Introduction to 2D and 3D Computer Graphics

The Rendering Process

Hidden Line
and
Hidden Surface Removal
(HLHSR)
Discuss HLHSR Techniques

- General Concepts
- Trival Rejection
- Backface Culling
- Depth Sorting
- Z Buffering
Many 3D objects have surfaces that are planar polygons... for example, cubes, pyramids, prisms, etc.

Many more complex objects are often built from these.

Curved surfaces can often be approximated by planar polyhedrons joined together... simulating the actual surface.
The curvature of a cylinder can be represented by many long narrow rectangles.

The advantage of using planar polygons to represent your objects is...

- that the display algorithms can take advantage of this and produce the objects by simply drawing the edges for wireframes or by simply filling the polygons.

But...objects may appear ambiguous because edges may be shown that wouldn't be seen in real life!
The Rendering Process

Hidden Line and Hidden Surface Removal

Introduction

- HLHSR is the process of determining which lines or surfaces of objects are visible, either from the center of projection for perspective projections or along the direction of projection for parallel projections.

- Allows only the visible lines or surfaces to be displayed.
The Rendering Process

*Hidden Line and Hidden Surface Removal*

*Introduction*

- Provides both *image-precision* and *object-precision* methods

- Image-precision is performed at the resolution of the device and determines the visibility of each pixel; it is performed AFTER rasterization

- Object-precision is performed in NPC space (after viewing); it is followed by operations to map to device coordinates and physically render the picture
The Rendering Process

Hidden Line and Hidden Surface Removal

Introduction

- Image-precision...
  - ...is a simple approach for each pixel in the picture:
  - (a) determines the object closest to the viewer that is pierced by the projector through the pixel, and
  - (b) draws the pixel in the appropriate color
Object-precision...

- ...compares objects directly with one another
  ...for each object in the picture:
  - (a) determines those parts of the object whose
    view is unobstructed by other parts of it or any
    other object
  - (b) draws those parts in the appropriate color
n Image-precision...and...object-precision...
 – ...require a number of costly operations such as:
   – ...determining whether or not a projecter and an object intersect and where they intersect
   – ...determining the intersection for multiple projections/objects
   – ...for intersections, determining which object is closest to the viewer and therefore visible
The Rendering Process

Hidden Line and Hidden Surface Removal

Introduction

- HLR/HSR algorithms can be optimized using techniques of...
  - coherence
  - depth comparisons
  - trivial rejection: *pruning*
  - backface culling

- Common HLR/HSR algorithms include...
  - backface culling
  - depth-sorting
  - Z-buffering
Coherence...

- ...is the degree to which parts of a projection of an object have similar features
- ...takes advantage of properties of objects that vary smoothly from one part to another
- ...allows calculations to be reused without modification or with incremental changes more efficiently than recalculating the information from scratch
Coherence includes...

- ...object coherence: takes advantage of one object's location over another's and only compares objects to see if they are completely separate

- ...face coherence: since surface properties typically vary smoothly across a face, computations for one part of a face can be modified incrementally to apply to adjacent parts
n Coherence includes...

- ...edge coherence: *an edge may change visibility only where it crosses behind a visible edge or penetrates a face*

- ...depth coherence: *adjacent parts of the same surface are typically close in depth; once the depth is calculated, the depth of the rest of the surface can be calculated using simple difference equations*
The Rendering Process

Hidden Line and Hidden Surface Removal
Optimized Techniques

Depth comparisons...

- ...determine if two or more points obscure one another (*i.e.*, *if two or more points lie on the same projector*)
- ...determine which point is closer when points have the same projector
The Rendering Process

Hidden Line and Hidden Surface Removal

Optimized Techniques

- If two points have the same projector, then the closer one obscures the other.

- Remember, HLR/HSR occurs in 3D space prior to mapping to 2D, since depth info is lost in 2D...
  - ...it also occurs in NPC space after viewing, which simplifies the comparisons for parallel and perspective projections, making projectors parallel to the z axis with the center of projection at infinity on the positive z axis. (points are obscured if $x_1 = x_2$ and $y_1 = y_2$)
Trivial rejection...called *pruning*

- ...is a simple way to avoid unnecessary comparisons
- ...again uses the concept that viewing (i.e., perspective) has already been applied
- ...requires simple orthographic projections of objects onto the XY plane by ignoring the z values
- ...if extents do not overlap, projections do not need to be tested for overlap with one another
  - ...if extents do overlap, projectors must be tested for overlap...however this is not a guarantee that overlap occurs
The Rendering Process

Hidden Line and Hidden Surface Removal

Optimized Techniques

- Trivial rejection...called *pruning*

Two objects with their projections onto the XY plane and the extents surrounding the projections

Examples where extents overlap & projectors are tested for overlap; remember there is no guarantee!
The Rendering Process

Hidden Line and Hidden Surface Removal

Optimized Techniques

- Trivial rejection...using bounding volumes...
  - ...uses a bounding volume around each entire object instead of using XY projections

- Trivial rejection...using a single dimension...
  - ...selects a single dimension to determine if overlap occurs
  - ...for example, could determine if objects overlap in z
The Rendering Process

Hidden Line and Hidden Surface Removal

Optimized Techniques

- Trivial rejection...

Two objects with their bounding volumes

Using one dimensional extends to determine if objects overlap
Backface culling...an object-precision technique...

- ...is useful for objects approximated by polyhedra
- ...eliminates backfacing polygons (i.e., polygonal faces whose surface normals point away from the observer and whose visibility is completely blocked by other closer polygons)
- ...since viewing has already occurred, the direction of projection is (0,0,-1) which means surface normals with negative z coordinates are backfacing
n Backface culling...an object-precision technique...

- ...for a single convex polyhedra, backface culling is the only HLR/HSR algorithm needed!

*Backfacing polygons are eliminated but frontfacing polygons are retained*
The Rendering Process
Hidden Line and Hidden Surface Removal
Optimized Techniques
To perform HLHSR requires...
  - considerable computation to distinguish between visible and invisible edges of an object

Instead, it is easier to find those planar polygons that are not to be shown, rather than to find their edges!
The Rendering Process

*Hidden Line and Hidden Surface Removal*

*Back Face Removal*

- Backface culling...
  - ...is particularly useful for polygon mesh objects
  - ...removes a significant portion of the hidden lines from a wireframe display
  - ...removes ALL hidden lines from convex objects
  - ...is a low cost operation ...will only remove about 50% of surfaces when there are multiple overlapping objects
The Rendering Process

Hidden Line and Hidden Surface Removal

Back Face Removal
These algorithms assume...
  - ...we are describing our objects as polygon meshes

Remember a polygon mesh...
  - ...contains many planar surfaces and straight edges
  - ...which shouldn't be thought of as simply unrelated sets of polygons ...instead, it consists of a mesh of polygons that are adjacent
The Rendering Process

Hidden Line and Hidden Surface Removal
Back Face Removal

- With a polygon mesh...
  - ...we should record the sequence of vertices or edges that compose the individual polygons

- A polygon mesh should allow us to...
  - ...identify a specific polygon in the mesh
  - ...identify all edges belonging to a polygon
  - ...identify those polygons that share an edge
  - ...identify the vertices of an edge
  - ...change the mesh
  - ...display the mesh
An explicit polygon mesh...

- ...stores each vertex in a "vertex table" ...defines a polygon as a sequence of vertices ...which can be realized by defining the polygons as linked lists of pointers into the vertex list.
The advantages of this approach is...
  – ...it requires the least amount of storage...and easily allows for the mesh to be changed
  – (i.e., to change one vertex - we only have to change its coordinates in the vertex list)
n An explicit edge mesh...

- ...is an alternative representation which overcomes the disadvantages of an explicit polygon mesh
- **Table of vertices**: each vertex is stored once
- **Linked list of edges**: where an edge is connected by two vertices ... where every edge has a pair of pointers to the vertex table, a pointer into the polygon list and a counter showing the # of polygons that share edge
- **List of polygons**: a linked list of pointers into the edge list which access (in the correct order) all of the edges that compose that particular polygon
The Rendering Process

Hidden Line and Hidden Surface Removal
Back Face Removal

Polygon List

polygon 1: E1 → E2 → E3 → 0

polygon 2: E2 → E4 → E5 → 0

List of Edges

Edge 1: V1 → V2 → 1 → P1 → 0

Edge 2: V2 → V3 → 2 → P1 → P2 → 0

Edge 3: V3 → V1 → 1 → P1 → 0

Edge 4: V2 → V4 → 1 → P2 → 0

Edge 5: V4 → V3 → 1 → P2 → 0
The Rendering Process

Hidden Line and Hidden Surface Removal

- Backface culling...
  - ...uses a viewpoint (usually the center of projection)
  - ...determines whether or not a polygon is visible from the viewpoint
  - ...determines visibility by calculating the angle between the polygon surface normal and the line-of-sight vector
  - ...visibility = \( \text{Normal} \cdot \text{Viewpoint} \)
The Rendering Process

Hidden Line and Hidden Surface Removal

- Viewpoint
- Line-of-sight vectors
- Visible
- Invisible
- >90°
- <90°
- Surface Normal
Backface culling...

- ...calculates the surface normal from three noncollinear vertices
- ...these define two vectors contained in the plane of the surface
- ...takes the cross-product which gives the surface normal
- ...if all three points occur in a counterclockwise direction around the surface, when you look from the outside, the normal will point outwards
Another way to look at these calculations...

- ...is that if you have a point that lies on the same side of the plane (the positive side) as your viewpoint
- ...then, the angle between the vector created from this viewpoint to the plane and the surface normal must be less than $\frac{\pi}{2}$
- ...which means: (where $\tilde{\theta}$ is the angle between the vectors)
Surface Normal • Viewpoint Vector = |Surface Normal| * |Viewpoint Vector| * cos Ţ

will result in a value >0

This tells us that a positive result will be calculated for every point on the side of a plane that is within view of our viewpoint...

...and therefore, that a negative result will be given for all planes that cannot be seen from the viewpoint!
Let's step through an example...

- the first step is to apply the viewing transformation
- then calculate the surface normal... to do this, a common technique is to use three points on that plane

So, let's start our example, with 3 points P1, P2, and P3 with coordinates (x1,y1,z1), (x2,y2,z2), and (x3,y3,z3)

...these three points are noncollinear!
Note that the direction of projection is (0,0,-1)...

- ...in this case the dot product test reduces to checking the z-coordinate of the surface normal
- ...if the z-coordinate is negative then the surface is backfacing

Continuing with this example...

...we can use these three points to specify two vectors
...where Vector1 = (X2-X1, Y2-Y1, Z2-Z1)
...and where Vector2 = (X3-X1, Y3-Y1, Z3-Z1)
But watch out...

- ...remember that the points P1, P2, P3 must be specified in a counterclockwise direction around the surface, as you look from the outside so that the normal will point outwards.
  ...otherwise the direction of the normal vector will be reversed and you will get the opposite of the desired effect!
The Rendering Process

Backface Culling

Cross Product of the Two Vectors is

\[(Y_2 - Y_1)(Z_3 - Z_1) - (Z_2 - Z_1)(Y_3 - Y_1)\]
\[(Z_2 - Z_1)(X_3 - X_1) - (X_2 - X_1)(Z_3 - Z_1)\]
\[(X_2 - X_1)(Y_3 - Y_1) - (Y_2 - Y_1)(X_3 - X_1)\]
If we assume that the eye is positioned along the positive Z-axis looking toward the origin:

Where,

\[ P1 = (0, 0, 0) \]
\[ P2 = (10, 0, 0) \]
\[ P3 = (10, 10, 0) \]
\[ P4 = (0, 10, 0) \]
\[ P5 = (10, 0, -10) \]
\[ P6 = (0, 0, -10) \]
\[ P7 = (0, 10, -10) \]
\[ P8 = (10, 10, -10) \]
• Plane 1 (the front plane) is defined by the points:

\[ P1 = (x1,y1,z1) = (0,0,0) \]
\[ P2 = (x2,y2,z2) = (10,0,0) \]
\[ P3 = (x3,y3,z3) = (10,10,0) \]

Where,

\[ \text{Vector1} = (x2-x1,y2-y1,z2-z1) \]
\[ \text{Vector2} = (x3-x1,y3-y1,z3-z1) \]
To determine if plane 1 is visible, we must calculate the surface normal... ...which is the cross product of vector1 and vector2:

\[
(y_2 - y_1)(z_3 - z_1) - (z_2 - z_1)(y_3 - y_1) \\
(z_2 - z_1)(x_3 - x_1) - (x_2 - x_1)(z_3 - z_1) \\
(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)
\]
The Rendering Process

Backface Culling

- Plane 2 (the back plane) is defined by the points:

\[
P5 = (10, 0, -10) \\
P6 = (0, 0, -10) \\
P7 = (0, 10, -10)
\]

Where,

Vector1 = \( (x6-x5, y6-y5, z6-z5) \)
Vector2 = \( (x7-x5, y7-y5, z7-z5) \)
To determine if plane 2 is visible, we must calculate the surface normal... ...which is the cross product of vector1 and vector2:

\[
\begin{align*}
(y_6 - y_5)(z_7 - z_5) - (z_6 - z_5)(y_7 - y_5) \\
(z_6 - z_5)(x_7 - x_5) - (x_6 - x_5)(z_7 - z_5) \\
(x_6 - x_5)(y_7 - y_5) - (y_6 - y_5)(x_7 - x_5)
\end{align*}
\]
The Rendering Process

Backface Culling

- Plane 3 (the bottom) is defined by the points:

\[ P5 = (10,0,-10) \]
\[ P2 = (10,0,0) \]
\[ P1 = (0,0,0) \]

Where,
\[ \text{Vector1} = (x_2-x_5,y_2-y_5,z_2-z_5) \]
\[ \text{Vector2} = (x_1-x_5,y_1-y_5,z_1-z_5) \]
To determine if plane 3 is visible, we must calculate the surface normal... ...which is the cross product of vector1 and vector2:

\[
\begin{aligned}
(y_2 - y_5)(z_1 - z_5) - (z_2 - z_5)(y_1 - y_5) \\
(z_2 - z_5)(x_1 - x_5) - (x_2 - x_5)(z_1 - z_5) \\
(x_2 - x_5)(y_1 - y_5) - (y_2 - y_5)(x_1 - x_5)
\end{aligned}
\]
In summary, backface culling is best used to remove hidden surfaces or hidden lines for a single convex object. 

- ...which means the front faces are always visible ...and the only thing that needs to get done is to remove the back faces

If an object is concave...

- ...some front faces can be partly or totally hidden by other parts of the object
If several objects are displayed...even if they are all convex...
  - ...one object may cover parts of another

Backface culling does not handle either of these cases...
  - ...but is still useful as a preprocessor for more general HLHSR techniques
  - ...and, will typically reduce the number of polygons that more general algorithms need to process by about 50%
The Rendering Process

Depth-Sorting

Depth-sorting algorithms...

- ...are more powerful than backface culling
  ...because they are not restricted to single convex objects

- ...allowing pictures to contain multiple concave and convex objects

- ...but they still require that pictures be defined using all planar polygonal surfaces
The Rendering Process

*Depth-Sorting*

- Depth-sorting algorithms do impose restrictions...
  - ...pictures cannot have surfaces that mutually penetrate each other or penetrate themselves
  - ...polygons are stored in one or more polygon meshes
  - ...viewing transformations are done prior to depth-sorting ...which means that perspective projection has already been performed
The Rendering Process

*Depth-Sorting*

- Depth-sorting paints polygons in order of decreasing distance from the viewpoint *(painters algorithm)*...
  - ...by sorting all polygons according to the smallest z values
  - ...by resolving ambiguities from polygons with overlapping z extents by splitting up the polygons
  - ...by scan converting each polygon in ascending order of smallest z coordinate
Once we have determined which polygons are not backfacing...

- ...we decompose the polygon into triangles...
  creating the front faces of the entire picture in the form of triangles ...
  ...for example:

Where, we start with vertices: A, B, C, D

And, we decompose it into two triangles:

A, B, C
A, C, D

So, a polygon with 4 vertices is decomposed into two triangles
The Rendering Process

Depth-Sorting

- The next step...
  - ...is to sort these "nonbackfacing" triangles into order based on their depth

- To create a picture with solid objects...
  - ...we simply begin drawing the triangles that are farthest away first
  - ...which means that if triangles overlap, the closest triangle will appear on top -- automatically hiding all or part of the triangle further behind
The Rendering Process

Depth-Sorting

- Do you see now why this is called the painter's algorithm?
- To create a picture with wireframe objects...
  - ...we must remove hidden lines
  - ...which is actually more complex because a line can be totally hidden by some triangle or cut so that only one portion remains OR cut so that two portions remain!
The Rendering Process

*Depth-Sorting: decomposition*

Before we decide the depth for each polygon...
- ...it is easiest if we first decompose them into triangles
- ...because triangles are the simplest form of polygon
- ...and because triangles are always convex!

So, all polygons that consist of 4 or more sides...
- ...should be decomposed into triangles

If polygons are convex......decomposition is easy

If polygons are concave......decomposition is more complicated
The process of decomposition (for convex or concave polygons)...

- ...begins at the leftmost vertex
- ...which can be found by finding the smallest x-value of all polygon vertices
- ...and take the next two vertices adjacent to it
- ...which forms the leftmost triangle
The next step is to determine if the polygon is concave...

- ...if no other vertex of the polygon lies inside our newly formed triangle, we can cut it off as a triangle and keep it separate from the rest of the polygon
- ...which can be done by a simple min/max test to eliminate the simple cases
- ...by first finding the smallest rectangle containing the triangle and then checking to see if any other vertices of the polygon lie inside this rectangle.
For the elimination of the easy cases...

- ...let's first find the smallest rectangle containing the triangle and then check to see if the rest of the polygon vertices lie inside or outside of this rectangle:

Where, our leftmost vertex is A

and, we create our first triangle to have A, B, D as its vertices

For elimination, we create a bounding box (i.e., the smallest rectangle containing the triangle)
Now...we can quickly check if each vertex lies within the rectangle by using four checks: (if any of these conditions is true - then our polygon-vertex is definitely outside the triangle)

Polygon-Vertex $X < \text{minimum}(AX,BX,DX)$
Polygon-Vertex $Y < \text{minimum}(AY,BY,DY)$
Polygon-Vertex $X > \text{maximum}(AX,BX,DX)$
Polygon-Vertex $Y > \text{maximum}(AY,BY,DY)$

• But, since we know that the triangle is the leftmost one, we actually only need to perform the following checks:

Polygon-Vertex $Y < \text{minimum}(AY,BY,DY)$
Polygon-Vertex $X > \text{maximum}(BX,DX)$
Polygon-Vertex $Y > \text{maximum}(AY,BY,DY)$
Before we continue...

- ...let's look at some examples using this method of elimination.

Both of these cases would eliminate the need for any further testing to determine if another polygon vertex lies within the leftmost triangle.
The Rendering Process

Depth-Sorting: decomposition

- More examples using this method of elimination...

These cases require additional processing to determine if another polygon vertex lies within the leftmost triangle; we now have to actually compute whether the polygon vertice(s) fall inside the triangle.
Before we develop our algorithm to test if a point lies within a triangle...

...remember the formula for a line through two points?

\[ f(x,y) = (x-x_1)(y_2-y_1)-(x_2-x_1)(y-y_1) \]

where, \( f(x,y) = 0 \) if you lie on the line
and, \( f(x,y) < 0 \) if you are on ONE side of the line
or, \( f(x,y) > 0 \) if you are on the OTHER side of the line
So, this means that any two points that lie on the same side of the line will have the same sign for both

- ...which means for a triangle, a point lies inside if it lies on the same side as the vertex that is opposite the line
- ...this check must be done for all three sides of the triangle ...and for any of the three checks the point doesn't lie inside, then we know that it is outside the triangle
These tests must be performed for the leftmost triangle...against every polygon vertex

- ... that does not belong to that triangle
- ... and that has not been eliminated by the Min/Max test

let's look at an example where:
A=(0,0), B=(15,15), C=(13,8), D=(8,5)

C was not eliminated using the bounding box because  
CX < max (Bx,Dx)  \{yes 13 < 15\}
and CY < max(Ay,By,Dy)  \{yes 8 < 15\}
and CY > min(Ay,By,Dy)  \{yes 8 > 0\}
Now...we check polygon vertex C against all three sides of the triangle:

\[ A=(0,0), B=(15,15), C=(13,8), D=(8,5) \]

We first check C against Side AB:
\[ f(13,8) = (13-0)(15-0)-(15-0)(8-0) = 75 \]
and compare that to where D lies:
\[ f(8,5) = (8-0)(15-0)-(15-0)(5-0) = 45 \]
Since they are both positive...they lie on the same side.

Then we check C against Side BD:
\[ f(13,8) = (13-15)(5-15)-(8-15)(8-15) = -29 \]
and compare that to where A lies:
\[ f(0,0) = (0-15)(5-15)-(8-15)(0-15) = 45 \]
Since they are of opposite signs ...
this means that A and C are on opposite sides of the line formed by B & D.
Let's step through one more example...  
...using decomposition for the leftmost triangle

Let's look at an example where:
A=(0,0), B=(15,20), C=(18,10), D=(20,5)

C was not eliminated using the bounding box because ...

\[ CX < \max(Bx, Dx) \]  \{yes 18 < 20\}
\[ CY < \max(Ay, By, Dy) \]  \{yes 10 < 20\}
\[ CY > \min(Ay, By, Dy) \]  \{yes 10 > 0\}
Now...we check polygon vertex C against all three sides.

Remember:
A=(0,0), B=(15,20), C=(18,10), D=(20,5)

We first check C against Side AB:
\[ f(18,10) = (18-0)(20-0) - (15-0)(10-0) = 210 \]
and compare that to where D lies:
\[ f(20,5) = (20-0)(20-0) - (15-0)(5-0) = 325 \]
Since they are both positive...they lie on the same side.

Then we check C against Side BD:
\[ f(18,10) = (18-15)(5-20) - (20-15)(10-20) = 5 \]
and compare that to where A lies:
\[ f(0,0) = (0-15)(5-20) - (20-15)(0-20) = 325 \]
Since they are both positive...they lie on the same side.

One more test to perform...this last test had better also show that C is inside!
The Rendering Process
Depth-Sorting: decomposition

- So, our final check is...

  **Remember:**
  \[ A=(0,0), \ B=(15,20), \ C=(18,10), \ D=(20,5) \]

  To check \( C \) against Side \( DA \):
  \[ f(18,10)=(18-20)(0-5)-(0-20)(10-5) = 110 \]
  and compare that to where \( B \) lies:
  \[ f(15,20)= (15-20)(0-5)-(0-20)(20-5) = 325 \]

  Since they are both positive...they lie on the same side

  **THEREFORE...**\( C \) lies **within** this triangle
Once we have decomposed the leftmost triangle...
- ...such that there are no vertices inside...
- ...it can be split off
- ...producing two new polygons from the old one
- ...where the first new polygon is the triangle
- ...and the second new polygon is formed by what is left after extracting the leftmost vertex
If there are vertices inside the leftmost triangle...

- ...then we have to split the triangle down further

...by taking the leftmost vertex that lies within the triangle and connecting it to the first vertex of the "triangle"

where $C$ is the leftmost vertex that lies within the triangle...so now use it instead of $D$ to form the first triangle
We continue this process...
- ...until every polygon of the picture is split into triangles
- ...which enables us to now sort them by depth
- ...this process is also called: geometric sorting
To begin with...

- ...we know that our triangles will be in front of or behind other triangles
- ...which means we need to put them in order so that the farthest triangle is drawn first
- ...which will allow closer triangles to hide those farther away
To handle the cases where triangles penetrate each other...

- ...or where multiple triangles overlap one another...

- ...would require us to further decompose our triangles into smaller triangles to exclude such cases.
The easiest way to do this...

- ...is to compare if the X/Y coordinates of triangles to see if they overlap...
- ...since we want to find the places where the X and Y coordinates of 2 or more triangles are the same but whose z values differ
- ...which then will let us order their depth depending on the z-coordinates
  ...and only when they do -- do we need to compare their z coordinates
As you would guess...

...it is best to perform a bounding box method of trivial elimination before doing much mathematics ...with this we can tell right away if two triangles are even possibly overlapping ...where, for each triangle we compute the smallest bounding box and check if they overlap...if the rectangles don't overlap - neither do the triangles!
As you can guess...

- ...we apply increasingly more expensive tests as we narrow down the cases that remain after performing trivial rejection...for example, to check if two triangles overlap, we have to check each edge of one triangle against each edge of the other triangle...as soon as we find an intersection, we can stop since we know they overlap....but, what if...
After the bounding box check...

- ...we can easily see if edges intersect as the next "trivial rejection method" by performing the following:
  - Edge 1 -- \((x_1, y_1) \rightarrow (x_2, y_2)\)
  - Edge 2 -- \((x_3, y_3) \rightarrow (x_4, y_4)\)

Then, if any of the following conditions is true...

- ...we will know that the edges do not intersect
  - \(\max(x_1, x_2) < \min(x_3, x_4)\)  \(\max(x_3, x_4) < \min(x_1, x_2)\)
  - \(\max(y_1, y_2) < \min(y_3, y_4)\)  \(\max(y_3, y_4) < \min(y_1, y_2)\)
If we still think that maybe there is a possible overlap...

- ...but none of the previous tests could tell us for sure
- ...then we need to check to see if the edges of the triangles are parallel
- ...this is true if the edges have the same slopes...which is true if: 
  \[(x3-x4)(y1-y2)-(x1-x2)(y3-y4) = zero\]

Therefore, the bottom line is that if the slopes are identical, the edges are parallel and there is no intersection!
However, if the edges are not parallel, we need to continue...

- ...finding the intersection points for each edge...testing against each edge
- As a result of all of these tests... ...we now know if each pair of triangles overlaps
- If the triangles do not overlap... ...their relative depths are not important and it doesn't matter which order they are drawn in
The Rendering Process

Depth-Sorting: geometric sorting

- If the triangles do overlap...
  - ...we have to decide which triangle is in front

- To do this...it is best to start with our Min/Max test...
  - ...since if all z-coordinates of one triangle are smaller than all z-coordinates of another - we know that one triangle is in front of the other
The Rendering Process

Depth-Sorting: geometric sorting

- If some of the z-coordinates overlap for any two triangles (which overlap in XY)...  
  - ...we must determine where the z-intersections occur
  - ...and potentially further break-down the triangles

- Given all of this information...
  - ...we are finally able to determine for each pair of triangles whether they overlap and if they do which one is nearer
At this point we have...
- ...removed the backfaces thru culling
- ... decomposed the remaining polygons into triangles
- ...and sorted them into depth order

As you can guess...
- ...after all of this work, the painter's algorithm is no big deal!...we simply draw all of the triangles that are farthest away from us ...then we draw all of the triangles one step closer
Notice -- this algorithm works...

- ...only on raster devices-excluding devices such as plotters
- ...only if solid objects are drawn by filling polygons
- ...and only if drawing modes are not used
Depth Buffer Methods...

- ...are the most powerful and general techniques for hidden surface removal
- ...require an enormous memory/buffer space
- ...can give a realistic display of the most complex pictures
Depth Buffer Methods only require...

- ...that objects be displayed using polygons (polygon meshes) allowing polygons to be any shape (doesn't have to be convex) and allowing surfaces to penetrate one another
  ...should be preceded by a simple back face culling operation

Doing back face culling first will...

- ...cut the computation time about in half
The Rendering Process
Z-Buffer or Depth Buffer Methods

- These methods work for raster displays...
  ...and therefore cannot be used on vector devices or plotters

- Takes advantage that pictures can be decomposed into pixels...
  - ...which means we don't have to compute intersections of lines or polygons with other lines or polygons  ...we simply reduce to testing whether a given pixel is or is not within a certain polygon
A depth computation is necessary to find the distance a point lying within the polygon is from the center of projection.

These methods are conceptually very simple...

- ...but require intensive computations and memory
- ...which can handle all hidden surface problems no matter how complex
Think for a moment...

- ...as we look at a polygon in space
- ...what we actually see is its projection
- ...and the image on the screen is made visible by setting the pixel values that lie within the projection of this polygon's boundary
Therefore...it can happen that several polygons - that are totally disjoint in our world

- ...can overlap on the screen due to the projection calculations (i.e., viewing transformation)
- ...which means that the same pixels can be occupied by the projections of different polygons
- ...in such a case - we only want to set the pixels to the color of the polygon which is closer at this point
  ...which means...the distance calculation has to be done for every pixel in the projection of a polygon!
One Depth Buffer Method is... ...Z-Buffering

Z-buffering....
  – ...is an image-precision algorithm
  – ...is one of the simplest algorithms to implement
  – ...requires memory for a z-buffer to store pixels' z values
  – ...allows objects to be drawn in any order
  – ...occurs as part of the rasterization process, after objects are scan converted
The Rendering Process

Z-Buffer or Depth Buffer Methods

n Z-buffering....

- ...draws on top of current images if the object being drawn is no farther from the viewer than is the point whose color and depth are currently being evaluated....the new point's color and depth replace the old values

- ...can be implemented as a full z-buffer storing each pixel's z value or a scan line z-buffer storing only a scan line at a time
The Rendering Process

Z-Buffer or Depth Buffer Methods

- Z-buffering requires...
  - ...no presorting of objects
  - ...no object by object comparisons

- Instead, the projection of each polygon is scanned...
  - ...scan line by scan line
  - ...and within each scan line pixel by pixel
For each pixel the distance or depth at that pixel is stored in the Z-buffer (also known as the depth buffer)

This stored value is then compared to the depths of other polygon points that project onto this same pixel.

To do this...

- Z-buffering has as many storage cells as there are pixels on the screen... ...where each storage cell must be able to hold the value of the depth
Remember...

- ...our goal with Z-buffering is to put into the Z-buffer the distance of the point of the polygon that is closest to the screen.

So...

- ...one way to do this would be to initialize the Z-buffer with a number as large or larger than any distance could be
- ...as we would do with any minimum-finding procedure
Then... when we scan a polygon...

- ...for each pixel we enter the distance of the corresponding polygon point if it is less than (or equal) the number already in the Z-buffer location

- ...and, at the same time, we set the frame buffer for that pixel to the color of that polygon point

- Which means, only the depth of the polygon closest will survive in the Z-buffer and we will only use the color for that surviving polygon
Let's take a look at the Z-buffer algorithm...

- ...for a single polygon in a picture with many polygons

- ...where the steps described must be repeated for each polygon

Let's also start with an assumption that we have already sent the polygon thru the viewing transformation...

- ...and we are working in Normalized Projection Coordinates (NPCs) which range from 0.0 to 1.0
The Rendering Process

Z-Buffer or Depth Buffer Methods

So, our Z-buffer might be able to contain depth values for each pixel -- which range between 0.0 and 1.0.

Given all of this, we also must assume that the Z-buffer is originally filled with all 1's...
  ...which is the highest relative depth in NPC space

For the (x,y) coordinates of the polygon...
  we must also transform them from NPCs to Device Coordinates......and then scan convert to locate the pixels for this polygon
The Rendering Process
_Z-Buffer or Depth Buffer Methods_

Then, for a given pixel on the screen...

- ...the z-coordinate in NPCs is the relative depth
- ...where every depth value in the polygon will be compared with the Z-buffer value for that pixel
- ...if the depth of the pixel is smaller (or equal) than the Z-buffer value previously stored for this pixel, we replace the Z-buffer entry with the smaller value and set this pixel in the frame buffer to the color of the polygon
If we are doing smooth shading...
  - ...which we will learn about next class
  - ...we compute this pixel's illumination and set the pixel accordingly

Now...we go back and process the next polygon...
  ...repeating until we have processed all polygons
The Rendering Process

Z-Buffer or Depth Buffer Methods

- Z-buffering algorithm...
  - for each object {
    - for each pixel in the object's projection {
      - if (object's z value at (x,y) \(\geq\) current pixel's z value) {
        - write the object's pixel value (x,y)
        - write the object's z value into z-buffer
      }
    }
  }
Another way to implement Depth Buffering Methods...

- ...is by using scan line algorithms
- ...where we use a Z-buffer the size of one scan line ...called a scan line buffer

With this method - is it far more efficient to first put the polygons in an order that will facilitate processing
To get this order...

- ...we must process a polygon one at a time
- ...and perform the following steps:
  - 1) Performing viewing and clipping on the polygon
  - 2) Transform the vertices to device coordinates
  - 3) Add this polygon to a data structure that contains the projected, clipped polygons in device coordinates
We can then begin processing -- once a polygon table is built to include all polygons being drawn...

- ...starting at the first scan line
- ...where we examine using Z-buffering for that particular scan line all polygons whose ymax is greater or equal than this scan line and who ymin values are less than or equal
Which means we don't have to process each polygon -- or search through each vertex (just the max/min values)...keeping the table of polygons in order (like we did when we were dealing with edges for filling) allows us to immediately know which polygons to include and which polygons to exclude
For each scan line...
- ...the algorithm will initialize the scan line buffer to the maximum depth
- ...and update the pointers into the polygon table to include only those polygons which are within range
As we begin processing, the first step is to find out which polygons intersect the current scan line... ...

- which includes checking the two y-values (min and max) against the scan line
- ...if one is larger and the other smaller than the scan line, then we know that the edge intersects the scan line
The next step...

- ...is to compute the x-value of the intersection (in DCs) which includes rounding
- Which is followed by saving the x-value in another table (commonly called the x table) ... which is sorted in ascending order with all other x values of edges that intersect the scan line for this polygon being processed
- ...which means there must be an even number of entries in the x table or you've done something seriously wrong!
Then, along this scan line...

- ...from the first x-value to the second x-value
  ...we compute the Z-value for all pixels
  ...and compare the depth already stored in that position of the scan line buffer and if it is smaller then we enter it into the buffer at that position
- ...also...if it is smaller...we enter in the color into the frame buffer at that position
n • We then take a look at the next intersecting polygon for that same scan line... ...and process it in the same way

n • When all polygons that intersect any given scan line are completely processed... ...we are done with processing that scan line ...and we can move down (by 1) to the next scan line
Some interesting points to notice:

- a) The order in which you see a polygon drawn into the frame buffer may not correspond to the same order in which you asked it to be drawn.

- b) The order in which one polygon is drawn on scan line A may be different than the order in which that same polygon is drawn on scan line B.

- c) If the polygon with the smallest Z value isn't drawn first, you will end up seeing pieces of polygons drawn which are behind the polygon you intended to see...just for a brief moment!