Graphics and Visualization

Holger Kenn

International University Bremen

Spring Semester 2006

Recap

- Coordinate System
- Cohen Sutherland Clipping
- Implicit curves
- Parametric curves

Coordinate System

- The space in which objects are described uses *world coordinates*.
- The part of this space that we want to display is called *world window*.
- The window that we see on the screen is our *viewport*.
- In order to know where to draw something, we need the *world-to-viewport transformation*
- Note that these terms can be used both for 2D and for 3D.

Cohen Sutherland

- Compute 4 test bits for the endpoints of a line segment
- Trivial Accept: all tests false, all bits 0
- Trivial Reject: the words for both points have 1s in the same position
- Deal with the rest: neither trivial accept nor reject

Cohen Sutherland (2)

- Identify which point is outside and to which side of the window
- Find the point where the line touches the world window border
- Move the outer point to the border of the window
- repeat all until trivial accept or reject

Implicit form of curves

- The implicit form is good for testing if a point is on a curve.
- For some cases, we can use the implicit form to define an "inside" and an "outside" of a curve: $F(x, y) < 0 \rightarrow$ inside, $F(x, y) > 0 \rightarrow$ outside
- some curves are *single valued* in x: $F(x, y) = y - g(x)$ or in y:$F(x, y) = x - h(y)$
- some curves are neiter, e.g. the circle needs two functions $y = \sqrt{R^2 - x^2}$ and $y = -\sqrt{R^2 - x^2}$

Parametric form of curves

- The parametric form of a curve suggests the movement of a point through time.
- Example: $x(t) = A_x + (B_x - A_x)t, y(t) = A_y + (B_y - A_y)t, t \in [0,1]$
- Example: $x(t) = W\cos(t)$, $y(t) = H\sin(t), t \in [0, 2\pi]$
- In order to find an implicit form from a parametric form, we can use the two $x(t)$ and $y(t)$ equations to eliminate t and find a relationship that holds true for all t.
- For the Ellipse: $\left(\frac{x}{W}\right)^2 + \left(\frac{y}{H}\right)^2 = 1$

Superellipses

- A *superellipse* is defined by the implicit form $\left(\frac{x}{W}\right)^n + \left(\frac{y}{H}\right)^n = 1$
- A *supercircle* is a superellipse with $W = H$.
- $x(t) = W\cos(t)|\cos(t)^{2/n-1}|$
- $y(t) = H\sin(t)|\sin(t)^{2/n-1}|$

Superellipses

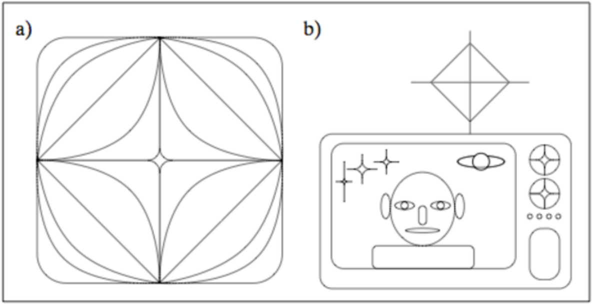


Image from Hill, Pg 125

Polar coordinates

- Polar coordinates can be used to draw parametric curves.
- The curve is represented by a distance to the center point r and an angle θ.
- $x(t) = r(t)\cos(\theta(t))$,$y(t) = r(t)\sin(\theta(t))$ (general form)
- $x(\theta) = f(\theta)\cos(\theta)$,$y(t) = f(\theta)\sin(\theta)$ (simple form)

Polar Coordinates

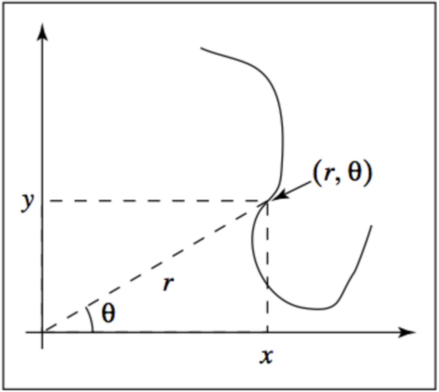


Image from Hill, Pg 126

Polar coordinate shapes

- Cardioid $f(\theta) = K(1 + \cos(\theta))$
- Rose Curves $f(\theta) = K\cos(n\theta)$
- Archimedian Spiral $f(\theta) = K\theta$
- Conic sections $f(\theta) = \frac{1}{1 \pm e\cos(\theta)}$
- Logarithmic Spiral $f(\theta) = Ke^{a\theta}$

Examples

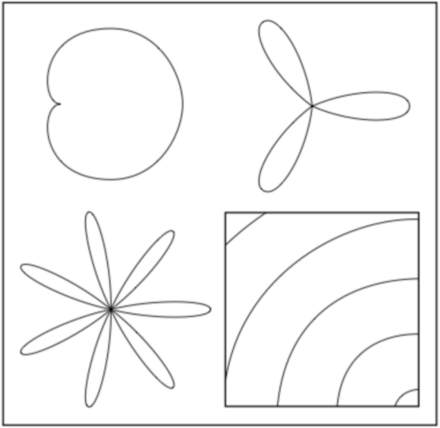


Image from Hill, Pg 126

Examples

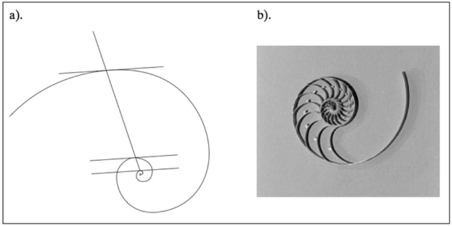


Image from Hill, Pg 127

3D parametric curves

- We can also specify 3d curves using three functions $x(t), y(t), z(t)$
- Helix: $x(t) = \cos(t), y(t) = \sin(t), z(t) = bt$
- Toroidal spiral:
  - $x(t) = (a\sin(ct) + b)\cos(t)$
  - $y(t) = (a\sin(ct) + b)\sin(t)$
  - $z(t) = a\cos(ct)$

Examples

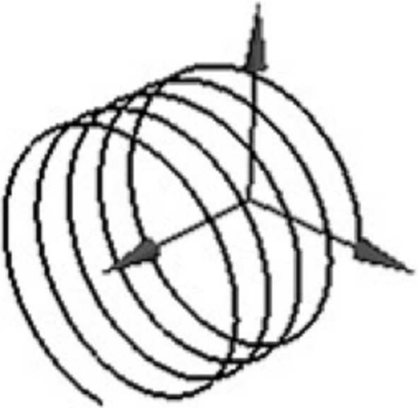


Image from Hill, Pg 128

Examples

Animation w. double buffering

- When we do a fast animation, the image starts to flicker.
- This results from the time it takes to draw the lines.
- We can avoid this via double-buffering
- in OpenGL, double buffering is simple:
- `glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);`
- `glutSwapBuffers();`

Representing Objects

- We have now seen that we can represent complex objects using many techniques
- Relative drawing lets us move objects around on the screen
- Parametric curves can represent classes of objects, e.g. Superellipses
- Polar coordinates can be used to draw round or curved objects
- And this also works in 3D.
- But it's not very practical: We don't want to use the clumsy relative drawing functions and we don't want to define a parametric representation for every complex form we want to draw.

Vectors

- We all remember what vectors are, right?
- Properties of vectors in CG:
- The difference of two points is a vector
- The sum of a point and a vector is a point
- A linear combination $a\vec{v} + b\vec{w}$ is a vector
- Let's write $w = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n$
- If $a_1 + a_2 + \cdots + a_n = 1$ this is called an affine combination
- if additionally $a_i \geq 0$ for $i = 1 \ldots n$, this is a convex combination
- To find the length of a vector, we can use Pythagoras: $|\vec{w}| = \sqrt{w_1^2 + w_2^2 + \cdots + W_n^2}$

Vectors

- When we know the length, we can normalize the vector, i.e. bring it to unit length: $\hat{a} = \vec{a}/|\vec{a}|$. We can call such a unit vector a *direction*.
- The dot product of two vectors is $\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} \vec{v}_i \vec{w}_i$ has the well-known properties
  - $\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$ (Symmetry)
  - $(\vec{a} + \vec{c}) \cdot b = \vec{a} \cdot \vec{b} + \vec{c} \cdot \vec{b}$ (Linearity)
  - $(s\vec{a}) \cdot \vec{b} = s(\vec{a} \cdot \vec{b})$ (Homogeneity)
  - $|\vec{b}|^2 = \vec{b} \cdot \vec{b}$
- We can play the usual algebraic games with vectors (simplification of equations)

Angles between vectors

- We can use the dot product to find the angle between two vectors: $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$. If the dot product of two (non-zero-length) vectors is 0 then they are *perpendicular* or *orthogonal* or *normal* to eachother.
- In 2D, we can find a perpendicular vector by exchanging the two components and negate one of them: If $\vec{a} = (a_x, a_y)$ then $\vec{b} = (-a_y, a_x)$ and we call this the *counterclockwise perpendicluar* vector of $\vec{a}$ or short $\vec{a}^\perp$

The 2D “Perp” Vector

- The “perp” vector is useful for projections (see book, page 157)
- The distance from a point C to the line through A in direction $\vec{v}$ is $|\vec{v}^{\perp} \cdot (C - A)|/|\vec{v}|$.
- Projections are used to simulate reflections

The cross product

- Everybody remembers $\vec{a} \times \vec{b}$
- One trick to write the cross product: Let $\vec{i}, \vec{j}, \vec{k}$ be the 3D standard unit vectors. Then the cross product of $\vec{a} \times \vec{b}$ can be written as the *determinant* of a matrix:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

- and we have the usual algebraic properties: antisymmetry, linearity, homogeneity...

Coordinate Systems and Coordinate Frames

- A coordinate system can be defined by three mutually perpendicular unit vectors.
- If we put these unit vectors into a specific point ϑ called origin, we call this a coordinate frame.
- In a coordinate frame, a point can be represented as $P = p_1\vec{a} + p_2\vec{b} + p_3\vec{c} + \vartheta$.
- This leads to a distinction between points and vectors by using a fourth coefficient in the so-called homogenous representation of points and vectors.

Homogenous Representation

- A vector in a coordinate frame:

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

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.

Representing lines and planes

- A line can be represented by its endpoints B and C
- It can also be represented parametrically with a point and a vector $L(t) = C + \vec{b}t$.
- A line can also be represented in *point normal form* $\vec{n} \cdot (R - C)$
- For $\vec{n}$ we can use $\vec{b}^{\perp}$ with $\vec{b} = B - C$
- A plane can be represented by three points
- It can also be represented parametrically by a point and two nonparallel vectors: $P(s, t) = C + \vec{a}s + \vec{b}t$
- It can also be represented in a point normal form with a point in the plane and a normal vector. For any point R in the plane $n \cdot (R - B) = 0$.
- A part of the plane restricted by the length of two vectors is called a *planar patch*.

intersections

- Every line segment has a *parent line*.
- We can first find the intersection of the parent lines
- and then see if the intersection point is in both line segments
- In order to intersect a plane with a line, we describe the line parametrically and the plane in the point normal form. Solving this equation gives us a "hit time" t that can be put into the parametric representation of the line to identify the *hitpoint*.

polygon intersections

- In convex polygons, the problem is rather easy: we can test all the bounding lines/surfaces.
- In order to know which side of a line/plane is “outside”, we represent them in a point normal form.
- We have to find exactly two “hit times” t_{in} and t_{out}.
- The right t_{in} will be the maximal “hit time” before the ray enters the polgon.
- The right t_{out} will be the minimal “hit time” after the ray exits the polgon.
- This approach can be used to clip against convex polygons. This is called the Cyrus-Beck-Clipping Algorithm.

Polygon Intersection

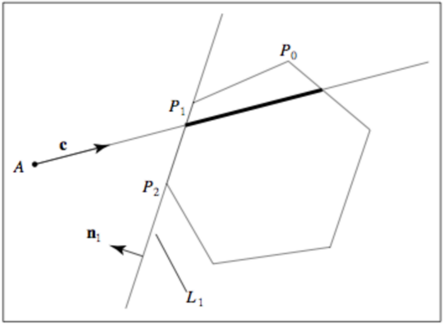


Image from Hill 4.43

Polygon Intersection

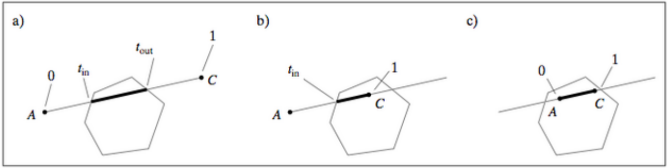


Image from Hill 4.44

Polygon Intersection

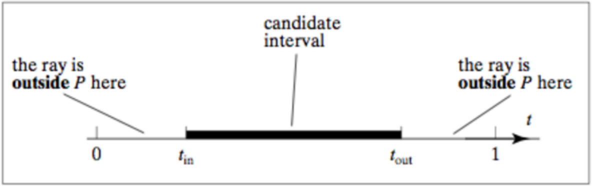


Image from Hill 4.45

Polygon Intersection

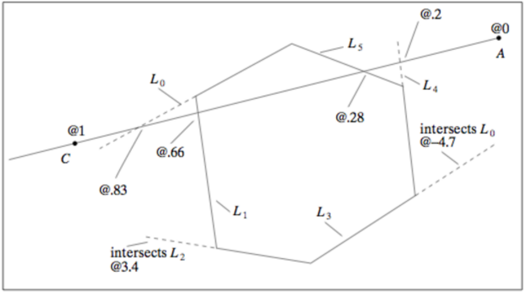


Image from Hill 4.46

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.

2d affine transformations

- A subset of transformations that uses transformation functions that are linear in the coordinates of the original point are the affine transformations.
- We can write them as a class of linear functions:

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{21}P_x + m_{22}P_y + m_{23} \\ 1 \end{pmatrix}$$

2d affine transformations

- or we can just use matrix multiplication

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- or we can also transform vectors with the same matrix

$$\begin{pmatrix} W_x \\ W_y \\ 0 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V_x \\ V_y \\ 0 \end{pmatrix}$$

standard transformations

- Translation

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & m_{13} \\ 0 & 1 & m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- scaling (and reflection for $S_{\{x,y\}} < 0$)

$$\begin{pmatrix} W_x \\ W_y \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} V_x \\ V_y \\ 1 \end{pmatrix}$$

standard transformations

- Rotation (positive θ is CCW rotation)

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- shearing

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & h & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

Inverse transformations

- inverse Rotation (positive θ is CW rotation)

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- inverse Scaling

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

Inverse transformations

- inverse shearing

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & -h & 0 \\ -g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

- inverse translation

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -m_{13} \\ 0 & 1 & -m_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

Inverse transformations

- In general (provided that M is nonsingular)

$$P = M^{-1}Q$$

- But as M is quite simple:

$$\det M = m_{11}m_{22} - m_{12}m_{21}$$
$$M^{-1} = \frac{1}{\det M}\begin{pmatrix} m_{22} & -m_{12} \\ -m_{21} & m_{11} \end{pmatrix}$$

composing affine transformations

- As affine transformations are simple matrix multiplications, we can combine several operations to a single matrix.
- In a matrix multiplication of transformations, the sequence of translations can be read from right to left.
- We can also take this combined matrix and reconstruct the four basic operations $M =$(translation)(shear)(scaling)(rotation) (this is for 2D only)

Some more facts

- Affine transformations preserve affine combinations of points
- Affine transformations preserve lines and planes
- Affine transformations preserve parallelism of lines and planes
- The column vectors of an affine transformation reveal the effect of the transformation on the coordinate system.
- An affine transformation has an interesting effect on the area of an object: $\frac{\text{area after transformation}}{\text{area before transformation}} = |\det M|$

The same game in 3D...

- 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:

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

Some facts about Rotations 3D

- 3D affine transformations can be composed as in 2D
- 3D rotation matrices do not commute (unlike 2D).
- Question: how to rotate around an arbitrary axis?
- Every 3D affine transformation can be decomposed into (translation)(scaling)(rotation)(shear_1)(shear_2).
- A 3D affine transformation has an effect on the volume of an object: $\frac{\text{volume after transformation}}{\text{volume before transformation}} = |\det M|$

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 $M_1 M_2 P$ in F_1.
- To apply the sequence of transformations M_1, M_2, M_3 to a point P, calculate $Q = M_3 M_2 M_1 P$. An additional transformation must be *premultiplied*.
- To apply the sequence of transformations M_1, M_2, M_3 to a coordinate system, calculate $M = M_1 M_2 M_3$. 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)`.

The 2D transformations

- Scaling in 2d:

```
glMatrixMode(GL_MODELVIEW);
glScaled(sx,sy,1.0);
```

- Translation in 2d:

```
glMatrixMode(GL_MODELVIEW);
glTranslated(dx,dy,0);
```

- Rotation in 2d:

```
glMatrixMode(GL_MODELVIEW);
glRotated(angle,0.0,0.0,1.0);
```

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);
```

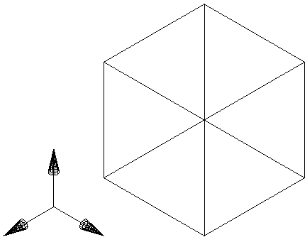
- A wireframe sphere:

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

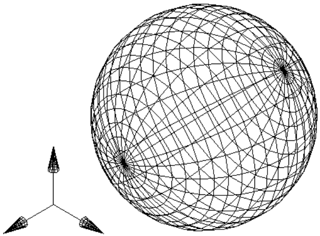
- A wireframe torus:

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

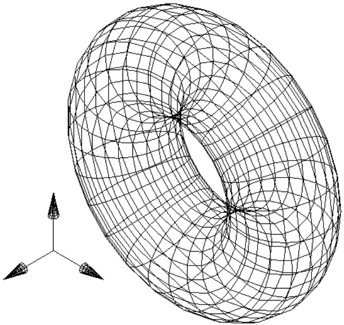
Cube



Sphere



Torus

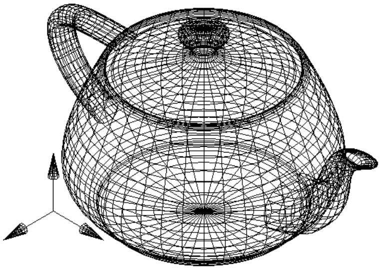


And the most famous one...

- The Teapot

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

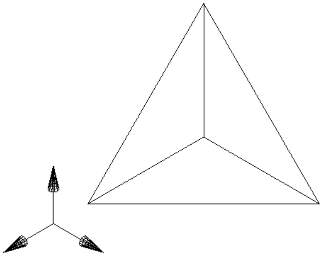
The Teapot



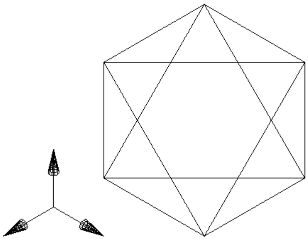
The five Platonic solids

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

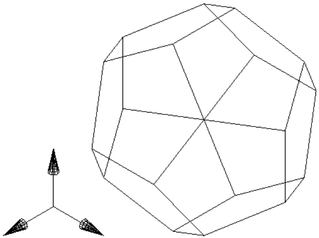
Tetrahedron



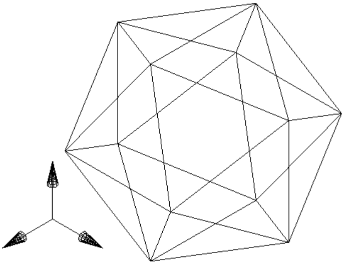
Octahedron



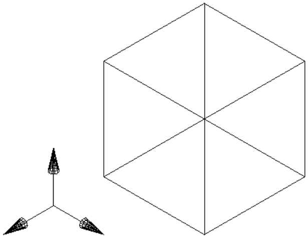
Dodecahedron



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

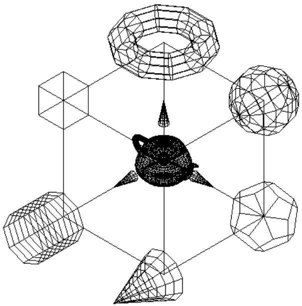


Image from Hill, Figure 5.60 (regenerated)

Summary

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