Clipping & Hidden Surface Removal

Teacher: A. Prof Chengying Gao (高成英)

E-mail: mcsgcy@mail.sysu.edu.cn

School of Data and Computer Science
Viewing Pipeline Review

Diagram showing the viewing pipeline with steps:
1. View Orientation
2. Projection
3. Mapping

Diagram elements include:
- Coordinate axes (X, Y, Z)
- View orientation arrows
- Projection and mapping stages

Hongxin Zhang, 2015
Projection

Orthographic

Perspective

Computer Graphics @ ZJU

Hongxin Zhang, 2015
Why eliminating invisible objects?

- Hidden surface removal (HSR) may reduce ambiguity

(a) Cube wireframe; (b) B is the nearest; (c) C the nearestest
Why eliminating invisible objects?

• **Visible** and **invisible** portions of objects
• **Enhance reality** (增加图形的真实感)
  • Projection: 3D space → 2D space
  • 2D space: sorting according to depth may add 3D cueing
Visible Surface Determination

• Goal
  - Given: a set of 3D objects and Viewing specification,
  - Determine: those parts of the objects that are visible when viewed along the direction of projection

• Elimination of hidden parts (hidden lines and surfaces)

• Visible parts will be drawn/shown with proper colors and shade
Outline

- Clipping

- Hidden Surface Removal
Clipping

• Clipping of primitives is done usually before scan converting the primitives

• Reasons being
  • Scan conversion needs to deal only with the clipped version of the primitive, which might be much smaller than its unclipped version
How would we clip?

• 2D clipping

• Clipping is easy for Line and Polygons

• Clipping is hard for curve and Text
  • They can be converted to lines and polygons first
2D Clipping Methods

• Brute force approach:
  • compute intersections with all sides of clipping window
• Inefficient: one division per intersection (需要计算除法)
Segment-Segment Intersection

Intersection: $x$ & $y$ values equal in both representations - two linear equations in two unknowns $(r, t)$

Test if resulting $r$ & $t$ are inside the $[0,1]$ range

\[
\begin{align*}
x_0^1 + (x_1^1 - x_0^1)t &= x_0^2 + (x_1^2 - x_0^2)r \\
y_0^1 + (y_1^1 - y_0^1)t &= y_0^2 + (y_1^2 - y_0^2)r
\end{align*}
\]
Intersection with axis-aligned lines

Intersection: \( x \) & \( y \) values equal in both representations - two linear equations in two unknowns \((r,t)\)

\[
\begin{align*}
G_1 &= \begin{cases} 
   x^1(t) = x_0^1 + (x_1^1 - x_0^1)t \\
   y^1(t) = y_0^1 + (y_1^1 - y_0^1)t
   \end{cases}, \\ 
G_2 &= \begin{cases} 
   x^2(r) = x_0^2 \\
   y^2(r) = y_0^2 + (y_1^2 - y_0^2)r
   \end{cases}
\end{align*}
\]

\( t \in [0,1] \), \( r \in [0,1] \)

\( x_0^1 + (x_1^1 - x_0^1)t = x_0^2 \)

\( t = \frac{x_0^2 - x_0^1}{x_1^1 - x_0^1} \), if \( t < 0 \) or \( t > 1 \) no intersection

\( y_0^1 + (y_1^1 - y_0^1)t = y_0^2 + (y_1^2 - y_0^2)r \), (relevant only for segments)
2D Clipping Methods

• Cohen-Sutherland: Codeing
• Mid-point clipping(中点分割裁剪): Divided by 2, shift operation
• Parametric clipping (梁友栋-Barsky 裁剪): High efficiency
• Nicholl-Lee-Nicholl: More precise
• ......
Cohen-Sutherland Algorithm

• Idea: eliminate as many cases as possible without computing intersections

• Start with four lines that determine the sides of the clipping window

\[
\begin{align*}
    &x = x_{\min} \\
    &x = x_{\max} \\
    &y = y_{\min} \\
    &y = y_{\max}
\end{align*}
\]
The Cases

• Case 1: both endpoints of line segment inside all four lines
  • Draw (accept) the line segment as is
    
    \[ x = x_{\min} \quad y = y_{\max} \]
    \[ x = x_{\max} \quad y = y_{\min} \]

• Case 2: both endpoints of line segment on same side of a line
  • Discard (reject) the line segment
The Cases

• Case 3: One endpoint inside, one outside
  • Must do at least one intersection

• Case 4: Both outside
  • May have part inside
  • May the whole segment be out of windows

\[ x = x_{\text{min}} \quad \quad y = y_{\text{max}} \quad \quad x = x_{\text{max}} \]
Defining Outcodes

• For each endpoint, define an outcode:

\[ b_0 \ b_1 \ b_2 \ b_3 \]

\[ b_0 = 1 \text{ if } y > y_{\text{max}}, \ 0 \text{ otherwise} \]
\[ b_1 = 1 \text{ if } y < y_{\text{min}}, \ 0 \text{ otherwise} \]
\[ b_2 = 1 \text{ if } x > x_{\text{max}}, \ 0 \text{ otherwise} \]
\[ b_3 = 1 \text{ if } x < x_{\text{min}}, \ 0 \text{ otherwise} \]

\begin{tabular}{c|c|c|c}
\hline
& 1001 & 1000 & 1010 \\
\hline
y = y_{\text{max}} & 0001 & 0000 & 0010 \\
\hline
y = y_{\text{min}} & 0101 & 0100 & 0110 \\
\hline
x = x_{\text{min}} & x = x_{\text{max}} & & \\
\end{tabular}

• Outcodes divide space into 9 regions
• Computation of outcode requires at most 4 subtractions
(outcode_1 OR outcode_2) == 0    line segment is inside

(outcode_1 AND outcode_2) != 0    line segment is totally outside

(outcode_1 AND outcode_2) == 0    line segment potentially crosses clip region

False positive

Some line segments that are classified as potentially crossing the clip region actually don’t
Using Outcodes

• Consider the 5 cases below
• AB: (outcode(A) OR outcode(B) == 0)
  • Accept line segment
Using Outcodes

• EF: \((\text{outcode}(E) \ \text{AND} \ \text{outcode}(F) \neq 0)\)
  
  • Both outcodes have a 1 bit in the same place
  
  • Line segment is outside of corresponding side of clipping window

• reject
Using Outcodes

• CD: \((\text{outcode (C) AND outcode(D) } == 0)\)

  • Compute intersection

  • **Location** of 1 in outcode(D) determines which edge to intersect with

  • Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two interceptions
Using Outcodes

- GH and IJ: same outcodes, logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm
Check Line $P_1P_2$:

(1) If $P_1P_2$ is completely inside, accept it; if $P_1P_2$ is completely outside, reject it; otherwise go to step 2;

(2) Find an end point $P_1$ (or $P_2$) of line $P_1P_2$ outside of region;

(3) Find the intersection point $P'_1$ to replace $P_1$ (or $P_2$);

(4) If $P_1P_2$ is completely inside, then accept this line, else go to step 2.
Efficiency

• In many applications, the clipping window is small relative to the size of the entire data base
  • Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
• Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step
Polygon clipping

- It’s harder than clipping segment.
  - Clipping a segment produce a segment at most.
  - Clipping a polygon may produce several polygons.

- To convex polygon, clipping a polygon only produces a polygon.
Polygon clipping

- One method is to replace non-convex (c) polygons with a group of triangles, this process is called tessellation.
- This also makes filling simpler.
- In GLU, there is a tessellation code, but the best method is for the user to perform it themselves.
Sutherland-Hodgeman algorithm

- Present the vertices in pairs
  - \((v_n, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\)
  - For each pair, what are the possibilities?
  - Consider \(v_1, v_2\)

---

Diagram:
- Inside | Outside
- \(v_1\) | \(v_2\) output \(v_2\)
- \(v_2\) | \(v_1\) output \(i\)
- Inside | Outside
- \(v_2\) | \(v_1\) no output
- Inside | Outside
- \(v_1\) | \(v_2\) output \(i\) and \(v_2\)
Example

$V_5, V_1$

$V_2, V_3, V_4, V_5$

Inside, Inside
Output $v_1$

Outside

Inside

Outside

Current
Output
Inside, Inside
Output \( v_2 \)

Current Output
\( v_4, v_5 \) – last edge...

Outside, Inside
Output \( i_2, v_5 \)

Current Output
不是直接对复杂多边形进行裁剪，而是先用一个方向与坐标轴平行的立方体或其它形状包围多边形

- 包围盒应尽可能得小
- 容易计算出坐标的最大值与最小值
通过直接基于包围盒确定多边形的接受与抛弃

接受

抛弃

需要更仔细的裁剪处理
Outline

• Clipping

• Hidden Surface Removal
Hidden surface removal

- **Object Space Method** (对象空间)
  - a.k.a. Object Precision
  - Work in 3D before scan conversion
  - Usually independent of resolution
    - Important to maintain independence of output device (screen/printer etc.)
  - Hidden Line/surface Remove

- **Image Space Method** (图像空间)
  - a.k.a. Image Precision
  - Work on per-pixel/per of fragment after scan conversion
  - Much faster, but resolution dependent
  - Z-Buffer/Depth Buffer
for(each object in the world) {
    determine those parts of the object whose view is unobstructed by other parts of it or any other object;
    draw those parts in the appropriate color;
}

Framework of HSR in object space
Features

• High preciseness, independent of resolution of display devices (适合于精密的CAD工程领域)

• Complexity $O(n^2)$:
  • Each object should be compared with the other
  • $n$: object number

• Back surface culling,...
Framework of HSR in image space

for (Each pixel in the image) {
    connect the pixel and the viewpoint
    find the nearest object;
    compute the color for the pixel;
}

Diagram showing the framework of HSR in image space with a perspective view of a 3D scene.
Features

• The image is constrained by resolution of the display devices

• Complexity O(nN):
  • Objects should be sorted for each pixel (use coherence!)
  • n: the number of primitives (polygons)
  • N: the number of pixels

• Algorithms: z-buffer
Object Space Method

- Determine visibility on object or polygon level
  - Using camera coordinates
- Resolution independent
  - Explicitly compute visible portions of polygons
- Early in pipeline
  - After clipping
- Requires depth-sorting
  - Painter’s algorithm
  - BSP trees
Back face culling

- In a closed polygonal surface
  - i.e. the surface of a polyhedral volume or a solid polyhedron
  - The faces whose outward normals point away from the viewer are not visible
  - Such back-facing faces can be eliminated from further processing

- Elimination of back-faces is called back-face culling
Back face culling

• Let $V$ be the viewing direction from the object to the camera; $n$ the normal of the face to be tested
  
  • $N \cdot V < 0$: invisible
  • $N \cdot V \geq 0$: visible
Back face culling

- Determine back & front faces using sign of inner product $nv$
  \[ n \cdot v = n_x v_x + n_y v_y + n_z v_z = \|n\| \cdot \|v\| \cos \theta \]

- In a convex object:
  - Invisible back faces
  - All front faces entirely visible $\Rightarrow$ solves hidden surfaces problem

- In non-convex object:
  - Invisible back faces
  - Front faces can be visible, invisible, or partially visible
Limitations

• Only applicable to convex polyhedra
Painter’s Algorithm

- Simple: render the polygons from back to front, “painting over” previous polygons

- Draw cyan, then green, then red
- Will this work in general?
For 2D application

Draw items one at a time
For 2D application

Draw items one at a time
For 2D application

Draw items one at a time
Painter’s Algorithm: Problem

What Order?
Painter’s Algorithm: Problem

What Order?

“Depth” is changing
Painter’s Algorithm: Problem

- Intersecting polygons present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:
Z-buffer algorithm

• Image precision algorithm

- Apart from a frame buffer $F$ in which color values are stored,
- it also needs a z-buffer, of the same size as the frame buffer, to store depth ($z$) values

F-Buffer  Z-Buffer

A.K.A. depth-buffer method
Z-buffer algorithm

- What happens if multiple primitives occupy the same pixel on the screen?
- Which is allowed to paint the pixel?
Z-buffer algorithm
Z-Buffer Pseudo-code

- for ( j=0; j<SCREEN_HEIGHT; j++ )
  - for ( i=0; i<SCREEN_WIDTH; i++ ) {
    - WriteToFrameBuffer(i, j, BackgroundColor);
    - WriteToZBuffer(i, j, MAX);
  }

- for ( each polygon )
  - for ( each pixel in polygon's projection ) {
    - z = polygon's z value at (i, j) ;
    - if ( z < ReadFromZBuffer(i, j) ) {
      - WriteToFrameBuffer(i, j, polygon's color at (i, j));
      - WriteToZBuffer(i, j, z);
    }
  }
Z-Buffer Pros

- Simple!!!
- Easy to implement in hardware
  - Hardware support in all graphics cards today
- Polygons can be processed in arbitrary order
- Easily handles polygon interpenetration
Z-Buffer cons

- Poor for scenes with high depth complexity
  - Need to render all polygons, even if most are invisible

- Shared edges/overlaps handled inconsistently
  - *Ordering dependent*
Binary Space Partitioning Trees

- **BSP Tree**
  - Very efficient for a static group of 3D polygons as seen from an arbitrary viewpoint
  - Correct order for Painter’s algorithm is determined by a suitable traversal of the binary tree of polygons
BSP Tree
BSP Tree
Binary Space Partition Trees

- **BSP Tree**: partition space with binary tree of planes
- **Idea**: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene

  - **Preprocessing**: create binary tree of planes
  - **Runtime**: correctly traversing this tree enumerates objects from back to front
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Splitting Objects

- No bunnies were harmed in previous example
- But what if a splitting plane passes through an object?
  - Split the object; give half to each node
Traversing BSP-Trees

- Tree creation independent of viewpoint
  - Preprocessing step
- Tree traversal uses viewpoint
  - Runtime, happens for many different viewpoints
Traversing BSP-Trees

- Each plane divides world into near and far
  - For given viewpoint, decide which side is near and which is far
    - Check which side of plane viewpoint is on independently for each tree vertex
    - Tree traversal differs depending on viewpoint!

- Recursive algorithm
  - Recurse on far side
  - Draw object
  - Recurse on near side
Traversing BSP-Trees

rasterize(C)
rasterize(A)
rasterize(B)
rasterize(B)
rasterize(A)
rasterize(C)
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A

- decide independently at each tree vertex
- not just left or right child!
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint A
BSP-Trees: Viewpoint B
BSP-Trees: Viewpoint B
BSP Tree Construction: Polygons

- The binary tree is constructed using the following principle:
  - For each polygon, we can divide the set of other polygons into two groups
  - One group contains those lying in front of the plane of the given polygon
  - The other group contains those in the back
  - The polygons intersecting the plane of the given polygon are split by that plane
BSP Tree Traversal: Polygons

- Split along the plane defined by any polygon from scene
- Classify all polygons into positive or negative half-space of the plane
  - If a polygon intersects plane, split polygon into two and classify them both
- Recurse down the negative half-space
- Recurse down the positive half-space
Summary: BSP Trees

Pros:
- Simple, elegant scheme
- Correct version of painter’s algorithm back-to-front rendering approach
- Still very popular for video games (but getting less so)

Cons:
- Slow(ish) to construct tree: $O(n \log n)$ to split, sort
- Splitting increases polygon count: $O(n^2)$ worst-case
- Computationally intense preprocessing stage restricts algorithm to static scenes
BSP Demo

• Useful Demo
  
  http://www.symbolcraft.com/graphics/bsp/