

UNIT 4

Visible Surface Determination and Computer Graphics Algorithms

(15 Hours) — Computer Graphics and Animation (CAC305)

4.1 Image Space and Object Space Techniques

Visible surface determination (also called hidden surface removal) is the process of identifying which surfaces, edges, or parts of a 3-D scene are visible from a given viewpoint and eliminating those obscured by other, closer objects. Because rendering an object that is ultimately invisible wastes computation, visible surface algorithms are an essential stage in the 3-D graphics pipeline. Visible surface algorithms are broadly classified into two categories based on where — in which coordinate space — the visibility comparisons are actually performed.

Object-Space Techniques

Object-space methods compare objects and surfaces to each other directly using their original 3-D world-coordinate definitions, before any conversion to a raster or pixel grid. Each pair of surfaces (or each surface against every other surface) is examined analytically — using geometric and algebraic tests — to determine which parts of which surface are visible.

- Calculations are carried out with the precision of the object definitions themselves — not limited by screen/display resolution.
- The output is a set of visible surfaces (or visible surface fragments) that can, in principle, be displayed at any resolution or scaled without recomputing visibility.
- Well suited to scenes with relatively few, well-defined objects, since comparing every object against every other object grows expensive as the number of objects increases.
- Example approach: the Back-Face Detection method and certain formulations of the Painter's Algorithm.

Image-Space Techniques

Image-space methods perform visibility determination after the scene has been projected onto the 2-D view plane, working at the resolution of individual pixels. For each pixel in the final image, the algorithm decides which of the (possibly many) surfaces that project onto that pixel is nearest to the viewer, and therefore visible.

- Visibility decisions are tied directly to the resolution of the output device — accuracy is limited by pixel size.
- Generally simpler to implement and highly parallelizable, since each pixel (or scan line) can often be processed independently.
- Scales well with scene complexity because processing cost is driven primarily by the number of pixels, not the number of object-pairs.
- Example approaches: the Z-Buffer (Depth-Buffer) Algorithm and the Scan-Line Algorithm.

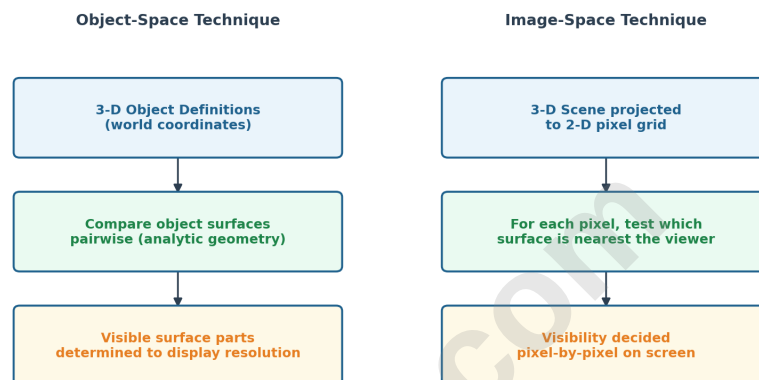


Fig 4.1 — Object-space visibility is resolved by comparing 3-D surfaces directly (left); image-space visibility is resolved pixel-by-pixel after projection onto the screen (right).

Object-Space vs Image-Space — Quick Comparison

- ✓ Object-space works in world coordinates before projection; image-space works in screen (pixel) coordinates after projection.
- ✓ Object-space accuracy is limited only by numerical precision; image-space accuracy is limited by display resolution.
- ✓ Object-space cost grows with the number of object/surface pairs; image-space cost grows with the number of pixels.
- ✓ Most practical systems combine both — e.g. back-face removal (object-space) as a fast pre-filter before a Z-buffer pass (image-space).

4.2 Hidden Surface Removal — Depth Comparison

The fundamental principle behind almost every visible-surface algorithm is depth comparison: along any single line of sight from the viewer through a given pixel, multiple surfaces in the scene may lie on that same projection line at different distances (depths). Only the surface with the smallest depth value — i.e. the one nearest the viewer — should contribute its colour to that pixel; all farther surfaces along the same line of sight are hidden and must be discarded or overwritten.

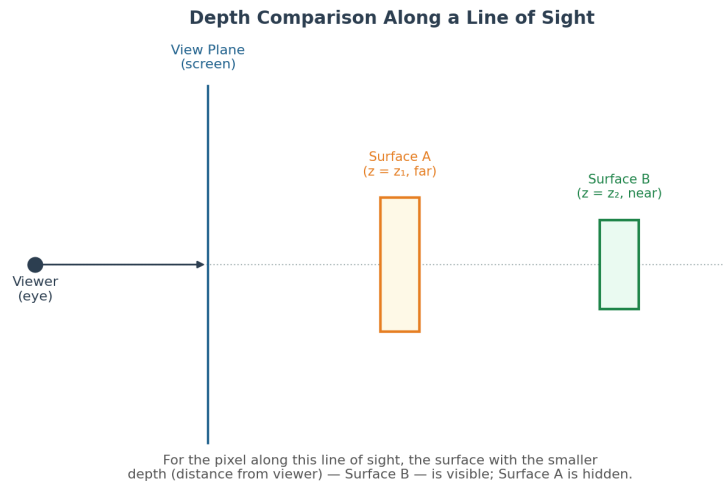


Fig 4.2 — Depth comparison along a single line of sight: of the two surfaces intersecting the same projector, only the nearer surface (smaller z -value) is visible at that pixel.

General Depth-Comparison Procedure

Example 4.1 — Generic Depth-Comparison Test

For a given pixel (x, y) and a candidate surface S with depth z_s at that pixel:

1. If no surface has yet been recorded for (x, y) , store S and z_s as the current visible surface.
2. Otherwise, compare z_s to the currently stored depth z_{current} .
3. If $z_s < z_{\text{current}}$ (S is nearer the viewer), replace the stored surface and depth with S and z_s .
4. If $z_s \geq z_{\text{current}}$, discard S at this pixel — it is hidden.

Note: The convention of "smaller z means nearer to the viewer" assumes a viewing system where depth increases away from the eye; some coordinate conventions reverse this, so the comparison direction must always match the coordinate system in use.

Different visible-surface algorithms apply this same depth-comparison idea at different stages and granularities: the Z-Buffer algorithm applies it independently at every single pixel; the Painter's Algorithm applies it once per whole polygon (by sorting polygons before drawing); and the Scan-Line algorithm applies it per horizontal span of pixels along each scan line.

4.3 Z-Buffer Algorithm

The Z-Buffer (or Depth-Buffer) algorithm is one of the simplest and most widely used image-space visible-surface methods. In addition to the usual frame buffer, which stores the colour of every pixel, it maintains a second array of equal dimensions — the depth (z) buffer — which stores the depth of the surface currently visible at each pixel.

Algorithm Steps

Example 4.2 — Z-Buffer Algorithm

1. Initialise every entry of the depth buffer to the maximum possible depth (often ∞ , or the depth of the back clipping plane).
2. Initialise every entry of the frame buffer to the background colour.
3. For each polygon in the scene, for each pixel (x, y) that the polygon covers:
 - a. Compute the polygon's depth z at (x, y) .
 - b. If $z < \text{depthBuffer}[x, y]$:
 $\text{depthBuffer}[x, y] = z$
 $\text{frameBuffer}[x, y] = \text{polygon's colour/intensity at } (x, y)$
4. After all polygons are processed, the frame buffer holds the final, correctly visible image.

Depth Buffer — Initialized to ∞ (Frame buffer set to background)					Depth Buffer After Processing (smallest z at each pixel wins)				
∞	∞	∞	∞	∞	-	-	4.2	4.0	-
∞	∞	∞	∞	∞	-	5.1	3.8	3.6	4.5
∞	∞	∞	∞	∞	6.0	4.9	3.5	3.4	4.1
∞	∞	∞	∞	∞	-	5.0	3.7	3.9	-
∞	∞	∞	∞	∞	-	-	4.4	-	-

Fig 4.3 — Depth buffer contents before processing (all ∞ , left) and after processing several overlapping polygons (right); each cell keeps the smallest depth seen so far.

Characteristics of the Z-Buffer Algorithm

Aspect	Description
Order independence	Polygons can be processed in any order — the depth test always keeps the correct nearest surface
Simplicity	Requires only a simple per-pixel comparison; easy to implement in both software and hardware
Memory cost	Needs an extra buffer the same size as the frame buffer to store one depth value per pixel
Hardware support	Implemented directly in graphics hardware (GPUs), making it the dominant method in real-time 3-D graphics
Aliasing	Like other image-space methods, accuracy is limited to pixel resolution, which can cause jagged edges without anti-aliasing

Note: Because the Z-Buffer algorithm processes each polygon and pixel independently with a simple comparison, it requires no pre-sorting of polygons by depth — unlike the Painter's Algorithm — which makes it far more robust for complex, interpenetrating scenes.

4.4 Back-Face Removal

For a solid, opaque object such as a polyhedron, any surface (polygon face) whose outward-facing normal points away from the viewer is, by definition, part of the far side of the object and is always hidden behind the near-side (front-facing) surfaces. Back-face removal (or back-face culling) is a fast object-space technique that discards such surfaces before any further, more expensive visibility processing is performed.

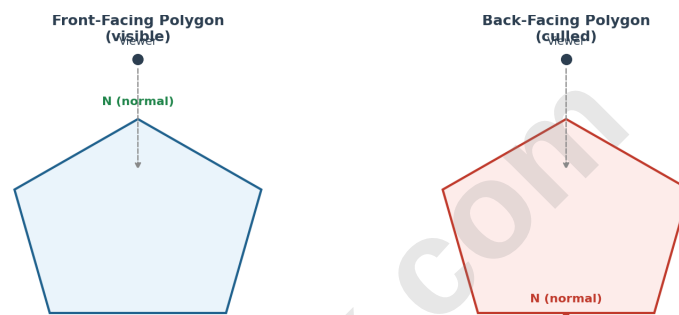


Fig 4.4 — A front-facing polygon has its normal pointing toward the viewer and is kept (left); a back-facing polygon has its normal pointing away from the viewer and is culled (right).

Back-Face Test

For a polygon with a known outward surface normal $N = (A, B, C)$ — the coefficients of its plane equation $Ax + By + Cz + D = 0$ — and a viewing direction vector V from the polygon toward the viewer (camera), the polygon is classified using the sign of the dot product of N and V .

Example 4.3 — Back-Face Test (Simple Case: Viewer Along +z Axis)

If the viewer looks along the negative z-direction and C is the z-component of the surface normal $N = (A, B, C)$:

- If $C > 0$ — the surface normal has a positive z-component, meaning it points toward the viewer — the polygon is front-facing and is kept.
- If $C < 0$ — the normal points away from the viewer — the polygon is back-facing and can be discarded.
- If $C = 0$, the polygon is viewed exactly edge-on and is typically treated as invisible (zero projected area).

Note: Back-face removal alone correctly hides only the surfaces of a single convex solid object; it cannot, by itself, resolve visibility between separate objects or concave shapes that partially occlude each other — for that, it is typically combined with an image-space method such as the Z-Buffer or a depth-sorting method such as the Painter's Algorithm.

- Extremely cheap to compute — a single dot product per polygon.
- Commonly eliminates roughly half of all polygon faces in a typical closed solid before any per-pixel work is done.
- Used almost universally as a first-pass optimisation in real-time rendering pipelines, ahead of Z-buffering.

4.5 The Painter's Algorithm

The Painter's Algorithm (also called the Depth-Sort algorithm) mimics the way a painter builds up an oil painting: distant background elements are painted first, and nearer objects are painted over them afterward, naturally obscuring whatever lies behind. Applied to computer graphics, all polygons in a scene are sorted by depth and then rendered back-to-front, so that nearer polygons simply overwrite farther ones at any pixel they share.

Algorithm Steps

Example 4.4 — Painter's Algorithm

1. Compute a representative depth (e.g., the depth of the farthest or average vertex) for every polygon in the scene.
2. Sort all polygons into a list ordered from farthest to nearest.
3. Render the polygons in that sorted order into the frame buffer, allowing each new polygon to overwrite pixels already painted by farther polygons.
4. The final frame buffer shows only the nearest surface at every pixel.

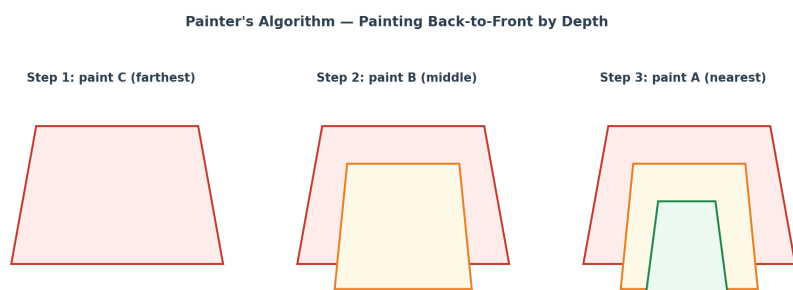


Fig 4.5 — Painter's algorithm paints polygon C (farthest) first, then B, then A (nearest) last, so nearer polygons correctly cover farther ones.

The Depth-Ordering Ambiguity Problem

A simple sort by a single representative depth value fails whenever polygons overlap in depth in more complex ways — for example, when two polygons intersect each other, or when three or more polygons cyclically obscure one another (A partly in front of B, B partly in front of C, and C partly in front of A). In such cases, no single back-to-front ordering correctly renders every pixel, and polygons may need to be split before sorting.

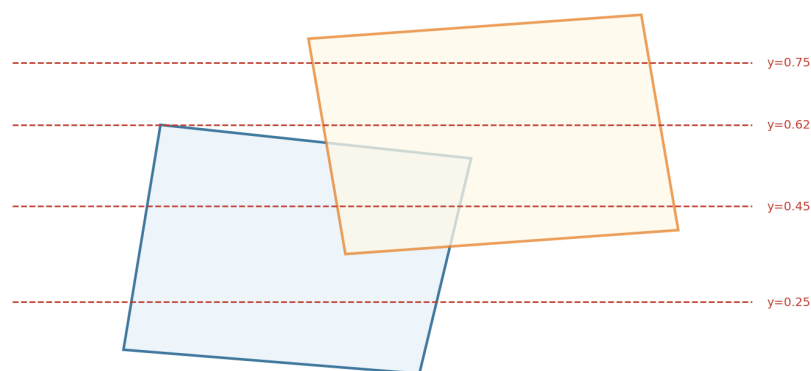
Ambiguous Case	Required Fix
Two polygons intersect (cross through each other)	Split both polygons along their line of intersection before sorting
Cyclic overlap (A over B, B over C, C over A)	Detect the cycle and split at least one polygon to break it
One polygon's projection surrounds another's on screen but at ambiguous depth	Use additional depth tests (e.g., bounding-box or plane tests) to resolve order

Note: Because of these ordering ambiguities, the Painter's Algorithm is best suited to relatively simple scenes with well-separated polygons; complex, highly interpenetrating scenes are handled far more robustly by the Z-Buffer algorithm.

4.6 Scan-Line Algorithm

The scan-line algorithm is an image-space method that determines visibility one horizontal scan line at a time, processing an entire row of pixels together rather than testing each pixel in isolation against every polygon in the scene. For each scan line, it finds where every polygon edge crosses that line, sorts these intersection points left to right, and then resolves which polygon is visible along each span between successive intersections.

Scan-Line Algorithm — Scan Lines Crossing Polygon Edges



At each scan line, edge intersections are sorted left-to-right; spans between paired intersections are filled using the nearest surface's depth and colour.

Fig 4.6 — Two overlapping polygons intersected by four scan lines; edge intersections on each scan line are sorted and grouped into spans that are filled with the nearest surface's colour.

Data Structures

- Edge Table (ET): stores every polygon edge, indexed by the scan line at which it first becomes active, together with its ending y-coordinate, x-intercept, and the inverse slope ($1/m$) used to incrementally update the x-intercept on successive scan lines.
- Active Edge List (AEL) / Active Edge Table (AET): holds only the edges that intersect the current scan line, kept sorted by x-coordinate; edges are added when a scan line reaches their starting y-value and removed once it passes their ending y-value.
- Polygon (Surface) Table: stores each polygon's plane equation coefficients (for depth calculation), colour/shading information, and an in/out flag for the current scan line.

Algorithm Steps

Example 4.5 — Scan-Line Visible-Surface Algorithm

1. Build the Edge Table from all polygon edges in the scene.
2. For each scan line y , from top to bottom of the screen:
 - a. Update the Active Edge List: add edges starting at this y , remove edges that have ended, and update x-intercepts using the inverse slope.
 - b. Sort the Active Edge List by x-coordinate.
 - c. Between each pair of consecutive intersections (a span), determine which surfaces are active, and if more than one, use depth comparison at the span's endpoints to decide the nearest surface.
 - d. Fill the span with the visible surface's colour or intensity.
3. Move to the next scan line and repeat until the whole image is complete.

Note: Because the x-intercepts of each edge can be updated incrementally ($x_{\text{new}} = x_{\text{old}} + 1/m$) from one scan line to the next, the scan-line algorithm avoids recomputing full edge intersections from scratch at every row, making it considerably more efficient than a naive pixel-by-pixel depth test across the entire image.

Visible-Surface Algorithms — Quick Comparison

- ✓ Z-Buffer: simplest, order-independent, per-pixel depth test; needs extra depth-buffer memory.
- ✓ Back-Face Removal: very fast object-space pre-filter; only removes a solid's own hidden faces.
- ✓ Painter's Algorithm: sorts and paints polygons back-to-front; can fail on intersecting or cyclically overlapping polygons.
- ✓ Scan-Line Algorithm: processes one row at a time using edge and active-edge tables; efficient for coherent, span-based rendering.

4.7 Light and Color — Color Models (RGB, CMY, YIQ)

Light and Color — Basic Concepts

Visible light is electromagnetic radiation with wavelengths roughly between 400 and 700 nanometres. The perceived colour of an object depends on the wavelengths of light it reflects (or, for a light source, emits) and how the human eye's receptors — cone cells sensitive to red, green, and blue wavelength ranges — respond to that light. A colour model (colour space) is a mathematical scheme for specifying colours as a set of numeric component values, allowing colours to be stored, manipulated, and reproduced consistently by graphics hardware and software.

RGB Color Model

The RGB model represents a colour as a combination of three additive primary components — Red, Green, and Blue — each typically ranging from 0 (no contribution) to 1 (full intensity), or 0–255 in 8-bit-per-channel systems. Combining light of these three primaries at full intensity produces white; combining none produces black.

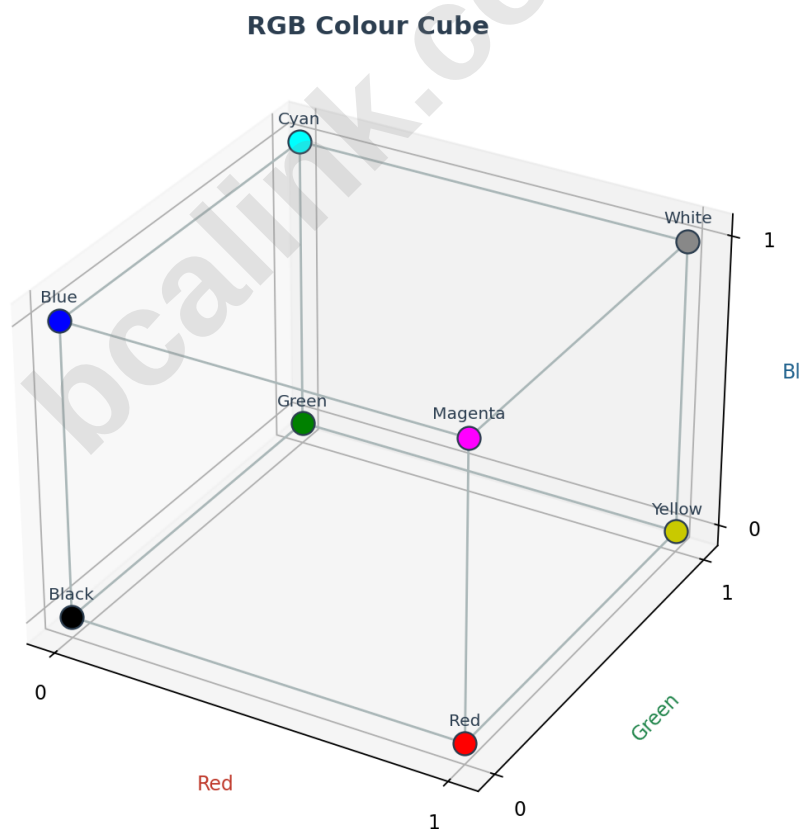


Fig 4.7 — The RGB colour cube: each axis represents one primary component, with black at the origin and white at the diagonally opposite corner; the diagonal itself contains all shades of grey.

- RGB is an additive model — colours are formed by adding light together — matching how monitors, TVs, and projectors emit light directly to the eye.

- The three coordinate axes of the RGB colour cube correspond to the R, G, and B components; the main diagonal from (0,0,0) to (1,1,1) represents the grey scale.
- Most common bit depth: 24-bit "true colour," using 8 bits per channel for $256 \times 256 \times 256 \approx 16.7$ million possible colours.

CMY / CMYK Color Model

The CMY model uses Cyan, Magenta, and Yellow as its three primary components and is a subtractive model: each component represents how much of a corresponding colour of light is removed (absorbed) from white light, which is the natural way pigments and inks — as used in printing — produce colour by reflecting only the wavelengths that are not absorbed.

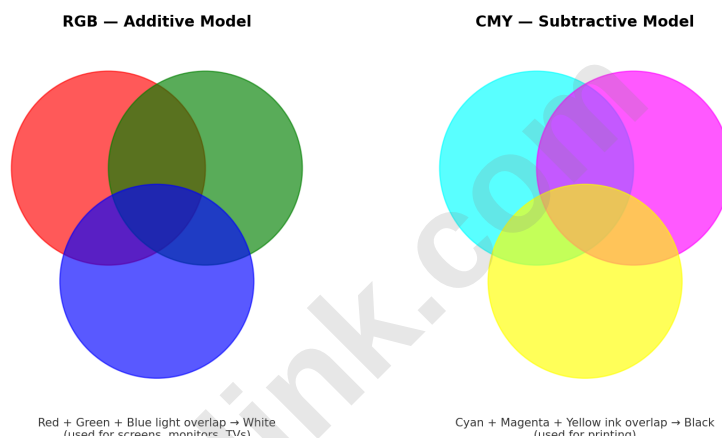


Fig 4.8 — RGB additive mixing of coloured light produces white where all three overlap (left); CMY subtractive mixing of inks produces black where all three overlap (right).

The RGB and CMY models are complementary and easily converted between one another, since cyan is the absence of red, magenta the absence of green, and yellow the absence of blue (assuming normalised 0–1 component values):

Example 4.6 — RGB ↔ CMY Conversion

$$C = 1 - R$$

$$M = 1 - G$$

$$Y = 1 - B$$

$$\text{(and conversely)} \quad R = 1 - C, \quad G = 1 - M, \quad B = 1 - Y$$

Note: In practice, printing uses CMYK — adding a separate black (K) ink channel — because mixing full-strength cyan, magenta, and yellow ink produces a muddy dark brown rather than a true black, and using a dedicated black ink is also cheaper and produces sharper black text.

YIQ Color Model

The YIQ model was developed for NTSC television broadcasting to remain compatible with older black-and-white receivers. It separates a colour into a luminance component (Y) — the brightness information that a monochrome set can display directly — and two chrominance components, I (in-phase) and Q (quadrature), that together carry the colour information.

Example 4.7 — RGB to YIQ Conversion (Standard NTSC Coefficients)

$$Y = 0.299R + 0.587G + 0.114B$$

$$I = 0.596R - 0.275G - 0.321B$$

$$Q = 0.212R - 0.523G + 0.311B$$

- The Y (luminance) component alone reproduces a full greyscale version of the image — this is exactly what early black-and-white TVs displayed.
- Because the human eye is far less sensitive to changes in chrominance (I, Q) than to changes in luminance (Y), the I and Q channels can be transmitted with much less bandwidth than Y — a major reason YIQ (and similar models like YUV/YCbCr) are still used in video compression today.
- The coefficients for Y mirror the human eye's relative sensitivity to red, green, and blue light — green contributes most to perceived brightness, blue the least.

Model	Type	Components	Typical Use
RGB	Additive	Red, Green, Blue	Monitors, TVs, digital cameras, web/screen graphics
CMY / CMYK	Subtractive	Cyan, Magenta, Yellow (+ Black)	Colour printing and offset/inkjet output
YIQ	Luminance–Chrominance	Y (luma), I, Q (chroma)	NTSC television broadcasting; bandwidth-efficient video transmission

Key Takeaways — Unit 4

- ✓ Visible surface algorithms fall into object-space techniques (compare 3-D surfaces directly) and image-space techniques (compare depths per pixel after projection).
- ✓ Every hidden-surface method ultimately relies on depth comparison: the surface nearest the viewer along a line of sight is the one displayed.
- ✓ The Z-Buffer algorithm keeps a per-pixel depth value, is order-independent, and is the standard method in modern graphics hardware.
- ✓ Back-face removal cheaply discards a solid object's own rear-facing polygons using the sign of the surface normal's viewing-direction component.
- ✓ The Painter's Algorithm sorts and paints polygons back-to-front but can fail for intersecting or cyclically overlapping polygons.
- ✓ The scan-line algorithm resolves visibility one row at a time using edge tables and an active edge list, exploiting coherence between adjacent scan lines.
- ✓ Colour models — RGB (additive, for displays), CMY/CMYK (subtractive, for printing), and YIQ (luminance–chrominance, for broadcast video) — represent colour numerically for different technologies and purposes.

— *End of Unit 4* —