3D Graphics with OpenGL: Hierarchical Models, Interaction, Animation

Z-Buffer Hidden Surface Removal

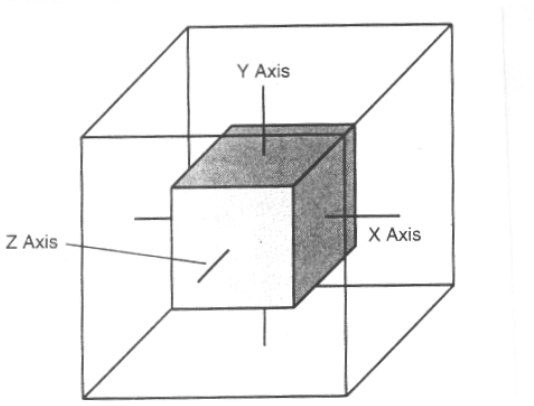
Illumination and Shading

3D Graphics Using OpenGL

- OpenGL 3D Coordinate System
- 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



Defining 3D Polygons in OpenGL

- e.g., front face of a cube centered at origin

```
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

Model-View and Projection Transformations

- Each vertex in model passes through two transformations
  - Defined by two 4X4 matrices
    - Model-view and projection matrices
  - Model-view matrix
    - Position objects relative to camera
  - Projection matrix
    - Forms the image through projection to a projection plane and helps with clipping

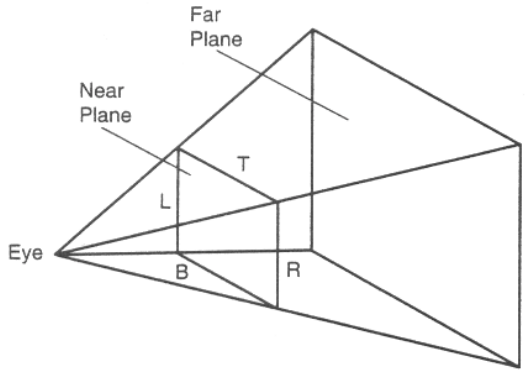
Projection Transformation

- 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

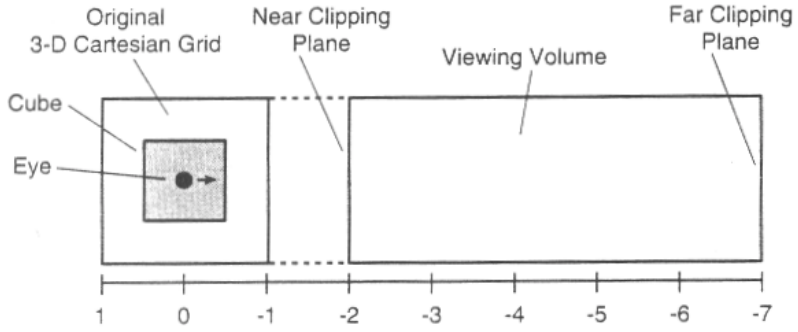
- Everything outside viewing volume is clipped
- Think of near plane as being window's client area



Modelview Transformation

Our cube as specified is not visible

It lies in front of near clipping plane



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

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:

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); // twice as big
  parameters: sx, sy, sz

glTranslatef(2.0f, 3.5f, 1.8f); // move object
  parameters: tx, ty, tz

glRotatef(30.0f, 0.0f, 0.0f, 1.0f); // 30 degrees about z-axis
  parameters:
  - angle
  - (x,y,z) -> 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

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslate(xf,yf,zf);
glRotate(45, 0.0,0.0,1.0);
glTranslate(-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

```
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
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 Tesselation
      - 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

```
GLUquadricObj *mySphere
mySphere=gluNewQuadric();
    //create the new sphere object
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

- An alternative to GLU Quadrics with many more 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

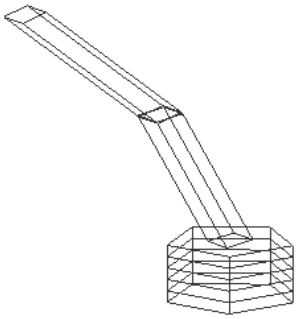
GLUT Menus

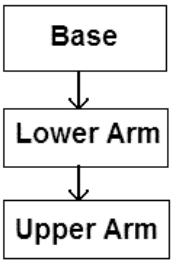
- Can create popup menus and add menu items with:
  - glutCreateMenu (menu-ftn(ID))
    - Menu-ftn(ID) is callback function called when user selects an item from the menu
    - ID identifies which item was chosen
  - glutAddMenuEntry(name, ID_value)
    - Adds an entry with name displayed to current 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 ➔ lower and upper arm also rotate
    - Lower arm rotates ➔ upper arm also rotates

Simple Robot Arm Hierarchical Model



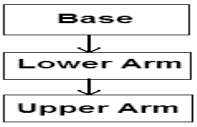


Use of Matrix Stacks in OpenGL to Implement Hierarchies

- 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
- Buffers are swapped 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

General Hidden Surface Removal

Z-Buffer Hidden Surface Removal Algorithm

Hidden Surface Removal

- Determination of surfaces not visible to the viewer
- Many different techniques
  - Back face culling, for single objects only
  - Z-Buffer
  - Depth Sort

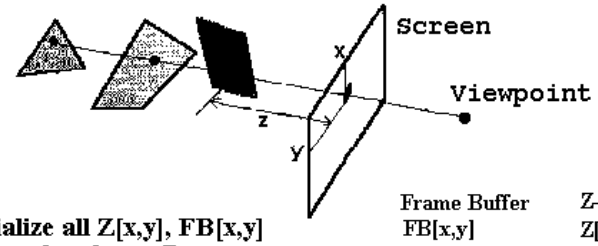
Z-Buffer Hidden Surface Removal Algorithm

- Basic Idea:
  - At a given pixel we want to plot color of closest surface that projects to that pixel
  - We're looking for minimum zv
  - Use a buffer (array) parallel to the frame buffer
    - Store minimum values of zv
    - One for every pixel
    - Called the Z-Buffer

Z-Buffer Technique Applied to a Polygon Mesh

- Initialize Z-Buffer and Frame Buffer
- Look at each polygon
  - Look at each point (xs,ys) projected to by the polygon
  - Compute zv of the point on the polygon
    - If zv is closer than value stored at [x,y] in Z-Buffer
      - Replace value in Z-Buffer with zv
      - Update corresponding element in frame buffer with color of the polygon

Z-Buffer Hidden Surface Removal



Initialize all Z[x,y], FB[x,y]
For each polygon P
  For each pixel (x,y) covered by P
    if (z < Z[x,y])
      Z[x,y] = z
      FB[x,y] = color of P

Frame Buffer FB[x,y]

b	b	b	b	b
b	b	b	b	b
b	b	b	b	b

b = background color initially

Z-Buffer Z[x,y]

i	i	i	i	i
i	i	i	i	i
i	i	i	i	i

i = infinity (largest value) initially

When completed, each position (pixel) in the frame buffer will contain the color of the closest polygon, and each position in the Z-buffer will contain the distance to the intersection with that polygon.

Z-Buffer Algorithm Applied to Convex Polygons

Data Structures:

- For each polygon
  - Polygon color
  - Polygon vertex coordinates: xs, ys, and zv
    - Note mixed coordinates
  - Edge table (xmin, ymin, zmin, xmax, ymax, zmax)
  - Active edge list (AEL) with active edges intersected by current scanline sorted on xs
    - (See scanline polygon fill notes)

Other Data Structures

- Frame Buffer FBuf[x][y]
  - Will store the color of each pixel (x,y)
- Z-Buffer ZBuf[x][y]
  - Will store the zv distance of point on closest polygon that projects to pixel (x,y) on screen
- Initialize each element of FBuf[][] to background color
- Initialize each element of ZBuf[][] to infinity (largest possible value)

The Algorithm

For each polygon

  For each scanline y spanning the polygon

    Get left & right active edges from AEL

    Get x,z coordinates of endpoints from edge table

    Compute scanline/edge intersection pts (xL,zL,xR,zR)

      (Use x-y & z-y interpolation)

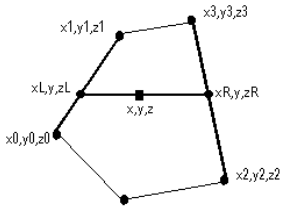
    For (x=xL to xR)

      Compute z by z-x interpol.

      If (z < ZBuf[x,y])

        ZBuf[x,y] = z

        FBuf[x,y] = polygon color



Double Interpolation

- We know (from Edge Table):

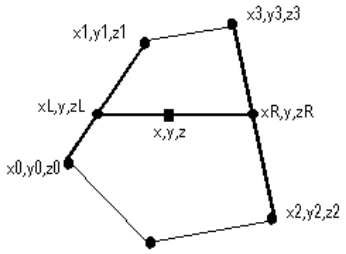
  lower/upper vertices of left active edge:

    (x0,y0,z0) and (x1,y1,z1)

  lower/upper vertices of right active edge:

    (x2,y2,z2) and (x3,y3,z3)

- We also know y of current scanline



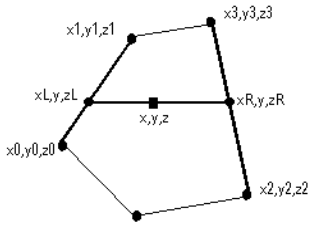
x-y Interpolation:

- Find x coords of intersection pts (xL,xR)
- Left Edge:

  xL-x0 y-y0

  ------- = -------

  x1-x0 y1-y0



- Solving for xL:

  xL = (x1-x0)*(y-y0)/(y1-y0) + x0
- Similarly for xR on right edge:

  xR = (x3-x2)*(y-y2)/(y3-y2) + x2

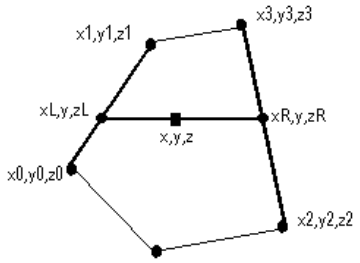
z-y Interpolation

- Find z coordinates of intersection points of scan line (y) with left and right edges
- Done the same way as x-y interpolation
- x coordinates replaced by z coordinates
- Results:
- zL = (z1-z0)*(y-y0)/(y1-y0) + z0
- zR = (z3-z2)*(y-y2)/(y3-y2) + z2

z-x Interpolation

- Find z value on polygon at pixel x on current scanline (y)
- Interpolate between the left and right edge intersection points:

$$\frac{z - zL}{zR - zL} = \frac{x - xL}{xR - xL}$$



Solving for z:

$$z = (zR - zL)*(x - xL)/(xR - xL) + zL$$

Speeding up the Algorithm

- Do interpolations incrementally
  - Get new values from old values by adding correct increments
  - xL,xR,zL,zR (in the outer loop)
  - z (in the inner loop)
  - Avoids multiplications and divisions inside algorithm loops

Z-Buffer Performance

- Outer loop repeats for each polygon
- Complex scenes have more polygons
  - So complex scenes should be slower
- But:-- More polygons usually means smaller polygons
  - So inner loops (y and x) are faster
- For most real scenes, performance is approximately independent of scene complexity

Disadvantage of Z-Buffer

- Memory requirements
- Z-Buffer is at least as big as the frame buffer
- For best results, need floating point or doubles for z values
- Example 1000 X 1000 resolution screen
  - Assume 8 bytes to store a double
  - 8 Megabytes required for Z-Buffer
- But memory has become cheap
- Z-Buffer used very commonly now
- Often implemented in hardware