Planes, Polygons and Objects

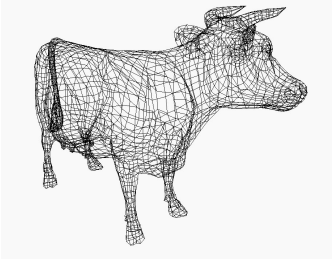
Overview

- Polygons
- Planes
- Creating an object from polygons

No More Spheres

- Most things in computer graphics are not described with spheres!
- Polygonal meshes are the most common representation
- Look at how polygons can be described and how they can used in ray-casting

Polygonal Meshes



Polygons

- A polygon (face) Q is defined by a series of points

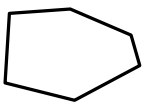
$$[p_0, p_1, p_2, \ldots, p_{n-1}, p_n]$$

$$p_i = (x_i, y_i, z_i)$$

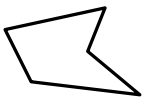
- The points are must be **co-planar**
- 3 points define a plane, but a 4th point need not lie on that plane

Convex, Concave

- Convex



- Concave



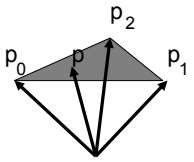
  - CG people dislike concave polygons
  - CG people would prefer triangles!!
    - Easy to break convex object into triangles, hard for concave

Equation of a Plane

$$ax + by + cz = d$$

- a,b,c and d are constants that define a unique plane and x,y and z form a vector P.

Deriving a,b,c & d (1)



- The cross product
  $n = (p_1 - p_0) \times (p_2 - p_0)$
  defines a **normal** to the plane
- There are two normals (they are opposite)
- Vectors in the plane are **all** orthogonal to the plane normal vector

Deriving a,b,c & d (2)

- So every p-p_0 is normal to n therefore

$$n \cdot (p - p_0) = 0 \quad \Leftrightarrow$$

$$n \cdot p = n \cdot p_0$$

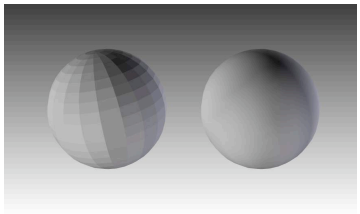
- But if $n = (a,b,c)$ and $p = (x,y,z)$ and $d = n.p0 = n_1 * x_0 + n_2 * y_0 + n_3 * z_0$

$$\Rightarrow \quad ax + by + cz = d$$

Half-Space

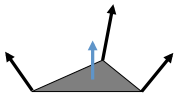
- A plane cuts space into 2 **half-spaces**
- Define $l(x, y, z) = ax + by + cz - d$
- If $l(p) = 0$
  - point on plane
- If $l(p) > 0$
  - point in **positive** half-space
- If $l(p) < 0$
  - point in **negative** half-space

Vertex Normals



Vertex Normals

- Compute/store a normal at each vertex



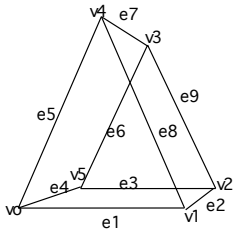
- Improves shading
- Compute by averaging neighbouring "face normals" (blue)

Polyhedra

Polyhedra

- Polygons are often grouped together to form polyhedra
  - Each edge connects 2 vertices and is the join between two polygons
  - Each vertex joins 3 (or more) edges
  - No faces intersect
- #Vertices - #Edges + #Faces = 2
  - For cubes, tetrahedra, cows etc...

Example Polyhedron



- F0=v0 v1 v4
- F1=v5 v3 v2
- F2=v1 v2 v3 v4
- F3=v0 v4 v3 v5
- F4=v0 v5 v2 v1

- V=6, F=5, E=9
- V-E+F=2

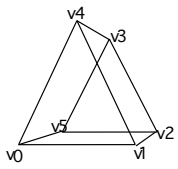
Representing Polyhedron (1)

- Exhaustive (array of vertex lists)
  - faces[1] = (x0,y0,z0),(x1,y1,z1),(x3,y3,z3)
  - faces[2] = (x2,y2,z2),(x0,y0,z0),(x3,y3,z3)
  - etc ….
- Very wasteful since same vertex appears at 3 (or more) points in the list
  - Is used a lot though!

Representing Polyhedron (2)

- Indexed Face Set
  - Vertex array
    - vertices[0] = (x0,y0,z0)
    - vertices[1]= (x1,y1,z1)
    - etc …
  - Face array (list of indices into vertex array)
    - faces[0] = 0,2,1
    - faces[1] = 2,3,1
    - etc ...

Vertex order matters

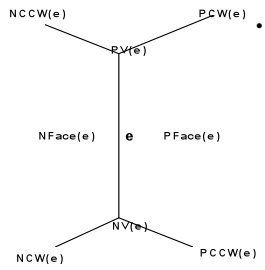


- Polygon v0,v1,v4 is NOT equal to v0,v4,v1
- The normal points in different directions
- Usually a polygon is only visible from points in its positive half-space
- This is known as **back-face culling**

Representing Polyhedron (3)

- Even Indexed face set wastes space
  - Each face edge is represented twice
- Winged edge data structure solves this
  - vertex list
  - edge list (vertex pairs)
  - face list (edge lists)

The Edge List Structure



- Edge contains
  - Next edge CW
  - Next edge CCW
  - Prev edge CW
  - Prev edge CCW
  - Next face
  - Prev face
  - Next vertex
  - Prev vertex

Advantages of Winged Edge

- Simple searches are rapid
  - find all edges
  - find all faces of a vertex
  - etc…
- Complex operations
  - polygon splitting is easy (LOD)
  - silhouette finding
  - potentially efficient for hardware
  - etc…

Building the WE

- Build indexed face set
- Traverse each face in CCW order building edges
  - label p and n vertices, p and n faces and link previous CCW edge
    - we fill in next CCW on next edge in this face
    - we fill in next CW and prev CW when traversing the adjacent face.

Exercises

- Make some objects using index face set structure
- Verify that V-E+F=2 for some polyhedra
- Think about testing for intersection between a ray and a polygon (or triangle)

Recap

- We have seen definition of planes and polygons and their use in approximating general shapes
- We have looked at data structures for shapes
  - Indexed face sets
  - Winged edge data sets
- The former is easy to implement and fast for rendering
- The latter is more complex, but makes complex queries much faster
- It is possible, though we haven't shown how, to convert between the two

Coursework

- Ray-Tracing!
- On the web-page now
- Deadline:
  4/Nov/2009 by 12pm

3080/GV10 Coursework 1

Ray tracing: You are given a functioning ray casting program which computes the ambient, diffuse and specular component of light on a point on an object. You are asked to implement new functionalities to the ray tracer. This coursework has 2 parts.

Part 1: (50%)

You are required to add shadow rays to ray tracing

1. Implementing shadow rays requires you to shoot rays from the intial hit point on the object towards each light to test if that point is illuminated.

Part 2: (50%)

You are required to implement recursive ray tracing.

1. Implementing recursive casting (reflections) requires you to reflect rays from the hit point using the method described in the notes. Note that there is a maximum recursion depth.

Extension:

If you want, you can get extra points by implementing an acceleration method by testing the interestion of rays with bounding boxes of objects, instead of the polygons. You can extend this to the use of a hierarchical bounding box technique to accelererate the computation of the recursive ray tracer.

1. Create the hierarchical bounding box structure. You can group boxes in a random way.
2. Use this structure to accelerate the ray tracing computation.
3. Update the hierarchical bounding box structure by grouping objects that are near in space and/or with similar size.

Important:

*You need to hand in a print out of your commented code (only yours with the context), a **short report** refering to the technique you use - description of equations if necessary, the choice you made in your algorithm, and analysed results. I.e., you should include some pictures you produced in your report.*

These images show the before and after adding ray-tracing:

Question 1+2:

These images are based upon a scene file called *complexscene.dat* and are rendered using the C++ version. Materials vary from object to object and the lights are point and directional lights. The first image is done with the vanilla code (no shadows or reflections), whereas the second image was created with shadows and recursion (MaxDepth set to 4). You can choose a lower value so that your code runs more quickly. Your result images should look similar to the latter.

Java: You can continue from your own code from lab2, or start from the model answer to lab exercise 2. This code can be found here and the Javadoc is available here. Again a ZIP file of the code and documentation can be found here. Copy all the .java files to your home directory. Also copy the simplescene.dat. Compile the java code, and then run by executing "java SimpleRay simplescene.dat".

Question 1:

The part of the code you must edit is in Scene.java, in the method calculateColour.

Hint 1

Note when tracing towards lights that you want to know if there is an object between the hit point and the light. The easiest way to do this is to set up a ray so that when t=0 you are at the hit point, and when t=1 you are at the light. If you cast the ray and the first intersection point occurs when t>0 and t<1 then something blocks the light. I.e., make a ray with

```
Origin = p;
Direction = direction;
```

BUT do not normalise direction. Your test for success will look then check if t is in the correct range.

Note that you still must normalise the direction to the light before doing the specular contribution, otherwise the vector h will be skewed. Think carefully what should happen in the case of directional lights.

Hint 2

In the Question code there is one change from the answer to Exercise 1 - the label "nextlight" has been added in calculateColour (see comment in the code). Look up what "continue" does.

Finally

The software described here and in the associated directories are for your personal use only, and in conjunction with the coursework for this course.

On no account may they be shown or in any way given to any person not on this course without permission from Anthony Steed.

They may be copied to your personal machine for use at home, without requiring permission.

On no account may any aspect of the software be used or modified for use for any commercial reason whatsoever.

Anthony Steed, A.Steed@cs.ucl.ac.uk
Jan Kautz, j.kautz@cs.ucl.ac.uk

Last modified: 10/16/2009 09:38:19