Programming with OpenGL: PART II

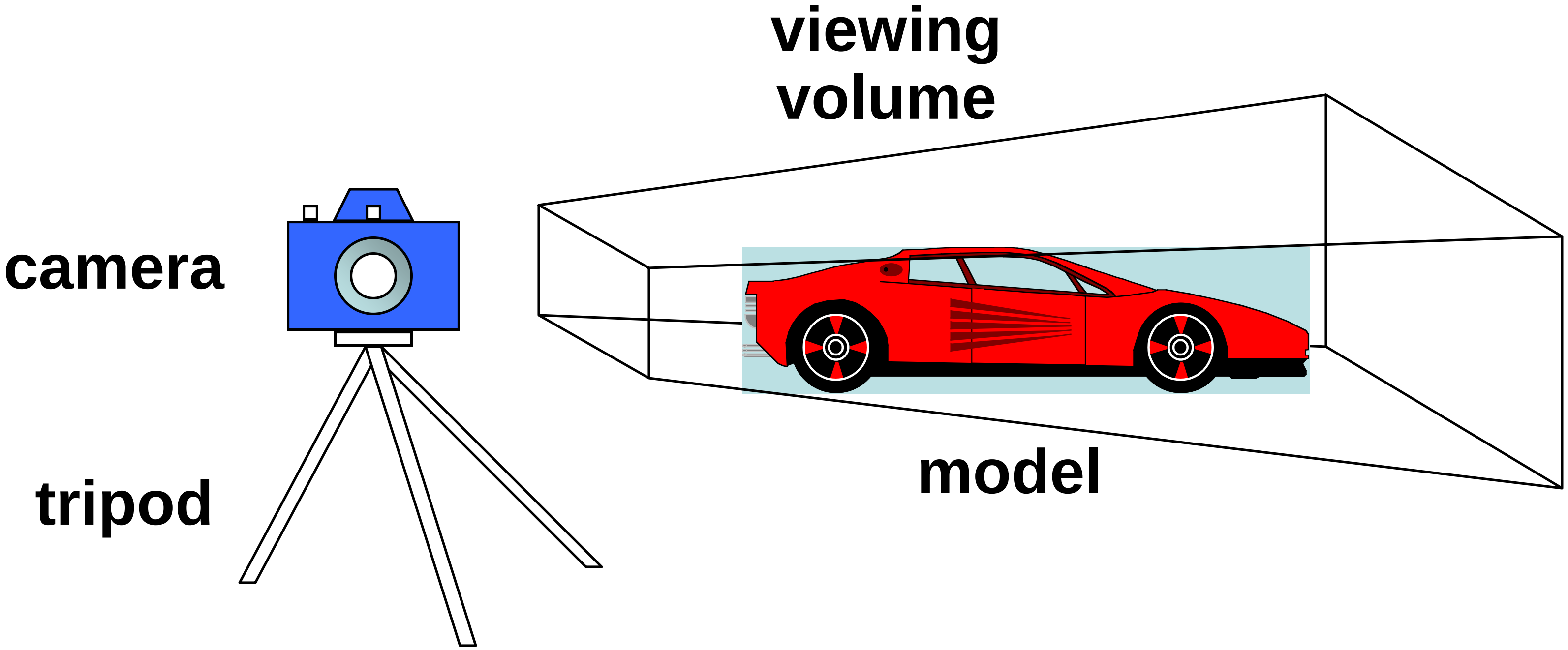
Serena Morigi

A.A.2018/2019

Camera Analogy

To display a **3D world onto a 2D screen**

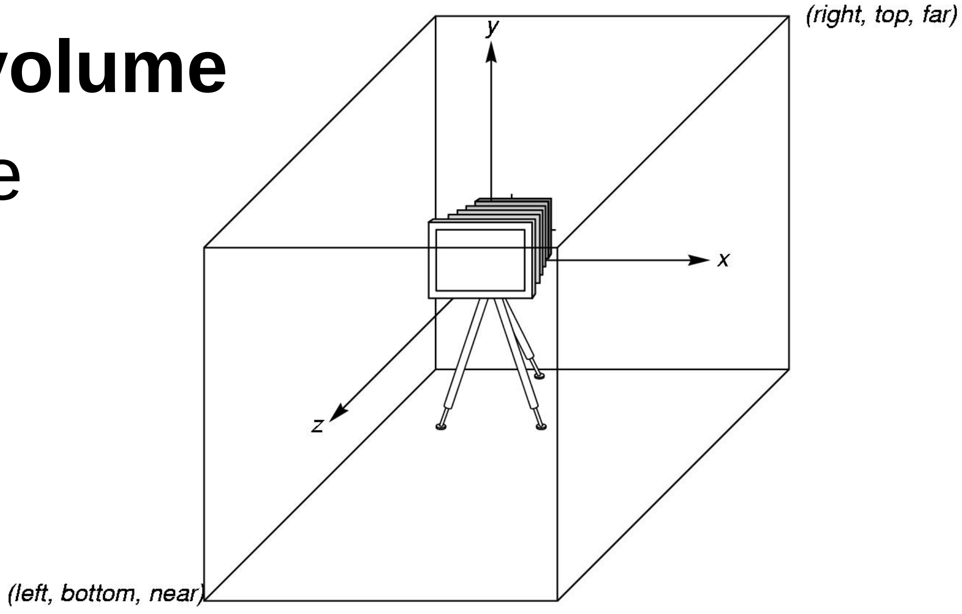
3D is just like taking a photograph (lots of photographs!)



OpenGL Camera

- OpenGL places a camera at the origin in object space pointing in the negative z direction

- The default **viewing volume** is a box centered at the origin with sides of length 2



Transformations and camera analogy

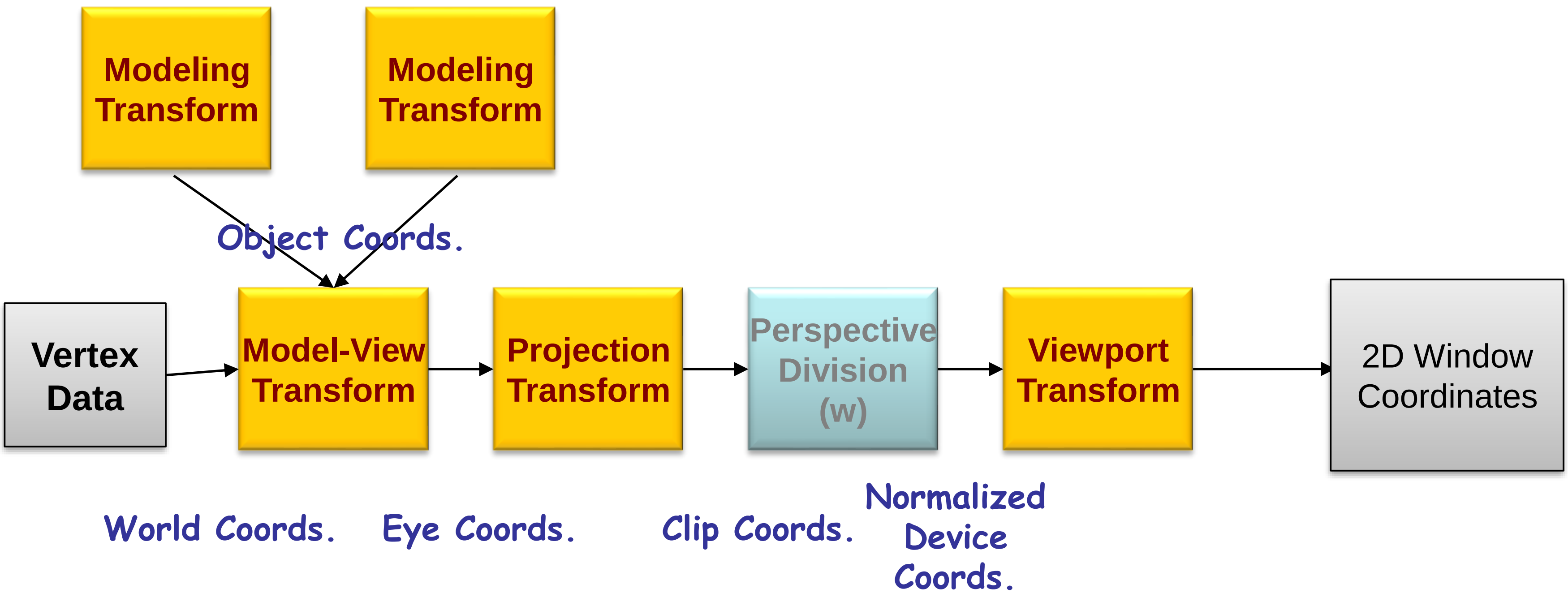
- Modeling transformations
  - moving the model
- Viewing transformations
  - camera–define position and orientation of the viewing volume in the world
- Projection transformations
  - adjust the lens of the camera
- Viewport transformations
  - enlarge or reduce the physical photograph

Moving camera is equivalent to moving every object in the world towards a stationary camera

Transfomations in OPENGL

Transformations take us from one “space” to another, used to realize the 3D scene and the rendering pipeline

We use transformations represented by 4×4 matrices



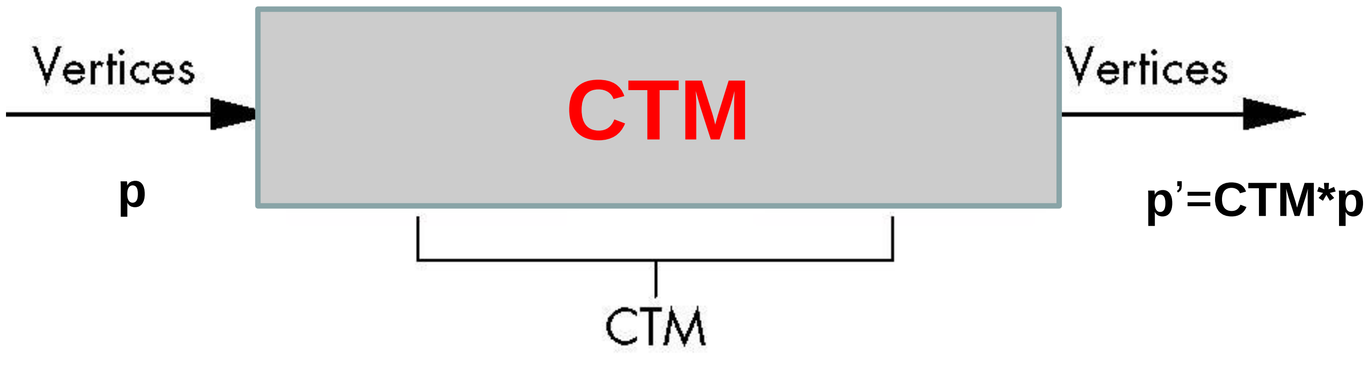
Transfomations in OPENGL

- In OpenGL, the transformation matrices are part of the state, they must be defined prior to any vertices to which they are to apply.
- Multiple types
  - Model-View matrix (**GL_MODELVIEW**)
  - Projection matrix (**GL_PROJECTION**)
  - Texture matrix (**GL_TEXTURE**)

- **OpenGL provides matrix stacks for each type of supported matrix** (modelview, projection, texture) to store matrices

Current Transformation Matrix (CTM)

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated (multiplied) together to form the CTM
- CTM is a column-major 4 x 4 homogeneous coordinate matrix, which is applied to all vertices that pass down the pipeline.



Transformations using CTM operations

- The CTM can be altered either (C is the CTM)
  - by initializing the current matrix stack

    Load an identity matrix: $\mathbf{C} \leftarrow \mathbf{I}$

  - by specifying the transformation

    Load a translation matrix: $\mathbf{C} \leftarrow \mathbf{T}$
    Load a rotation matrix: $\mathbf{C} \leftarrow \mathbf{R}$
    Load a scaling matrix: $\mathbf{C} \leftarrow \mathbf{S}$

  - by loading a new 4x4 matrix

    Load an arbitrary matrix: $\mathbf{C} \leftarrow \mathbf{M}$

Post-multiplication

Postmultiply by an arbitrary matrix M: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{M}$
Postmultiply by a translation matrix T: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{T}$
Postmultiply by a rotation matrix R: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{R}$
Postmultiply by a scaling matrix S: $\mathbf{C} \leftarrow \mathbf{C}\,\mathbf{S}$

Specify the current matrix stack:

- **glMatrixMode**(GL_MODELVIEW/ GL_PROJECTION)

Specify the transformation:

- **glLoadIdentity()** (C=I) replaces the matrix on the top of the current matrix stack with the identity matrix

- **glRotate{fd}(angle, x, y, z),**
- **glTranslate{fd}(x, y, z),**
- **glScale{fd}(x, y, z)**

(C=C * T)

(Each has a float (f) and double (d) format)

Post-multiply the CTM with a transformation R,T,S;

- glPushMatrix() glPopMatrix()

Apply transformations between push and pop then reload the previous CTM

- glLoadMatrix(pointer_to_matrix**)** load a matrix

- glMultMatrix(pointer_to_matrix**)** post-multiply the matrix on the top of the current matrix stack (Arbitrary 4× 4 matrices specified by a pointer to a one-dimensional array of 16 entries organized by the columns of the desired matrix)

Reading back matrix:

double m[16];

- glGetFloatv(GL_MODELVIEW, m); access matrix C

Model Transformations

First define the transformations on the geometric model then the geometric model itself.

```
//select model-view matrix stack
glMatrixMode(GL_MODELVIEW);

//initialize identity matrix
glLoadIdentity( );
....
glTranslatef( dx, dy, dz );          //matrix C
glScalef( sx, sy, sz );              //matrix B
glRotatef( angle, rx,ry,rz );        //matrix A

// Draw a cube using indices
glDrawElements(GL_QUADS,24,GL_UNSIGNED_BYTE, allIndices);
```

$$p' = (C(B(Ap)))$$

Apply the tranformations in inverse order w.r.t. their definitions

Rotation about a Fixed Point

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

$\mathbf{C} \leftarrow \mathbf{I}$	Start with identity matrix
$\mathbf{C} \leftarrow \mathbf{CT}^{-1}$	Move fixed point back
$\mathbf{C} \leftarrow \mathbf{CR}$	Rotate
$\mathbf{C} \leftarrow \mathbf{CT}$	Move fixed point to origin

Each operation corresponds to one function call in the program. Note that the last operation specified is the first executed in the program

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, 1.0);
glTranslatef(-1.0, -2.0, -3.0);
draw_object()
```

Each vertex p that is specified after the CTM has been set will be multiplied by C, thus forming the new vertex q=Cp

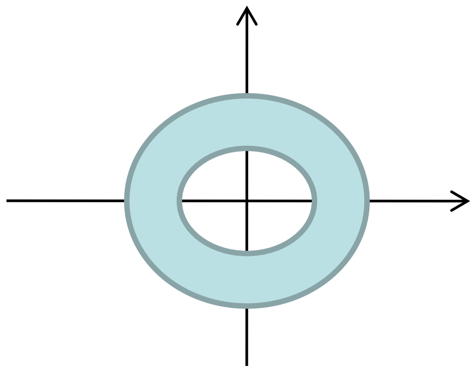
$$q = \boxed{T(1.,2.,3.)R(30,0,0,1)T(-1.,-2.,-3.)}p$$

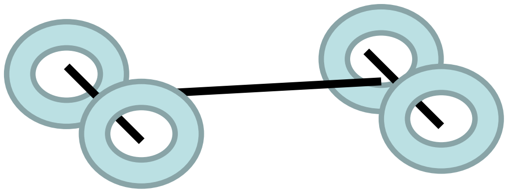
Stack of matrices

- OpenGL maintains stacks for each type of matrix
- *glPushMatrix()* save C (C_SAVED) for use later
- *glPopMatrix()* reload C_SAVED

Example

```
// position of the car on the road
glTranslate(CarPos)
glRotate(CarDirection,0,1,0)
DrawCarBody();
For (i=0;i<4;++i) {
  glPushMatrix()
  //relative single wheel position
  glTranslate(wheelPos[i]);
  DrawWheel();
  glPopMatrix();
}
```





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

```
void main(int argc, char **argv) {
…
glutIdleFunc(spinCube);
glutMouseFunc(mouse);
…
}
```

Idle and Mouse callbacks

```
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;
}
```

variables such as theta and axis must be defined as globals

Display callback

```
void display() {
glClear(GL_COLOR_BUFFER_BIT
                        |GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
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();
}
```

Using the Model-view Matrix

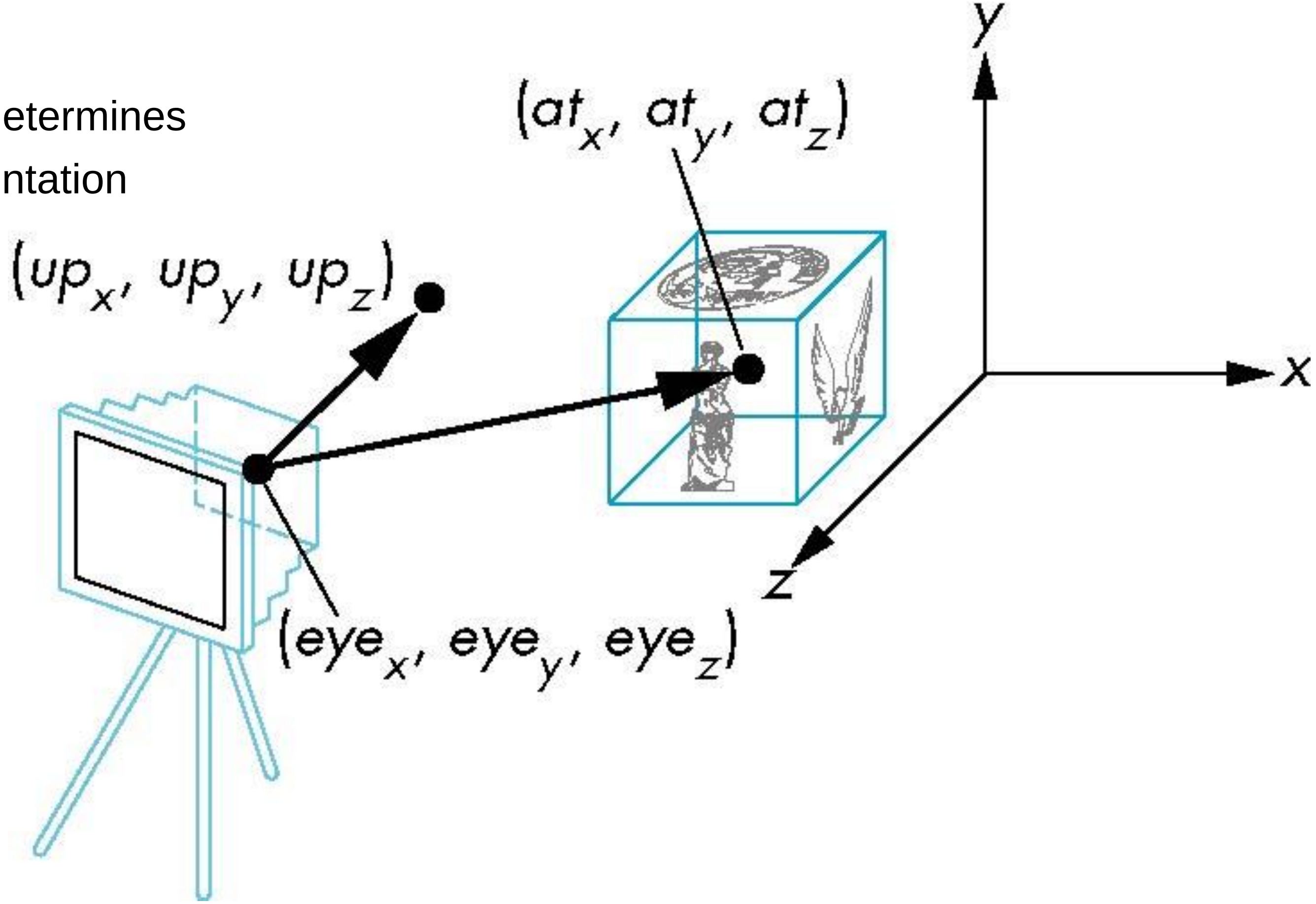
- In OpenGL the same model-view matrix is used to
  - **Position the camera (view transform)**
    - Can be done by rotations and translations but is often easier to use a LookAt function
  - **Build models of objects (model transform)**
- The **projection transform** is used to define the view volume and to select a camera lens

- CTM (Model-view matrix composed with Projection matrix) is finally applied to the vertices

View Transformation

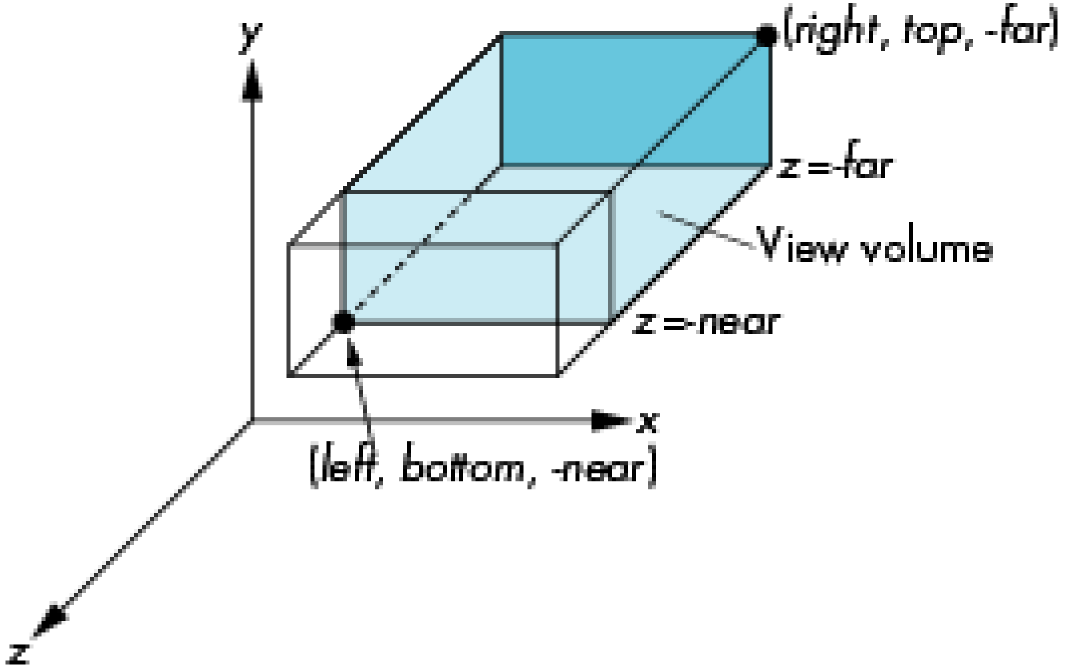
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
gluLookAt(eyex,eyey,eyez,aimx,aimy,aimz,upx,upy,upz);
```

up vector determines
unique orientation



OpenGL Orthographic projection

```
glMatrixMode( GL_PROJECTION )
glLoadIdentity( );
glOrtho(left,right,bottom,top,near,far)
```



`near` and **`far`** measured <u>from</u> camera

2D Graphics orthographic viewing

In 2D, the clipping **volume** becomes a clipping **rectangle**

Equivalent to calling **glOrtho** with *near* = *-1* and *far* = *1*

```
gluOrtho2D(left, right, bottom, top);
```

define a 2D orthographic projection matrix used to set up the world window.

Example: If we wanted a world window with

x varying from -1.0 to 1.0 and

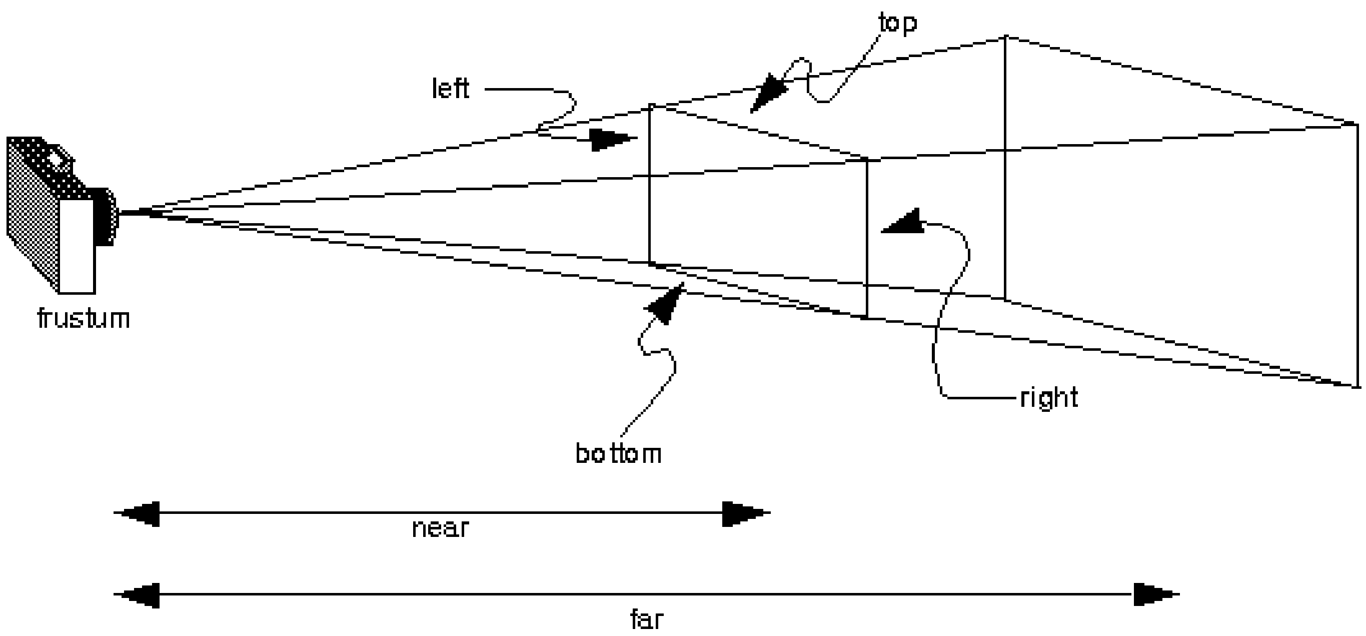
y varying from 3.0 to 5.0,

we would use the following code to accomplish this:

```
– glMatrixMode(GL_PROJECTION);
– glLoadIdentity( );
– gluOrtho2D(-1.0, 1.0, 3.0, 5.0);
```

OpenGL Perspective

```
glMatrixMode( GL_PROJECTION )
glLoadIdentity( );
glFrustum(left,right,bottom,top,near,far)
```



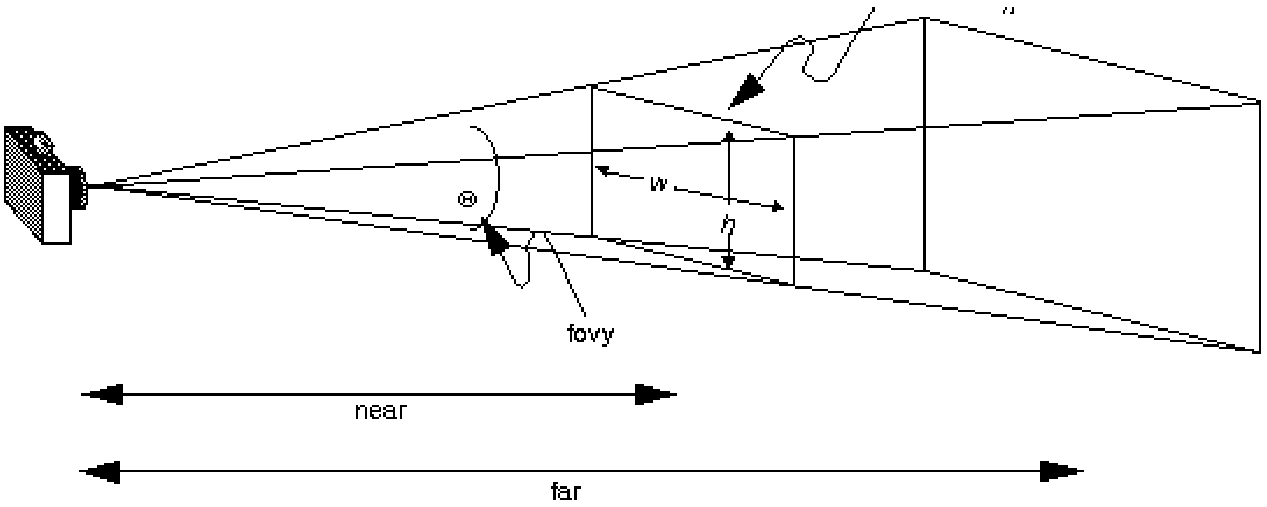
Using Field of View

With **`Frustum`** it is often difficult to get the desired view

`gluPerspective(fovy, aspect, near, far)`

often provides a better interface

`aspect = w/h`

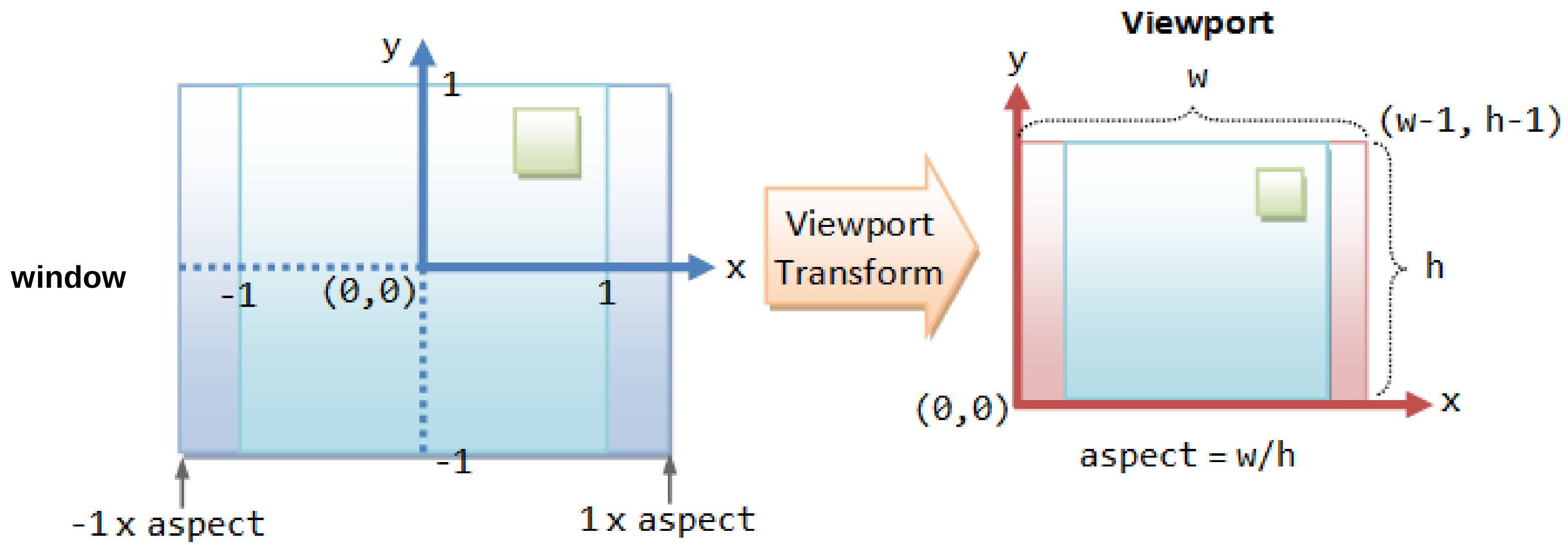


FOVY is the angle between the top and bottom planes (between 0.0 and 180.0)

Viewport Transformations

- Viewport = rectangular region of the screen where the image is projected
- Default : viewport is the window open in the screen
- Modify the viewport with the function

`glViewport(x,y,width,height)`



x y specify the lower left corner of the viewport rectangle, in pixels, default (0,0)

Reshape callback

Registered as callback for **glutReshapeFunc()**

```
/* called for window resize */
void resize( int w, int h )
{
glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 65.0, (GLfloat) w / h,1.0, 100.0 );

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt( 0.0, 0.0, 5.0,0.0, 0.0, 0.0,0.0, 1.0, 0.0);
}
```

resize() uses the viewport's width and height values as the aspect ratio for gluPerspective() which eliminates distortion.

Advanced Graphics Primitives

- **GL represents points, segments, polygons and Bézier curves and surfaces**
- **GLU represents quadric surface (sphere, torus, cylinder, ..)**
- **GLU represents spline curves and surfaces (NURBS)**

Quadric primitives (GLU)

- Define a quadric object (in main( ))

obj = gluNewQuadric( );

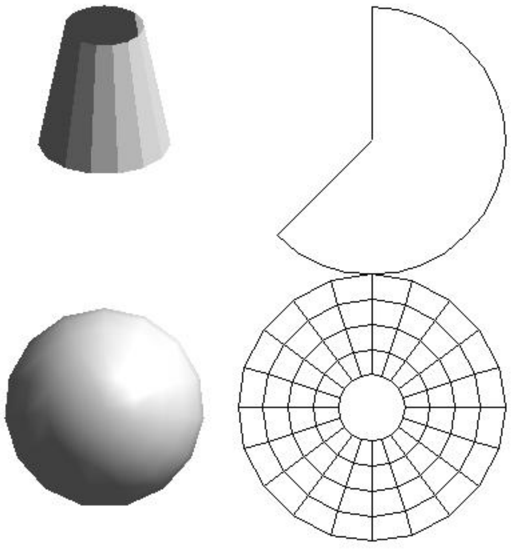
- Specify rendering attributes (in main( ))

gluQuadricDrawStyle(obj, GLU_LINE);
gluQuadricNormals(…)
gluQuadricTexture(…)

……………………

- Draw (in display( ))

gluSphere(obj, 1.0, 12, 12);
gluCylinder(obj, 1.0, 0.5, 1.0, 12, 12);
gluDisk(obj, 0.5, 1.0, 10, 10);
gluPartialDisk(obj, 0.5, 1.0, 10, 10, 0.0, 45.0);
glutWireTeapot(1.0);
glutWireTorus(0.5, 1.0, 10, 10);
glutWireCone(1.0, 1.0, 10, 10);

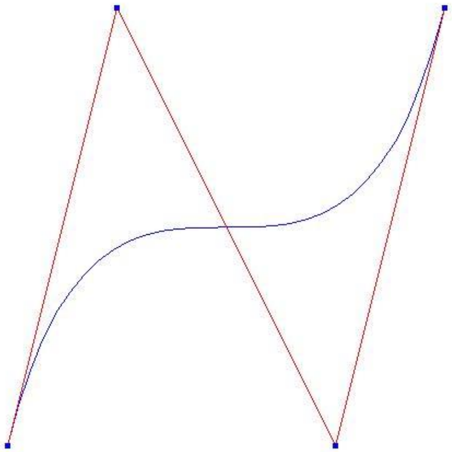


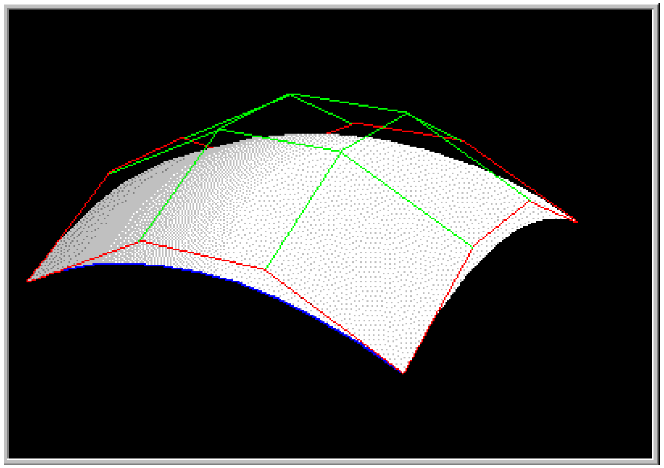
- Cancel

glutDeleteQuadric( );

Bézier Curves and surfaces

- **Evaluators:** compute curve and surfaces values at parametric values





Bézier Curve

Glfloat cpts[4][3]={ { -4.0, -4.0, 0.0 }, { -2.0, 4.0, 0.0 },

{ 2.0, -4.0, 0.0 }, { 4.0, -4.0, 0.0 } };

1. Define Evaluator **(init):**

```
glEnable(GL_MAP1_VERTEX_3);
glMap1f(type, u_min, u_max, stride, order, cpoint_array);
```

stride: offset between two control points in u direction

Example:

Cubic 3D Bézier curve in [0,1]

```
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &cpts[0][0]);
```

Bézier Curve

2. Draw the curve:

Evaluate the curve at u parametric value:

```
glEvalCoord1f(u);
```

Evaluate a set of values:

```
glBegin(GL_LINE_STRIP);
   for (i=0;i<100;i++) glEvalCoord1f((GLfloat) i/100.0);
glEnd();
```

For uniform spaced values u:

```
glMapGrid1f(100, 0.0, 1.0);           (in init())
glEvalMesh1(GL_LINE, 0, 100);    (in display())
```

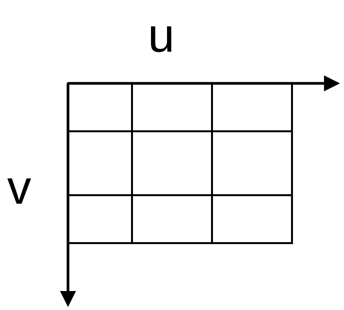
Bézier Patch

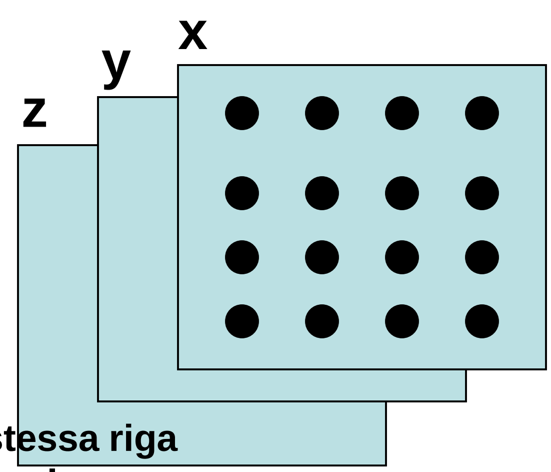
1. Define Evaluator **(init):**

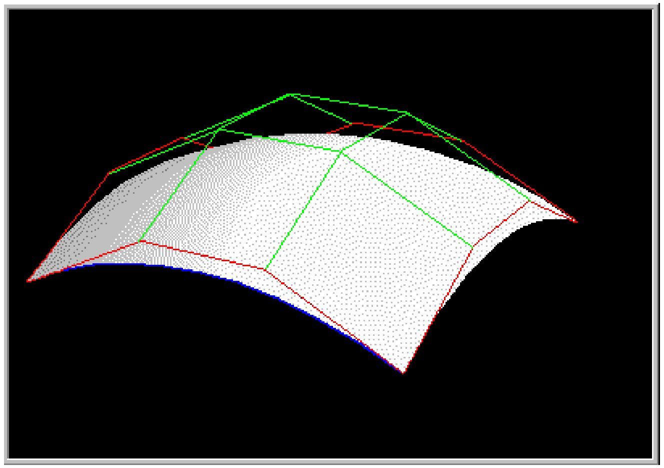
```
glEnable(GL_MAP2_VERTEX_3);
glMap2f(type, u_min, u_max, strideu, orderu, v_min,
            v_max, stridev, orderv, cpoint_array);
```

Es: bicubic patch in [0,1]x[0,1]:

```
glMap2f(GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4,
                0.0, 1.0, 12, 4, &ctps[0][0][0]);
```







Step 3 prossimo elemento stessa riga
Step 12 prossimo elemento colonna

Bézier Patch: mode I (wireframe)

2. **Draw the Patch (Evaluate a grid of points):**

```
for (j = 0; j <= 100; j++)
{

  glBegin(GL_LINE_STRIP);
  for (i = 0; i <= 100; i++)
   glEvalCoord2f( (GLfloat) i/100.0, (GLfloat) j/100.0);
  glEnd( );

  glBegin(GL_LINE_STRIP);
  for (i = 0; i <= 100; i++)
   glEvalCoord2f( (GLfloat) j/100.0, (GLfloat) i/100.0);
  glEnd();
}
```

Bézier Patch: mode II

Draw the Patch (Evaluate a grid of points):

```
for (j = 0; j < 99; j++)
{
   glBegin(GL_QUAD_STRIP);
      for (i = 0; i <= 100; i++){
glEvalCoord2f((GLfloat) i/100.0, (GLfloat) j/100.0);
glEvalCoord2f((GLfloat) (i+1)/100.0, (GLfloat)j/100.0);
      }
  glEnd();
}
```

Bézier Patch

```
glEnable(GL_AUTO_NORMAL);
```

Draw the Bézier Patch in regular grid:

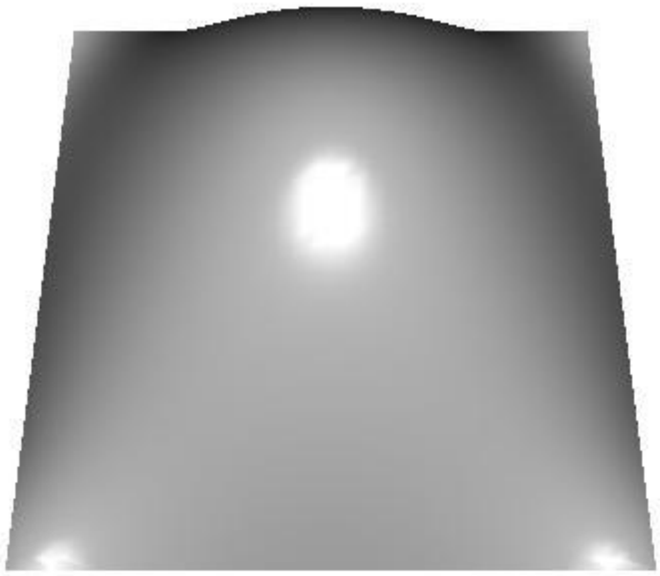
(init:)

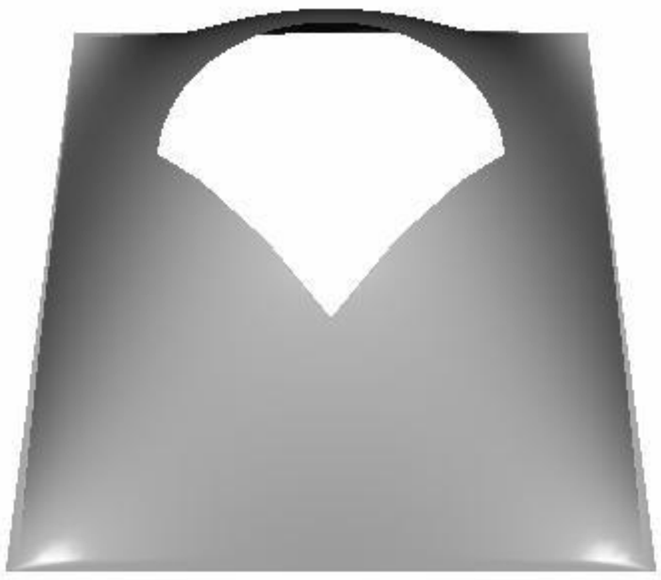
```
glMapGrid2f(100,0.0,1.0,100,0.0,1.0);
```

(display:)

```
glEvalMesh2(GL_FILL,0,100,0,100);
```

NURBS curves and surfaces and Trimmed NURBS





NURBS curves and surfaces

- Define the surface properties (init.):

```
theNurb = gluNewNurbsRenderer( );
gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
```

- Initialize the surface S(u,v)   (display):

Replace with **Curve**

```
gluBeginSurface(theNurb);
    gluNurbsSurface(theNurb,
      u_numknots, uknots,   num. knots in u, nodal vector in u
      v_numknots, vknots,
      4 * v_numpoints,   u_stride:offset between control points
      4,                 v_stride
      &ctrlpoints[0][0][0],
      u_order, v_order, GL_MAP2_VERTEX_4);
gluEndSurface(theNurb);
```

Trimmed NURBS surfaces

- A **trimming curve** can be either a NURBS curve (**gluNurbsCurve**) or a polyline (**gluPwlCurve**)

```
gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb, 8, knots, 8, knots, 4 * 3, 3,
      &ctlpoints[0][0][0], 4, 4, GL_MAP2_VERTEX_3);
   gluBeginTrim (theNurb);
    gluNurbsCurve (theNurb, 8, curveKnots, 2,
                &curvePt[0][0],4,GLU_MAP1_TRIM_2);

gluPwlCurve(theNurb,3,&pwlPt[0][0],2,GLU_MAP1_TRIM_2);
  gluEndTrim (theNurb);
 gluEndSurface(theNurb);
```