# 5. Polygon Mesh Rendering

5.1 The Polygon Mesh Rendering Pipeline

5.2 Back Face Culling

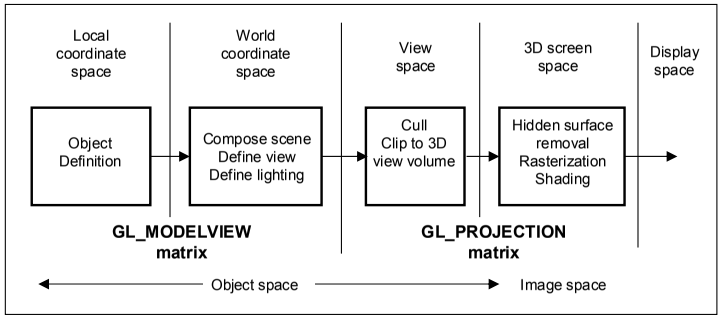
5.3 View Frustum Culling

5.4 Hidden Surface Removal



## 5.1 The Polygon Mesh Rendering Pipeline

- **Reference: Chapter 6 of the book "3D Games"**
- Rendering Pipeline



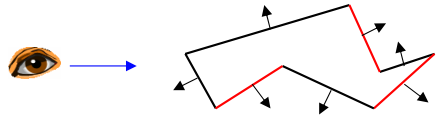



## OpenGL Geometric Transformations

- Modeling: glTranslate*, glRotate*, glScale
- Scene composition (instancing): glTranslate*, glRotate*, glScale*
- Viewing
  - position, orientation: glTranslate*, glRotate*, gluLookAt
  - Projection
    - glOrtho - parallel (orthographic)
    - glFrustum – arbitrary general 3D, including symmetric and asymmetric perspective
    - gluPerspective – symmetric perspective
  - Game viewing:
    - First person: game camera view = player view
    - Third person: game camera fixed offset from player (follow)



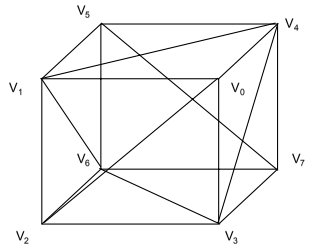
## 5.2 Back Face Culling



- Back Face Culling
  - Faces of 3D polyhedra modeled with polygon mesh have an "outside" and an "inside" face. On avg. ½ polys = "inside".
  - For closed polyhedron (no "holes") "inside" faces never visible – always obscured by an "outside" face
  - Principle: eliminate back-facing polygons with a simple test before they reach the complex steps of clipping, rasterization, Hidden Surface Removal (HSR), and shading in pipeline
  - Note: a scene with 1 convex polyhedron can be rendered using only back-face cull then rasterize without HSR

# Back Face Culling (cont'd)

- "inside" and "outside" determined by **winding order**
  - Clockwise (CW) or Counter-Clockwise (CCW) ordering of vertices of a polygon
  - OpenGL = CCW is default




```
glBegin( GL_TRIANGLES );
  glVertex3f( v0 );
  glVertex3f( v1 );
  glVertex3f( v2 );

  glVertex3f( v2 );
  glVertex3f( v3 );
  glVertex3f( v0 );

  glVertex3f( v4 );
  glVertex3f( v0 );
  glVertex3f( v3 );

  glVertex3f( v3 );
  glVertex3f( v7 );
  glVertex3f( v4 );

  glVertex3f( v0 );
  glVertex3f( v4 );
  glVertex3f( v1 );

  glVertex3f( v5 );
  glVertex3f( v1 );
  glVertex3f( v4 );

  glVertex3f( v5 );
  glVertex3f( v4 );
  glVertex3f( v7 );

  glVertex3f( v7 );
  glVertex3f( v6 );
  glVertex3f( v5 );

  glVertex3f( v1 );
  glVertex3f( v5 );
  glVertex3f( v6 );

  glVertex3f( v6 );
  glVertex3f( v2 );
  glVertex3f( v1 );
glEnd();
```

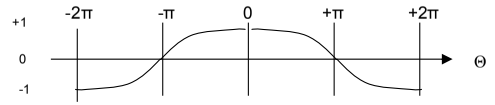


# Back Face Culling (cont'd)

- Mathematics of back-face removal
  - Face normal vector, **n** – must be correct direction vector, but need not be normalized (unit length). If using lit surfaces, normal vector part of face data structure (else, compute).
  - Line of sight vector, **v** – direction from any point on face to the viewpoint (eye point), e.g., any vertex or center, need not be normalized (note: sqrt() is slow):
  - Face is back face if angle φ between **n** and **v** has the property

    -90 <= φ <= +90 degrees ⇔ cos φ >= 0

    ⇔ **v** • **n** >= 0 (since **v** • **n** = |**v**| |**n**| cos φ )





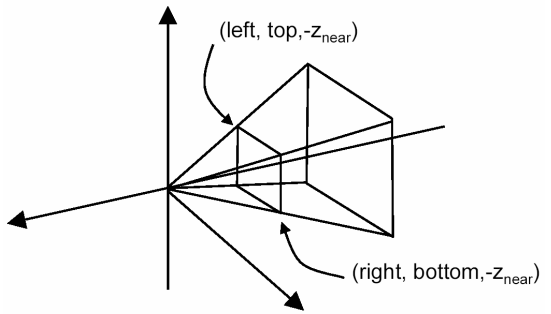

# Back Face Culling (cont'd)

- OpenGL implementation of back-face cull
  - Set cull mode, `void glCullFace( GLenum mode )`
    mode = `GL_FRONT`, `GL_BACK` (default), or
    `GL_FRONT_AND_BACK`
  - Enable culling, `void glEnable( GLenum cap )`
    cap = `GL_CULL_FACE`
- Usage
  - Normal: enable back face (use front face for some modelers)
  - Special cases, e.g., transparent objects.



# 5.3 View Frustum Culling

- View volume corners specified by the points of a frustum:
  `glFrustum(left, right, bottom, top, znear, zfar)`



## View Frustum Culling (cont'd)

- Principle: discard polygons that lie outside view frustum with a simple test before they reach the complex steps of clipping, rasterization, HSR, and shading in pipeline.
- For most games `znear` clip distance very near 0, so simplify view frustum to pyramid bounded by 2 side, 2 top, and far clipping planes (determined by 5 points)
- Test: substitute vertex to be tested into clip plane's plane equation to obtain vertex distance value, $dist = ax + by + cz + d$ $dist > 0$ ➔ outside frustum (for outward pointing plane normal)
- VFC must be implemented by application, no automatic OpenGL function.
  - Speed up by clipping bounding volumes



## 5.4 Hidden Surface Removal (HSR)

### Z-Buffer algorithm

- Algorithm: triply nested loop that interpolates *x,y,z* and **c**
- Initialization (once per frame, glClear())

  For all *x,y* zBuffer[x,y] = maximum_depth
- For each polygon

  Construct edge list from the vertices, EdgeList[] : (*x,y,z*,**c**) for each edge endpoint

  For y = ymin to ymax

  For each segment in EdgeList[y]

  Get segment values Xleft, Xright, Zleft, Zright, Cleft, Cright

  For Xleft to Xright

  Linearly interpolate z and **c** between Zleft, Zright, Cleft, Cright

  If z < zBuffer[x,y] then

  zBuffer[x,y] = z; colorBuffer[x,y] = c



## Z-Buffer Algorithm

- **Motivation**
  - Image space (screen space) algorithm
  - Reasonably fast
  - De facto standard HSR algorithm. Used by nearly all graphics HW
- **Advantages**
  - Processing is O(*N***M*) (*N*= #polys, *M* = # pixels/poly) compare with algorithms that rely on sorting, O($N^2$) or O(*N* log*N*)
  - Independent of polygon processing order (no sorting or priority)
  - Independent of model: requires (*x,y,z*) screen space pixels regardless of how they are generated
- **Disadvantages**
  - Inefficient because many pixels are rendered and later obscured
  - Requires very FAST depth buffer memory same size as color buffer
  - Z precision errors – depends on # depth bits, typically 16, 24, or 32

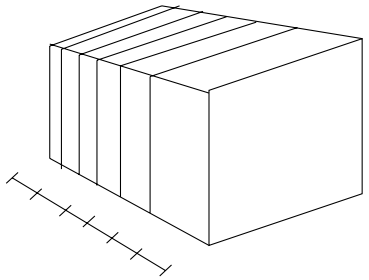


## Z-Buffer Algorithm (cont'd)

- Problem: Z-Buffer Memory requirements
  - Example: 1280 x 1024, $10^6$ polys @ ~ 64 pixels/poly, and 30 Hz rendering rate
    - Z-Buffer processing time = 33 msec – clear time = 32
    - # pixels Read/Modify = 64 x 106
    - sec/pixel = 32 x $10^{-3}$ seconds / 64 x $10^6$ pixels = .5 x $10^{-9}$ = ½ nanosec/pixel !!!
- Problem: z value precision
  - View projection mapping from $z_{World}$ to $z_{Screen}$ to $z_{Buffer}$ transform to $z_{Screen}$ [ 0.0, 1.0 ] then to $z_{Buffer}$ [ 0, 65535 ]

# Z-Buffer Algorithm (cont'd)

- Mapping is non-linear due to perspective division
- All $z_{World}$ values that map to same z Screen within $1/2^n$ Bits are assigned same z buffer value, e.g. if $z_{W2} < z_{W1}$ but $z_{B2} = z_{B1}$ 1st poly's color is written to color buffer
- But, if in next frame $z_{B2} = z_{B1} - 1$, then 2nd poly color is written to color buffer





# The Polygon rendering Pipeline - Review

- Rendering pipeline review
- 3D viewing
- Back-face culling
  - OpenGL implementation
  - Enable/disable for scene components when appropriate
- View Frustum and VFC
  - Computation: compare with frustum plane equations
  - Implemented by Game Engine
- Z-buffer
  - Implemented on graphics card
  - Deficiencies
    - Writes pixels that may be overwritten
    - Memory speed requirements (calculate nsec/pixel)
    - Integer buffer precision errors
  - Alternative methods can be implemented in Game Engine