CMSC 427: Chapter 8
Depth Cues and Hidden Surface Removal

Reading: Chapt. 8 in Shirley.
Overview:
- Depth Cues: Transparency, blending, fog.
- Culling: View frustum culling, back-face culling, visibility culling.
- Hidden surface removal algorithms
  - Painter’s Algorithm and BSP Acceleration
  - Z-Buffer Algorithm

Elements of Realistic Rendering

Realism in Rendering:
- Perspective projection
- Illumination and shading
- Texture mapping and surface detail
- Shadows, Reflection, and other effects
- Hidden surface removal
- Color
Overview

• Depth Cues

• Culling Invisible Objects

• Hidden Surface Removal
  • Painter's Algorithm
  • Subdivision Methods and BSP Trees
  • Z-Buffer Algorithm

Hidden Surface Removal

We have seen how to compute the projection of objects onto the image plane and color them. Next, we consider the question of how to determine which objects are visible and which are not.

Hidden Surface Removal: Involves determining which objects (or portions thereof) are visible and which are obscured.

Assumptions:
- Objects presented as solid polygons (or have been triangulated).
- Objects are opaque. (Otherwise blending is needed.)

Output precision:
  Object precision: Output is a set of visible object polygons. Can be scaled arbitrarily.
  Image precision: The algorithm computes its results to the precision of a pixel of the image. Cannot be rescaled without losing accuracy.
Depth Cues

Hidden Surface Removal is but one of many means to provide depth cues, that is, a sense of distance from the viewer.

- **Perspective Viewing**: distant objects appear smaller.
- **Kinetic-Depth Effect**: distant objects appear to move slower.
- **Stereopsis**: Difference in appearance of the scene between the left and right eyes. (Can be achieved using 3-D glasses or special head-mounted displays.)
- **Occlusion**: More distant objects are obscured by closer ones (hidden surface removal).
- **Illumination Attenuation**: Distant objects appear fainter (fog effect).

Opacity and Blending

- **Alpha Channel**: Recall that color in OpenGL is based on RGBA. The A component is called alpha. It allows different levels of opacity amongst objects:
  - \( \alpha = 1 \Rightarrow \text{Perfectly opaque. (Default)} \)
  - \( \alpha = 0 \Rightarrow \text{Perfectly transparent.} \)
  - \( 0 < \alpha < 1 \Rightarrow \text{Different levels of translucency.} \)

- **Blending**: The mixing colors of two sets of pixels, source and destination.
  - **Destination pixel**: The pixel already stored in the frame buffer.
  - **Source pixel**: The incoming pixel that’s about to be blended with the destination pixel.
  - OpenGL provides many possible functions to be used to combine the source and destination pixels into a new pixel in the frame buffer.
Blending in OpenGL

```c
setEnabled ( GL_BLEND )
gBlendFunc ( (source_factor), (destination_factor) )
  - where factors are:
    GL_ONE, GL_ZERO, GL_SRC_ALPHA,
    GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA,
    GL_ONE_MINUS_DST_ALPHA

See the OpenGL documentation for formal definitions of these
blending functions.
```

Chapter 9, Slide 7
Copyright © D. M. Mount and A. Varshney

Blending in OpenGL

Typical Transparency Blend Function:
```c
gBlendFunc ( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA )
```

Note: Blending only occurs if the source pixel is deemed to be visible. For example, if depth-buffering is enabled, and the source pixel is farther from the viewer than the destination pixel, the source pixel will not be blended.

Trick: Turn off depth buffering when blending overlapping objects.
```c
gDisable( GL_DEPTH_TEST ) ... glEnable( GL_DEPTH_TEST )
```
Depth Cueing and Fog

**Depth Cueing**: Draw objects farther from the viewer darker.

**Fog**: Draw objects farther from the viewer whiter.

Let the color to be added with depth be $C_f$, the color of the pixel be $C_p$ and the factor for blending be $f$, then:

$$C_p = f C_p + (1 - f) C_f$$

**Example**: We want fog of color $fcolor$ to vary exponentially with depth, e.g. $f = e^{-0.5z}$.

```gl
    glEnable ( GL_FOG );
    glFogf ( GL_FOG_MODE, GL_EXP ); // glFogf sets fog parameters
    glFogf ( GL_FOG_DENSITY, 0.5 );
    glFogfv ( GL_FOG_COLOR, fcolor ); // glFogf sets fog parameters
```

Overview

- Depth Cues
  - Culling Invisible Objects
- Hidden Surface Removal
  - Painter's Algorithm
  - Subdivision Methods and BSP Trees
  - Z-Buffer Algorithm
Culling

Most of the objects in a typical complex scene are not visible. (Imagine a walkthrough simulation of a large building.) Culling is the process of rapidly eliminating objects that cannot be seen.

Common Types of Culling:

- **View Frustum Culling:** Removal objects outside the view volume.
- **Back-face Culling:** Removal of faces on the back side of objects.
- **Visibility-based Culling:** Removal of objects that cannot be seen based on knowledge of the environment.

Why do Culling? Hidden surface elimination can be expensive. No need to apply to objects that are obviously not visible.

View-Frustum Culling

View Frustum Culling:
- Remove objects that are entirely outside the view frustum (or more generally the view volume).
- Culling proceeds by checking the object against the six planes of the view volume.
View-Frustum Culling

Buddha is entirely within the frustum ⇒ not culled.
Castle is entirely outside the frustum ⇒ completely culled.
Bunny is partly inside frustum ⇒ partly culled.

Efficient Testing:
- Build a bounding volume for each object. Examples:
  - Bounding box (axis-parallel rectangle): Very easy to compute (min-max coordinate values).
  - Bounding sphere: A bit harder to compute, but a bit easier to test (just need distance to center point).
  - Convex hull: Best approximation, but hardest to compute and hardest to test.
- Test whether bounding volume intersects the view volume.

Handling Complex Objects and Partial Overlaps:
- Build a bounding-volume hierarchy for each object.
- Test starts at root level.
- if outside ⇒ cull,
  if inside ⇒ keep,
  otherwise ⇒ recurse on next lower level of the hierarchy.
Back-face Culling

Assume that objects are solid polyhedra, bounded by polygonal faces.
Assume that objects have been transformed into the view-frame coordinate system (view direction is the z-axis).
For a face f, if its normal vector n forms an obtuse angle with the z-axis, that is, \( n_z < 0 \), then this face lies on the back side of the object (relative to the viewer) and so may be culled.

In-Class Exercise

- You are given a tetrahedron whose vertices are at \((0, 0, 0)\), \((1, 0, 0)\), \((0, 1, 0)\), and \((0, 0, 1)\).
- It has four faces:
  - X: On the \( x = 0 \) plane.
  - Y: On the \( y = 0 \) plane.
  - Z: On the \( z = 0 \) plane.
  - F: Passes through \((1,0,0)\), \((0,1,0)\), \((0,0,1)\).
- The eye coordinates are \((e_x, e_y, e_z)\).
- Give four tests (one per face), which (based on the eye coordinates) determines whether this face may be culled.

**Hint:** The equation for the plane containing F is \( x+y+z = 1 \). (How do you compute the plane passing through three arbitrary points?)
Back-face Culling in OpenGL

Enabling culling:

```c
glCullFace (GL_BACK);        // which faces to cull
```

Also possible: `GL_FRONT, GL_FRONT_AND_BACK`

```c
glEnable (GL_CULL_FACE);     // turn culling on
```

Disabling culling:

```c
glDisable (GL_CULL_FACE);    // turn culling off
```

Dynamic: Some faces can be drawn with culling enabled and others with culling disabled.

Front side: The side from which the vertices appear in CCW order.

(This can be changed using `glFrontFace()`.)

---

Visibility-Based Culling

Visibility-Based Culling: Remove objects that cannot be seen in the current frame.

Based on scene knowledge: (Not done by OpenGL or GPU.)

Static: Determine the visibility of objects and viewing volume independent of viewpoint.

Example: Cannot see into a third-floor office from the second floor.

Dynamic: Use near (and large) objects to cull away far objects incrementally based on the viewer's location.

Example: In an office, some other offices may be invisible through the door, depending on the viewer's location.
Cells and Portals

Chapter 9, Slide 21
Copyright © D. M. Mount and A. Varshney

Chapter 9, Slide 22
Copyright © D. M. Mount and A. Varshney
**Cells and Portals**

- Establish *potentially visible sets (PVS)* from each cell through its portals.
- Use *portal graph* (based on doors and windows) to determine neighboring cells.
- Display geometric models only for PVS.

*Very fast in practice:* In practice only a small fraction of the domain is visible.

*Conservative Culling:* No visible objects are culled.

---

**Overview**

- Depth Cues
- *Culling Invisible Objects*
  - *Hidden Surface Removal*
    - Painter's Algorithm
    - Subdivision Methods and BSP Trees
    - Z-Buffer Algorithm
Algorithms for Hidden Surface Removal

General Approaches:
- Object-Order Algorithms: Sort the objects and then display them.
- Image-Order Algorithms: Scan-convert objects in arbitrary order and then depth sort the pixels.
- Hybrid of the above: Combinations of the above.

Popular Approaches:
- Painter's Algorithm: (a.k.a. Depth-Sort) Draw objects from back to front. (Object order).
- Spatial Decompositions: Data structures which store objects in a spatial decomposition as an aid to sorting objects.
- Z-buffer Algorithm: (a.k.a. Depth-Buffer) Each pixel stores distance to closest object.

Overview

- Depth Cues
- Culling Invisible Objects
- Hidden Surface Removal
  - Painter's Algorithm
  - Subdivision Methods and BSP Trees
  - Z-Buffer Algorithm
**Painter's Algorithm**

*Painter's Algorithms: An object-order algorithm:*
- Sort objects by depth.
- Display them in back-to-front order.
Later (closer) objects overwrite earlier (farther) ones.

**Cyclic and Intersecting Objects**

*Problem:* If objects intersect cyclically, it may not be possible to sort them.

*Solution:* Split one or more objects so that cyclic depth order exists.
Painter's Algorithm Basics

Terminology:
- **z-extents**: of an object is the interval it projects on the z-axis.
- Given a polygon P, let $H_P$ denote the plane containing P.

Even if the z-extents overlap, we may be able to safely draw P:

Painter's Algorithm

1. Sort the polygons by depth based on the farthest vertex.
2. While the sorted list is non-empty do:
   - (a) Let P be the polygon that is at the end of the list (farthest).
   - (b) Consider polygons Q whose z-extents overlaps P's. Q is "safe" if:
     1. The x-extents of P and Q disjoint, or
     2. The y-extents of P and Q disjoint, or
     3. P lies entirely on the far side of $H_Q$, or
     4. Q lies entirely on the near side of $H_P$, or
     5. The image-plane projections of P and Q are disjoint.
   - (c) If any overlapping Q is not safe: use $H_Q$ to split P into $P_1$ and $P_2$.
   - (d) else (all overlapping Q are safe): draw P.
Painter's Algorithm

Observations:

Running time: In theory this partitioning could generate as many $O(n^2)$ individual polygons (every polygon splits every other), but in practice the final number of polygons is $O(n)$.

Space: The depth-sort algorithm needs no storage other than the frame buffer and a linked list of the polygons (and their fragments).

Multiple Drawing: It suffers from the deficiency that each pixel is written as many times as there are overlapping polygons.

Image Precision: It outputs pixels.

Transparency: Can be handled easily by blending as we draw.

Overview

- Depth Cues
- Culling Invisible Objects
- Hidden Surface Removal
  - Painter's Algorithm
  - Subdivision Methods and BSP Trees
  - Z-Buffer Algorithm
Methods Based on Spatial Decomposition

**Painter’s Algorithm:** The polygons can be painted correctly if for each polygon $F$:
- Polygons on the far side of $F$ are painted before $F$.
- Polygons on the close side of $F$ are painted after $F$.

**Idea:** Preprocess the spatial relationships of polygons in the scene in a tree structure for rapid depth determination.

**Spatial Decomposition Trees:** Can be used to represent these spatial relationships efficiently.

---

Methods Based on Spatial Decomposition

**Octrees:**
- Based on a recursive decomposition of space into cubes.
- Each cube is subdivided into 8 identical cubes of half the size.
- Each node of the tree has 8 children.
- Generalizes quadtrees to 3-space.

**kd-trees:**
- Each node splits space using an axis-orthogonal plane.
- Splitting directions alternate cyclically.
- Cells are 3-D rectangles.
- Extra splitting flexibility helps improve performance.

**Binary-Space Partition Trees (BSP trees):**
- Each node splits space using an arbitrarily oriented plane.
- Cells are convex polygons.
Building a BSP Tree

```java
class BSPNode {
    Polygon splitter; // splitting polygon
    BSPNode frontChild, backChild; // children in the tree
}

BSPNode makeBSP (PolygonList P) { // recursive tree constructor
    if (list == null) return new BSPLeaf; // no more objects remain
    Remove a polygon s from P; // splitter polygon
    Split all polygons in P that intersect the plane passing through s;
    PF  polygons of P on front side of s; // partition P into front and back
    PB  polygons of P on back side of s;
    BSPNode node  new BSPNode; // initialize the node
    node.splitter  s;
    node.frontChild  makeBSP (PF); // build the subtrees
    node.backChild  makeBSP (PB);
    return node;
}
```

Building a BSP Tree (2-D Example)
Displaying a BSP Tree

```c
void displayBSP ( BSPNode p ) {
  if ( p = null ) return; // nothing here to draw
  if ( viewer is in front of p.splitter ) {
    displayBSP ( p.backChild ); // display back to front
    drawPolygon ( p.splitter );
    displayBSP ( p.frontChild );
  }
  else /* viewer behind splitter */ {
    displayBSP ( p.frontChild ); // display front to back
    drawPolygon ( p.splitter );
    displayBSP ( p.backChild );
  }
}
```

Painter's Algorithm using a BSP Tree

Far-to-near order: [4, 5b, 3, 5a, 2, 1].
Far-to-near order (with leaves): [e, 4, f, 5b, g, 5a, b, a, 2, c, 1, d]
BSP Trees: Discussion

Advantages:
- Efficient.
- View-independent.
- Easy transparency and antialiasing.

Disadvantages:
- Tree is hard to balance.
- Not efficient for small polygons.

Overview

- Depth Cues
- Culling Invisible Objects
- Hidden Surface Removal
  - Painter's Algorithm
  - Subdivision Methods and BSP Trees
    - Z-Buffer Algorithm
**Z-Buffer Algorithm**

**Z-Buffer (Depth-Buffer) Algorithm**: Image precision, image order.
- Scan-convert each object.
- Maintain the depth (in Z-buffer) and color (in color buffer) of the closest object at each pixel.
- Display the final color buffer.

*Simple and implementable in hardware (GPUs)* → This is the key reason for this algorithm's importance.

---

```plaintext
for each pixel (i, j) { // clear Z-buffer and frame buffer
    zBuffer[i, j] ← farPlaneZ
    colorBuffer[i, j] ← backgroundColor
}

for each polygon P { // scan convert each polygon
    for each pixel(i, j) in the projection of P {
        compute depth z and color c of P at (i, j)
        if (z < zBuffer[i, j]) { // if closer, store new depth and color
            zBuffer[i, j] ← z
            colorBuffer[i, j] ← c
        }
    }
}
```

---

Chapter 9, Slide 41
Copyright © D. M. Mount and A. Varshney
Efficient Z-Buffer Implementation

Incremental computation: When scan converting each polygon, we need to compute the depth of each pixel rapidly.

Plane equation: For each polygon scanned, compute the implicit representation of its plane (relative to projected coordinates):

\[ ax + by + cz + d = 0 \]

\[ z(x, y) = (-ax - by - d)/c \]

Horizontal Change: Given a pixel's depth, the change in depth for the next pixel in the same row is:

\[ \Delta z_i = z(x+1, y) - z(x, y) \]

\[ = -a(x+1) - by - d - (-ax - by - d) \]

\[ = \frac{-a}{c} \]

Incremental Computation: Next we consider how depth changes from one scan line to the next. Compute the slope \( m \) of the left edge.

Let \( m \) be the slope of this edge.

Then \( x' = x - (1/m) \), and thus the change in depth is

\[ \Delta z_2 = z(x', y-1) - z(x, y) = z(x - (1/m), y - 1) - z(x, y) \]

\[ = \frac{-a(x - (1/m)) - b(y - 1) - d - (-ax - by - d)}{c} \]

\[ \Delta z_2 = \frac{(a/m) + b}{c} \]
**Efficient Z-Buffer Implementation**

**Incremental Computation:** We explicitly compute the depth and location of the top vertex of the left edge using the implicit representation.

- Until reaching the right edge, we incrementally adjust the depth of each successive pixel in each row: $z \leftarrow z + \Delta z_1$.
- On reaching the right edge, we descend to the next scan line:
  
  $$
  y = y - 1 \\
  x = x' = x - \frac{1}{m} \\
  z = z + \Delta z_2.
  $$

---

**Z-Buffer Algorithm**

**Advantages:**
- Simple: Implementable in hardware.
- General: Objects need not be polygons. (Fancier depth update rules.)

**Disadvantages:**
- Space intensive: Separate buffer for depth information needed.
- Problems handling transparency: A partially transparent foreground object will completely obscure all background objects.
- Difficulty with aliasing: (Jagged edges) Only one visible object per pixel.
**A-Buffer Algorithm**

A-buffer (or accumulation) algorithm: An extension to the Z-buffer algorithm. Two types of pixels are stored, depending on the depth entry:

- **Depth ≥ 0**: This entry behaves exactly in the same way as in the Z-buffer algorithm.
- **Depth < 0**: The color field is a pointer to a linked list of objects whose projection overlaps this pixel. Each list entry contains:
  - RGB color,
  - A-channel (opacity value),
  - Depth,
  - Fraction of the pixel area covered,
  - ... (other surface properties that may be relevant)

This addresses some of the shortcomings of the Z-buffer algorithm, but space is high if many transparent objects.

---

**Summary**

Summary:

- **Depth Cues**
  - Transparency, Blending, Fog.
- **Culling**
  - View frustum culling, back-face culling, visibility-based culling.
- **Hidden surface removal algorithms**
  - Painter’s Algorithm and BSP Acceleration
  - Z-Buffer and A-Buffer