2IV60 Computer graphics set 11: Hidden Surfaces

Jack van Wijk
TU/e

Visible-Surface Detection 1

Problem:
Given a scene and a projection,
what can we see?

Visible-Surface Detection 2

Terminology:

Visible-surface detection vs. *hidden-surface removal*

Hidden-line removal vs. *hidden-surface removal*

Many algorithms:

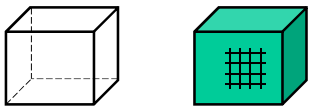
- Complexity scene
- Type of objects
- Hardware

Visible-Surface Detection 3

Two main types of algorithms:

Object space: Determine which part of the object are visible

Image space: Determine per pixel which point of an object is visible



Object space *Image space*

H&B 16-1:504

Visible-Surface Detection 4

Visible-surface detection = *sort* for depth

- what and in what order varies

Performance: use *coherence*

- Objects
- Position in world space
- Position in image space
- Time

H&B 16-1:504

Visible-Surface Detection 5

Four algorithms:

- Back-face elimination
- Depth-buffer
- Depth-sorting
- Ray-casting

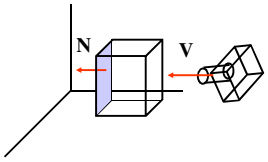
But there are many other.

H&B 16

Back-face elimination 1

We cannot see the back-face of solid objects:

Hence, these can be ignored



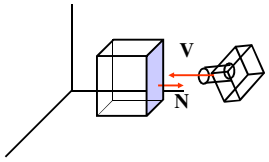
$\mathbf{V} \cdot \mathbf{N} > 0$: back face

H&B 16-2:504-505

Back-face elimination 2

We cannot see the back-face of solid objects:

Hence, these can be ignored

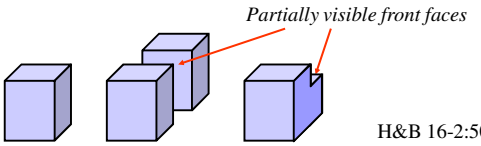


$\mathbf{V} \cdot \mathbf{N} < 0$: front face

H&B 16-2:504-505

Back-face elimination 3

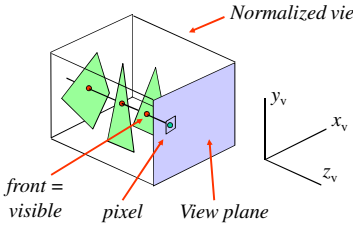
- Object-space method
- Works fine for convex polyhedra: ±50% removed
- Concave or overlapping polyhedra: require additional processing
- Interior of objects can not be viewed



H&B 16-2:504-505

Depth-Buffer Algorithm 1

- Image-space method
- Aka *z-buffer algorithm*



Algorithm:
Draw polygons,
Remember the color most in front.

H&B 16-3:505-508

Depth-Buffer Algorithm 2

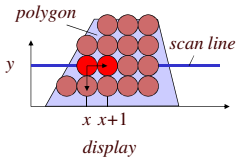
```
var zbuf: array[N,N] of real;          { z-buffer: 0=near, 1=far }
    fbuf: array[N,N] of color;         { frame-buffer }

For all 1<= i, j <=N do
        zbuf[i,j] := 1.0; col[i,j] := BackgroundColour;
For all polygons do                    { scan conversion }
    For all covered pixels (i,j) do
        Calculate depth z;
        If z < zbuf[i,j] then { closer! }
            zbuf[i,j] := z;
            fbuf[i,j] := surfacecolor(i,j);
```

Sorting

H&B 16-3:505-508

Depth-Buffer Algorithm 3

Fast calculation z: use coherence.



Plane : $Ax + By + Cz + D = 0$

Hence : $z(x, y) = \dfrac{-Ax - By - D}{C}$

Also : $z(x+1, y) = \dfrac{-A(x+1) - By - D}{C}$

Thus : $z(x+1, y) = z(x, y) - \dfrac{A}{C}$

Also : $z(x, y) = z(x, y-1) + \dfrac{B}{C}$

H&B 16-3:505-508

Depth-Buffer Algorithm 4

+ Easy to implement
+ Hardware supported
+ Polygons can be processed in arbitrary order
+ Fast: ~ #polygons, #covered pixels

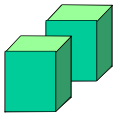
- Costs memory
- Color calculation sometimes done multiple times
- Transparancy is tricky

H&B 16-3:505-508

Depth-Sorting Algorithm 1

- Image and Object space
- Aka *Painter's algorithm*

1. *Sort surfaces for depth*
2. *Draw them back to front*

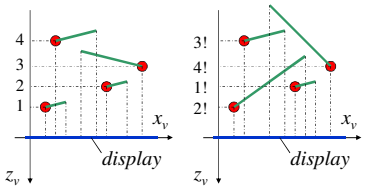


H&B 16-6:511-514

Depth-Sorting Algorithm 2

Simplistic version sorting:

- Sort polygons for (average/frontal) *z*-value



H&B 16-6:511-514

Depth-Sorting Algorithm 3

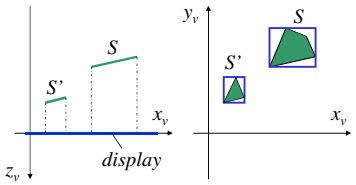
A polygon *S* can be drawn if all remaining polygons *S'* satisfy one of the following tests:

1. No overlap of *bounding rectangles* of *S* and *S'*
2. *S* is completely behind plane of *S'*
3. *S'* is completely in front of plane of *S*
4. Projections *S* and *S'* do not overlap

H&B 16-6:511-514

Depth-Sorting Algorithm 4

1. No overlap of *bounding rectangles* of *S* and *S'*

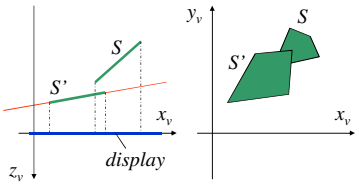


H&B 16-6:511-514

Depth-Sorting Algorithm 5

2. *S* is completely behind plane of *S'*

Substitute all vertices of *S* in plane equation *S'*, and test if the result is always negative.

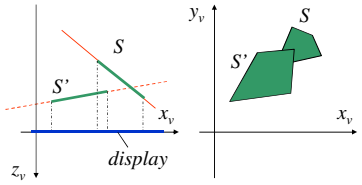


H&B 16-6:511-514

Depth-Sorting Algorithm 6

3. *S'* is completely in front of plane of *S*

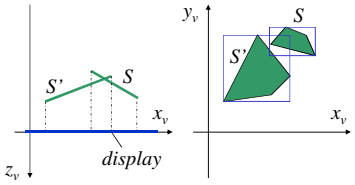
   Substitute all vertices of *S'* in plane equation of *S*, and test if the result is always positive



H&B 16-6:511-514

Depth-Sorting Algorithm 7

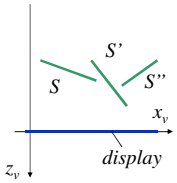
4. Projections *S* and *S'* do not overlap



H&B 16-6:511-514

Depth-Sorting Algorithm 8

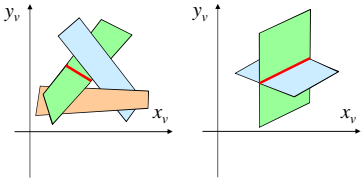
If all tests fail: Swap *S* and *S'*,
and restart with *S'*.



H&B 16-6:511-514

Depth-Sorting Algorithm 9

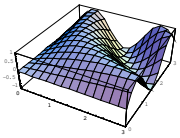
Problems: circularity and intersections
Solution: Cut up polygons.



H&B 16-6:511-514

Depth-Sorting Algorithm 10

- Tricky to implement
- Polygons have to be known from the start
- Slow: ~ #polygons^2

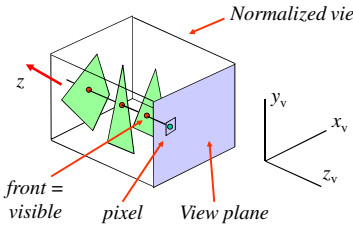


+ Fine for certain types of objects, such as plots of $z=f(x, y)$ or non-intersecting spheres
+ Produces exact boundaries polygons

H&B 16-6:511-514

Ray-casting Algorithm 1

- Image-space method
- Related to depth-buffer, order is different



Algorithm:
Per pixel:
- Calculate intersection points
- Determine front one

H&B 16-10:518-519

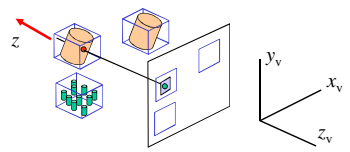
Ray-casting Algorithm 2

Var *fbuf*: **array**[*N,N*] **of** *colour*; { *frame-buffer* }
n : *integer*; { *#intersections* }
z : **array**[*MaxIntsec*] **of** *real;* { *intersections* }
p : **array**[*MaxIntsec*] **of** *object;* { *corresp. objects* }
For *all 1<= i, j <=N* **do** { *for alle pixels* }
For *all objects* **do**
Calculate intersections and add these to z and p, keeping z and p sorted;
if *n > 1* **then** *fbuf*[*i,j*] := *surfacecolor*(*p*[1], *z*[1]);

H&B 16-10:518-519

Ray-casting Algorithm 3

Acceleration intersection calculations:
Use (hierarchical) bounding boxes



H&B 16-10:518-519

Ray-casting algorithm 4

+ Relatively easy to implement
+ For some objects very suitable (for instance spheres and other quadratic surfaces)
+ Transparency can be dealt with easily

- Objects must be known in advance
- Slooooow: ~ #objects*pixels, little coherence
+ Special case of *ray-tracing*

H&B 16-10:518-519

Comparison

- Hardware available? Use depth-buffer, possibly in combination with back-face elimination or depth-sort for part of scene.
- If not, choose dependent on complexity scene and type of objects:
  - Simple scene, few objects: depth-sort
  - Quadratic surfaces: ray-casting
  - Otherwise: depth-buffer
- Many additional methods to boost performance (kD-trees, scene decomposition, etc.)

H&B 16-11:519

OpenGL backface removal

```
glEnable(GL_CULL_FACE);        // Turn culling on
glCullFace(mode);              // Specify what to cull
mode = GL_FRONT or GL_BACK     // GL_BACK is default
```

Orientation matters! If you want to change the default:

```
glFrontFace(vertexOrder); // Order of vertices
vertexOrder = GL_CW or    // Clockwise
              GL_CCW;     // Counterclockwise (default)
```

H&B 16-11:523-525

OpenGL depth-buffer functions

```
glEnable(GL_DEPTH_TEST);       // Turn testing on
glClear(GL_DEPTH_BUFFER_BIT);  // Clear depth-buffer,
                               // typically once per frame
glDepthFunc(condition);        // Change test used
condition: GL_LESS             // Closer: visible (default)
           GL_GREATER          // Farther: visible
```

Note: range between z_{near} and z_{far} is mapped to [0,1], using one or two bytes precision. If z_{far} has an unnecessarily high value, you loose precision, and artifacts can appear.

H&B 16-11:523-525

Next…

- We know how to determine what is visible, and we looked at illumination and shading.
- Next: Let's consider more advanced shading.