed in the program

CTM in OpenGL with Shader

- The CTM in OpenGL is defined in the user program and passed down to the vertex processor
  - Vertex shader will multiply the matrix to vertex coordinates
- Programmers should create and manage them in their own applications
- Matrix / vector class can help
  - mat.h, vec.h (textbook source code)
    - Error corrected version of vector class (vec_fixed.h) is included in assign_1 skeleton code

ModelView/Projection 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 lookAt function in mat.h
  - Build models of objects
- The projection matrix is used to define the view volume and to select a camera lens



Basic Matrix Functions (mat.h)

- Create an identity matrix

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

- Fill it with components

```
mat4 m = mat4(0,1,2,3,4,5,6,7,
              8,9,10,11,12,13,14,15);
```

- By vectors

```
mat4 m = mat4( vec4(0,1,2,3),
               vec4(4,5,6,7),
               vec4(8,9,10,11),
               vec4(12,13,14,15) );
```

Rotation, Translation, Scaling

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

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

- Also have rotateX, rotateY, rotateZ.
- Do same with translation and scaling:

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

Rotation about A Fixed Point using mat.h

- Fixed point: (4, 5, 6)
- Rotation angle: 45 degrees
- Rotation axis: the line through the origin and the point (1, 2, 3)
- Remember that last matrix specified in the program is the first applied

```
mat4 m = Identity();
m = Translate(4.0, 5.0, 6.0)*
  Rotate(45.0, 1.0, 2.0, 3.0)*
  Translate(-4.0, -5.0, -6.0);
```

How to pass matrix to shader?

- GL Shader

```
#version 150
in  vec4 vPosition;
in  vec4 vColor;
out vec4 color;
uniform mat4 model_view;
uniform mat4 projection;
void main()
{
    gl_Position = projection*model_view*vPosition;
    color = vColor;
}
```

How to pass matrix to shader?

- User code (C++)

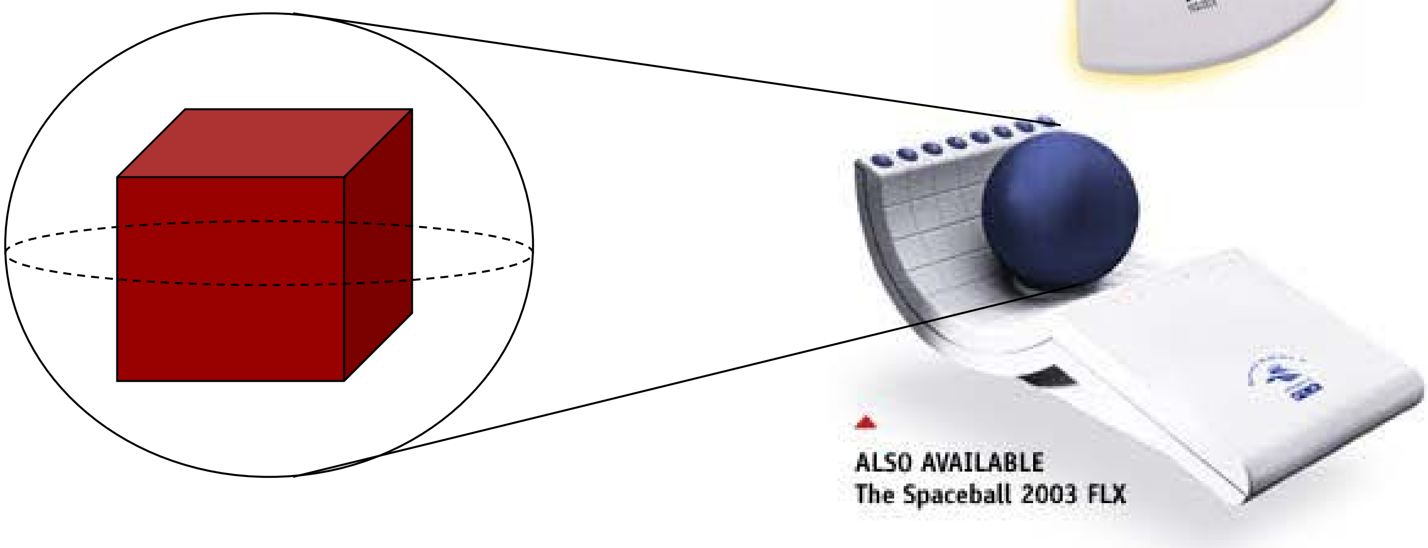
```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    point4  eye( 0, 0, -100,  1.0 );
    point4  at( 0.0, 0.0, 0.0, 1.0 );
    vec4    up( 0.0, 1.0, 0.0, 0.0 );
    mat4  mv = LookAt( eye, at, up );
    glUniformMatrix4fv( model_view, 1, GL_TRUE, mv );
    mat4  p = Perspective( fovy, aspect, zNear, zFar );
    glUniformMatrix4fv( projection, 1, GL_TRUE, p );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}
```

Outline

- OpenGL transformation
- Virtual trackball

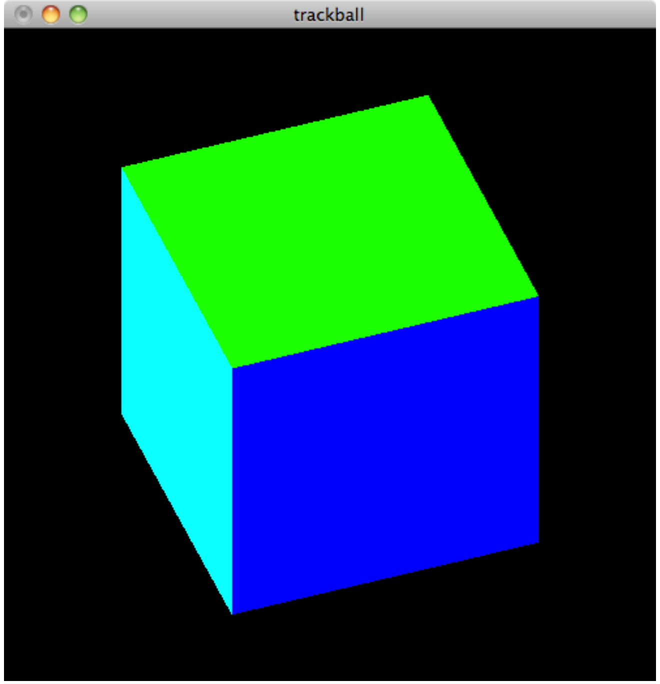
3D Rotations with Trackball

- Imagine the objects are rotated along with a imaginary hemi-sphere



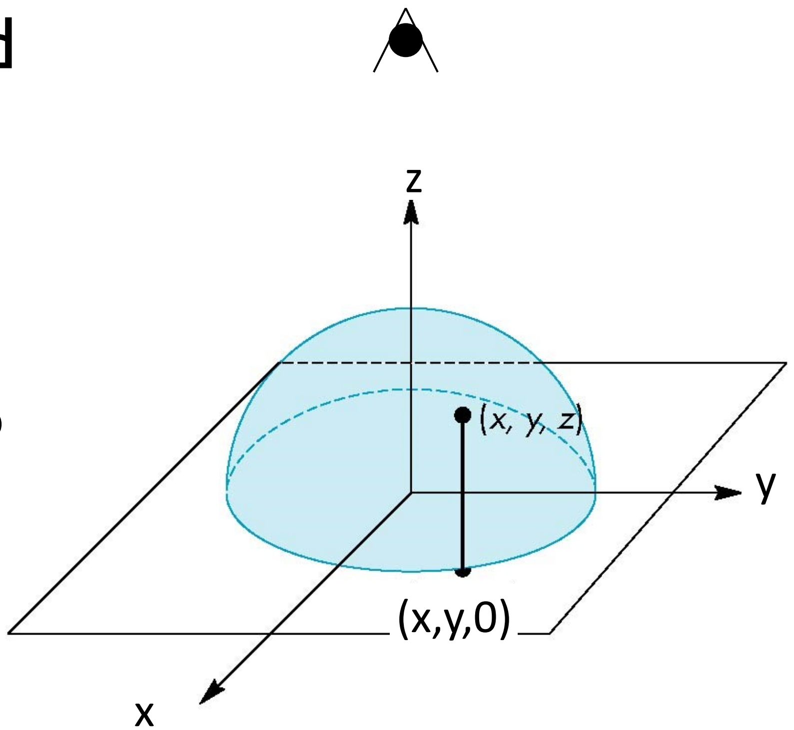
Virtual Trackball

- Allow the user to define 3D *rotation* using mouse click in 2D windows
- Work similarly like the hardware trackball devices



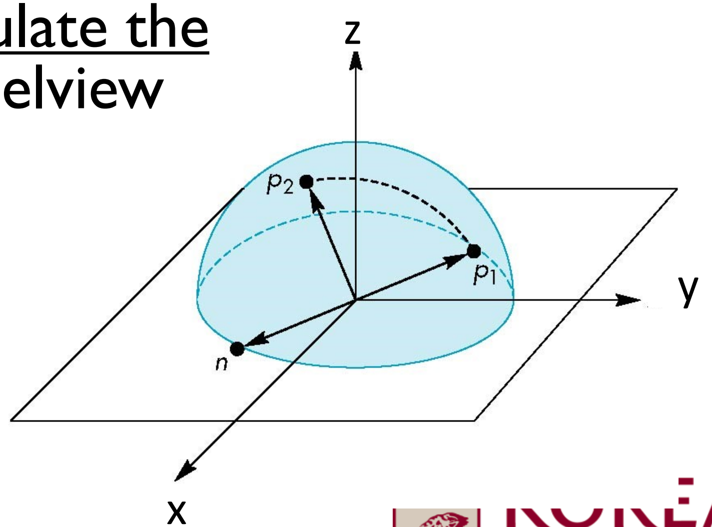
Virtual Trackball

- Superimpose a hemi-sphere onto the viewport
- This hemi-sphere is projected to a circle inscribed to the viewport
- The mouse position is projected orthographically to this hemi-sphere



Virtual Trackball

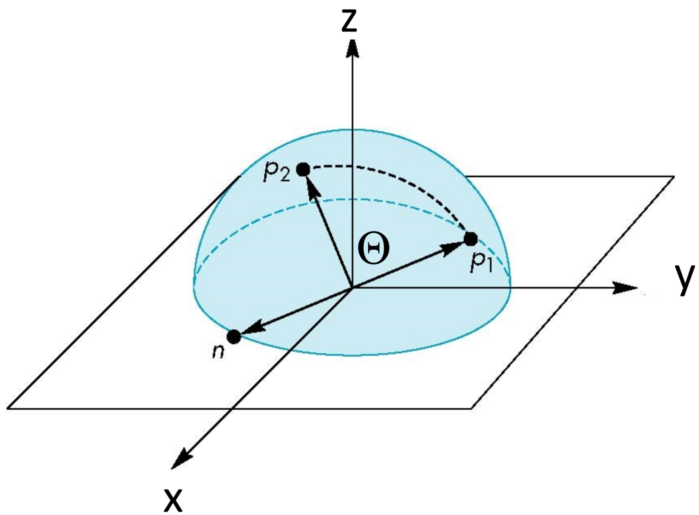
- Keep track the previous mouse position and the current position
- Calculate their projection positions p1 and p2 to the virtual hemi-sphere
- We then rotate the sphere from p1 to p2 by finding the proper rotation axis and angle
- You should also remember to accumulate the current rotation to the previous modelview matrix



Virtual Trackball

- The axis of rotation is given by the normal to the plane determined by the origin, **p**1 , and **p**2

$$\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_1$$

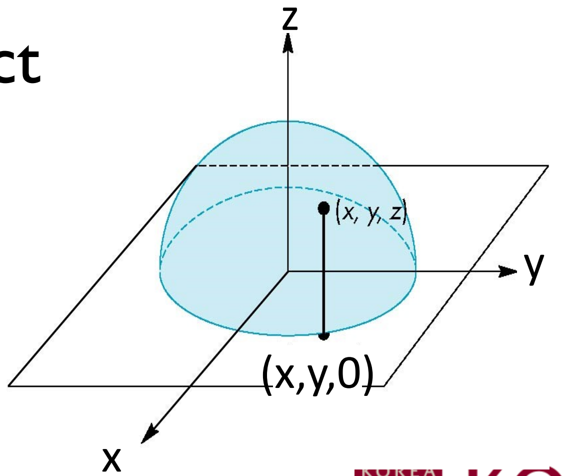


Virtual Trackball

- How to calculate p1 and p2?
- Assuming the mouse position is (x,y), then the sphere point P also has x and y coordinates equal to x and y
- Assume the radius of the hemi-sphere is *r*. Then the z coordinate of P is

$$\sqrt{r^2 - x^2 - y^2}$$

- If a point is outside the circle, project it to the nearest point on the circle (set z to 0 and renormalize (x,y))



glut Callback Functions

- glutMouseFunc(mouseButton)
  - Mouse click (UP/DOWN)
- glutMotionFunc(mouseMotion)
  - Mouse move
  - You need to count only when mouse is moving while mouse button is pressed

Mouse Callback Example

```
void mouseButton(int button, int state, int x, int y)
{
 if(button==GLUT_LEFT_BUTTON) switch(state)
 {
    case GLUT_DOWN:
        startMotion(x,y);
        break;
    case GLUT_UP:
        stopMotion(x,y);
        break;
  }
 }
```

glutMotionFunc Example

```
Void mouseMotion(int x, int y)
{
      float curPos[3],
      dx, dy, dz;

      /* compute position on hemisphere */
      trackball_ptov(x, y, winWidth, winHeight, curPos);

      if(trackingMouse)
      {
         /* compute the change in position
              on the hemisphere */
         dx = curPos[0] - lastPos[0];
         dy = curPos[1] - lastPos[1];
         dz = curPos[2] - lastPos[2];
```

⋮

glutMotionFunc Example

```
void trackball_ptov(int x, int y, int width, int height,
 float v[3])
{
  float d, a;

  /* project x,y onto a hemi-sphere centered within width,
 height */
  v[0] = (2.0F*x - width) / width;
  v[1] = (height - 2.0F*y) / height;
  d = (float) sqrt(v[0]*v[0] + v[1]*v[1]);
  v[2] = (float) cos((M_PI/2.0F) * ((d < 1.0F) ? d : 1.0F));
  a = 1.0F / (float) sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
  v[0] *= a;
  v[1] *= a;
  v[2] *= a;
}
```

glutMotionFunc Example

⋮

```
    if (dx || dy || dz)
    {
      /* compute theta and cross product */
      angle = 90.0 * sqrt(dx*dx + dy*dy + dz*dz);
      axis[0] = lastPos[1]*curPos[2] –
           lastPos[2]*curPos[1];
      axis[1] = lastPos[2]*curPos[0] –
           lastPos[0]*curPos[2];
      axis[2] = lastPos[0]*curPos[1] –
           lastPos[1]*curPos[0];
      /* update position */
      lastPos[0] = curPos[0];
      lastPos[1] = curPos[1];
      lastPos[2] = curPos[2];
      }
    }
    glutPostRedisplay();
}
```

Update Rotation Matrix

- Order is important!
  - rot * cmt for right-side multiplication

```
mat4 cmt; // current matrix

...

mat4 rot = Rotate(alpha, vx, vy, vz));

cmt = rot*cmt;
```

Notes about Glew

glew

- glewInit() must be called before any OpenGL command and after glutDisplayFunc()

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv)
    glutInitDisplayMode(...)
    ...
    glutCreateWindow(..)

    glewInit();

    glutMainLoop();
    return 1;
}
```

glew

- glew.h must be included before glut.h

```
#include <GL/glew.h>

#include <GL/glut.h>

....
```

glutPostRedisplay()

- Update framebuffer
  - Call display callback function registered by `glutDisplayFunc()`
- Do not call `glutPostRedisplay()` inside display callback function!
  - Infinite loop
- Call when you need to manually refresh screen
  - After mouse or keyboard events

glutSwapBuffers()

- Swap back and front buffers
- Render target must be back buffer
  - `glDrawBuffer(GL_BACK)`
- Call only once per each render pass
  - Only at the end of the display callback function

Questions?

Refraction Effect using Vertex Shader (Wikipedia)