Hidden Surface Elimination

A scene composed of 3D objects may have some of them obscuring all or parts of the others.

Draw only objects closest to the viewing position and eliminate those obscured.

A.k.a. hidden surface removal or visible surface detection or occlusion culling.

Changing viewpoint can change the obscuring relationship.

Benefit?

Hidden Surface Elimination

Algorithms are usually classified by whether they work on:

- object space:
  - painter's algorithm
  - BSP tree

- image space:
  - z-buffer
  - ray casting

There are others, but z-buffer, BSP tree, and ray casting are most commonly used in practice.

Painter’s Algorithm


It solves the visible object problem by painting, or filling, with opaque paint, where closer objects are painted over farther ones.

```plaintext
sort objects by z
for all objects {
  for all covered pixels(x,y)
  paint
}
```

Problem:

does not handle cyclic ordering.

Binary Space Partitioning

Very efficient visibility culling method for a static set of polygons

Trade off time-and-space intensive preprocessing against linear display step

Well suited for applications where viewpoint changes, but objects do not, e.g., 3D games such as Doom

Main idea: a polygon $A$ is painted in correct order if:

• polygons on the far side of $A$ are painted first
• $A$ is painted next
• polygons on near side of $A$ are painted last

Build a tree to recursively partition the space and group polygons

Two types:

• Polygon-aligned BSP
• Axis-aligned BSP

Implicit 3D Plane Eqn

Consider a plane through points $a$, $b$, and $c$

The normal of the plane is: $n = (b - a) \times (c - a)$

A point $p$ is on the plane if $n \cdot (p - a) = 0$

Implicit plane equation, for $a$ on plane, is thus: $f(p) = n \cdot (p - a) = 0$

Think of $(p - a) \cdot n$ as projecting $(p - a)$ onto $n$:

• $f(p) > 0$ if $p$ is on the same side as $n$
• $f(p) < 0$ if $p$ is on the other side of $n$

Let $p = (x, y, z)$, $n = (A, B, C)$, the implicit plane equation can also be expressed as:

$$f(x, y, z) = Ax + By + Cz + D = 0$$

where $D = -A x_a - B y_a - C z_a = -n \cdot a$

Polygon-aligned BSP

Use plane that is coplanar with each triangle as scene separator

First assume no triangle crosses such planes

BSP idea is simply:

• let $e$ be the eye-point and $f_T(e) = 0$ be the implicit plane equation for the plane coplanar with $T_1$
• if ($f_T(e) > 0$) {
  // eye is closer to $T_1$ than to $T_2$
  draw $T_2$
  draw $T_1$
} else {
  draw $T_1$
  draw $T_2$
}

Polygon-aligned BSP Tree

Using the BSP:

recurse on the far side (from eye) child
draw parent
recurse on the near side (to eye) child

render(node, eye) {
  if(node == 0) return;
  if(node.f(eye) > 0) { // in front
    render(node.child_back, eye);
    draw(node.T);
    render(node.child_front, eye);
  } else {
    render(node.child_front, eye);
    draw(node.T);
    render(node.child_back, eye);
  }
}
BSP Tree Construction

Add all the triangles in any order

Recursive algorithm:

```c
struct node_t {
    triangle_t T;
    plane_eqn f(point p);
    node_t *child_back, *child_front;
    void add(triangle_t *new_T); // add a child recursively
}
```

```c
node_t::add(triangle_t *new_T) {
    if (self.f(new_T[0])>0 && self.f(new_T[1])>0 && self.f(new_T[2])>0)
        if (self.child_front == 0) // no children in front
            self.child_front = node_t(new_T);
        else
            self.child_front->add(new_T);
    else if (all negative)
        ...
    else
        split_and_add_triangle(new_T);
}
```

**Polygon-aligned BSP**

Use plane coplanar with each triangle as scene separator

First assume no triangle crosses such planes

What if a triangle does cross the plane defined by another triangle?

**BSP Tree Construction**

Split triangle $T_2$:

- **one** of $T_2$'s vertices will be on the opposite side of $T_1$'s plane to the others
- find the intersections where the plane cuts the triangle (how?)

\[ p(t) = k + t(m - k) \]

if $p(t)$ is on $T_1$'s plane,

for a any vertex of $T_1$,

\[ f(p(t)) = n \cdot p(t) - n \cdot a = 0 \]

\[ n \cdot (k + t(m - k)) - n \cdot a = 0 \]

\[ t = \frac{n \cdot a - n \cdot k}{n \cdot (m - k)} \]

Similarly for $q = 1 + t_q(m - 1)$

- then tessellate $T_2$ into 3 triangles

**BSP Tree Construction**

Caveats when splitting triangle $T$:

- use implicit equation of plane to check which one of $T$'s vertices is on the opposite side to the others
- must maintain vertex ordering to preserve normal!
- be careful not to create a sliver of a triangle: if distance of $m$ from plane is $< \epsilon$, treat $m$ as if it is in the plane
- if one or more vertices are in the plane, no need to cut triangle
BSP Tree Performance

Add triangles to the BSP tree in any order

- tree shape doesn’t effect performance for hidden surface elimination (why?)
- but it is useful to keep BSP balanced for other uses, e.g., collision detection
- different ordering of triangle additions to the tree result in more (or less) tree nodes
  - one heuristics: in each round, pick 5 triangles at random, choose the plane with minimal triangle crossings as the separator

Axis-Aligned BSP Tree: Idea

Splitting plane aligned to x, y, or z axis

Minimal box

Split along plane

Split along plane

Axis-Aligned BSP Tree: Build

Each internal node holds a divider plane
Leaves hold geometry

A.k.a., kd-tree

Axis-aligned BSP-Tree: Usage

Test the planes against the point of view
Test recursively from root
To sort front to back, visit near side first
  - being able to do so and not display invisible back polygons was a major innovation in Quake

• does not give exact sorting when there are multiple objects per node, or when an object spans multiple nodes
Hidden Surface Elimination

Algorithms are usually classified by whether they work on:

- object space:
  - painter’s algorithm
  - BSP tree

- image space:
  - z-buffer
  - ray casting

There are others, but z-buffer, BSP tree, and ray casting are most commonly used in practice

z-Buffer or Depth Buffer

One of the simplest of all image-based hidden-surface elimination algorithms (Ed Catmull, 1974)

- originally considered too expensive
- now commonly implemented in hardware using fast memory (cheap now) and fast GPU
- geometry independent: hidden surface removal one fragment at a time!

Algorithm:

- at each pixel, store the z-value of the closest triangle rasterized so far
- change the pixel’s color (and depth value) only if a new z-value is closer to the viewer than the stored value

```
setpixel(int i, j, RGB c, real z) {
  if (z < z_buffer(i,j)) {
    screen(i,j) = c;
    z_buffer(i,j) = z;
  }
}
```
**z-Values**

Computation:
- use barycentric coordinates to interpolate depth value of each pixel from those of the vertices

Depth-value storage:
- as non-negative integers
- integers are represented in \( b (=16 \text{ or } 32) \) bits,
  giving a limited range of \( B (= 2^b) \) values \( \{0, 1, 2, \ldots, B-1\} \)

**z-Buffer visualization:**

---

**z-Buffer in OpenGL**

```c
glutInitDisplayMode(GLUT_DEPTH | . . . );
glClear(GL_DEPTH_BUFFER_BIT | . . . );
glEnable(GL_DEPTH_TEST);
// get viewing position and draw objects

Recall that OpenGL is a state machine
Boolean state settings can be turned on and off with glEnable() and glDisable()
Anything that can be set can be queried using glGet()
```