Visible Surface Determination

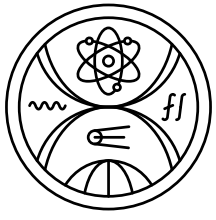
Algorithms for determining the visibility of objects and their parts

Róbert Bohdal

robert.bohdal@fmph.uniba.sk
https://flurry.dg.fmph.uniba.sk/webog/en/bohdal-teaching

Department of Algebra and Geometry
Faculty of Mathematics, Physics and Informatics
Comenius University Bratislava

Computer Graphics (1)
Lecture 5



Contents

1 Visibility of Object Parts in a Scene

2 Using Normal Vectors

3 Z-Buffer Algorithm (Depth Test)

4 Painter's Algorithms

5 Warnock's Area Subdivision Algorithm

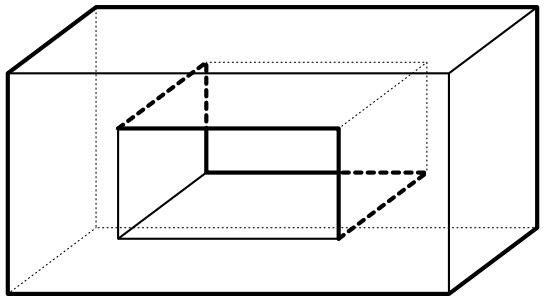
6 Rendering a Wireframe Model

7 Rendering a 3D Function Graph



Visibility of Object Parts in a Scene

- The task is to find the part of an object's surface that is visible to the viewer, i.e. the part that is not occluded by any other objects or by the object itself (in the case of non-convex objects)
- We will generally assume that the modelled objects are represented by polygon meshes
- Algorithms for determining visibility in a scene fall into two groups
  - **object space methods** – algorithms operating on objects, e.g. polygons in 3D
  - **image space methods** – algorithms operating on the image, e.g. bitmap of the 2D projection

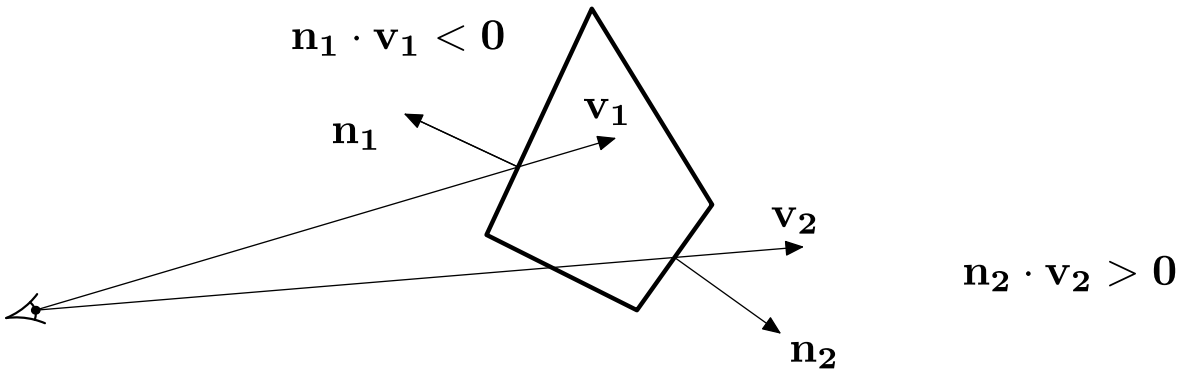


Example of visibility determination for a non-convex object with a hole



Visibility Determination Using Face Normals

- An object space method
- The algorithm represents objects using closed, **orientable**, 2-manifold polygon meshes. The vertices of each face are indexed counterclockwise when viewed from outside
- Once the viewer has been positioned in the scene, the dot product allows us to classify all mesh faces as:
  - **back-facing** – the normal vector and the direction away from the viewer form an acute angle ($< 90°$)
  - **front-facing** – the normal vector and the direction away from the viewer form an obtuse angle ($> 90°$)
- Front-facing faces are **potentially** visible. They may be occluded by other faces of the same object (for non-convex objects) or of another object. Back-facing faces need not be considered during rendering unless translucent faces must be handled



A truncated prism with 4 lateral faces (top view in orthographic projection)



Visibility Determination Using Face Normals

- For a single **convex** polyhedron, or several convex polyhedra whose 2D projections do not overlap, this algorithm completely solves the visibility problem
- For more complex objects, face normals can be used in a preprocessing step to eliminate back-facing polygons (i.e. invisible from the viewing direction)
- Let the vertices $A_0A_1 \ldots A_{n-1}$ of the face under consideration be ordered counterclockwise when viewed from outside. The polygon's normal vector $\mathbf{n}$ is then determined as follows:
  - if the polygon is convex, compute the normal using the cross product: $\mathbf{n} = (A_i - A_{i-1}) \times (A_{i+1} - A_i)$ (beware of the case $\overline{A_{i-1}A_i} \parallel \overline{A_iA_{i+1}}$)
  - otherwise, choose a coordinate whose values are not identical across the polygon, and find a vertex where it attains a maximum or minimum. At this vertex, use the same cross product of adjacent edges (assuming they are not collinear)
- If the polygon's vertices are not coplanar, there is no single exact normal for the entire face. Triangulate the face and use the sum of the unnormalized normals (if non-zero) of consistently oriented triangles as a **representative normal** direction (i.e. their vertices are ordered counterclockwise when viewed from outside):

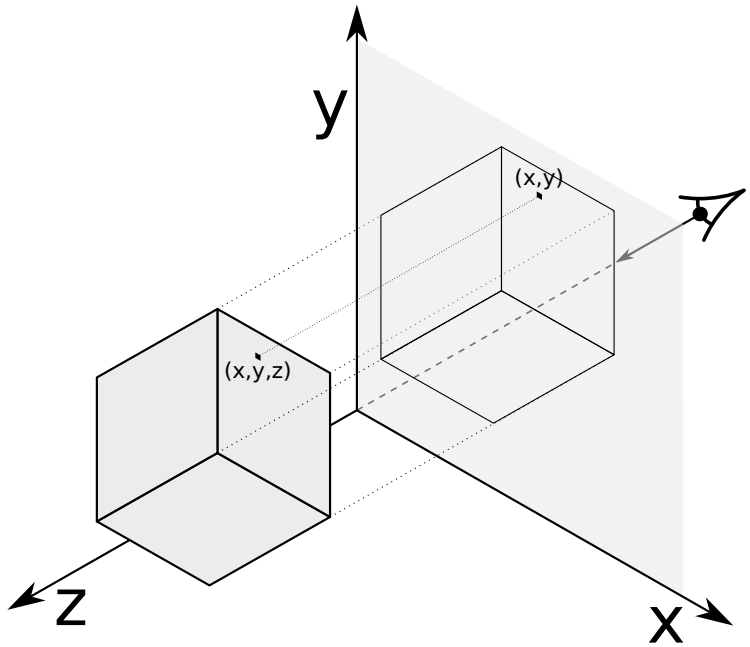
$$\mathbf{n} = \sum_{k=1}^{m} (Q_k - P_k) \times (R_k - P_k),$$

where P_k, Q_k, R_k are the consistently ordered (counterclockwise) vertices of the k-th triangle of the triangulation



Z-Buffer Algorithm (Depth Test)

- The algorithm belongs to the image space methods
- It is very popular because it is easy to implement
- It works well with various object representations that are visualised as sets of points – pixels
- The *Z-buffer* is a 2D array/bitmap of the same size as the window in which the scene is rendered
- Together with Gouraud and Phong shading, it is implemented in hardware on graphics processors



Orthographic projection onto the xy plane, the depth z increases away from the viewer



z-buffer - basic algorithm

In the following algorithm, we assume **orthographic** projection onto the xy plane, with depth z **increasing** away from the viewer

- Each polygon of an object is projected along the z axis (the camera's viewing direction) onto the xy plane (the projection plane) and rasterised.

  For each polygon, we compute:

  1. the coordinates (x, y) of all pixels representing the polygon's projection,
  2. the original (unprojected) z coordinate for each (x, y) value,
  3. the colour (intensity) for each (x, y) value.

- The z coordinates are computed from the equation of the polygon's plane or by successive linear interpolation: first along the polygon edges, then between the endpoints of each scanline span. If the depth test passes, the computed value is stored in the z-buffer
- The colour computed in step 3 is stored in the *frame-buffer*, which stores the rendered image



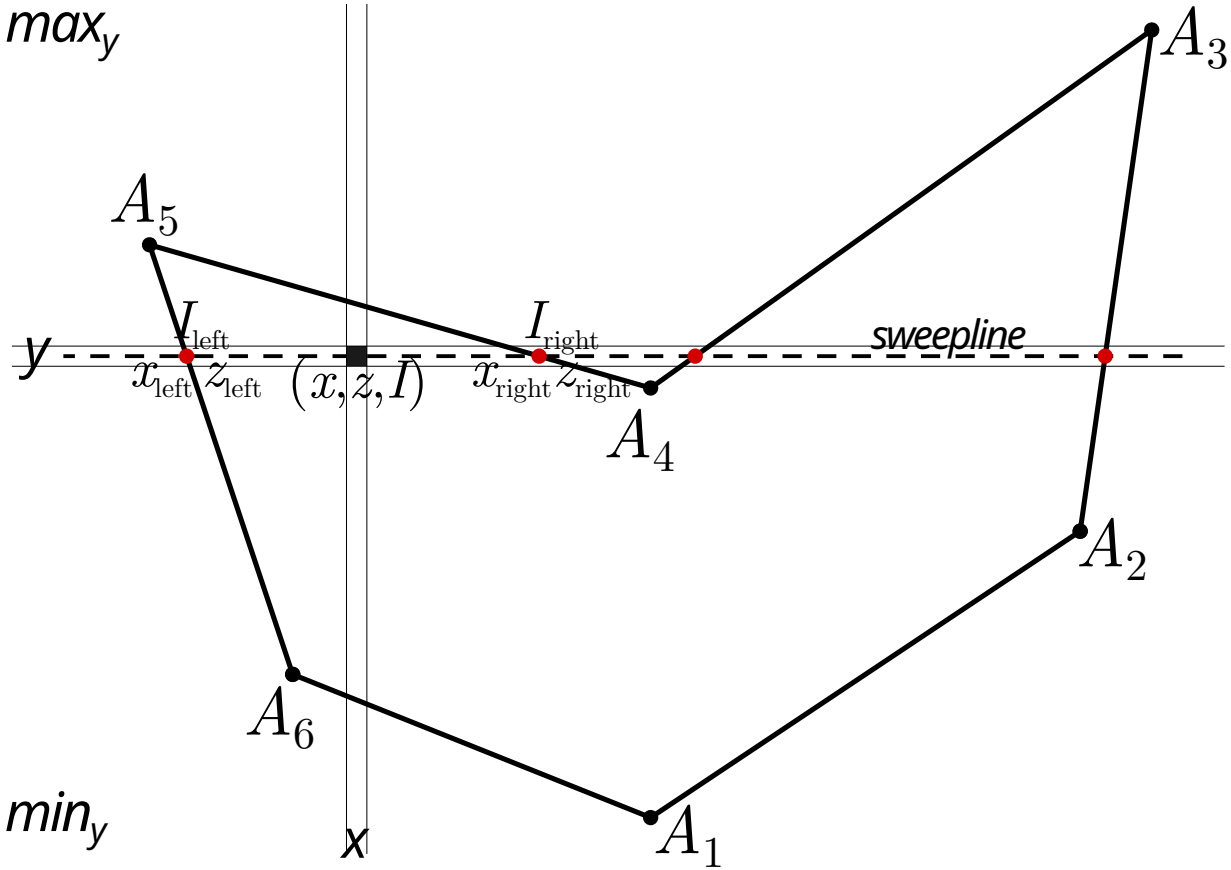
z-buffer - basic algorithm

```
// initialisation
for all points x, y in the window:
    z-buffer[x, y] := ∞ // an initial value
    frame-buffer[x, y] := background colour // optional step
// rendering
for each polygon in the scene:
    find all (x, y) values in the raster belonging to the polygon's projection onto the xy plane
    for each relevant (x, y) value:
      compute the corresponding z coordinate (i.e. z such that (x, y, z) lies in the polygon's plane)
      if z < z-buffer[x, y] then: // depth test
        z-buffer[x, y] := z
        frame-buffer[x, y] := colour computed for the polygon at point (x, y)
```



z-buffer - detailed algorithm

- Computations for a single polygon use a horizontal sweep line moving from its minimum y value to its maximum
- This line stores the x coordinates of the segment endpoints obtained by intersecting the current sweep line with the **polygon's projection**. It also stores the corresponding z coordinates and, optionally, colours
- For a non-convex polygon, the intersection may consist of several line segments



z-buffer - detailed algorithm

// initialisation
for all x, y:
  z-buffer$[x, y] := \infty$
  frame-buffer$[x, y]$:= background colour
for each polygon in the scene:
  let y_{min} and y_{max} be the minimum and maximum y coordinates of points in the rasterised polygon projection
  // construct the span list
  for all $y \in [y_{min}, y_{max}]$ in the raster:
    find the intersection of the sweep line and the polygon's projection (giving 1 or more line segments)
    for these segments, store in span_list$[y]$ the values $x_{left}, x_{right}, z_{left}, z_{right}$
      and optionally the colours at the segment endpoints I_{left}, I_{right}
  // process the segments in the span list
  for all $y \in [y_{min}, y_{max}]$:
    for each segment in span_list$[y]$:
      read $x_{left}, x_{right}, z_{left}, z_{right}$ (optionally also I_{left}, I_{right})
      for all $x \in [x_{left}, x_{right}]$ in the raster:
        compute (e.g. by linear interpolation) the z coordinate of the point projected onto (x, y)
        if $z <$ z-buffer$[x, y]$ then:
          z-buffer$[x, y] := z$
          frame-buffer$[x, y]$:= colour computed for the polygon at point (x, y)



z-buffer - evaluation

- A disadvantage is the high memory requirement at high resolutions. One solution is to process the image window in smaller parts, such as scanlines or tiles, reusing a smaller depth buffer for each part
- The basic z-buffer, which stores only the nearest depth, is insufficient for correctly rendering overlapping translucent surfaces. It does not store which object lies behind the first translucent surface. One solution is to render opaque objects first, followed by translucent surfaces from back to front with colour blending, depth testing enabled and depth buffer writes disabled
- Points are also rendered for objects that are invisible in the final image because they will later be covered by objects that are closer



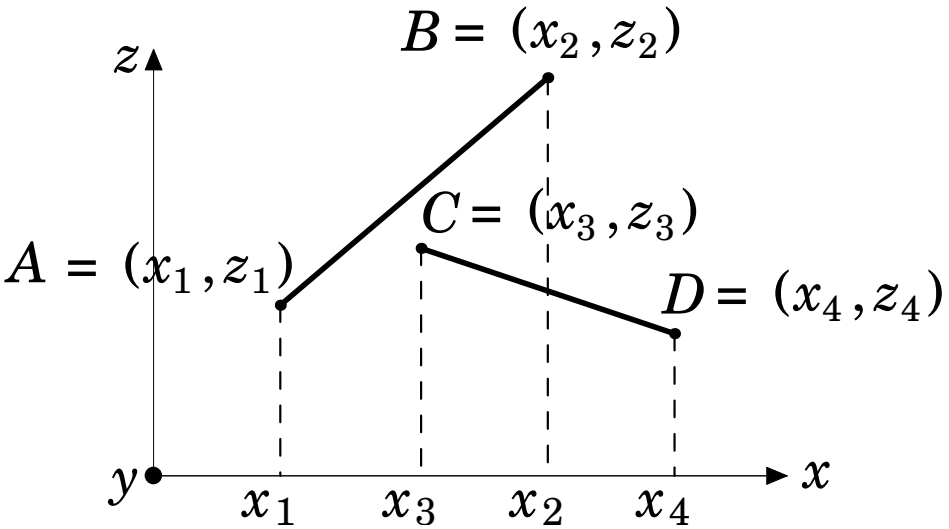
z-buffer with a sweep line (scanline)

- A z-buffer variant that processes the image one scanline at a time, using a depth buffer only one pixel row high
- The *z*-buffer is reinitialised after each scanline has been processed
- In the implementation, the order of the *for* loops is reversed. The outer loop iterates over scanlines rather than polygons. For each scanline, initialise the one-dimensional depth buffer and the corresponding row of the framebuffer

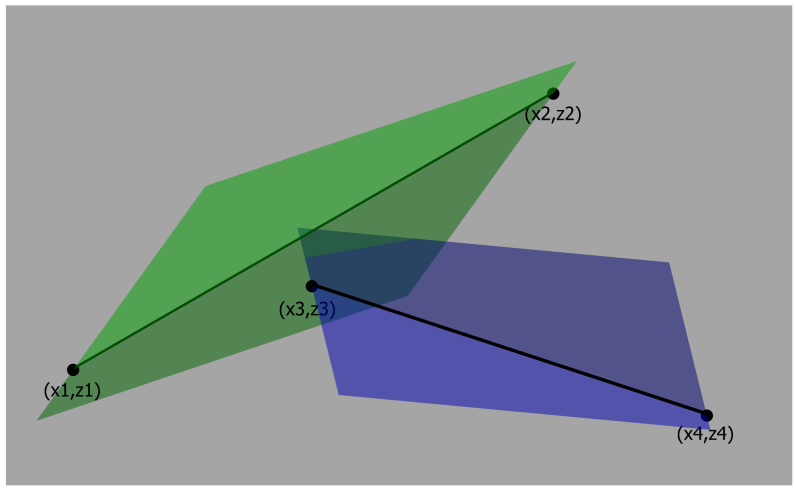


Sweep Line with Intervals

- The idea is that, for the sweep line at the current y value, visibility need not be tested at every point; a single test inside an interval is sufficient
- Divide the x axis at the projections of segment endpoints and of their mutual intersections in the cutting plane $y = c$, where c is the current y coordinate of the sweep line. Within each resulting interval, the depth order of the segments remains unchanged, so it is sufficient to identify the nearest segment at one interior point



Intervals on the sweep line (note: the figure is viewed along the y axis, not along the camera's viewing direction)



View of 2 polygons cut by the plane $y = c$
The positive y axis points towards the viewer



Ray Casting and Ray Tracing

- These algorithms handle visibility naturally. We will study them in more detail in the *Computer Graphics (2)* course later
- In ray casting, rays are sent through the pixels of the projection plane into the scene, and their **nearest** intersections with objects are found. Ray tracing extends this approach by tracing additional rays, for shadows, reflections and/or refraction



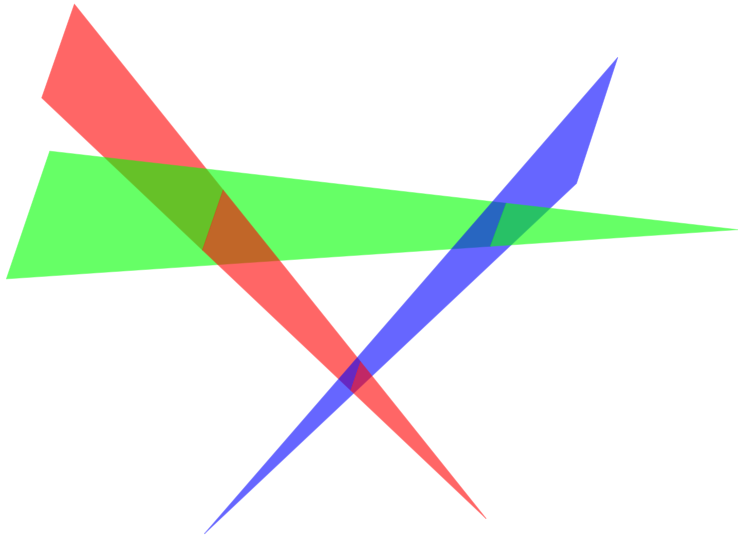
Painter's Algorithms

- A painter's algorithm renders objects in the same way as a painter would paint them on a canvas
- The most distant objects (e.g. mountains) are rendered first, followed by closer objects (e.g. trees in the foreground), so that distant objects that are not visible from a given viewpoint are gradually overpainted by nearer objects
- These algorithms combine ordering in object space with overpainting in image space. Although they require analysis of the scene in 3D, i.e. determining the relative order of objects along the viewing direction, rendering itself takes place after projection onto a 2D plane – in the 2D pixel space



Depth-Sorting Algorithm

- We assume that objects are represented by polygon meshes
- Objects are sorted approximately by their distance from the viewer
- If the scene is projected onto the xy plane (with the viewer looking along the z axis), objects are sorted by their z coordinate
- To determine the rendering order of **two polygons**, the following tests are performed (it is sufficient for the polygons under test to satisfy at least one of them):
  1. if the polygons' z ranges do not overlap, their rendering order can be determined
  2. If the bounding rectangles of the **projected polygons** do not overlap in the projection plane, we can render them in any order
  3. if all vertices of one polygon lie in front of/behind the plane containing the other polygon, their rendering order can be determined



A scene with objects to which the depth-sorting algorithm cannot be applied directly



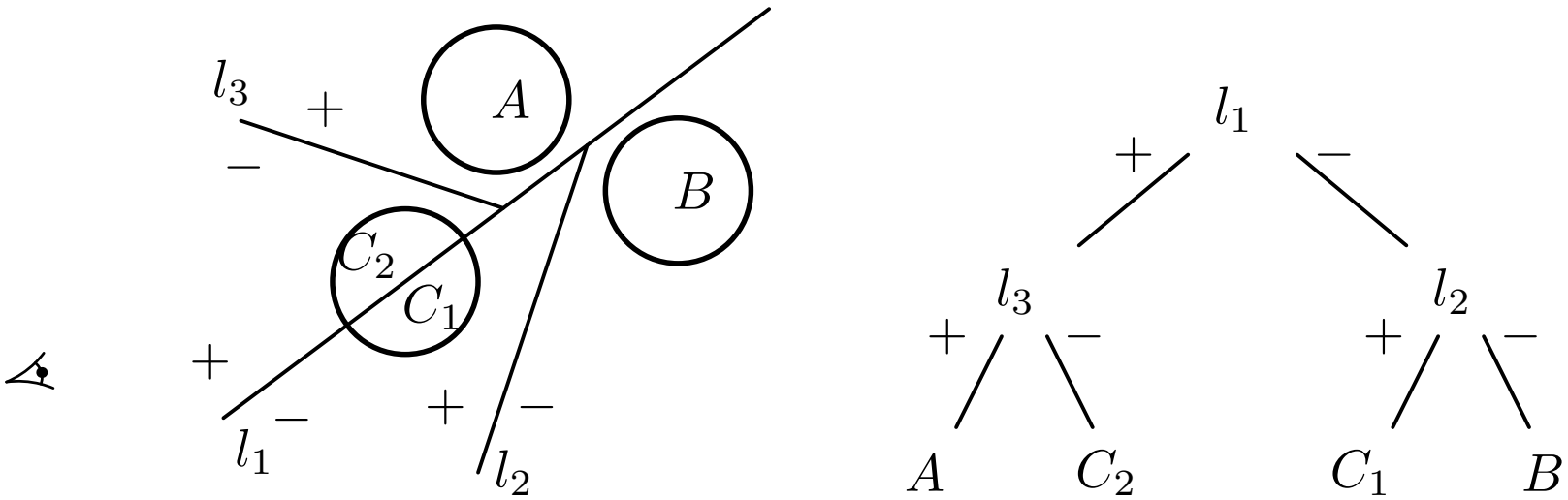
Depth-Sorting Algorithm

- Test 3 requires finding the plane equation $n_1x + n_2y + n_3z + d = 0$ for one polygon (e.g. use the cross product to determine the normal and substitute a vertex to determine d), then testing the vertices of the other polygon against this plane. The same half-space test can be expressed in two equivalent forms:
  1. Substitute the coordinates of the other polygon's vertices into the plane equation. If all resulting values have the same sign, the entire tested polygon lies in one half-space, allowing the rendering order to be determined
  2. Use the plane equation in the form $\mathbf{n} \cdot (X - P) = 0$, where P is a point in the plane. Substitute the coordinates of the other polygon's vertices for X and check whether the resulting values have the same sign
- Compare these signs with the sign obtained by substituting the viewer's position into the left-hand side of the plane equation. If the tested polygon lies in the half-space opposite to the viewer, render it before the polygon lying in the plane. If it lies in the same half-space, reverse the order
- These tests cannot determine the rendering order for every pair of polygons
- In some cases, no back-to-front order exists without splitting polygons (e.g. three cyclically overlapping polygons; see the previous slide)
- If a polygon cannot be placed in the rendering order, split it into two parts using the plane of another polygon so that a rendering order can be determined for the resulting polygons



Using a BSP Tree

- Building a BSP tree in 3D space serves as preprocessing for other algorithms, such as the painter's algorithm
- The scene is progressively partitioned and encoded in a BSP tree
- Each leaf contains a **convex** object or other simple geometry that **can** be rendered with correct visibility for a given viewpoint and viewing direction, without considering occlusion by geometry outside that leaf
- The hardest part is finding suitable splitting planes
- If no plane can separate the objects into two non-empty groups without cutting any of them, one or more objects must be split by a splitting plane (object $C = C_1 \cup C_2$ in the figure)



Organising a scene into a BSP tree

Using a BSP Tree

- In a polygonal representation, the leaves contain individual polygons. Splitting planes are often chosen to contain selected object faces
- Once the scene is represented by a BSP tree and the observer's position is selected, the tree is traversed recursively to render the scene using the following algorithm:
  1. read the splitting plane at the root of the tree
  2. if the viewer is in the *positive* half-space render the *negative* subtree first, then the *positive* subtree
  3. if the viewer is in the *negative* half-space render the *positive* subtree first, then the *negative* subtree
- In the illustrated example, the scene in the negative subtree of node l_1 is rendered first. At node l_2, the negative subtree is rendered first, while at node l_3, the positive subtree is rendered first. The objects are therefore rendered in the order B, C_1, A, C_2
- If an internal node stores polygons lying in its splitting plane, render them after the subtree farther from the viewer and before the nearer subtree



Using a BSP Tree

- An advantage of this algorithm is that the BSP tree is built independently of the viewer, although its traversal depends on the viewer's position
- In static scenes, where only the viewer moves, the costly construction of the BSP tree is performed once at the start. The scene can then be rendered very quickly by determining the object rendering order, for example in animations where only the camera moves
- The painter's algorithm also renders parts of objects that are occluded by objects closer to the viewer. If calculating the colour of a pixel is computationally intensive, opaque polygons can be processed in **reverse** order, from front to back. In this case, an *auxiliary array* is used to track whether each pixel in the image has already been rendered. If so, the colour calculation for that pixel is skipped. Otherwise, the colour is calculated, the pixel is rendered, and the corresponding entry in the auxiliary array is marked as rendered. At the start of rendering each image, all entries in the auxiliary array are marked as not rendered



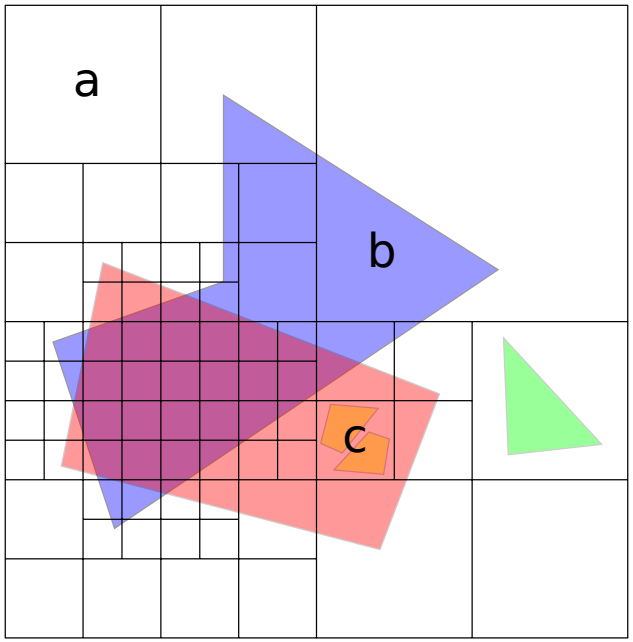
Using an Octree

- The algorithm is similar to the BSP approach, but the scene is partitioned into an octree
- It is mainly used when the input data are already represented by an octree
- After selecting the observer's position and ordering the octants from back to front relative to the viewer (similar to BSP), the tree is traversed in the specified order to render the individual objects



Warnock's Area Subdivision Algorithm

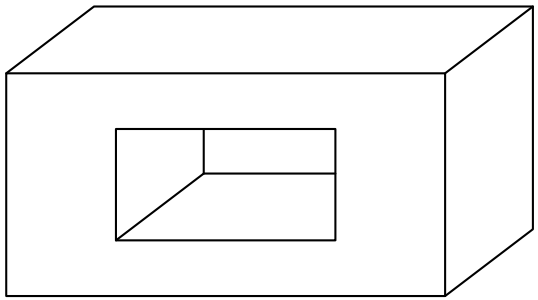
- Uses the "divide and conquer" principle: if the scene is sufficiently simple, render it directly; otherwise, divide the window into four quadrants and process each independently
- **Simple** (solvable) cases are rendered **directly**:
  - a: if the window is empty – fill the entire window with the background colour
  - b: if only one polygon intersects the window – render the portion of that polygon inside the window and fill the rest of the window with the background colour
  - c: if one polygon covers the entire window and lies in front of all other polygons within it – fill the window with its colour
- Otherwise, keep subdividing the window until a simple case is reached or the window is reduced to a single pixel; in the latter case, render the pixel corresponding to the nearest polygon part represented by that pixel



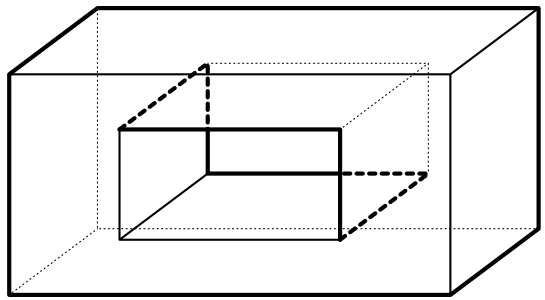


Rendering a Wireframe Model

- A wireframe model (the edges of a polygonal representation) is rendered by removing invisible portions of edges, including entire edges
- We can use the painter's algorithm: render polygons from back to front, first filling their interiors with the background colour and then drawing their boundaries
- A more specialised approach first identifies the visible portions of edges and then renders them
- Finding visible edge portions relies on classifying edges into three groups:
  - **back edges** – shared by two back-facing faces,
  - **front edges** – shared by two front-facing faces,
  - **silhouette edges** – shared by one front-facing and one back-facing face.
- Back edges are invisible; front and silhouette edges are **potentially** visible, so their visible portions must be found



Object with a hole



Highlighted silhouette edges



Roberts' Algorithm

- The algorithm removes invisible portions of edges of opaque **convex** polyhedra. Non-convex solids must first be decomposed into convex parts
- Use face normals to eliminate **back** edges, which are shared by two back-facing faces. Front and silhouette edges remain potentially visible.
- Test each potentially visible edge $P(t) = A + t(B - A)$, $t \in [0, 1]$, against the other polyhedra in turn
- Represent the tested polyhedron as the intersection of the half-spaces defined by its faces: $\mathbf{n}_i \cdot X + d_i \leq 0, \ \ i = 1, \ldots, m$, where $\mathbf{n}_i$ are the outward face normals
- A point $P(t)$ is occluded if the line segment from the viewer to that point passes through the interior of the tested polyhedron. Using the above half-space inequalities, determine the interval of parameter values t for which $P(t)$ is occluded
- Remove the occluded interval, leaving **zero**, **one** or **two** segments of the tested segment, which are then tested against the remaining polyhedra
- After processing all polyhedra, project and render the remaining visible edge segments



Hidden-Line Removal Using Edge-Face Testing

- The algorithm determines visible portions of edges of opaque closed polyhedra, which **need not** be convex
- Use computed face normals to eliminate **back** edges, which are shared by two back-facing faces. Front and silhouette edges remain potentially visible.
- Test each potentially visible edge against all non-incident faces in turn, including faces of the same object
- In the projection plane, find the portions of the projected edge that lie within the projection of the tested face. A non-convex face may produce several separate intervals of overlap. Retain the portions outside the face's projection
- In the overlapping portions, compare the depths of the edge and the face at corresponding points in the projection plane:
  - if the entire edge segment lies in front of the face, retain it,
  - if the entire edge segment lies behind the face, remove it,
  - if the edge crosses the face's plane within an overlapping segment, split the segment at the intersection and remove the portion behind the face
- Continue testing the remaining edge segments against the other faces. If the entire edge is removed, stop processing it
- After processing all faces, render the remaining visible segments. Depth sorting of faces is not required



Appel's Algorithm – Basic Idea

- Uses the classification into front, back and silhouette edges and "quantitative invisibility" (QI – *Quantitative Invisibility*), a number indicating how many occluding front-facing faces lie between the viewer and the tested point
- Potentially visible edges are **not clipped** by faces, but by silhouette edges, since edge visibility can change only where the projection of a potentially visible edge intersects the projection of a silhouette edge
- Determine QI for individual edge segments from its initial value and the changes occurring when passing **behind** silhouette edges. Render only segments with $QI = 0$
- After back edges are removed, the originally connected edge network of an object may split into several disconnected components. For each component, choose a starting vertex (*seed*) at which to compute the initial QI value
- The algorithm is relatively difficult to implement because special cases must be handled (e.g. when the projected edge passes through a projected endpoint of a silhouette edge)



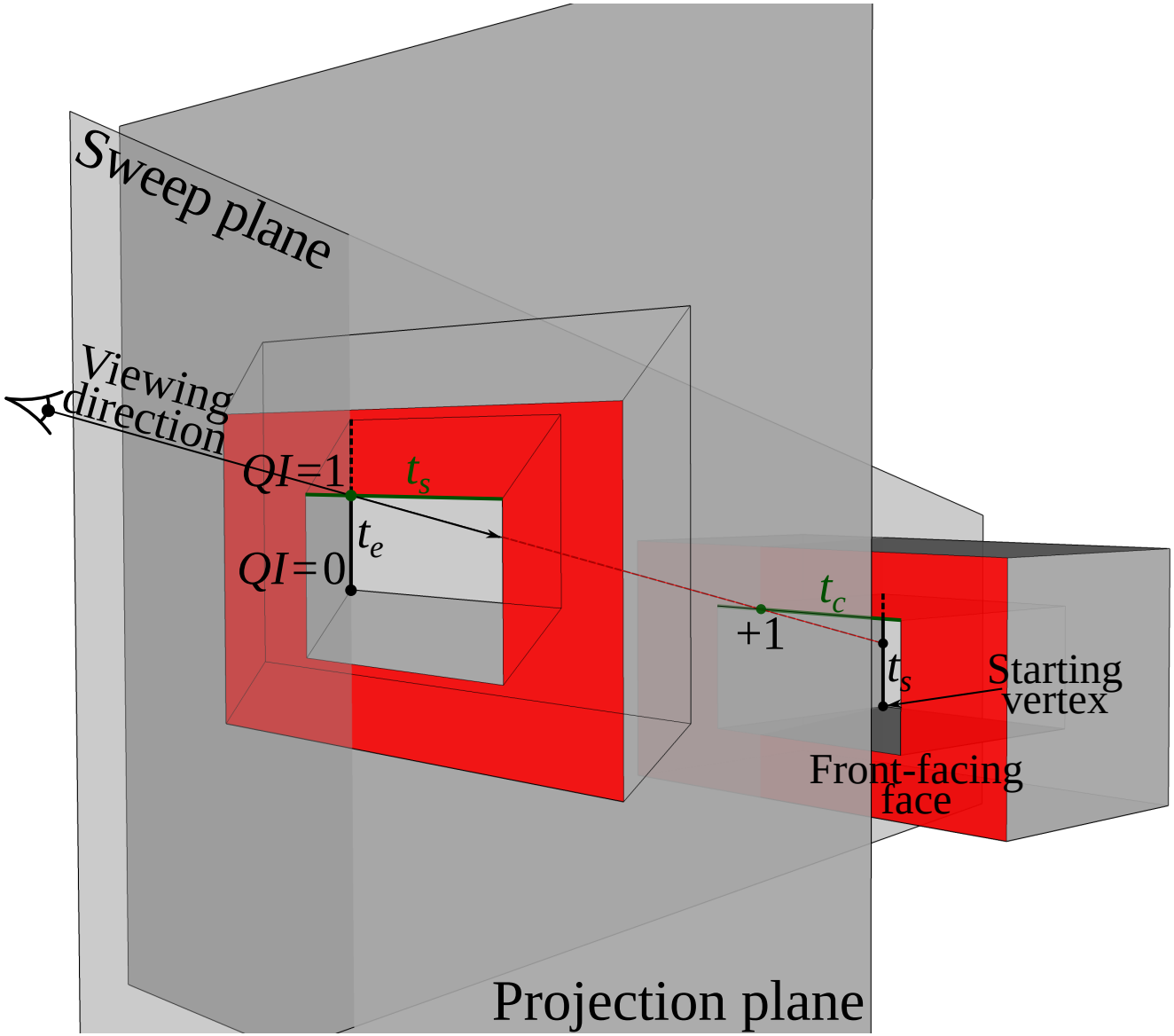
Appel's Algorithm – Edge Traversal for the Current Object

Preprocessing: Identify front-facing faces and silhouette edges throughout the scene. We assume that no edge intersects the interior of another face in 3D space (such an intersection may occur, for example, when objects interpenetrate)

1. Project the vertices onto the projection plane, retaining their 3D coordinates for subsequent depth comparisons
2. Choose a seed vertex for each connected component of the edge network being processed. Determine its QI as the number of front-facing faces intersecting the open segment between the viewer and the seed vertex
3. Traverse unprocessed front and silhouette (i.e. potentially visible) edges from vertices with known QI. For the current edge:
   1. find the intersections of its projection with the projections of silhouette edges throughout the scene,
   2. for each intersection, determine two corresponding 3D points: one on the tested edge at parameter t_e and one on the silhouette edge at parameter t_s; process only cases where the point on the silhouette edge is closer to the viewer,
   3. sort the intersections by parameter t_e in the traversal direction of the tested edge,
   4. proceed along the edge and update QI at each intersection; when entering the projection of the front-facing face adjacent to the silhouette edge, set $QI := QI + 1$; when leaving, set $QI := QI - 1$
4. Between consecutive intersections, QI remains unchanged. Render only segments with $QI = 0$, including the portions from the start of the edge to the first intersection and from the last intersection to the end of the edge
5. Derive the QI value at the end vertex and continue with the next edge. When moving between a vertex and the interior of an incident edge, account for any correction to QI caused by incident faces



Appel's Algorithm – Finding Intersections and Determining ΔQI



Rendering a 3D Function Graph

- To render the graph of a function $z = f(x, y)$ over the rectangular domain $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$, we can use the floating horizon algorithm
- Project each point (x, y, z) on the graph onto a 2D plane (the projection plane), denoting its coordinates by (x', y')
- First, render the isoparametric x-curves $(x, y_i, f(x, y_i))$ for fixed $y_0, y_1, \ldots, y_n \in [y_{min}, y_{max}]$ in front-to-back order
- For each projected x' coordinate in the projection window, store the two extreme (min/max) rendered values y'_{min}, y'_{max} in an auxiliary array
- Then use the same procedure to render the isoparametric y-curves $(x_i, y, f(x_i, y))$ for fixed $x_0, x_1, \ldots, x_n \in [x_{min}, x_{max}]$, storing the extreme values x'_{min}, x'_{max}



Rendering a 3D Function Graph – Isoparametric Curve Algorithm

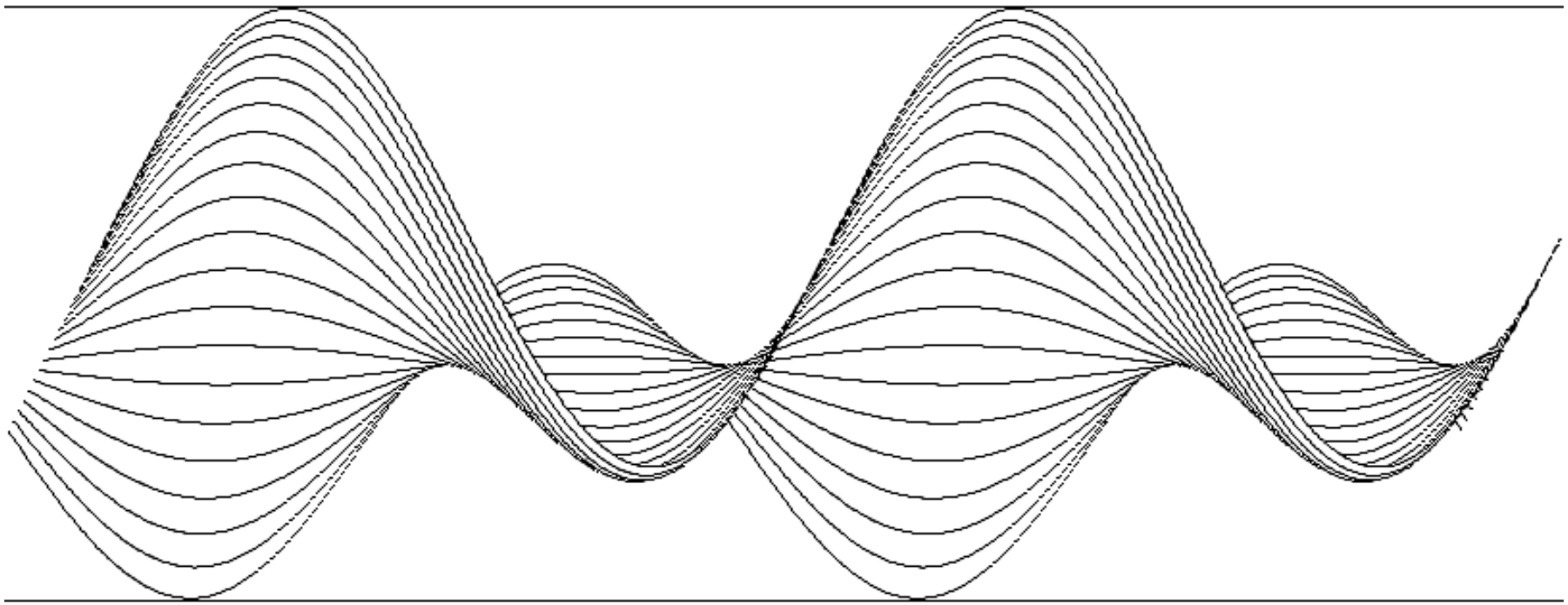
Process the isoparametric x-curves $(x, y_i, f(x, y_i))$ **from front to back**. Choose samples $x_0, \ldots, x_m$ densely enough that adjacent projected points do not skip image columns

// initialise the horizons
for each image column x':
  $y'_{\text{min}}[x'] := +\infty, \quad y'_{\text{max}}[x'] := -\infty$
for $i = 0, \ldots, n$ in front-to-back order: *// loop over all y values*
  P_{old} := rasterised projection of $(x_0, y_i, f(x_0, y_i))$ *// starting point of the x-curve*
  for $j = 0, \ldots, m$: *// loop for drawing the x-curve*
    let $P = (x', y')$ be the rasterised projection of $(x_j, y_i, f(x_j, y_i))$
    if $y' < y'_{\text{min}}[x']$ or $y' > y'_{\text{max}}[x']$: *// is the segment inside the horizon envelope?*
      draw the line segment between P_{old} and P
    $y'_{\text{min}}[x'] := \min(y'_{\text{min}}[x'], y')$ *// update the lower horizon envelope*
    $y'_{\text{max}}[x'] := \max(y'_{\text{max}}[x'], y')$ *// update the upper horizon envelope*
    $P_{\text{old}} := P$

Note: In this simplified approach, visibility is tested at the endpoints of short line segments. Accuracy at visibility boundaries therefore depends on the sampling density



Rendering a 3D Function Graph – Illustration



Part of the graph of $z = \sin(x) \cdot \cos(y)$ rendered using the floating horizon method

