Introduction to Modelling Polygonal geometry The rendering pipeline Z buffering High Performance Visible Surface Detection

433-324 Graphics and Interaction Rendering pipeline & object modelling

Adrian Pearce
Department of Computer Science and Software Engineering University of Melbourne

The University of Melbourne

Lecture outline

Introduction to Modelling

Polygonal geometry

The rendering pipeline

Z buffering

High Performance Visible Surface Detection
How are objects modelled in computer graphics?

Aim: understanding polygonal geometry and the basics of the rendering pipeline.

Reading:
- Foley Section 9.1.1 Representing polygon meshes.
- Foley Sections 14.4 Shadows, 14.9 The rendering pipeline and 14.3 Surface detail.

Additional Reading:
- Section 23.3 Level of detail in Computer graphics and virtual environments: from realism to real-time by Mel Slater, Anthony Steed and Yiorgos Crysanthou, Addison Wesley 2002 ISBN: 0201624206.

Introduction to modelling

The first problem in modelling geometric objects and their motion is to formulate a suitable mathematical model (does it have a solution? is the solution unique?).

A model based on general theories and laws of particular discipline (e.g. luminosity model of a 3D surface or physical model of a soccer ball).

- You may need to simplify the model and approximate to ensure tractability (use polyhedra model of soccer ball instead of parametric surface.)

- There will be a balance between what to include for completeness, and what to remove for tractability (don’t model the effect of wind on the soccer ball).
Polygons are very useful, both in themselves and as building blocks for approximating arbitrary curved arcs and regions.

Representation

- Either as a set of line segments,
- or an ordered sequence of vertices using absolute or relative coordinates,
- Walking order convention often applies, e.g. anti-clockwise for outer and clockwise for inner
**Polygon mesh: vertex list**

\[ V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1), \ldots, (x_4, y_4, z_4)) \]

\[ P_1 = (1, 2, 4) \]

\[ P_2 = (4, 2, 3) \]

**Polygon mesh: edge list**

\[ V = (V_1, V_2, V_3, V_4) = ((x_1, y_1, z_1), \ldots, (x_4, y_4, z_4)) \]

\[ E_1 = (V_1, V_2, P_1, 1) \]

\[ E_2 = (V_2, V_3, P_2, 1) \]

\[ E_3 = (V_3, V_4, P_2, 1) \]

\[ E_4 = (V_4, V_2, P_1, 1) \]

\[ E_5 = (V_4, V_1, P_1, 1) \]

\[ P_1 = (E_1, E_4, E_5) \]

\[ P_2 = (E_2, E_3, E_4) \]

Polygons represented as line segments (edges) makes polygon clipping and scan-line filling operations easier (we will see later).
Polygon types

- Convex
- Concave
- Non-simple
- Multiple-boundary
- Star

Polygon Properties

- In a convex polygon no internal angle is greater than 180 degrees
- In a concave polygon there are internal angles that can be greater than 180 degrees
- Concave polygons can be represented as a conjunction of convex polygons, sometimes desired as convex polygons have certain properties that simplify some geometric operations and tessellations.
- the multiple boundary polygons would need some classification of region (like Tasmania connected to the mainland of Australia).
The Rendering pipeline and the z-buffer

(Foley Figure 14.41)

Z Buffer

As well as pixel frame buffer, keep a parallel Z buffer, which at each point records the depth of corresponding pixel.

The Z buffer is initialised to some representation of “infinite” depth.

The frame buffer initialised to “background” colour then pixel writes take pixel value, \( x \), \( y \), and \( Z \). Updates only take place at \((x, y)\) only if new depth is closer.
Z Buffer

Depth must be in camera coordinates, any order-preserving function of true depth will do (pseudo-depth is OK).

Z buffer must have sufficient bits to resolve depth finely enough and may use some bits of frame buffer, if enough bits are available, and display lookup tables can be manipulated.

Can use buffer with higher spatial resolution, and block-average down for antialiasing.

The z-buffer drawing (a) first and (b) second polygons

- Assume looking down the z axis.
- Depends on spatial resolution (must have sufficient bits to resolve depth finely enough).
Interpolation of z values along edges for scan lines

\[ z_a = z_1 - (z_1 - z_2) \frac{y_1 - y_s}{y_1 - y_2} \]

\[ z_b = z_1 - (z_1 - z_3) \frac{y_1 - y_s}{y_1 - y_3} \]

\[ z_p = z_b - (z_b - z_a) \frac{x_b - x_p}{x_b - x_a} \]

Pros and cons of the z buffer

+ simple to implement.
+ implementable in hardware (generally is).
+ works for all kinds of objects (not just polyhedra).
- extra storage needed for Z buffer.
- many accesses to Z buffer and frame buffer—bad for display through a communications link.
- limited to space and depth resolution of buffers.
Properties of Z-buffers

- for each polygon or object in scene (piecewise Z buffering).
- (z-buffer) for each scan line (single-scan-line Z buffering)
- If shading computation is time-consuming, a rough front-to-back depth sort of objects to display closest one first improves efficiency (can use radix sort for this, at resolution of z-buffer).
- Don’t calculate shading for hidden surfaces — important if using Phong illumination model as we will see later.

Double buffering

The problem: animation and motion can cause the display to flicker.

The solution: double buffering can be used to reduce the effect of flickering, by sending all drawing commands to an off-screen buffer, then swapping buffers before redrawing.

In addition, only those pixels recently drawn are selectively erased at each step in the off-screen buffer.
Rendering pipeline for z-buffer (revisited)

(Foley Figure 14.41)

Rendering pipeline for z-buffer - the steps

See page 521 of Foley:

- **db traversal & modelling transformation** transform all (relatively defined) polygons or polygonal meshes to their correct location.
- **Trivial accept/reject**: Entirely outside & back face culling
- **Lighting**: calculate intensity (for vertices only: need to do in 3D before perspective projection)
- **Viewing transformation**: perspective transformation (e.g. perspective foreshortening).
- **Clipping**: clip to viewport (involves creation of new vertices on border).
- **Map to viewport**: Change of coordinate systems (Divide by W: based on homogeneous coordinate system).
- **Rasterisation**: scan-line drawing polygons (including z-buffering & shading interpolation)
High performance visible surface determination

Many methods and variations of visible surface determination are possible, which performs best depends on hardware available, and nature of graphics drawn, E.g.

- for a scene made of many small polygons, Z buffering might be best,
- for a scene made of a few large polygons, binary space partition techniques (BSPs) might be better.

The Z-buffer idea can also be used for a variety of tasks, such as

- piecewise Z buffering (for example polygon-by-polygon), or
- single-scan-line Z buffering (pixel-by-pixel).

Level of detail

Varying quality with distance - programmer provides several different versions of an object for viewing at different distances.
Example from Stanford University Computer Graphics Lab

Despite the difference in the number of used to model the figures (on the left), at a distance (on the right) they appear very similar (Figure from Slater text).

Vertex split method

(Figure from Slater text)
Example of levels of detail

Texture and bump mapping

- A *texture map* is used to scale one or more of the surfaces material properties, such as diffuse colour (RGB) components.

- A *bump map* is an array of displacements, each which perturbs the surface normal before it is used in the illumination model.
Summary

- Polygons are one of the most widely used models in computer graphics and are useful for representing both two-dimensional shapes and three-dimensional objects.
- The Rendering pipeline, based on z-buffering, is the basis of OpenGL.
- The rendering pipeline facilitates the systematic modelling, transformation and rendering of polygons.
- Various techniques exist for achieving high performance, including level-of-detail, texture and bump mapping.