Graphics and Visualization

Holger Kenn

International University Bremen

Spring Semester 2006

Recap

- Representing graphic objects by homogenous points and vectors
- Using affine transforms to modify objects
- Using projections to display objects

Homogenous Representation

- A vector in a coordinate frame:

$$\vec{v} = (\vec{a}, \vec{b}, \vec{c}, \vartheta) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix}$$

Homogenous Representation

- A point in a coordinate frame:

$$P = (\vec{a}, \vec{b}, \vec{c}, \vartheta) \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ 1 \end{pmatrix}$$

Homogenous coordinates

- The difference of two points is a vector
- The sum of a point and a vector is a point
- Two vectors can be added
- A vector can be scaled
- Any linear combination of vectors is a vector
- An affine combination of two points is a point. (An affine combination is a linear combination where the coefficients add up to 1.)
- A linear interpolation $P = (a(1 - t) + Bt$ is a point.
- This fact can be used to calculate a “tween” of two points.

Transformations

- Transformations are an easy way to reuse shapes
- A transformation can also be used to present different views of the same object
- Transformations are used in animations.

Transformations in OpenGL

- When we're calling a `glVertex()` function, OpenGL automatically applies some transformations. One we already know is the world-window-to-viewport transformation.
- There are two principle ways do see transformations:
  - *object transformations* are applied to the coordinates of each point of an object, the coordinate system is unchanged
  - *coordinate transformations* defines a new coordinate system in terms of the old coordinate system and represents all points of the object in the new coordinate system.
- A transformation is a function that mapps a point P to a point Q, Q is called the image of P.

3D affine transformations

- The general form of an affine 3D transformation

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Translation...

- As expected:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Scaling in 3D...

- Again:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Shearing...

- in one direction

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Rotations 3D...

- x-roll, y-roll and z-roll
- x-roll:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 1 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Rotations 3D...

- y-roll:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Rotations 3D...

- z-roll:

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

point vs coordinate system transformations

- If we have an affine transformation M, we can use it to transform a coordinate frame F_1 into a coordinate frame F_2.
- A point $P = (P_x, P_y, 1)^T$ represented in F_2 can be represented in F_1 as MP
- $F_1 \rightarrow^{M_1} F_2 \rightarrow^{M_2} \rightarrow F_3$ then P in F_3 is M_1M_2P in F_1.
- To apply the sequence of transformations M_1, M_2, M_3 to a point P, calculate $Q = M_3M_2M_1P$. An additional transformation must be *premultiplied*.
- To apply the sequence of transformations M_1, M_2, M_3 to a coordinate system, calculate $M = M_1M_2M_3$. A point P in the transformed coordinate system has the coordinates MP in the original coordinate system. An additional transformation must be *postmultiplied*.

And now in OpenGL...

- Of course we can do everything by hand: build a point and vector datatype, implement matrix multiplication, apply transformations and call `glVertex` in the end.
- In order to avoid this, OpenGL maintains a *current transformation* that is applied to every `glVertex` command. This is independent of the window-to-viewport translation that is happening as well.
- The current transformation is maintained in the *modelview matrix*.

And now in OpenGL...

- It is initialized by calling `glLoadIdentity`
- The modelview matrix can be altered by `glScaled()`,`glRotated` and `glTranslated`.
- These functions can alter any matrix that OpenGL is using. Therefore, we need to tell OpenGL which matrix to modify: `glMatrixMode(GL_MODELVIEW)`.

A stack of CTs

- Often, we need to “go back” to a previous CT. Therefore, OpenGL maintains a “stack” of CTs (and of any matrix if we want to).
- We can push the current CT on the stack, saving it for later use: `glPushMatrix()`. This pushes the current CT matrix and makes a copy that we will modify now
- We can get the top matrix back: `glPopMatrix()`.

3D! (finally)

- For our 2D cases, we have been using a very simple parallel projection that basically ignores the perspective effect of the z-component.
- the view volume forms a rectangular parallelepiped that is formed by the border of the window and the *near plane* and the *far plane*.
- everything in the view volume is parallel-projected to the window and displayed in the viewport. Everything else is clipped off.
- We continue to use the parallel projection, but make use of the z component to display 3D objects.

3D Pipeline

- The 3d Pipeline uses three matrix transformations to display objects
  - The modelview matrix
  - The projection matrix
  - The viewport matrix
- The modelview matrix can be seen as a composition of two matrices: a model matrix and a view matrix.

in OpenGL

- Set up the projection matrix and the viewing volume:

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

- Aiming the camera. Put it at eye, look at look and upwards is up.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x, eye_y, eye_z,
    look_x, look_y, look_z, up_x, up_y, up_z);
```

Basic shapes in OpenGL

- A wireframe cube:

```
glutWireCube(GLdouble size);
```

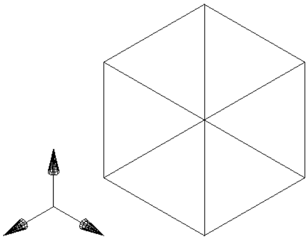
- A wireframe sphere:

```
glutWireSphere(GLdouble radius,
    GLint nSlices,GLint nStacks);
```

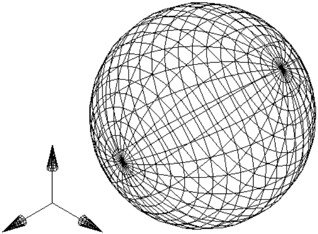
- A wireframe torus:

```
glutWireTorus(GLdouble inRad, GLdouble outRa
    GLint nSlices,GLint nStacks);
```

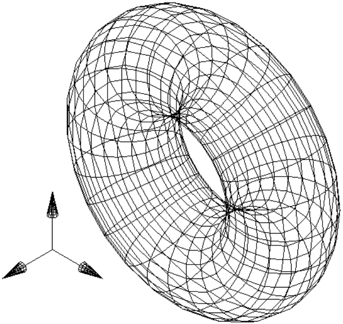
Cube



Sphere



Torus

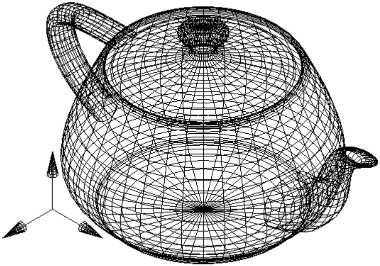


And the most famous one...

- The Teapot

```
glutWireTeapot(GLdouble size);
```

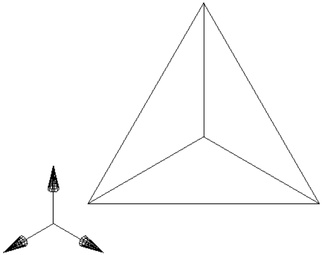
The Teapot



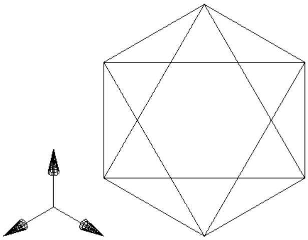
The five Platonic solids

- Tetrahedron: `glutWireTetrahedron()`
- Octahedron: `glutWireOctahedron()`
- Dodecahedron: `glutWireDodecahedron()`
- Icosahedron: `glutWireIcosahedron()`
- Missing one?

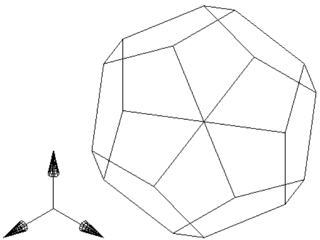
Tetrahedron



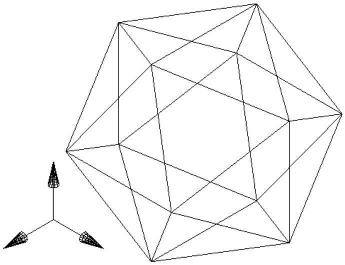
Octahedron



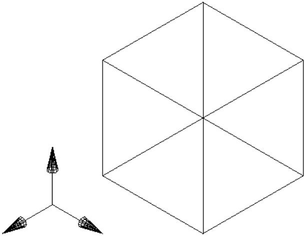
Dodecahedron



Icosahedron



Cube



...but we had that already.

Moving things around

- All objects are drawn at the origin.
- To move things around, use the following approach:

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslated(0.5,0.5,0.5);
glutWireCube(1.0);
glPopMatrix();
```

Moving things around

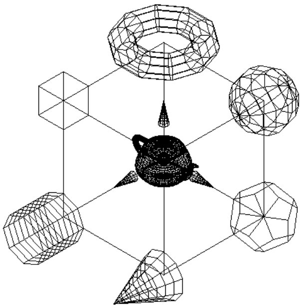


Image from Hill, Figure 5.60 (regenerated)

Rotating things

- To rotate things, use the following approach:

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glRotatef(angle,0.0,1.0,0.0);
glutWireTeapot(1.0);
glPopMatrix();
```

Hierarchical Modeling

- If we try to model an everyday object (like a house), we do not want to move all its components separately.
- Instead we want to make sure that if we move the house, the roof of the house move together with the walls.
- The CT stack gives us a simple way to implement this.

Global motion

- The simple case of hierarchical modeling is global motion.
- To implement it, we apply a number of transforms before we start drawing objects.

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glTranslated(x,y,z);
glRotatef(turnit,0.0,1.0,0.0);
drawMyScene();
glPopMatrix();
```

Local motion

- To implement local motion, apply an extra transformation before each object is drawn

```
drawmyteapot(){
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
   glRotatef(spinit,0.0,0.0,1.0);
   glutWireTeapot(1.0);
   glPopMatrix();
}
```

Robot Arm Example: Box

```
void Box(float width, float height, float depth) {
  char i, j = 0;
  glColor3f(1,0,0);
  glPushMatrix();
  glScalef(width,height,depth);
  glutWireCube(1.0);
  glPopMatrix();
}
```

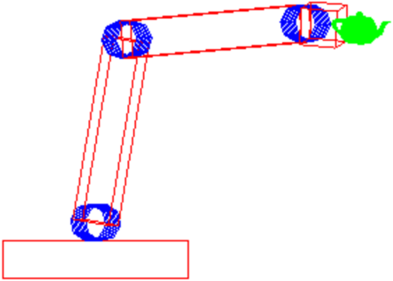
Robot Arm Example:Joint

```
void Joint() {
  glColor3f(0,0,1);
  glPushMatrix();
  GLUquadricObj * qobj;
  qobj=gluNewQuadric();
  gluQuadricDrawStyle(qobj,GLU_LINE);
  glRotatef(90,0,1,0);
  glTranslatef(0,0,-0.15);
  gluCylinder(qobj,0.1,0.1,0.3,8,8);
  glPopMatrix();
}
```

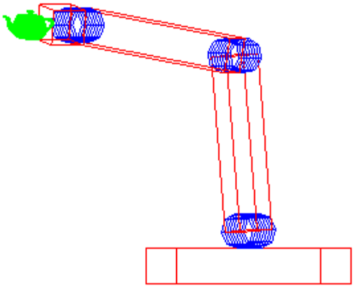
Robot Arm Example:Arm

```
void Arm() {
  glPushMatrix();
  Box(1,0.2,1);
  glTranslatef(0,0.2,0); /* on top of the box */
  glRotatef(dof[0],0,1,0);/* rotate around the y ax
  glRotatef(dof[1],1,0,0);/* rotate around the x ax
  Joint();
  glTranslatef(0,0.5,0); /* move to the middle of th
  Box(0.2,1,0.2); /* draw a box */
  glTranslatef(0,0.5,0); /* move to the end of the
  glRotatef(dof[2],1,0,0); /* rotate elbow joint */
  Joint();
...
```

Robot Arm



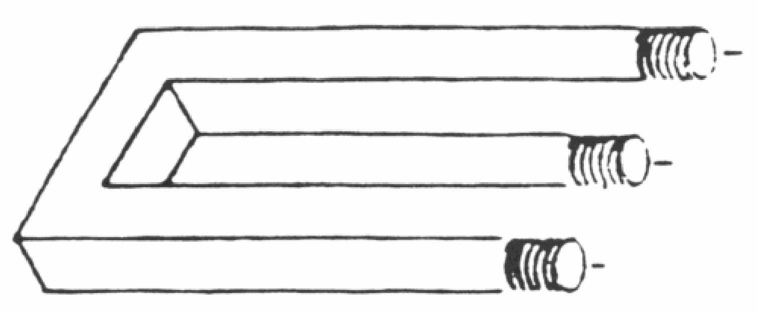
Robot Arm



Perspective

- Our current parallel projection is quite poor in giving us a “real” view of things.
- That is because it is “ignoring” the z component which leads to ambiguities.

Perspective



from http://www.leinroden.de/

Perspective in OpenGL

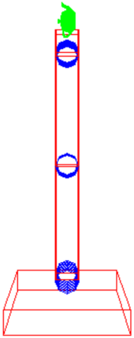
- Set up the projection matrix and the viewing volume:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(viewAngle, aspectRatio,N,F);
```

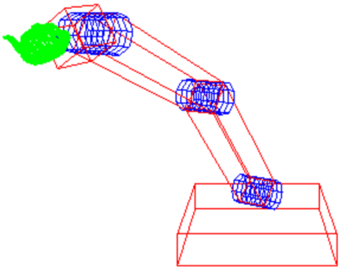
- Aiming the camera. Put it at eye, look at look and upwards is up. (no change here)

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye_x, eye_y, eye_z,
    look_x, look_y, look_z, up_x, up_y, up_z);
```

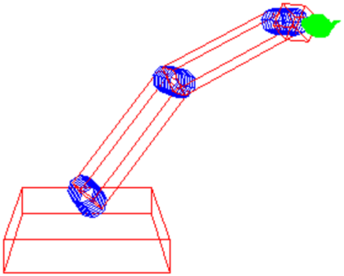
Robot Arm



Robot Arm



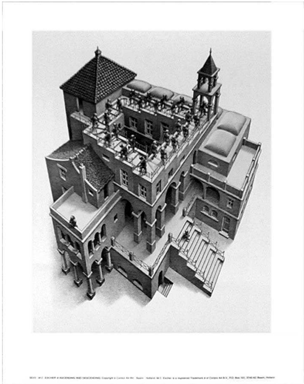
Robot Arm



Perspective

- The point perspective in OpenGL resolves some ambiguities
- but it cannot solve all ambiguities

Perspective



from http://www.worldofescher.com

Solid Modeling

- We can model a solid object as a collection of polygonal faces.
- Each face can be specified as a number of vertices and a normal vector (to define the inside and the outside)
- For clipping and shading, it is useful to associate a normal vector with every vertex. Multiple vertices can be associated with the same normal vector and a vertex can be associated with multiple normal vectors.
- To represent and object, we could store all vertices for all polygons together with a normal vector for every vertex. That would be highly redundant.

Storing polygonal meshes

- Instead, we can use three lists:
  - the vertex list
    It contains all distinct vertices
  - the normal list
    It contains all distinct normal vectors
  - the face list
    It only contains lists of indices of the two other lists

The basic barn

vertex	x	y	z
0	0	0	0
1	1	0	0
2	1	1	0
3	0.5	1.5	0
4	0	1	0
5	0	0	1
6	1	0	1
7	1	1	1
8	0.5	1.5	1
9	0	1	1

normal	n_x	n_y	n_z
0	-1	0	0
1	-0.707	0.707	0
2	0.707	0.707	0
3	1	0	0
4	0	-1	0
5	0	0	1
6	0	0	-1

The basic barn

face	vertices	normals
0	0,5,9,4	0,0,0,0
1	3,4,9,8	1,1,1,1
2	2,3,8,7	2,2,2,2
3	1,2,7,6	3,3,3,3
4	0,1,6,5	4,4,4,4
5	5,6,7,8,9	5,5,5,5,5
6	0,4,3,2,1	6,6,6,6,6

Finding the normal vectors

- We can compute the normal of a face using three vectors and the cross product $m = (V_1 - V_2) \times (V_3 - V_2)$ and normalize it to unit length.
- Two problems arrise:
  - What if $(V_1 - V_2)$ and $(V_3 - V_2)$ are almost parallel?
  - What to do with faces that are defined through more than three vertices?
- Instead, we can use Newell's method:
  - $m_x = \sum_{i=0}^{N-1}(y_i - y_{next(i)})(z_i + z_{next(i)})$
  - $m_y = \sum_{i=0}^{N-1}(z_i - z_{next(i)})(x_i + x_{next(i)})$
  - $m_z = \sum_{i=0}^{N-1}(x_i - x_{next(i)})(y_i + y_{next(i)})$

Properties of polygonal meshes

- Solidity (if the faces enclose a positive and finite amount of space)
- Connectedness (if there is a path between every two vertices along the polygon edges)
- Simplicity (if the object is solid and has no “holes”)
- Planarity (if every face is planar, i.e. every vertex of a polygon lies in a plane)
- Convexity (if a line connecting any two points in the object lies completely within the object)
- A Polyhedron is a connected mesh of simple planar polygons that encloses a finite amount of space

Properties of polyhedrons

- Every edge is shared by exactly two faces
- at least three edges meet at each vertex
- faces do not interpenetrate: they either touch at a common edge or not at all.
- Euler's formula for simple polyhedrons: $V + F - E = 2$ (E:Edges, F: Faces, V: Vertices)
- For non-simple polyhedrons: $V + F - E = 2 + H - 2G$ (G: holes in the polyhedron, H: holes in faces)

Summary

- Hierarchical Modeling
- Perspective vs Parallel Projection
- Representing solid objects

Shading

- Displaying Wireframe models is easy from a computational viewpoint
- But it creates lots of ambiguities that even perspective projection cannot remove
- If we model objects as solids, we would like them to look “normal”. One way to produce such a normal view is to simulate the physical processes that influence their appearance (Ray Tracing). This is computationally very expensive.
- We need a cheaper way that gives us some realism but is easy to compute. This is shading.

Types of shading

- Remove hidden lines in wireframe models
- Flat Shading
- Smooth Shading
- Adding specular light
- Adding shadows
- Adding texture

Toy-Physics for CG

- There are two types of light sources: ambient light and point light sources.
- If all incident light is absorbed by a body, it only radiates with the so-called blackbody radiation that is only dependent of its temperature. We're dealing with cold bodys here, so blackbody radiation is ignored.
- Diffiuse Scattering occurs if light penetrates the surface of a body and is then re-radiated uniformily in all directions. Scattered lights interact strongly with the surface, so it is usually colored.
- Specular reflections occur in metal- or plastic-like surfaces. These are mirrorlike and highly directional.
- A typical surface displays a combination of both effects.