Introduction to 3D Graphics

John E. Laird
Basic Issues

- Given an internal model of a 3D world, with textures and light sources how do you project it on the screen from any perspective fast.
  - Restrictions on geometry
  - Restrictions on viewing perspective
  - Lots of algorithms

- Questions
  - How do I draw polygons on the screens?
  - Which polygons should I draw?
  - How should I rasterize them (for lighting and texture)?
Overview: Simple 3D graphics

- 3D space
- Points, Lines, Polygons, and Objects in 3D
- Coordinate Systems
- Translation, Scaling, and Rotation in 3D
- Projections
- Solid Modeling
- Hidden-surface removal
- Z-Buffering
3D Space

Right-handed system

Left-handed system
Points, Lines, Polygons

- **Points**: x, y, z
- **Line**: two points
- **Polygon**: list of vertices, color/texture
Objects

- Made up of sets of polygons
  - Which are made up of lines
  - Which are made of points
  - No curved surfaces
- Just a “shell”
  - Not a solid object
- Everything is a set of points
  - *In local coordinate system*
Object Transformations

- Since all objects are just sets of points, we just need to translate, scale, rotate the points.
- To manipulate a 3D point, use matrix multiplication.
- Translation:

\[
\begin{bmatrix}
  x' \\
  y'
  z'
  1
\end{bmatrix} = \begin{bmatrix}
  x & y & z & 1
\end{bmatrix}
\begin{bmatrix}
  1 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 \\
  dx & dy & dz & 1 \\
\end{bmatrix}
\]
Scaling

**Constant Axis Scaling**

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{bmatrix} = \begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix} \begin{bmatrix}
  s & 0 & 0 & 0 \\
  0 & s & 0 & 0 \\
  0 & 0 & s & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

**Variable Axis Scaling**

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
\end{bmatrix} = \begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix} \begin{bmatrix}
  sx & 0 & 0 & 0 \\
  0 & sy & 0 & 0 \\
  0 & 0 & sz & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]
Rotation

- **Parallel to x-axis**
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} = \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix} \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & \cos r & \sin r & 0 \\
  0 & -\sin r & \cos r & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

- **Parallel to y-axis**
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} = \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix} \begin{bmatrix}
  \cos r & 0 & -\sin r & 0 \\
  0 & 1 & 0 & 0 \\
  \sin r & 0 & \cos r & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]

- **Parallel to z-axis**
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} = \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix} \begin{bmatrix}
  \cos r & \sin r & 0 & 0 \\
  -\sin r & \cos r & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \]
Three Coordinate Systems

- World-centered: Where objects are in the world
- Object-centered: Relative to position of object
- View-centered: Relative to the position of viewer

Simplest case is viewing down z-axis
Projections

- Mapping a 3D object onto a 2D viewing surface

- Perspective projection
- Parallel projection

View Plane
Projections

- **Parallel**
  - If viewing down z-axis, just discard z component

- **Perspective**
  - If viewing down z-axis, scale points based on distance.
    - $x_{\text{screen}} = x / z$
    - $y_{\text{screen}} = y / z$
Projections

- Usually not viewing down center of z axis.
- Usually $x = 0$ and $y = 0$ at bottom left.
- Correct by adding $1/2$ screen size
  - $x_{\text{screen}} = x/z + 1/2$ screen width
  - $y_{\text{screen}} = y/z + 1/2$ screen height

- To get perspective right, need to know field of view, distance to screen, aspect ratio.
  - Often add scaling factor to get it to look right
  - $x_{\text{screen}} = x \text{ scale} / z + 1/2$ screen width
Field of View

- To simulate human vision:
  - 110-120 degrees horizontally
  - < 90 vertically

- Think of the viewing pyramid or frustum
Clipping

near clip plane

Far clip plane

Viewing frustum

View plane
Drawing the Surface

- Split triangles and fill in as described earlier
Solid Modeling

- Which surfaces should be drawn?
  - Object space methods
    - Hidden Surface Removal
    - Painters Algorithm
    - BSP Trees
  - Image space methods
    - Z-Buffering
    - Ray Casting
Hidden Surface Removal

- **Step 1:**
  - Remove all polygons outside of viewing frustum

- **Step 2:**
  - Remove all polygons that are facing away from the viewer
  - If the dot product of the view vector and the surface normal is $\geq 90$ degrees, it is facing away.
    - Surface normal = cross product of two co-planar edges.
    - View vector from normal point to viewpoint

- **Step 3:**
  - Draw the visible faces in an order so the object looks right.
Testing if Surface is Visible

\[ N = U \times V \]
Painter’s Algorithm

- **Basic idea**
  - Sort surfaces and then draw so looks right.
  - If all surface are parallel to view plane, sort based on distance to viewer, and draw from back to front.
  - Worst-case is \( O(n^2) \)
  - Otherwise, have five tests applied to each pair to sort.
  - Order tests by cheap to expensive

- **Why can’t come up with order (max, min, mean)?**
Test 1: X overlap

- If the x extents of two polygons do not overlap, then order doesn’t matter and go to next pair.
- If x extents overlap, goto test 2.
Test 2: Y Overlap

- If the y extents of two polygons do not overlap, then order doesn’t matter.
- If y extents overlap, goto test 3.
Tests 3 & 4

- Extend polygons to be a cutting plane
  - If a polygon can be contained within the cutting plane of the other, that polygon should be drawn first.
  - If neither can be contained, go to step 5.

View direction

Polygon 1 is completely on the far side of the plane that extends from polygon 2.
Test 5

- Only needs to be considered if have concave polygons.
How to make it easier

- Use convex objects
- Avoid long objects (like walls) that can overlap each other in multiple dimensions
- Avoid intersecting objects and polygons
Z-buffer

- Z-buffer holds the z-coordinate of every pixel
  - Usually 16 or 32-bits/pixel
- Initialize all values to maximum depth
- Compute the z value of every point of every non-back facing polygon
  - Not too hard if all polygons are triangles or rectangles
  - Do this during the filling of the triangles
- If z of point < z in Z-buffer, save the color of the current point and update Z-buffer
  - Otherwise, throw away point and move on
- In all 3D hardware now
Ray Tracing

- Technique that mimics physical processes of light
- Extremely computationally intensive, but beautiful
  - Hidden surface removal
  - Transparency
  - Reflections
  - Refraction
  - Ambient lighting
  - Point source lighting
  - Shadows
Shading

- Compute lighting based on angle of light on polygon surface.
Gouraud Shading

- Compute shading for each pixel by averaging shading based on distance and shading of vertices.
Transparency

- Use an extra set of bits to determine transparency
  - Alpha
  - Blend present value of the color buffer with new values.
Texture Mapping

- **Apply stored bit map to a surface**

- **Average texels covered by pixel image**
3D Collision Detection

- Can’t be done in image space
- Usually use hierarchical approach
  - First find objects in same 3D cells
  - Second test for overlaps in bounding sphere or box
  - Third
    - Good enough!
    - Check for polygon collisions
- Accurate 3D collision detection is very expensive