Basics in Computer Graphics II
egon elbre
little bit of 3D stuff

3D glasses not required

(if you still decide to put them on, you will just look stupid)
Polygon mesh

Geometry defined by connected triangles.

(most 3D games/applications)
mesh terminology

vertices  edge  faces  polygons  surfaces
Constructive Solid Geometry (CSG)

Geometry defined by primitives and their compositions with boolean operations and transformations.

Primitives are usually spheres, cubes, cylinders, capsules

(CAD programs)
Voxels

Voxel = Volumetric Pixel

Geometry defined similarly to a bitmap.

(medical & scientific data)
making polygons smooth
aka finding subdivision surface

If you've seen 3D cartoons or any film using CG... it has been used there...

refine rough mesh by adding/replacing polygons recursively until it's denser and smoother

approximating

interpolating
Catmull-Clark

Edwin Catmull, Jim Clark 1978

1. face point: avg. of original points
2. edge point: avg. of two neighboring face points and two original points
3. original point: \( \frac{F}{n} + \frac{2R}{n} + \frac{P(n-3)}{n} \)
   - F: avg. of n face points, R: avg. of mid points
   - P: original point,
   - n: number of edge or face
we have some mesh
find *face points* = average of all the points of the face
find edge points = average between the center of the edge and center of segment made by face points
update vertices

with:

\[ m_1 = \frac{n - 3}{n} \]
\[ m_2 = \frac{1}{n} \]
\[ m_3 = \frac{2}{n} \]

\[ \text{new} = (m_1 \times \text{old_coords}) + (m_2 \times \text{avg_face_points}) + (m_3 \times \text{avg_mid_edges}) \]
update vertices

with:
\[ m1 = \frac{n - 3}{n} \]
\[ m2 = \frac{1}{n} \]
\[ m3 = \frac{2}{n} \]
\[ \text{new} = (m1 \times \text{old\_coords}) + (m2 \times \text{avg\_face\_points}) + (m3 \times \text{avg\_mid\_edges}) \]
and... create new polygons
Doo-Sabin

Daniel Doo, Malcolm Sabin 1978

1. face point: average of the vertices of the polygon
2. edge point: midpoint of the edge
3. new point: average of four points
Visible Surface Determination

If surface A is in front of surface B, then surface B shouldn't be seen.

Rendering invisible surface and later eliminating is a waste of time.
Painters Algorithm

1. sort objects by distance
2. draw from back to front

! slow, lot's of overwritten pixels

Reverse Painters Algorithm

1. sort objects by distance
2. draw from front to back

! problem with alpha

NB! problem with cycles
Scanline Rendering

Wylie, Romney, Evans and Erdahl 1967

rendering multiple polygons
extension to polygon scan-conversion algorithm
instead of one poly, we got set of polygons
1. Edge Table (ET)

linked list for all nonhorizontal edges of all polygons
entries are sorted into buckets based on smaller y coordinate

1. x coordinate with smaller y coordinate
2. y coordinate of the other end
3. x increment, Δx, used to step from one scanline to next
4. polygon identification number

---

**ET**

<table>
<thead>
<tr>
<th>x</th>
<th>y_{max}</th>
<th>Δx</th>
<th>ID</th>
</tr>
</thead>
</table>

**PT**

| ID | Plane eq | Shading | In-Out |
2. Polygon Table (PT)

1. coefficients of the plane equations
2. shading and color information
3. in-out boolean flag
3. Active Edge Table (AET)

edges being processed at a scanline kept in order of increasing x

<table>
<thead>
<tr>
<th>Scan line</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>AB AC</td>
</tr>
<tr>
<td>β</td>
<td>AB AC FD FE</td>
</tr>
<tr>
<td>γ, γ+1</td>
<td>AB DE CB FE</td>
</tr>
<tr>
<td>γ + 2</td>
<td>AB CB DE FE</td>
</tr>
</tbody>
</table>
add edges to Edge Table
initialize Active Edge Table

for each scan line {
    update Active Edge Table
    for each pixel on scan line
        for each edge in AET
            if enters/leaves polygon
                flip PT[PolyID].InOut
            z = PlaneEq(x,y)
            if z < closest_z
                closest_z = z
                closest_PolyID = PolyID
        determine closest pixel shade using PolyID
Scan line | Entries
---|---
\(\alpha\) | AB AC
\(\beta\) | AB AC FD FE
\(\gamma, \gamma+1\) | AB DE CB FE
\(\gamma + 2\) | AB CB DE FE
Z-Buffer

store z value with the pixel color and use that to determine overdrawning visibility

for(each polygon P in the polygon list){
  for(each pixel(x,y) that intersects P){
    calculate z-depth of P at (x,y)
    if (z-depth < z-buffer[x,y]){
      z-buffer[x,y] = z-depth;
      COLOR(x,y)=Intensity of P at(x,y);
    }
  }
}

A simple three-dimensional scene

Z-buffer representation
Warnock's Algorithm

subdivides each area into four equal squares
at each recursive step each polygon has one of four relationships to the area of interest

a. area of interest is contained
b. intersecting the area
c. completely inside the area
d. completely outside the area
case 1

if all the polygons are disjoint from the area
display the background color

case 2

if there's only one polygon intersecting/contained
display the background color
scan convert the contained polygon

case 3

if surrounding polygon, but no intersecting/contained polygons
scan convert the surrounding polygon
case 4

more than one polygon is intersecting/contained/surrounding, but one surrounding polygon is in front of all other polygons

test each polygon plane z values in the corners of the area

Fig. 15.43 Two examples of case 4 in recursive subdivision. (a) Surrounding polygon is closest at all corners of area of interest. (b) Intersecting polygon plane is closest at left side of area of interest. × marks the intersection of surrounding polygon plane; ○ marks the intersection of intersecting polygon plane; * marks the intersection of contained polygon plane.
Back-face culling

removing polygons that are facing the other way

can be easily done by discarding polygons where dot product between camera-to-polygon and polygon normal is

\[ N \cdot C < 0, \text{ keep} \]
\[ M \cdot C > 0, \text{ discard} \]

\[ a \cdot b = |a| |b| \cos \theta \]
Frustrum Culling

remove objects that are outside of camera view

brute-force: test whether each object is in frustrum
Optimization 1

1. test against bounding sphere center
2. test against bounding sphere
3. test against object
Optimization 2

1. test frustrum sphere
2. test cone
3. test bounding sphere center
4. test bounding sphere
5. test object

[Image of geometric shapes and spheres]
Area Subdivision

BSP, Quadtree, Octree, R-Tree
Binary Space Partitioning (BSP)
Fuchs, Kedem, Naylor 1980, 1983

slow pre-processing
simple tree traversal to get
correct order of things

3D games
especially in FPS-s for rendering
ray tracing, collision detection

Doom 1993,
 id Software
Algorithm BSPTree constructBSP(PolygonSet S)
Input: a set of polygons, S
Output: a binary search partitioning tree of S
{
    PolygonSet FrontSet, BackSet;
    if set S == {},
        return EmptyTree;

    P = choosePolyFrom(S)
    FrontSet = {}
    BackSet = {}

    for each polygon Q in S - {P} {
        if Q is in front of the plane containing P
            insert Q in FrontSet;
        else if Q is in back of the plane containing P
            insert Q in BackSet;
        else
            split Q into two polygons Qfront and Qback
            insert Qfront in FrontSet;
            insert Qback in BackSet;
    }
    return makeTree(P, constructBSP(FrontSet), constructBSP(BackSet));
}
for each polygon Q in S - {P} {  
    if Q is in front of the plane containing P  
        insert Q in FrontSet;  
    else if Q is in back of the plane containing P  
        insert Q in BackSet;  
    else  
        split Q into two polygons Qfront and Qback  
        insert Qfront in FrontSet;  
        insert Qback in BackSet;  
}
return makeTree(P, constructBSPTree(FrontSet), constructBSPTree(BackSet));
construction - step 1
construction - step 2
construction - step 3
construction - alternate rooting
Algorithm displayBSP(bspRoot node, viewPoint view)
Input: viewpoint v, the (x,y,z) coordinates of a viewpoint in 3D
Output: rendering of scene
{
    if node == EmptyTree
        return
    if view is in front of node {
        displayBSP( node->back, view)
        renderPoly( node->root )
        displayBSP( node->front, view)
    }
    else {
        displayBSP( node->front, view)
        renderPoly( node->root )
        displayBSP( node->back, view)
    }
}
Fig. 15.33 Two traversals of the BSP tree corresponding to two different projections. Projectors are shown as thin lines. White numbers indicate drawing order.
Quadtree

area recursively divided to 4

spatial indexing
2D collision detection
view frustum culling
Octree

Space divided into eight octants

spatial indexing
3D collision detection
frustrum culling
sparse voxel octree
Boolean Operations on Quad/Octrees

finding union

examine quadrants:
if S.q or T.q is black:
  \( U.q = \text{black} \)
else if S.q is white
  \( U.q = T.q \)
else if T.q is white
  \( U.q = S.q \)
else
  recurse
U.q all nodes same
replace with one node

union          intersection
Finding Neighbors

Finding a node that is adjacent to the original quadtree has 8 possible neighbors N,S,E,W and diagonally NW,NE,SW,SE

Finding a neighbor node in a specified direction

start at original node
ascend until you reach common ancestor
descend to find the desired neighbor

Samet 1989
## Drawing Octrees

drawing voxel data

### VPN

<table>
<thead>
<tr>
<th>$z$</th>
<th>$y$</th>
<th>$x$</th>
<th>Back-to-front order</th>
<th>Visible faces*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>7,6,5,3,4,2,1,0</td>
<td>B,D,L</td>
</tr>
<tr>
<td>$-$</td>
<td>$-$</td>
<td>$+$</td>
<td>6,7,4,2,5,3,0,1</td>
<td>B,D,R</td>
</tr>
<tr>
<td>$-$</td>
<td>$+$</td>
<td>$-$</td>
<td>5,4,7,1,6,0,3,2</td>
<td>B,U,L</td>
</tr>
<tr>
<td>$-$</td>
<td>$+$</td>
<td>$+$</td>
<td>4,5,6,0,7,1,2,3</td>
<td>B,U,R</td>
</tr>
<tr>
<td>$+$</td>
<td>$-$</td>
<td>$-$</td>
<td>3,2,1,7,0,6,5,4</td>
<td>F,D,L</td>
</tr>
<tr>
<td>$+$</td>
<td>$-$</td>
<td>$+$</td>
<td>2,3,0,6,1,7,4,5</td>
<td>F,D,R</td>
</tr>
<tr>
<td>$+$</td>
<td>$+$</td>
<td>$-$</td>
<td>1,0,3,5,2,4,7,6</td>
<td>F,U,L</td>
</tr>
<tr>
<td>$+$</td>
<td>$+$</td>
<td>$+$</td>
<td>0,1,2,4,3,5,6,7</td>
<td>F,U,R</td>
</tr>
</tbody>
</table>

*R = right, L = left; U = up; D = down; F = front; B = back.
Marching Cubes Algorithm

voxels $\rightarrow$ mesh

process 8 voxels at a time

create surfaces that represent those 8 voxels

$2^8$ possible combinations
14 unique cases, lookup table can be created by mirroring and rotating those 14 cases

(something similar can also be done in 2D)

Lorensen, Cline 1987
Ray tracing

Simulation of light movement

Select camera position for each pixel
determine ray from camera to pixel
for each object in scene
if object is intersected and is closest
record intersection and object
for each light
determine how much light reaches intersection
set pixel's color

Simulation
bounding volumes

instead of testing against object directly
test against bounding volume first
hierarchy traversal

Use hierarchical scene representation

BSP, Octtree, R-tree...

Z-buffer

pre-process scene with scan conversion to find intersections
use found intersections to calculate rays

shooting rays individually against each object is inefficient
Constructive Solid Geometry

Mesh geometry contains lots of triangles to test against. CSG contains only primitives that can be tested faster.

\[
\text{intersect}(\text{Ray } r, \text{ node } n)\{
\text{span left, right}
\text{if ( n is composite )}\{
\text{left = intersect(ray, n.left)}
\text{if left == NULL && node.op != UNION}
\text{return NULL}
\text{else}
\text{right = intersect(ray, n.right)}
\text{return combine(node.op, left, right)}
\} \text{ else}
\text{return intersect with n's object}
\}\
\]
Sparse Voxel Octree

Efficient storage of voxels and efficient way of doing ray casts.

Voxels with additional contour data.

Figure 5: Left: cubical voxels. Right: the same voxels with contours. In this kind of situation where surfaces are reasonably smooth, contours can provide several hierarchy levels’ worth of geometric resolution improvement. Note that the model has been deliberately undersampled to illustrate the effect.

Figure 4: Effect of contours on surface approximation. Top row: cubical voxels. Bottom row: voxels enhanced with contours. The resulting approximation follows the original surface much more closely than in the top row. Sharp corners can be approximated by taking contours on multiple levels into account simultaneously.
a little bit of texture & image generation...
Perlin Noise

procedural texture primitive

function of either (x,y), (x,y,z),(x,y,z,time)

noise that changes smoothly

often used for visual effects: fire, smoke, clouds
add together multiple levels of different noise
func noise(x, y)
    return PseudoRandom(x,y)
}

func SmoothNoise(x, y)
    corners = (Noise(x-1,y-1) + Noise(x+1,y-1) +
               Noise(x+1,y-1) + Noise(x+1,y-1)) / 16
    sides   = (Noise(x-1,y) + Noise(x,y-1) +
               Noise(x+1,y) + Noise(x,y+1)) / 8
    center  = Noise(x, y) / 4
    return corners + sides + center
}

func InterpolatedNoise(x,y)
    int_x = int(x); frac_x = x - int_x
    int_y = int(y); frac_y = y - int_y
    v1 = SmoothNoise(int_x, int_y)
    v2 = SmoothNoise(int_x + 1, int_y)
    v3 = SmoothNoise(int_x, int_y + 1)
    v4 = SmoothNoise(int_x + 1, int_y + 1)
    i1 = Interpolate(v1, v2, frac_x)
    i2 = Interpolate(v3, v4, frac_x)
    return Interpolate(i1, i2, frac_y)
}
func PerlinNoise_2D(x, y) {
    total = 0
    p = persistence
    n = number_of_octaves - 1
    for i = 0 to n {
        freq = 2^i
        amplitude = p^i

        total += InterpolatedNoise(x*freq, y*freq) * amplitude
    }
    return total
}
Fractals: Mandelbrot set

\[ z_{n+1} = z_n^2 + c \quad c \in \mathbb{C} \]

if \( \lim_{n \to \infty} |z_n| < f(c), \ c \in M \)

for each pixel {
    x0 : scaled (-2.5..1)
    y0 : scaled (-1..1)
    x,y = 0
    it = 0
    max = 1000
    while( x^2 + y^2 <= 2^2 and
            it < max ){
        xt = x^2 - y^2 + x0
        y = 2*x*y + y0
        x = xt
        it++
    }
    pixel.color = it == max ? black
    : colors[it % color_count]
}
Buddhabrot

\[ z_{n+1} = z_n^2 + c \quad c \in \mathbb{C} \]

if \( \lim_{n \to \infty} |z_n| < f(c), \ c \in M \)

create 2dim array of counters
randomly sample c values
if c escapes max-iterations
plot \( z_0 \ldots z_{\text{max}} \)

use counters to determine pixel gray values
Fractal Flames

Scott Draves 1992

a video of flame animated
Set of flame functions

\[
\begin{align*}
F_1(x, y), & \quad p_1 \\
F_2(x, y), & \quad p_2 \\
\ldots \\
F_n(x, y), & \quad p_n
\end{align*}
\]

in each iteration, choose one function with probability \( p_j \)
then compute next iteration by applying \( F_j \) on \((P.x, P.y)\)
each function has the following form:

\[
F_j(x, y) = \sum_{V_k \in \text{Variations}} w_k \cdot V_k(a_j x + b_j y + c_j, d_j x + e_j y + f_j)
\]

\( V_k \) are a set of predefined functions
\( V_0 = (x, y) \) [ linear ]
\( V_1 = (\sin x, \sin y) \) [ sinusoidal ]
\( V_2 = (x, y) / (x^2 + y^2) \) [ spherical ]
color is blended with associated function color

\[ P_{\text{new}}.c := \frac{(P.c + (F_i)_{\text{color}})}{2} \]

after each iteration, update histogram at the corresponding point \( P \)

\[
\text{histogram}[x][y][\text{freq}]++ \quad \text{histogram}[x][y][\text{color}] = \frac{(\text{histogram}[x][y][\text{color}] + P.c)}{2}
\]

**Rendering**

for each pixel\((x,y)\):

\[
\text{freq\_avg}[x][y] := \text{avg\_histogram\_cells\_freq}(x,y) \\
\text{color\_avg}[x][y] := \text{avg\_histogram\_cells\_color}(x,y)
\]

\[
\text{alpha}[x][y] := \frac{\log(\text{freq\_avg}[x][y])}{\log(\text{freq\_max})}
\]

\[
\text{pixel}[x][y] := \text{color\_avg}[x][y] \times \text{alpha}[x][y]^{(1/\gamma)}
\]