CISC454B: Computer Graphics

- computer graphics is concerned with producing pictures with a computer
  - this is a very broad definition…
- we are interested in three-dimensional computer graphics
  - topics:
    - rendering pipeline
    - mathematical foundations
    - representation of 3D objects
    - camera analogy
    - human vision (brief) and color
    - lighting, material properties, shading
    - rasterization
    - particle systems
    - other topics

Administrative Information

- instructor: Burton Ma
- office hours: 2:30-3:30pm Mon-Thurs Goodwin 735
- course web site
  - www.cs.queensu.ca/home/mab/454.html
  - still under construction
- text books (not required)
  - Computer Graphics Using OpenGL, 2nd Edition
  - OpenGL Programming Guide, 3rd Edition
  - The C++ Programming Language, Special Edition
- class notes
  - available online in PDF format
  - generally not complete (examples will be missing)

Administrative Information (cont)

- programming facilities
  - Walter Light Hall CASLAB
  - 24 Sun Ultra 10 workstations
  - C++, OpenGL

- marking
  - do whatever you want for 100 marks (must write first midterm)

✦ assignments	8 x 6%
✦ midterms	2 x 15%
✦ write a lecture	1 x 10%
✦ final exam	no more than 50%

Administrative Information (cont)

- assignments
  - written and programming
  - can work in groups (up to 3 people per group)
  - no extensions for any reason
    - don't leave them to the last minute
- midterms
  - in class
    - Thursday, February 1
    - Thursday, March 15
  - must write first midterm
  - closed book

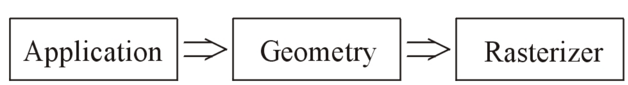
Administrative Information (cont)

- write a lecture
  - produce a lecture based on a research paper
  - can work with one other person
  - due before last week of class
- exam
  - notes and textbook permitted

- comments
  - don't let math intimidate you
  - don't let C++ intimidate you
  - a lot of work
    - budget your time wisely

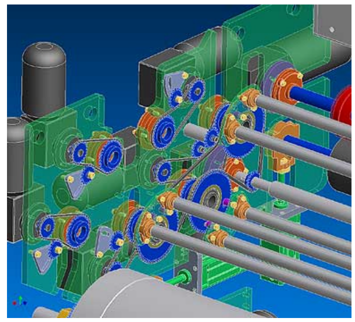
Rendering Pipeline

- rendering modeled as pipeline process
- primitives go in one end, move from stage to stage, and come out at the other end
- pipeline speed determined by slowest stage
- each stage also a pipeline or parallel pipelines
- each stage may discard primitives for efficiency



Application Stage

- application software e.g. video game, computer assisted design (CAD)
  - any application program that needs to send output to the screen



Application Stage (cont)

- defines:
  - geometry to draw (points, lines, polygons, and others)
  - material properties
  - lighting
  - viewing or camera parameters
- also performs other tasks:
  - user interaction (Hill 1.5 for examples of input devices)
  - animation
  - collision detection
  - speed-up techniques
  - many others
- output is scene to be drawn

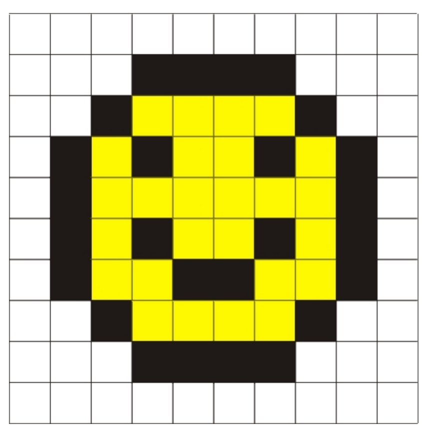
Geometry Stage

- performs most per-polygon and per-vertex operations
- implemented in software or hardware
- Hill calls this stage the graphics pipeline (Figures 5.52, 8.18)

- output is transformed geometry, colour and texture information

Rasterizer Stage

- rasterizes geometry
  - fills in pixels with correct colour to produce final image
- raster image is an array of picture elements (pixels)
  - example:
  - also see Hill 1.3-1.4



Rasterizer Stage (cont)

- implemented in hardware
- performs:
  - hidden surface removal
  - texturing
  - compositing
  - stenciling
  - accumulation
- output is image on screen

Summary

- application stage
  - what to draw and how to draw it
- geometry stage
  - computes 3D appearance of scene from viewer/camera point of view
- rasterizer stage
  - draws 2D screen image

Mathematics for Computer Graphics

- in this course we rely mostly on simple linear algebra
  - more advanced graphics techniques also rely on calculus, statistics, numerical methods
- most of polygon-based computer graphics uses vectors and points defined in 3-dimensional real Cartesian space
- most common family of transformations represented by 4x4 matrix

Vectors

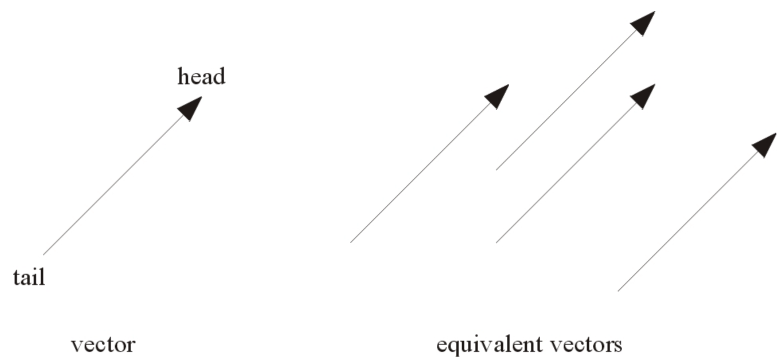
- R^3 is the 3-dimensional real Euclidean space
- vector in R^3 is a 3-tuple of real numbers

$$\vec{v} = (v_0, v_1, v_2) = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} v_0 & v_1 & v_2 \end{bmatrix}^T$$

$$\vec{w} = (1, 1, -1) = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix}^T$$

Drawing Vectors

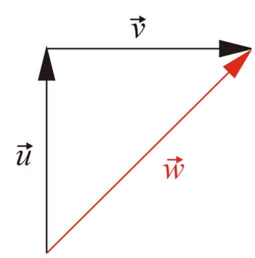
- vector has magnitude and direction
  - but no location



Vector Operations

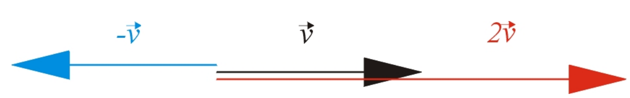
- formally only 2 operations
- vector-vector addition

$$\vec{w} = \vec{u} + \vec{v} = \begin{bmatrix} u_0 + v_0 \\ u_1 + v_1 \\ u_2 + v_2 \end{bmatrix}$$



- scalar-vector multiplication

$$a\vec{v} = \begin{bmatrix} av_0 \\ av_1 \\ av_2 \end{bmatrix}$$



Vector Properties

$(\vec{u}+\vec{v})+\vec{w}=\vec{u}+(\vec{v}+\vec{w})$	associative
$\vec{u}+\vec{v}=\vec{v}+\vec{u}$	commutative
$\vec{0}+\vec{v}=\vec{v}$	zero identity
$\vec{v}+(-\vec{v})=\vec{0}$	additive inverse
$(ab)\vec{u}=a(b\vec{u})$	associative
$(a+b)\vec{u}=a\vec{u}+b\vec{u}$	distributive
$a(\vec{u}+\vec{v})=a\vec{u}+a\vec{v}$	distributive
$1\vec{u}=\vec{u}$	multiplicative identity

Dot Product

- Hill 4.3
- in Euclidean space dot product (inner product) is defined

$$\begin{aligned} d &= \vec{a}\cdot\vec{b} \\ &= \begin{bmatrix} a_0 & a_1 & a_2 \end{bmatrix}\cdot\begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix} \\ &= a_0b_0 + a_1b_1 + a_2b_2 \end{aligned}$$

- properties

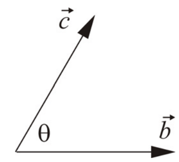
$\vec{a}\cdot\vec{b}=\vec{b}\cdot\vec{a}$	symmetry
$(\vec{a}+\vec{c})\cdot\vec{b}=\vec{a}\cdot\vec{b}+\vec{c}\cdot\vec{b}$	linearity
$(s\vec{a})\cdot\vec{b}=s(\vec{a}\cdot\vec{b})$	homogeneity
$\lvert\vec{b}\rvert^2=\vec{b}\cdot\vec{b}$	magnitude

Dot Product (cont)

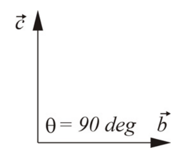
- angle between two vectors (Hill 4.3.2)

  $\vec{b} \cdot \vec{c} = |\vec{b}||\vec{c}|\cos(\vartheta)$

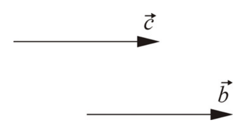
  $\therefore \cos(\vartheta) = \dfrac{\vec{b} \cdot \vec{c}}{|\vec{b}||\vec{c}|}$



- two vectors are perpendicular (orthogonal) if $\vec{b} \cdot \vec{c} = 0$



- two vectors are parallel if $\dfrac{\vec{b} \cdot \vec{c}}{|\vec{b}||\vec{c}|} = 1$



- what can you say about two vectors if their dot product is negative?

Vector Norm

- norm or magnitude of vector defined as

$$|\vec{a}| = \sqrt{\vec{a} \cdot \vec{a}} = \sqrt{{a_0}^2 + {a_1}^2 + {a_2}^2}$$

  - example

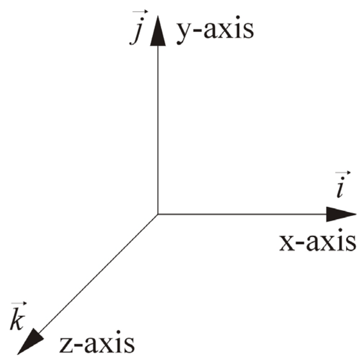
$$\left|[3 \quad 0 \quad -4]^T\right| = \sqrt{3^2 + 0^2 + (-4)^2} = 5$$

- in Euclidean space gives us notion of length or distance
- a unit vector has norm of 1
  - important!
  - to normalize a vector divide by its norm
  - example: normalize $[3\ \ 0\ \ -4]^T$

$$\frac{[3 \quad 0 \quad -4]^T}{5} = [0.6 \quad 0 \quad -0.8]^T$$

Basis

- in R^3 a basis is a set of 3 non-parallel vectors
- common to use orthonormal basis
  - basis vectors are mutually orthogonal
  - basis vectors have unit magnitude
- basis most students are familiar with



$$\vec{i} = [1 \quad 0 \quad 0]^T$$
$$\vec{j} = [0 \quad 1 \quad 0]^T$$
$$\vec{k} = [0 \quad 0 \quad 1]^T$$
$$\vec{i} \cdot \vec{j} = \vec{i} \cdot \vec{k} = \vec{j} \cdot \vec{k} = 0$$
$$|\vec{i}| = |\vec{j}| = |\vec{k}| = 1$$

Direction (cont)

- can write any vector as a linear combination of basis vectors

$$\begin{bmatrix} -3 \\ 2 \\ 7 \end{bmatrix} = -3\vec{i} + 2\vec{j} + 7\vec{k}$$

Cross Product

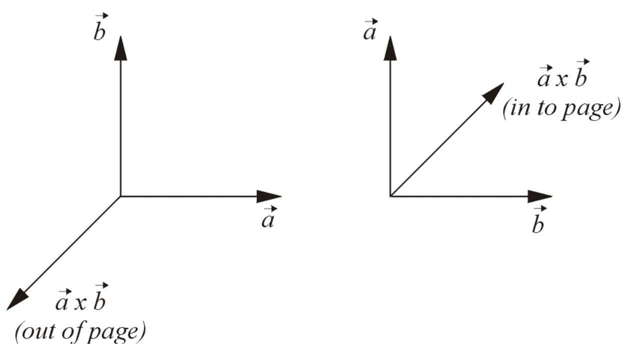
- Hill 4.4
- only defined in R^3
  - defined in terms of standard basis

$$\vec{a}\times\vec{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T \times \begin{bmatrix} b_x & b_y & b_z \end{bmatrix}^T$$

$$\begin{aligned} &(a_y b_z - a_z b_y)\vec{i} + \\ =\ &(a_z b_x - a_x b_z)\vec{j} + \\ &(a_x b_y - a_y b_x)\vec{k} \\ =\ &\vec{c} \end{aligned}$$

Cross Product (cont)

- cross product of two vectors is a vector that is orthogonal to the original two vectors
  - direction given by right hand rule



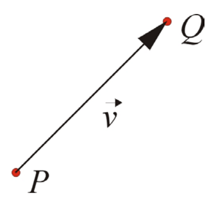
Cross Product (cont)

- properties

$$\begin{aligned} \vec{a}\times\vec{b} &= -\vec{b}\times\vec{a} && \text{antisymmetry} \\ \vec{a}\times(\vec{b}+\vec{c}) &= \vec{a}\times\vec{b}+\vec{a}\times\vec{c} && \text{linearity} \\ (s\vec{a})\times\vec{b} &= s(\vec{a}\times\vec{b}) && \text{homogeneity} \\ \vec{i}\times\vec{j} &= \vec{k} \\ \vec{j}\times\vec{k} &= \vec{i} \\ \vec{k}\times\vec{i} &= \vec{j} \end{aligned}$$

Points

- a point represents location (has zero size)
- to move between points use a vector



$$\begin{aligned} Q &= P+\vec{v} \\ &= \vec{v}+P \end{aligned}$$

- 1 operation defined with points
  - point-point subtraction (yields a vector)

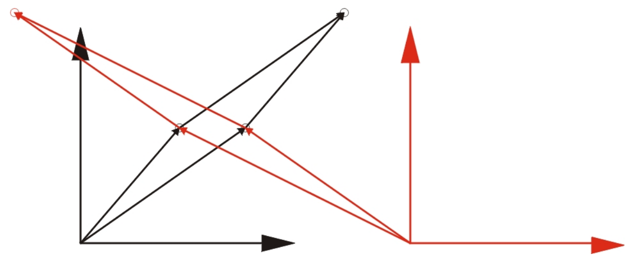
    $Q - P = \vec{v}$

- in Euclidean space distance between two points defined as

  $$\text{distance}(P,Q) = \sqrt{(P-Q)\cdot(P-Q)}$$

Why Only One Operation?

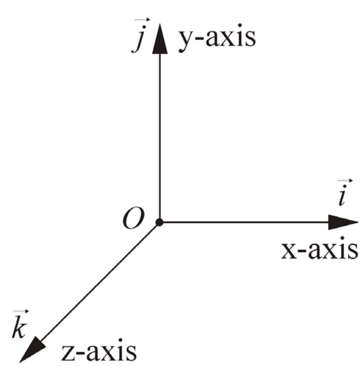
- why can we not add points?
  - not independent of coordinate frame



  - cannot multiply points by scalar for same reason
- but affine sum of points is legal (Hill 4.5.2)

Frames

- very important in computer graphics
  - you've probably been using them since high school
- a frame is a basis and a point called the origin
- most students should be familiar with the standard basis in Cartesian space



$$O = [0 \quad 0 \quad 0]^T$$
$$\vec{i} = [1 \quad 0 \quad 0]^T$$
$$\vec{j} = [0 \quad 1 \quad 0]^T$$
$$\vec{k} = [0 \quad 0 \quad 1]^T$$
$$\vec{i} \cdot \vec{j} = \vec{i} \cdot \vec{k} = \vec{j} \cdot \vec{k} = 0$$
$$|\vec{i}| = |\vec{j}| = |\vec{k}| = 1$$

Homogeneous Representation

- points and vectors are different objects but they look the same

$$P = [x \quad y \quad z]^T \qquad \vec{v} = [x \quad y \quad z]^T$$

- homogeneous representation of points and vectors distinguishes between points and vectors

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \vec{v} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

Homogeneous Representation (cont)

- the difference becomes clear when we consider the frame

$$P = [\vec{i} \quad \vec{j} \quad \vec{k} \quad O] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = x\vec{i} + y\vec{j} + z\vec{k} + O$$

$$\vec{v} = [\vec{i} \quad \vec{j} \quad \vec{k} \quad O] \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = x\vec{i} + y\vec{j} + z\vec{k}$$

- notice that
  - vector = linear combination of vectors
  - point = vector + point

Homogeneous Representation (cont)

- to go from ordinary to homogeneous coordinates
  - if the object is a point, append a 1
  - if the object is a vector, append a 0
- to go from homogeneous to ordinary coordinates
  - if the object is a point, delete the 1
    - this rule will change later on
  - if the object is a vector, delete the 0

Points in OpenGL

- OpenGL represents a point with a set of floating-point numbers called a vertex
- to draw a group of points use

```
GLfloat x0, y0, z0, x1, y1, z1, xn, yn, zn;
// assign values to x0, y0, z0, etc. here
// ...
glBegin(GL_POINTS);
glVertex3f(x0, y0, z0);      // point with coordinates (x0, y0, z0)
glVertex3f(x1, y1, z1);
// and so on…
glVertex3f(xn, yn, zn);
glEnd();
```

- every call to glVertex() sends a vertex down the geometry stage

Points in OpenGL (cont)

- many versions of glVertex()

  ```
  void glVertex3f(…)
  ```

- number (here 3) indicates number of coordinates
  - can be 2, 3, or 4
- letter (here f) indicates data type
  - can be
    - s   GLshort
    - i   GLint
    - f   GLfloat
    - d   GLdouble
- examples:

  ```
  glVertex2i(3, 4);
  glVertex3f(-1.0f, 2.0f, 3.5f);
  glVertex4d(1.2, 4.5, 3.9, 1.0);
  ```

Points in OpenGL (cont)

- glVertex() can also take an array as an argument
  - add a “v” to the function name
  - example:

    ```
    GLint one_pt[3] = { 1, 2, 3 };
    GLdouble two_pts[6];
    two_pts[0] = 1.0; two_pts[1] = 2.0; two_pts[2] = 3.0;
    two_pts[3] = 3.0; two_pts[4] = 2.0; two_pts[5] = 1.0;
    glBegin(GL_POINTS);
    glVertex3iv(one_pt);
    glVertex3dv(two_pts);          // point (1.0, 2.0, 3.0)
    glVertex3dv(two_pts+3);        // point (3.0, 2.0, 1.0)
    glEnd();
    ```

Matrices

- only need 3x3 and 4x4 matrices

$$M = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \qquad M = \begin{bmatrix} m_{00} & m_{00} & m_{00} & m_{00} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

- identity matrix

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Vector Multiplication

- can postmultiply a matrix with a column vector

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} m_{00}x + m_{01}y + m_{02}z \\ m_{10}x + m_{11}y + m_{12}z \\ m_{20}x + m_{21}y + m_{22}z \end{bmatrix}$$

Matrix Multiplication

- can multiply two 3x3 or two 4x4 matrices together
  - just treat second matrix like 3 or 4 vectors

Matrix Multiplication Properties

$$(LM)N = L(MN)$$
$$L(M + N) = LM + LN$$
$$(L + M)N = LN + MN$$
$$A(sB) = sAB$$
$$MI = IM = M$$
$$MN \neq NM$$

Transpose

- swap rows and columns
  - the transpose of M is M^T

$$M = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

Transpose Properties

$$\begin{aligned} (aM)^T &= aM^T \\ (M+N)^T &= M^T + N^T \\ (M^T)^T &= M \\ (MN)^T &= N^T M^T \end{aligned}$$

Determinant

- determinant of a matrix is a scalar value
- usually only need 2x2 and 3x3 matrix determinants
  - the determinant of M is $|M|$

$$
\begin{aligned}
|M| &= \begin{vmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{vmatrix} \\
&= m_{00}\begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} - m_{01}\begin{vmatrix} m_{10} & m_{12} \\ m_{20} & m_{22} \end{vmatrix} + m_{02}\begin{vmatrix} m_{10} & m_{11} \\ m_{20} & m_{21} \end{vmatrix} \\
&= m_{00}m_{11}m_{22} + m_{01}m_{12}m_{20} + m_{02}m_{10}m_{21} - \\
&\quad - m_{02}m_{11}m_{20} - m_{01}m_{10}m_{22} - m_{00}m_{12}m_{21}
\end{aligned}
$$

Determinant Properties

- for an n x n matrix

$$
\begin{aligned}
|M^{-1}| &= 1/|M| \\
|MN| &= |M||N| \\
|sM| &= s^n|M| \\
|M^T| &= |M|
\end{aligned}
$$

Inverse

- exists only if determinant is nonzero
- multiplicative inverse

$$MM^{-1} = M^{-1}M = I$$

- properties

$$(MN)^{-1} = N^{-1}M^{-1}$$
$$(M^T)^{-1} = (M^{-1})^T$$

- computing inverse?
  - Cramer's rule (we'll see this soon)
  - Gaussian elimination and other methods

Cofactor

- need this for Cramer's rule
- cofactor of matrix element m_{ij} is $(-1)^{i+j}$ times determinant of the matrix obtained by deleting row i and column j from M
  - example (adapted from Hill A2.1.5)

$$M = \begin{bmatrix} 2 & 0 & 6 & 0 \\ 8 & 1 & -4 & 0 \\ 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Adjoint

- adjoint is the transpose of matrix of cofactors

$$\text{cofactor}(M) = C = \begin{bmatrix} 27 & -56 & 40 & 0 \\ 30 & 14 & -10 & 0 \\ -6 & 56 & 2 & 0 \\ 0 & 0 & 0 & 294 \end{bmatrix}$$

$$\text{adjoint}(M) = C^T = \begin{bmatrix} 27 & 30 & -6 & 0 \\ -56 & 14 & 56 & 0 \\ 40 & -10 & 2 & 0 \\ 0 & 0 & 0 & 294 \end{bmatrix}$$

Cramer's Rule

- inverse of M is

$$M^{-1} = \frac{\text{adjoint}(M)}{|M|}$$

Summary

- in graphics the most commonly used concepts are
  - 2x2, 3x3, and 4x4 matrices
  - matrix-vector and matrix-matrix multiplication
  - matrix inverse
- Hill reviews these concepts (and many more) in Appendix 2

Transformations

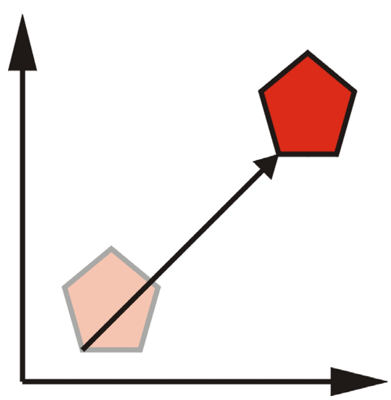
- in graphics, transformations map vectors to vectors and points to points
- transformations can be arbitrarily complex but
  - for efficiency (implementation in geometry pipeline hardware) need to restrict generality of transformations
- we will study the family of affine transformations

Affine Transformations

- transformation T is said to be affine
  - T maps vectors to vectors and points to points
  - T is a linear transformation on vectors
    - $T(a\vec{u} + b\vec{v}) = aT(\vec{u}) + bT(\vec{v})$
  - $T(P + \vec{v}) = T(P) + T(\vec{v})$
- Hill proves several properties of affine transformations (Section 5.2.7)
- only a few affine transformations
  - translation
  - scale
  - rotation
  - shear
- all can be represented by a 4x4 matrix

Translation

- moves points by a vector amount
- does not affect vectors (because vectors have no location)



Applying Translation

- translation leaves vectors unchanged

$$\vec{v} = \begin{bmatrix} x & y & z & 0 \end{bmatrix}^T$$

$$T\vec{v} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

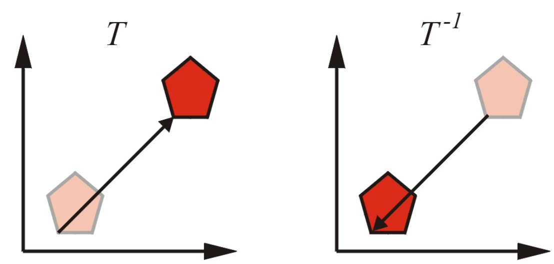
- translation moves points by a vector amount

$$P = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$$

$$TP = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

Inverse of Translation

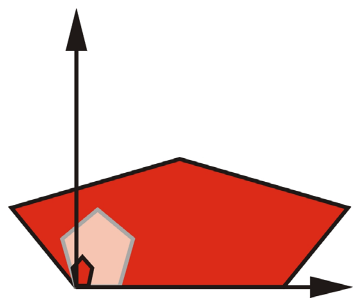
- inverse of a transformation undoes the transformation



- check that $TT^{-1} = T^{-1}T = I$

Scale

- enlarge or shrink an object
- scales objects about the x, y, and z-directions
  - origin is invariant



- if $0 < sx < 1$, then object shrinks by a factor of sx in x-direction
- if $sx > 1$, then object grows by a factor of sx in x-direction
- what if $sx < 0$? if $sx = 0$?

Inverse of Scale

- if an object is scaled by a factor of s
  - then the inverse scales by a factor of 1/s

Shear

- six basic shearing transformations

$$H_{xy}=\begin{bmatrix}1&h&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\quad H_{xz}=\begin{bmatrix}1&0&h&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\quad H_{yx}=\begin{bmatrix}1&0&0&0\\h&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

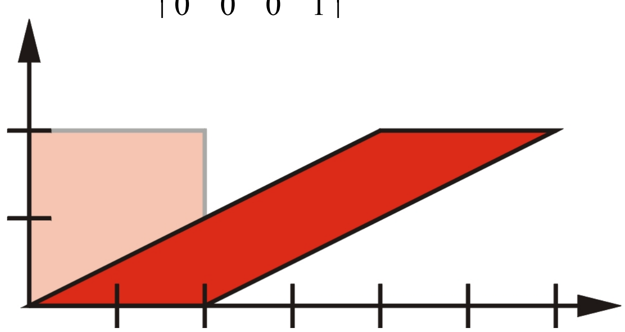
$$H_{yz}=\begin{bmatrix}1&0&0&0\\0&1&h&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\quad H_{zx}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\h&0&1&0\\0&0&0&1\end{bmatrix}\quad H_{zy}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&h&1&0\\0&0&0&1\end{bmatrix}$$

- first subscript: which coordinate is changed
- second subscript: which coordinate does the shearing

Shear Example

- suppose we have h=2 for Hxy

$$H_{xy}=\begin{bmatrix}1&2&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$



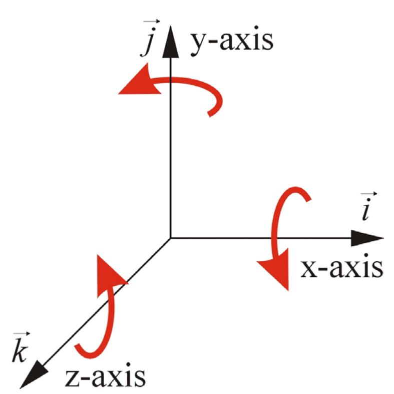
- invariant points?

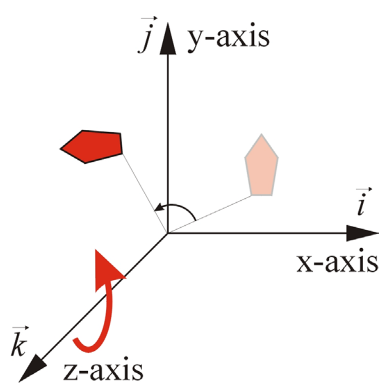
Inverse of Shear

- think about it

Rotation

- rotation about x, y, and z-axis
- points on axis of rotation are invariant
- positive angle of rotation causes a counterclockwise rotation about the axis when you look along the axis towards the origin





Rotation (cont)

- three basic rotation matrices (one for each axis)
  - check that points on the axes of rotation are invariant

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\beta) & -\sin(\beta) & 0 \\ 0 & \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\beta) & -\sin(\beta) & 0 & 0 \\ \sin(\beta) & \cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Inverse of Rotation

- rotation matrix is orthogonal
- fact: inverse of an orthogonal matrix is the transpose
  - for ANY rotation matrix: $R^{-1} = R^T$
- geometrically
  - if you rotate about an axis by β degrees then the inverse is a rotation about the same axis by -β degrees

Composition or Concatenation of Transformations

- rare to perform only one elementary transformation
- composition of affine transformations is also affine
- order transformations are applied in matters
  - matrix multiplication does not commute
  - example: translate then scale vs. scale then translate

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \quad & \quad & \quad & \quad \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \quad & \quad & \quad & \quad \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Composition or Concatenation of Transformations

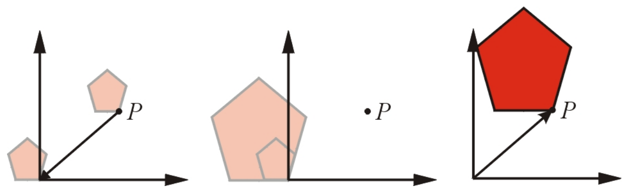
- reading left to right transformation matrices appear in reverse order
  - example: apply A then B then C

$$A\vec{a} = \vec{b}$$
$$B\vec{b} = \vec{c}$$
$$C\vec{c} = \vec{d}$$
$$\therefore \vec{d} = C(B(A\vec{a}))$$

  - overall transformation is $T = CBA$

Composition or Concatenation of Transformations (cont)

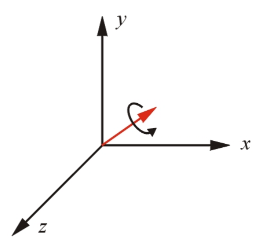
- example: scale about arbitrary point P = (px, py, pz)
  - translate P to origin
  - scale
  - translate back to original point P



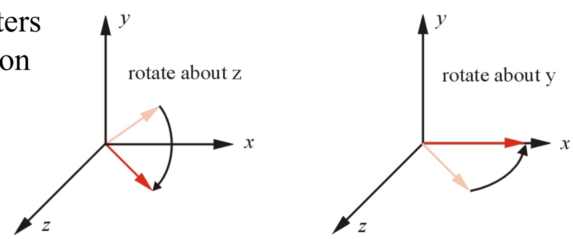
$$\begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Composition or Concatenation of Transformations (cont)

- example: rotation about arbitrary axis (hard way)
  - apply two rotations to align axis with x-axis
    - illustrated right and below
  - rotate about x-axis
  - undo first two rotations

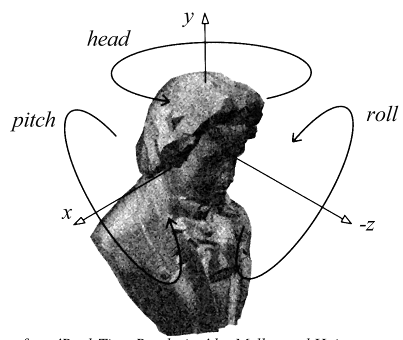


only need 3 parameters
to specify a rotation



Rotations Revisited

- rotations are common
- many different ways of specifying arbitrary rotation
- example: Euler transformations
  - 24 different Euler transformations
  - head (yaw), pitch, roll is common



from 'Real-Time Rendering' by Moller and Haines

$$R(h,p,r) = R_z(r)R_x(p)R_y(h)$$

Rotations Revisited (cont)

- Goldman (in Graphics Gems 1)
- for rotation of β degrees about an axis with normalized direction vector $\vec{u} = (u_x, u_y, u_z)$

$c =$

$s =$

$t =$

$$R = \begin{bmatrix} c + tu_x^2 & tu_xu_y - su_z & tu_xu_z + su_y & 0 \\ tu_xu_y + su_z & c + tu_y^2 & tu_yu_z - su_x & 0 \\ tu_xu_z - su_y & tu_yu_z - su_x & c + tu_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- don’t bother memorizing this

Affine Transformations and the Determinant

- the determinant of an affine transformation matrix tells you how much the transformation scales the volume of an object by
  - if an object D has volume V then applying an affine transformation M to the object produces a new object with volume |M|V
- you only need to compute the determinant of the upper left 3x3 matrix

$$|M| = \begin{vmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$= -0\begin{vmatrix} m_{01} & m_{02} & m_{03} \\ m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{vmatrix} + 0\begin{vmatrix} m_{00} & m_{02} & m_{03} \\ m_{10} & m_{12} & m_{13} \\ m_{20} & m_{22} & m_{23} \end{vmatrix} - 0\begin{vmatrix} m_{00} & m_{01} & m_{03} \\ m_{10} & m_{11} & m_{13} \\ m_{20} & m_{21} & m_{23} \end{vmatrix} + 1\begin{vmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{vmatrix}$$

Interpreting Transformations

- we have assumed that affine transformations transform points and vectors
- this is not the only interpretation
  - transformation can transform the coordinate frame
    - this is a common interpretation in OpenGL
    - we'll see this a little later
  - transformation can transform from one affine space to another affine space

Affine Transformations in OpenGL

- OpenGL maintains a stack of transformations called modelview matrix stack
- several functions modify the top-of-stack element by postmultiplying top-of-stack with a matrix

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();              // tos = I
glMultMatrixf(N);              // tos = I*N
glMultMatrixf(M);              // tos = I*N*M
glMultMatrixf(L);              // tos = I*N*M*L
glBegin(GL_POINTS);
glVertex3f(x, y, z);           // transformed by tos
glEnd();
```

- transforms vertex by N*M*L
- top-of-stack is called current transformation

Translation, Scale, and Rotation

`void glTranslatef(GLfloat x, GLfloat y, GLfloat z);`

- postmultiplies current transformation by translation matrix T(x,y,z)

`void glScalef(GLfloat x, GLfloat y, GLfloat z);`

- postmultiplies current transformation by scale matrix S(x,y,z)

`void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);`

- postmultiplies current transformation by rotation matrix corresponding to rotation of angle degrees about the axis from the origin to the point (x,y,z)

- OpenGL calls these transformations modeling transformations

Other Affine Transformations

- notice that no shear function
- must specify all 16 values of transformation matrix for “custom” transformations
  - OpenGL requires an array with the 16 elements specified like so:

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

```
GLfloat S[16];          // a scale matrix
S[0] = 3.0f;   S[4] = 0.0f;   S[8] = 0.0f;    S[12] = 0.0f;
S[1] = 0.0f;   S[5] = 5.0f;   S[9] = 0.0f;    S[13] = 0.0f;
S[2] = 0.0f;   S[6] = 0.0f;   S[10] = 7.0f;   S[14] = 0.0f;
S[3] = 0.0f;   S[7] = 0.0f;   S[11] = 0.0f;   S[15] = 1.0f;
```


Other Affine Transformations (cont)

```
void glLoadMatrixf(const GLfloat* M);
```

- sets the 16 values of current transformation matrix to those in the array M

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();            // tos = I
glLoadMatrixf(S);            // tos = I*S
```

```
void glMultMatrixf(const GLfloat* M);
```

- postmultiplies current transformation by matrix defined by M
- remember: if current matrix is C then current matrix is replaced with C*M

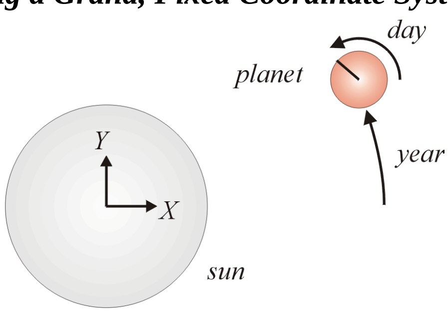
Thinking About Transformations in OpenGL

- a single grand, fixed coordinate system (often called the ‘world’)
  - matrix multiplications affect position, orientation, and size of objects
  - this is how we’ve been interpreting transformations so far
- transformations are specified in opposite order
- example: rotation followed by translation

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf( T );        // or glTranslatef()
glMultMatrixf( R );        // or glRotatef()
// draw object here…
```

- example: sun and planet

Using a Grand, Fixed Coordinate System



- assume that we know how to draw a sphere centered at the origin
- the sun is at already at the origin, we just have to draw it
- the planet needs to be transformed to its orientation and position starting from the origin
  - rotate by ‘day’ degrees about z-axis
  - translate in x-direction by radius of planet’s orbit
  - rotate by ‘year’ degrees about z-axis

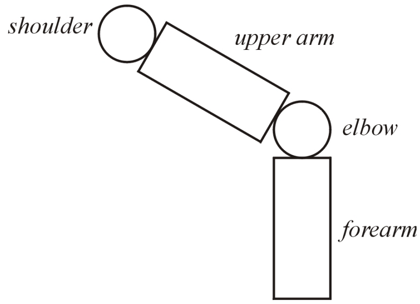
Using a Grand, Fixed Coordinate System (cont)

- in OpenGL

```
glMatrixMode(GL_MODELVIEW);
drawSun();          // draws a sphere at the origin with sun size
glRotatef(year, 0.0f, 0.0f, 1.0f);
glTranslatef(orbit, 0.0f, 0.0f);
glRotatef(day, 0.0f, 0.0f, 1.0f);
drawPlanet();       // draws a sphere at the origin with planet size
```

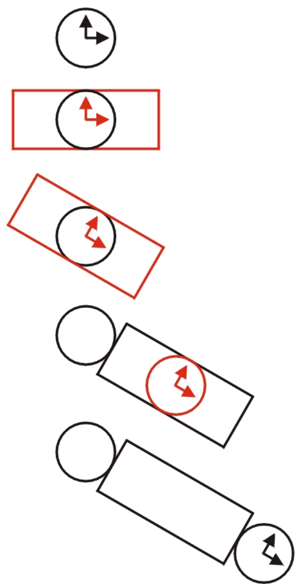

Using a Local Coordinate System

- instead of a world coordinate system, consider a coordinate system local to model
  - matrix multiplications affect position, orientation, and scale of local coordinate frame
- transformations appear in "natural" order
- especially useful for drawing articulated or hierarchical models



position of objects are related to one another

Using a Local Coordinate System (cont)



drawCircle();   // need to define this

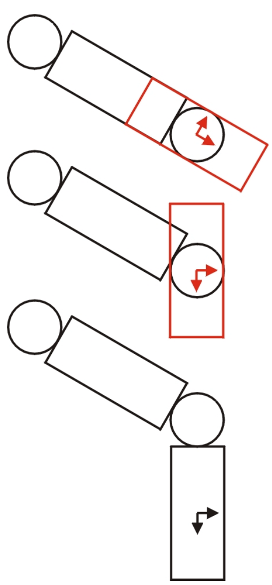
glRotatef(-30.0f, 0.0f, 0.0f, 1.0f);

glTranslatef(5.0f, 0.0f, 0.0f);
drawRectangle(); // need to define this

glTranslatef(5.0f, 0.0f, 0.0f);
drawCircle();

9

Using a Local Coordinate System (cont)



glRotatef(-60.0f, 0.0f, 0.0f, 1.0f);

glTranslatef(5.0f, 0.0f, 0.0f);
drawRectangle();

10

Using a Local Coordinate System (cont)

- beware if you use scale transformations when thinking in terms of a local coordinate system
  - glScalef() will change the scale of the coordinate axes!
- we can apply an inverse scale (after we're done with the original scale) but there is a better way
  - we can manipulate the matrix stack
    - we'll study this a little later on

11

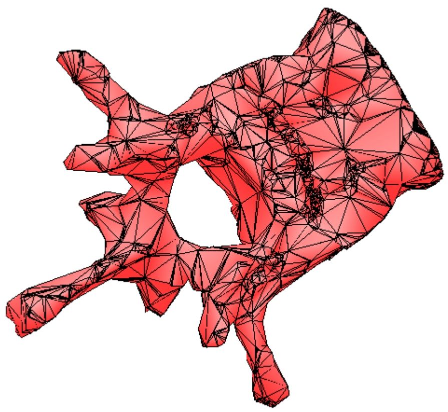
Affine Transformations Summary

- affine transformations in 3D can be represented with a 4x4 matrix
- four different types of basic affine transformations and their inverses
  - translation, scaling, shear, rotation
- when applying multiple transformations, write matrices from right to left (if you think of transforming points and vectors)
- remember how to invert a concatenation of transformations
- determinant of an affine transformation matrix tells you the factor by which the volume of an object changes when you apply the transformation to the object

12

Representation of Object Surfaces

- most common representation of objects is polygonal mesh/net
  - collection of polygons that approximate the outer surface or skin of the object



13

Representation of Object Surfaces

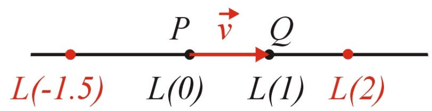
- modern hardware capable of rendering simple polygons fast
  - NVIDIA GeForce2 Ultra: 31 million polygons/s
  - PlayStation2: more than 60 million polygons/s (raw speed)
    - many factors can affect these numbers (polygon size, image size, lighting, type of shading, etc) so don’t take them at face value
- if polygons are small enough (i.e. if sufficiently large number of polygons are used) resulting images can be realistic
  - “reality is 80,000,000 polygons per frame”
    - Carpenter, Catmull, and Cook
  - 2.4 billion polygons per second
    - complexity of scenes grows faster than hardware speed
    - we’re still many years away from this number

14

Lines

- a line is 1-dimensional
    - has infinite length, but no other dimension
- a line is defined by 2 noncoincident points P and Q
    - or by a point P and a vector parallel to the line
        - any point L on a line is given by:

$$
\begin{aligned}
L &= L(t) \\
&= P + (Q - P)t \\
&= P + \vec{v}t \\
&= (1-t)P + tQ \quad \text{an affine sum of points}
\end{aligned}
$$



- L(t) is called the parametric form of a line
- can produce a finite line (called a line segment) by restricting the domain of L(t)

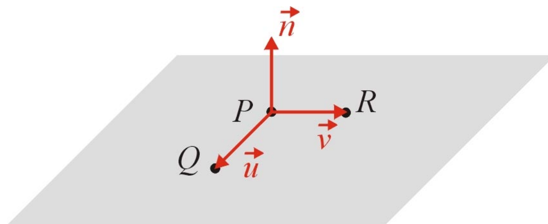
Planes

- a plane is 2-dimensional
    - has infinite length and width, but no other dimension
- a plane can be defined by 3 noncollinear points P, Q, and R in the plane
    - or by a point P and two nonparallel vectors parallel to the plane
        - any point on the plane is given by:

$$
\begin{aligned}
A &= A(s,t) \\
&= P + (Q - P)s + (R - P)t \\
&= P + \vec{u}s + \vec{v}t \\
&= (1-s-t)P + sQ + tR \quad \text{an affine sum of points}
\end{aligned}
$$

- A(s,t) is the parametric form of the plane

Planes (cont)



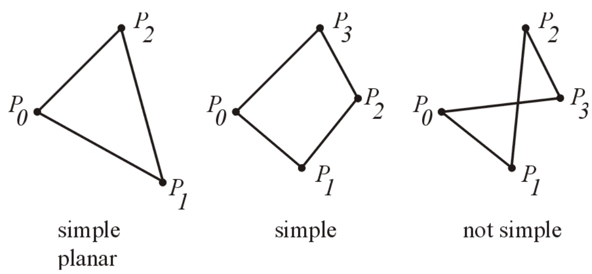
- a plane can also be defined by a point P and a vector perpendicular to the plane
  - for every point X in the plane

    $$\vec{n} \cdot (X - P) = 0$$

  - this is called the point-normal equation of a plane
- the vector $\vec{n}$ is called the normal vector to the plane
- given P, Q, and R it is easy to compute $\vec{n}$

17

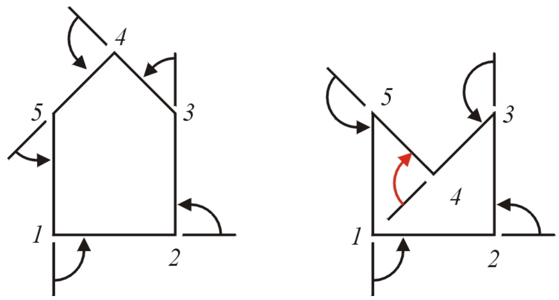
Polygons



- a polygon is an ordered set of points (vertices) with adjacent points connected by edges (line segments)
  - polygons are closed: first and last points are connected
- we will use counterclockwise convention (when looking at the outside surface or front face of the polygon)
- a polygon is simple if no two edges intersect
- a polygon is planar if it is mathematically flat (contained by a plane)

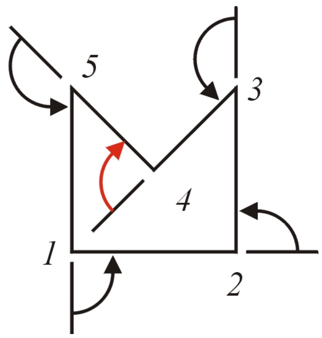
18

Polygons: Turning Angles



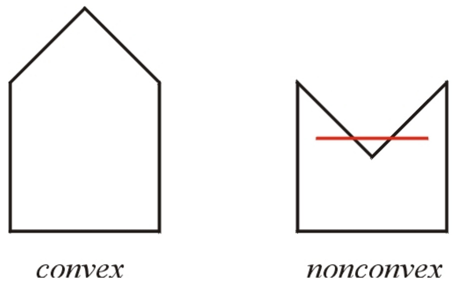
- angle by which you turn at vertex called turning angle
  - for counterclockwise ordering of vertices
    - turn left: turning angle is positive
    - turn right: turning angle is negative
- interior angle = 180 – turning angle (degrees)
  - or π - turning angle (radians)

Polygons: Turning Angles (cont)



- how do you compute the sign of the turning angle?
  - hint: at vertex P_i consider the edge vectors (P_i-P_{i-1}) and (P_{i+1} - P_i)
    - now consider the normal vector of the polygon

Polygons: Convexity

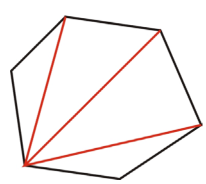


- a polygon is convex if it has no indentations
  - any two points in a convex polygon can be connected with a straight line that never leaves the polygon
  - all interior angles less than 180 degrees (π radians)
  - all turning angles have same sign
- a nonconvex polygon also called concave

21

Polygons: Winding Number

- sum of interior angles = (n-2)*180 degrees for convex polygon of n sides
  - proof:

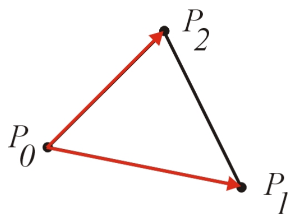


- sum of turning angles = 360 degrees for convex polygon
  - proof?
- winding number = (sum of all turning angles) / (360 degrees)
  - for a convex polygon, the winding number = 1

22

Polygonal Meshes

- many algorithms assume triangular meshes
  - hardware support
  - always convex
  - always planar
    - polygon normal vector easy to compute

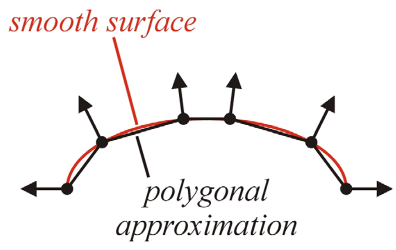


$$\vec{N} = (P_1 - P_0) \times (P_2 - P_0)$$

$$\vec{n} = \frac{\vec{N}}{|\vec{N}|}$$

Polygonal Meshes: Per Vertex Normal Vectors

- a mesh usually approximation for smooth surface
  - for shading want normal vector of smooth surface
    - store this information only at mesh vertices
      - example: normal vectors shown as arrows, vertices as dots
      - important: normal vectors are perpendicular to “true” smooth surface



- usually most convenient to store normalized (unit) normal vectors

Polygonal Meshes: Operations

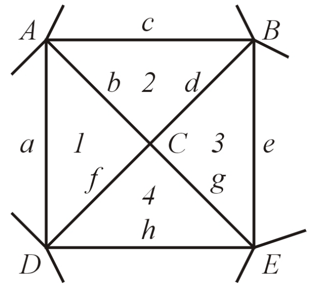
- rendering
- simplification
  - given a mesh, compute a new mesh that looks the same as the old mesh but has fewer vertices and faces
- smoothing
  - given a mesh, compute a new mesh that looks smoother than the old mesh
- animation or warping
- slicing
  - cut a mesh into two or more meshes

Polygonal Meshes: Operations (cont)

- adjacency relationship queries

Given	Find all adjacent
vertex	vertices
	edges
	faces
edge	vertices
	edges
	faces
face	vertices
	edges
	faces

Polygonal Meshes: Operations (cont)



- adjacency examples
- vertex C is adjacent to:
  - vertices A, B, D, E
  - edges b, d, f, g
  - faces 1, 2, 3, 4
- edge b is adjacent to:
  - vertices A, C
  - edges a, c, d, f, g, and others
  - faces 1, 2
- face 2 is adjacent to:
  - vertices A, B, C
  - edges b, c, d
  - faces 1, 3, and one other

Polygonal Meshes: Data Structures

- efficiency
  - memory or storage
  - time to access specific geometry
  - time to perform specific operations (e.g. answer adjacency query)
  - of rendering?
- meshes often store
  - position of vertices (geometry)
  - how the vertices are connected (topology)
  - normal direction at vertices (orientation)
    - other stuff too
      - material properties
      - texture coordinates
      - colors

Polygonal Meshes: A Simple Data Structure

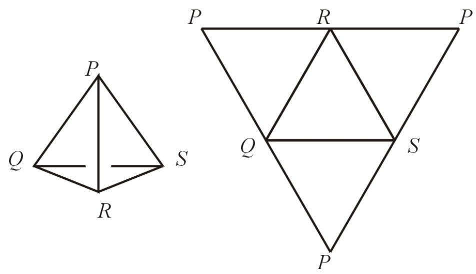
- mesh is a collection of polygons (commonly called faces)
- simplest data structure stores every face
  - example for triangle mesh

face	vertex 0	normal 0	vertex 1	normal 1	vertex 2	normal 2
0	x_{00} y_{00} z_{00}	n_{x00} n_{y00} n_{z00}	x_{01} y_{01} z_{01}	n_{x01} n_{y01} n_{z01}	x_{02} y_{02} z_{02}	n_{x02} n_{y02} n_{z02}
1	x_{10} y_{10} z_{10}	n_{x10} n_{y10} n_{z10}	x_{11} y_{11} z_{11}	n_{x11} n_{y11} n_{z11}	x_{12} y_{12} z_{12}	n_{x12} n_{y12} n_{z12}
etc						

Polygonal Meshes: A Simple Data Structure (cont)

- storage requirements
  - each vertex requires 3 floating point numbers
  - each vertex normal requires 3 floating point numbers
  - each face has 3 vertices and 3 vertex normals
    - F faces require 18*F floating point numbers
- vertices repeated
- normals repeated
- information about edges not explicit
- adjacency operations are inefficient
- rendering straightforward but inefficient

Polygonal Meshes: A Simple Data Structure Example



tetrahedron is not a smooth surface
so normal vectors are constant
for each face

face	vertices	normals
0	P_x, P_y, P_z Q_x, Q_y, Q_z R_x, R_y, R_z	a_x, a_y, a_z a_x, a_y, a_z a_x, a_y, a_z
1	R_x, R_y, R_z Q_x, Q_y, Q_z S_x, S_y, S_z	b_x, b_y, b_z b_x, b_y, b_z b_x, b_y, b_z
2	R_x, R_y, R_z S_x, S_y, S_z P_x, P_y, P_z	c_x, c_y, c_z c_x, c_y, c_z c_x, c_y, c_z
3	Q_x, Q_y, Q_z P_x, P_y, P_z S_x, S_y, S_z	d_x, d_y, d_z d_x, d_y, d_z d_x, d_y, d_z

Polygonal Meshes: Shared Vertex Data Structure

- avoid repetitive storage of vertices
  - store each vertex only once
- requires
  - vertex list to store geometric information
    - store each distinct vertex once
  - normal list to store orientation information
    - store each distinct normal vector once
    - not necessarily equal to number of vertices
  - face list to store connectivity or topological information
    - each face stores pointers or array indices or identifiers into the vertex and normal lists
- this is the mesh format Hill uses

Polygonal Meshes: Shared Vertex Data Structure (cont)

- vertex list (V vertices)

vertex	coordinates
0	x_0, y_0, z_0
1	x_1, y_1, z_1
…	
V-1	$x_{V-1}, y_{V-1}, z_{V-1}$

- normal list (N normal vectors)

normal vector	coordinates
0	nx_0, ny_0, nz_0
1	nx_1, ny_1, nz_1
…	
N-1	$nx_{N-1}, ny_{N-1}, nz_{N-1}$

Polygonal Meshes: Shared Vertex Data Structure (cont)

- face table (F faces)
  - note that numbers in vertices and normals columns are for example only

face	vertices	vertex normals
0	0, 4, 5 (array indices into vertex list)	0, 1, 2 (array indices into normal list)
1	3, 6, 9	5, 1, 9
…		
F-1	the vertices of the face in counterclockwise order	the normal vector associated with each of the vertices

Polygonal Meshes: Shared Vertex Data Structure (cont)

- storage requirements?
  - need relationship between number of vertices and number of faces
- if the mesh has no holes (e.g. not a doughnut or torus), and if every edge is shared by exactly two polygons
  - Euler's formula: V – E + F = 2
    - # vertices – # edges + # faces = 2
  - triangle mesh: 3F ≈ 2E
  - under these assumptions:

$$
\begin{aligned}
V - E + F &= 2 \\
V - 3F/2 + F &\approx 2 \\
V - F/2 &\approx 2 \\
V &\approx F/2
\end{aligned}
$$

35

Polygonal Meshes: Shared Vertex Data Structure (cont)

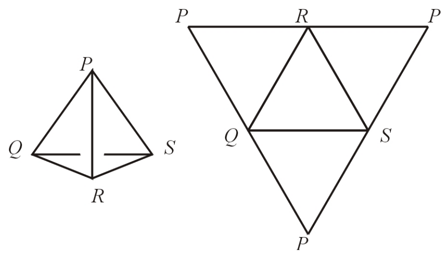
- what about the number of normal vectors?
  - impossible to say in general, but assume N ≈ V
  - the storage requirements are:

$$
\begin{aligned}
&3V + 3N + 6F \\
&\approx 3V + 3V + 6F \\
&= 6V + 6F \\
&\approx 3F + 6F \\
&= 9F
\end{aligned}
$$

- this is half the storage requirement of the first simple data structure

36

Polygonal Meshes: Shared Vertex Data Structure Example



vertex	coordinates
P	P_x, P_y, P_z
Q	Q_x, Q_y, Q_z
R	R_x, R_y, R_z
S	S_x, S_y, S_z

normal	coordinates
a	a_x, a_y, a_z
b	b_x, b_y, b_z
c	c_x, c_y, c_z
d	d_x, d_y, d_z

face	vertex	normal
0	P, Q, R	a, a, a
1	R, Q, S	b, b, b
2	R, S, P	c, c, c
3	Q, P, S	d, d, d