• Hidden Surface Removal
  • Back Face Culling
• 3D Surfaces
  • Bicubic Parametric Bezier Surface Patches
• 3D Graphics with OpenGL

Back-Face Culling

• Define one side of each polygon to be the visible side
  – That side is the outward-facing side
• Defining each polygon in the polygons array:
  – Systematically number vertices in counter-clockwise fashion as seen from outside of the object
First: Review of Vector Products

• Dot (Scalar) Product
  \[ s = A \cdot B \]
  \[ s = |A| \cdot |B| \cdot \cos(\theta) \]
  \( \theta \) is the angle between vectors A and B
  In terms of components (RH coord system):
  \[ s = Ax \cdot Bx + Ay \cdot By + Az \cdot Bz \]

\[ \theta \]
\[ B \]
\[ A \]

Cross (Vector) Product

• \( V = A \times B \), a vector
• Magnitude: \( |V| = |A| \cdot |B| \cdot \sin(\theta) \)
  \( \theta \) is angle between vectors A and B
• Direction: Given by right-hand rule
  – 1. Align fingers of right hand with first vector
  – 2. Rotate toward second
  – 3. Thumb points in direction of V
In the following diagram:

\[ V = A \times B \] would point out of the screen toward the observer

In terms of components (RH coordinate system):

\[
\begin{vmatrix}
i & j & k \\
Ax & Ay & Az \\
Bx & By & Bz \\
\end{vmatrix}
\]

(a determinant)

i, j, k are unit vectors along x,y,z axes

---

**Triple Product**

\[ A \cdot (B \times C) \]

\[
\begin{vmatrix}
Ax & Ay & Az \\
Bx & By & Bz \\
Cx & Cy & Cz \\
\end{vmatrix}
\]

(determinant)

(Components in terms of RH coord system)
Back-Face Culling

- Consider triangle with vertices 0, 1, 2
- Visible side of the triangle: 0,1,2
  - Vertices numbered in counter-clockwise order
  - Invisible side is: 0,2,1
    - (clockwise vertex ordering)

Define vector N
- Outward normal to triangle

Define Vector V0
- Vector from observer to vertex 0

Some Cases:
- N and V0 nearly parallel ($V0 \cdot N = 1$)
- Visible side of triangle 0 1 2 invisible to viewer
• Rotate triangle about side 01 by 90 degrees
  – Now N and V0 are perpendicular (V0 · N = 0)
  – Triangle is about to become visible
  – At all other points between these two orientations:
    • V0 · N is positive
    • And triangle is invisible to viewer

• Continue rotation about side 01
• Triangle becomes visible to the viewer
• 90 degrees more, N and V0 are antiparallel
  V0 · N = -1
  Triangle facing toward viewer and is visible
  – At all intermediate orientations:
    • Triangle is visible
    • And V0 · N is negative
Criterion for Invisibility

- If $V_0 \cdot N > 0$, triangle 012 is invisible
- Now place triangle 012 in an arbitrary position relative to viewer V

Outward normal $N$ is vector (cross) product of $V_{01}$ and $V_{02}$

- $V_{01}$ is vector from vertex 0 to vertex 1
- $V_{02}$ is vector from vertex 0 to vertex 2

So: $N = V_{01} \times V_{02}$

Criterion for invisibility:
$V_0 \cdot (V_{01} \times V_{02}) > 0$

But:
$V_{01} = V_1 - V_0$
$V_{02} = V_2 - V_0$
Substituting we get:
\[ V_0 \cdot [(V_1 - V_0) \times (V_2 - V_0)] > 0, \text{ invisibility} \]

Expanding:
\[
V_0 \cdot (V_1 \times V_2) - V_0 \cdot (V_1 \times V_0) - V_0 \cdot (V_0 \times V_2) \\
+ V_0 \cdot (V_0 \times V_0) > 0
\]

Last Term = 0
(Cross product of any vector with itself = 0)

Middle two terms:
Quantity inside ( ) is a vector perpendicular to \( V_0 \)
So dot product of either vector with \( V_0 \) is 0

So: \( V_0 \cdot (V_1 \times V_2) > 0 \)
- For right-handed coordinate system, triple product can be expressed as a determinant
\[
\begin{vmatrix} X_0 & Y_0 & Z_0 \\ X_1 & Y_1 & Z_1 \\ X_2 & Y_2 & Z_2 \end{vmatrix}
\]

- \((X_0,Y_0,Z_0), (X_1,Y_1,Z_1), (X_2,Y_2,Z_2)\) are viewing coordinates \((xv,yv,zv)\) of vertices 0, 1, and 2
- But viewing coordinate system is left-handed
- So sign of the determinant must be reversed
Final Criterion for Invisibility

\[
\begin{vmatrix}
X0 & Y0 & Z0 \\
X1 & Y1 & Z1 \\
X2 & Y2 & Z2 \\
\end{vmatrix} < 0
\]

- Result can be applied to any planar polygon
- Use viewing coordinates of three consecutive polygon vertices
- Could implement as a “visibility” function
  - Computes and returns value of determinant
    - Positive means visible, negative invisible

3-D Surfaces

- Explicit Representation
  \[ z = f(x,y) \]
- Plotting
  - Fix values of y and vary x
  - Gives a family of curves
    \[
    \begin{align*}
    z0 &= f(x,0) \\
    z1 &= f(x,\delta) \\
    z2 &= f(x,2\delta) \\
    z3 &= f(x,3\delta) \\
    \text{etc.}
    \end{align*}
    \]
Plotting 3D Surfaces, continued

- Then fix values of $x$ and vary $y$
- Gives another family of curves
  \[ z_0' = f(0,y) \]
  \[ z_1' = f(\delta,y) \]
  \[ z_2' = f(2\delta,y) \]
  \[ z_3' = f(3\delta,y) \]
  etc.

Plotting 3D Surfaces, continued

- Result is a wireframe that represents the surface
- Could be broken up into polygons
Parametric Representation of 3D Surfaces

- Need two parameters, say t and s
- \( x = x(t,s), \ y = y(t,s), \ z = z(t,s) \)
- both t and s vary over a range (0 to 1)
- To plot:
  - Fix values of s and for each vary t over range
    - gives one family of isoparametric curves
  - Fix values of t and for each vary s over range
    - gives another family of isoparametric curves

Cubic Bezier Curves (Review)

- In matrix form, points on curve P [P = x,y] are given in terms of parameter t and four control points P0, P1, P2, P3
- Result:
  \[
  P = a*t^3 + b*t^2 + c*t + d, \quad 0 \leq t \leq 1
  \]
  - Can be written in a more compact form:
  \[
  P = T \cdot Bg \cdot Pc
  \]
  T: row vector of parameter powers \([ \ t^3 \ t^2 \ t \ 1 \ ]\)
  Bg: the constant 4 X 4 Bezier Geometry matrix
  Pc: column vector of the control points
Bicubic Bezier Surface Patches

- Define 4-vectors S and T:

\[
S = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix}, \quad 0 \leq s \leq 1
\]

\[
T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}, \quad 0 \leq t \leq 1
\]

- Define points on surface patch \( Q(s,t) \) \( [Q = x,y,z] \) as:

\[
Q(s,t) = S \ast M_B \ast \begin{bmatrix} P_0(t) \\ P_1(t) \\ P_2(t) \\ P_3(t) \end{bmatrix}
\]

Control points \( P_0, P_1, P_2, P_3 \) are themselves parameterized by \( t \).

\( M_B \) is the Bezier Geometry Matrix we’ve seen before.

So \( P_0(t) = T \ast M_B \ast \begin{bmatrix} P_{00} \\ P_{01} \\ P_{02} \\ P_{03} \end{bmatrix} \)

Transposing:

\[
P_0(t) = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \end{bmatrix} \ast M_B^T \ast T^T
\]

Do the same for \( P_1(t), P_2(t), P_3(t) \)

Result:

\[
Q(s,t) = S \ast M_B \ast \begin{bmatrix} P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} \ast M_B^T \ast T^T
\]
A Bicubic Bezier Surface Patch

Expanding and Rearranging Terms -- x(s,t) Equation

\[ x(s,t) = (1-s)^2 \left[ x_{10} (1-t)^2 + 3 x_{11} (1-t)^2 t + 3 x_{12} (1-t) t^2 + x_{13} t^2 \right] \\
+ 3 (1-s)^2 s \left[ x_{00} (1-t)^2 + 3 x_{01} (1-t)^2 t + 3 x_{02} (1-t) t^2 + x_{03} t^2 \right] \\
+ 3(1-s)s^2 \left[ x_{20} (1-t)^2 + 3 x_{21} (1-t)^2 t + 3 x_{22} (1-t) t^2 + x_{23} t^2 \right] \\
+ s^2 \left[ x_{30} (1-t)^2 + 3 x_{31} (1-t)^2 t + 3 x_{32} (1-t) t^2 + x_{33} t^2 \right] \]

- Similar equation for y(s,t)
Plotting One Set of Isoparametric Curves

For \((s=0; s<=1; s+=\delta)\)
- Compute & store \(x(s,0), y(s,0), z(s,0)\)
- Project to screen and store --> \(xs(s,0), ys(s,0)\)
- MoveTo\((xs(s,0), ys(s,0))\)
- For \((t=0; t<=1; t+=\delta)\)
  - Compute & store \(x(s,t), y(s,t), z(s,t)\)
  - Project to screen and store --> \(xs(s,t), ys(s,t)\)
  - LineTo\((xs(s,t), ys(s,t))\)

Plotting the Other Set of Isoparametric Curves

For \((t=0; t<=1; t+=\delta)\)
- MoveTo\((xs(0,t), ys(0,t))\)
- For \((s=0; s<=1; s+=\delta)\)
  - LineTo\((xs(s,t), ys(s,t))\)
Introduction to 3D Graphics with OpenGL

3D Graphics Using OpenGL

- Building Polygon Models
- ModelView & Projection Transformations
- Quadric Surfaces
- User Interaction
- Hierarchical Modeling
- Animation
OpenGL 3D Coordinate System

- A Right-handed coordinate system
  - Viewpoint is centered at origin initially

Define 3D Polygons in OpenGL

- e.g., front face of a cube
  ```
  glBegin(GL_POLYGON)
  glVertex3f(-0.5f, 0.5f, 0.5f);
  glVertex3f(-0.5f, -0.5f, 0.5f);
  glVertex3f(0.5f, -0.5f, 0.5f);
  glVertex3f(0.5f, 0.5f, 0.5f);
  glEnd();
  ```
- need to define the other faces
Projection Transformation

- First tell OpenGL you're using the projection matrix
  `glMatrixMode(GL_PROJECTION);`
- Then initialize it to the identity matrix
  `glLoadIdentity();`
- Then define the viewing volume, for example:
  `glFrustum(-1.0, 1.0, -1.0, 1.0, 2.0, 7.0);`
  - (left, right, bottom, top, near, far)
    - near & far are positive distances, near < far
  - Viewing volume is the frustum of a pyramid
  - Used for perspective projection
  or `glOrtho(-1.0, 1.0, -1.0, 1.0, 2.0, 7.0);`
  - Viewing volume is a rectangular solid
  - for parallel projection
- For both the viewpoint (eye) is at (0,0,0)

The Viewing Volume

- Everything outside viewing volume is clipped
- Think of near plane as being window's client area
Modelview Transformation
Our cube is not visible
It lies in front of near clipping plane

![Diagram showing original 3-D Cartesian grid with near clipping plane and viewing volume.]

Positioning the Camera
- By default it’s at (0,0,0), pointing in −z direction, up direction is y-axis
- Can set the camera point
- And the “lookat” point
- And the up direction
  \[
  \text{gluLookAt}(xc, yc, zc, xa, ya, za, xu, yu, zu);
  \]
  (xc, yc, zc) coordinates of virtual camera
  (xa, ya, za) coordinates of lookat point
  (xu, yu, zu) up direction vector
- Example:
  \[
  \text{gluLookAt}(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
  \]
  camera at (2,2,2), looking at origin, z-axis is up
Modelview Transformation

- Used to perform geometric translations, rotations, scalings
- Also implements the viewing transformation
- If we don’t position the camera, we need to move our cube into the viewing volume
  
  ```
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslate(0.0f, 0.0f, -3.5f);
  - Translates cube down z-axis by 3.5 units
  ```

- OpenGL performs transformations on all vertices
- First modelview transformation
- Then projection transformation
- The two matrices are concatenated
- Resulting matrix multiplies all points in the model
OpenGL Geometric Transformations

- "Modeling" Transformations

\[
glScalef(2.0f, 2.0f, 2.0f); \quad // \text{twice as big} \\
\text{parameters: } sx, sy, sz
\]

\[
glTranslatef(2.0f, 3.5f, 1.8f); \quad // \text{move object} \\
\text{parameters: } tx, ty, tz
\]

\[
glRotateref(30.0f, 0.0f, 0.0f, 1.0f); \quad // \text{30 degrees about z-axis} \\
\text{parameters:} \\
\quad \text{angle} \\
\quad (x,y,z) -> \text{coordinates of vector about which to rotate}
\]

OpenGL Composite Transformations

- Combine transformation matrices
- Example: Rotate by 45 degrees about a line parallel to the z axis that goes through the point (xf,yf,zf) – the fixed point
  \[
  \text{glMatrixMode(GL_MODELVIEW);} \\
  \text{glLoadIdentity();} \\
  \text{glTranslatef(xf,yf,zf);} \\
  \text{glRotatef(45, 0.0,0.0,1.0);} \\
  \text{glTranslatef(-xf,-yf,-zf);} \\
  \]

- Note last transformation specified is first applied
  - Because each transformations in OpenGL is applied to present matrix by postmultiplication
Typical code for a polygon mesh model

```c
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-1.0, 1.0, -1.0, 1.0, 2.0, 7.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0f, 0.0f, -3.5f);         // translate into viewing frustum
glRotatef(30.0f, 0.0f, 0.0f, 1.0f);      // rotate about z axis by 30
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);   // set background color
glClear(GL_COLOR_BUFFER_BIT);           // clear window
glColor3f(0.0f, 0.0f, 0.0f);            // drawing color
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
   // define polygon vertices here
   // define polygon vertices here
glEnd();
```

- See 3dxform example program

The OpenGL Utility Library (GLU) and Quadric Surfaces

- Provides many modeling features
  - Quadric surfaces
    - described by quadratic equations in x,y,z
    - spheres, cylinders, disks
    - Polygon Tessellation
      - Approximating curved surfaces with polygon facets
  - Non-Uniform Rational B-Spline Curves & Surfaces (NURBS)
- Routines to facilitate setting up matrices for specific viewing orientations & projections
Modeling & Rendering a Quadric with the GLU

1. Get a pointer to a quadric object
2. Make a new quadric object
3. Set the rendering style
4. Draw the object
5. When finished, delete the object

OpenGL GLU Code to Render a Sphere

```c
GLUquadricObj *mySphere;
mySphere=gluNewQuadric();
gluQuadricDrawStyle(mySphere,GLU_FILL);
    // some other styles: GLU_POINT, GLU_LINE
    gluSphere(mySphere,1.0,12,12);
    // radius, # longitude lines, # latitude lines
```
The GLUT and Quadric Surfaces

- Many predefined quadric surface objects
  - glutWire***()
  - glutSolid***()
  - Some examples:
    - glutWireCube(size); glutSolidCube(size);
    - glutWireSphere(radius,nlongitudes,nlatitudes);
    - glutWireCone(rbase,height,nlongitudes,nlatitudes);
    - glutWireTeapot(size);
    - Lots of others
  - See cone_perspective example program

Interaction in OpenGL

- OpenGL GLUT Callback Functions
  - GLUT’s version of event/message handling
  - Programmer specifies function to be called by OS in response to different events
  - Specify the function by using glut***Func(ftn)
    - We’ve already seen glutDisplayFunc(disp_ftn)
    - disp_ftn called when client area needs to be repainted
      - Like Windows response to WM_PAINT messages
  - All GLUT callback functions work like MFC On***() event handler functions
Some Other GLUT Callbacks

- `glutReshapeFunc(ftn(width,height))`
  - Identifies function ftn() invoked when user changes size of window
    - height & width of new window returned to ftn()

- `glutKeyboardFunc(ftn(key,x,y))`
  - Identifies function ftn() invoked when user presses a keyboard key
  - Character code (key) and position of mouse cursor (x,y) returned to ftn()

- `glutSpecialFunction(ftn(key,x,y))`
  - For special keys such as function & arrow keys

Mouse Callbacks

- `glutMouseFunc(ftn(button, state, x, y))`
  - Identifies function ftn() called when mouse events occur
    - Button presses or releases
    - Position (x,y) of mouse cursor returned
    - Also the state (GLUT_UP or GLUT_DOWN)
    - Also which button
      - GLUT_LEFT_BUTTON, GLUT_RIGHT_BUTTON, or GLUT_MIDDLE_BUTTON
Mouse Motion

- Move event: when mouse moves with a button pressed –
  - glutMotionFunctionFunc(ftn(x,y))
    - ftn(x,y) called when there's a move event
    - Position (x,y) of mouse cursor returned
- Passive motion event: when mouse moves with no button pressed
  - glutPassiveMotionFunctionFunc(ftn(x,y))
    - ftn(x,y) called when there's a passive motion event
    - Position (x,y) of mouse cursor returned

GLUT Menus

- Can create popup menus and add menu items with:
  - glutCreateMenu (menu-ftn(ID))
    - Menu-ftn(ID) is callback function called when user selects an item from the menu
    - ID identifies which item was chosen
  - glutAddMenuEntry(name, ID_value)
    - Adds an entry with name displayed to current menu
    - ID_value returned to menu_ftn() callback
  - glutAttachMenu(button)
    - Attaches current menu to specified mouse button
    - When that button is pressed, menu pops up
Hierarchical Models

- In many applications the parts of a model depend on each other
- Often the parts are arranged in a hierarchy
  - Represent as a tree data structure
  - Transformations applied to parts in parent nodes are also applied to parts in child nodes
  - Simple example: a robot arm
    - Base, lower arm, and upper arm
    - Base rotates \( \rightarrow \) lower and upper arm also rotate
    - Lower arm rotates \( \rightarrow \) upper arm also rotates

Simple Robot Arm Hierarchical Model
Use of Matrix Stacks in OpenGL to Implement Hierarchies

- Matrix stacks store projection & model-view matrices
- Push and pop matrices with:
  - `glPushMatrix();`
  - `glPopMatrix();`
- Can use to position entire object while also preserving it for drawing other objects
- Use in conjunction with geometrical transformations
- Example: Robot program

OpenGL Hierarchical Models

- Set up a hierarchical representation of scene (a tree)
- Each object is specified in its own modeling coordinate system
- Traverse tree and apply transformations to bring objects into world coordinate system
- Traversal rule:
  - Every time we go to the left at a node with another unvisited right child, do a push
  - Every time we return to that node, do a pop
  - Do a pop at the end so number of pushes & pops are the same
GLUT Animation

- Simple method is to use an “idle” callback
  - Called whenever window’s event queue is empty
  - Could be used to update display with the next frame of the animation
  - Identify the idle function with:
    • `glutIdleFunc(idle_ftn())`
  - Simple Example:
    ```
    void idle_ftn()
    {
      glutPostRedisplay();
    }
    ```
    - Posts message to event queue that client area needs to be repainted
    - Causes display callback function to be invoked
    - Effectively displays next frame of animation

Double Buffering

- Use two display buffers
- Front buffer is displayed by display hardware
- Application draws into back buffer
- Swap buffers after new frame is drawn into back buffer
- Implies only one access to display hardware per frame
- Eliminates flicker
- In OpenGL, implement by replacing `glFlush()` with `glutSwapBuffers()` in display callback
- In initialization function, must use:
  ```
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  ```
- See `anim_square` & `cone_anim` examples