3D Modelling

Lecture 10
Mesh Representations

Patrick Min

Utrecht University

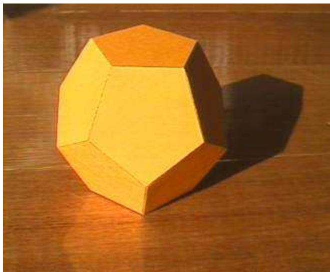
March 15, 2005

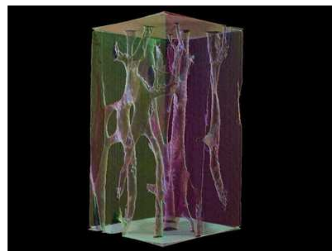
Outline

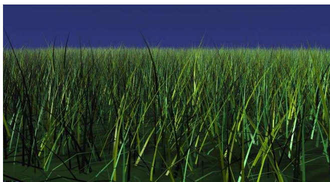
1. Introduction
2. Volume representations
3. Surface representations

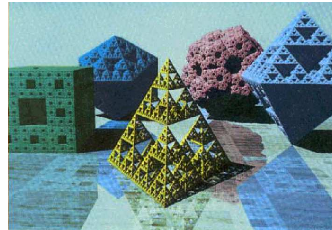
Which representation?

How would you represent these objects?









Classification of representations

- surface vs. volume
- explicit vs. implicit vs. procedural
- there are more types, and classification criteria
- at the end of the day, you want to unambiguously specify a shape, in a way that's suited to your application

Equivalence of Representations

"A 3D model representation is a (formal) language describing geometry"

- eventually, each representation results in a discrete sampling of the surface and/or interior of an object
- so the object is built up out of "discrete elements" (a vertex, a voxel, ...) and their connections (edges, adjacency, ...)
- each "discrete element" may have additional information associated with it (connectivity, appearance, physical properties, etc.)
- the elements may also be organized (in groups (parts), in a hierarchy, ...)

"All models are wrong, but some are useful"

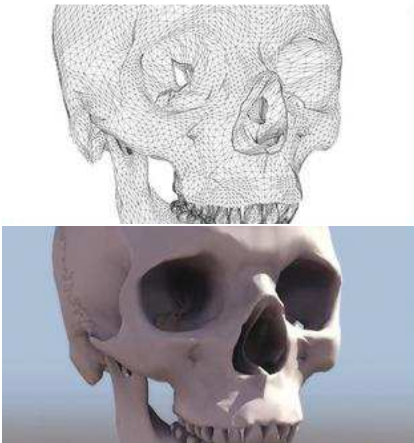
George Box, 1979, (on theories of scientific models)

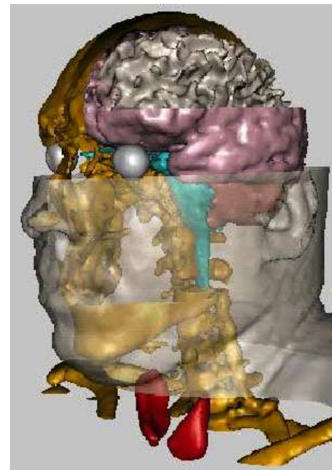
Representation possibilities

- the representation determines what are simple or complex operations
- so the required operations determine the best representation
- most common application: **rendering/visualization**
- so the most common representations are the **surface** representations (what you don't see, you don't have to represent)

Surface vs. Volume

→ inherently 2D and inherently 3D representation





images c/o de Espona Infografica, The Visible Human Project

Outline

1. Introduction
2. Volume representations
3. Surface representations
   - Mesh data structures

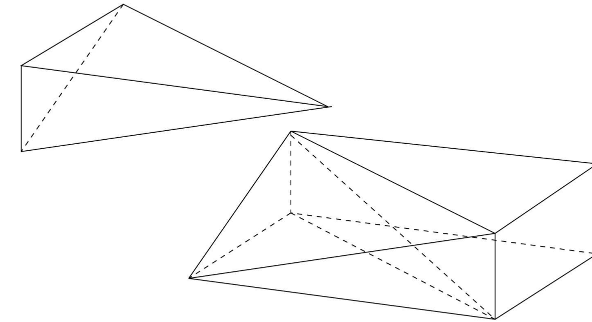
Volume representations

distinguished by the type of *volume element* used:

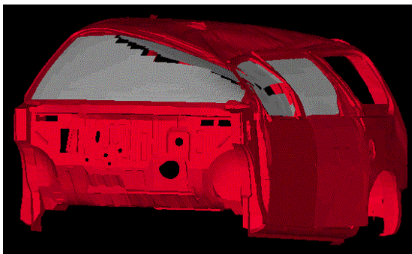
- voxel grids: voxels
- spatial subdivisions: part of space
- tetrahedral mesh: tetrahedron
- ...

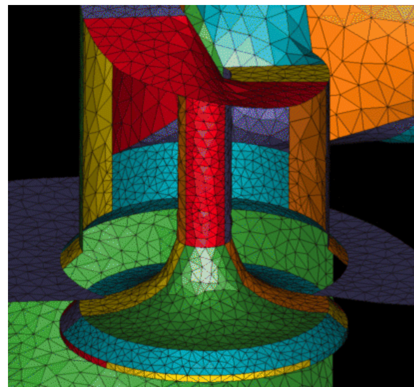
Example representation: tetrahedral mesh

- How many tetrahedra make up this block?
- Answer: **six**



Tetrahedral mesh applications



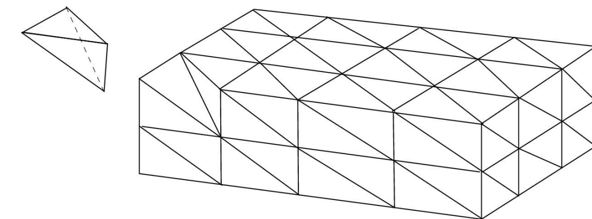


Finite Element Analysis (FEM)

- crush analysis
- heat transfer modelling (in engines, for example)
- geological surveying
- pressure flow modelling (around aircraft, for example)

Tetrahedral mesh generation

Starting from a surface triangulation:



Current methods are much more sophisticated

Outline

1 Introduction

2 Volume representations

3 Surface representations
- Mesh data structures

Surface representations

Explicit:

- parametric surface
- spline patches
- polygonal mesh

Implicit:

- zero set of an implicit function

The polygonal mesh is the most common surface representation.

Polygonal meshes

This is by far the most popular format, because visualization is by far the most popular application.

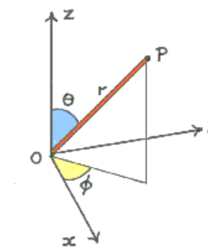
We mostly deal with *triangular* meshes: if a mesh is not triangular, it can easily be triangulated.

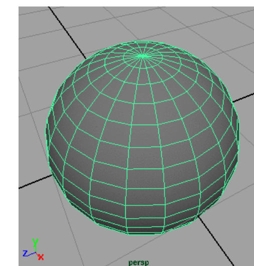
Advantages of the triangle primitive:

- the simplest object (with area)
- always planar (so easy to rasterise)
- specified with 3 vertices
- supported in hardware

Discretization of continuous surfaces

- It is easy to create a polygonal mesh from a parameterized surface
- Given, for example, the sphere $x^2 + y^2 + z^2 = 1$
- In polar coordinates, this is: $(r\cos(\phi)\sin(\theta), r\sin(\phi)\sin(\theta), r\cos(\theta))$ (with $r = 1$)



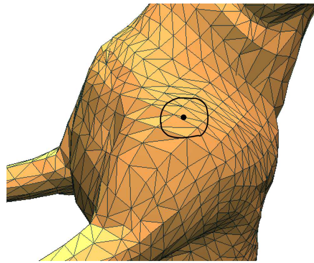


Creating more complex/simpler versions is straightforward

2-manifold polygonal meshes

A **2-manifold** surface: every local neighbourhood of a point on the surface is homeomorphic (= can be deformed into) a disc.
A triangular mesh is 2-manifold if:

- all boundaries between triangles are exactly one edge
- each edge borders exactly one or two triangles



(the **2** refers to the fact that the manifold is 2-dimensional)

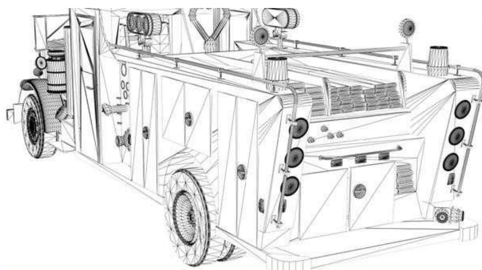
2-manifold polygonal meshes

An **oriented** 2-manifold mesh has an inside and an outside, so:

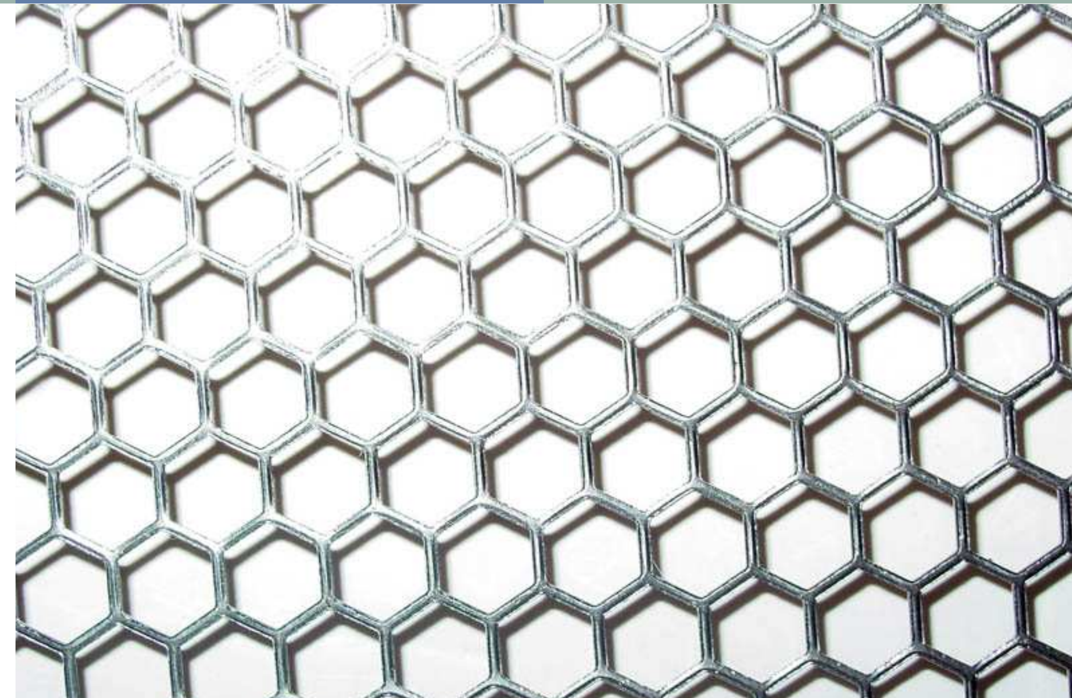
- this can be determined for each face
- as a result, there is some volume information

Polygon soup

But our goal was visualization anyway, so who cares about 2-manifold meshes?



- the **polygon soup** is a common 3D model representation...
- but: models are often made up of parts that are 2-manifold



Mesh data structures

A polygonal mesh consists of:

- vertices
- edges
- faces

The vertices specify the **geometrical** information (locations), the faces and/or edges the **topological** information (how the vertices are connected)

How to encode this in a datastructure?

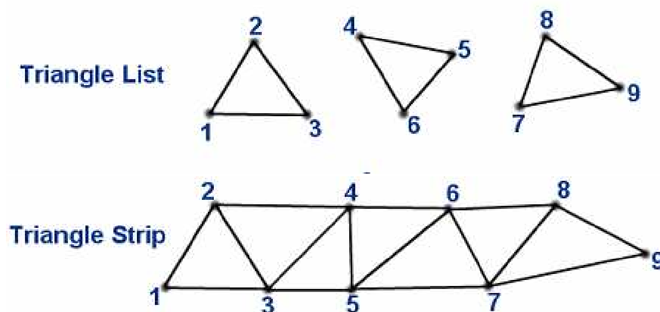
A practical aside: OpenGL

OpenGL = Open Graphics Language

- derived from Iris GL, by Silicon Graphics
- OS-independent API to graphics hardware
- successful because it is easy to program
- several libraries now exist on top of OpenGL:
  - GLU (for modelling (quadrics, NURBS, etc.))
  - GLUT (windowing, user I/O)
  - etc.
- the Java package `jogl` has Java bindings for OpenGL

Triangles, strips and fans

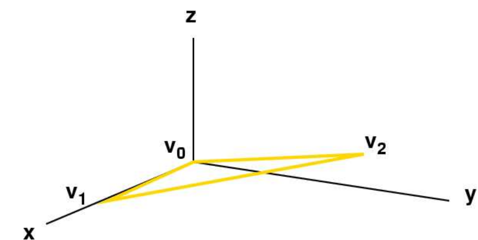
How to encode sequences of triangles?



OpenGL example

A yellow triangle, $v_0 = (0, 0, 0)$, $v_1 = (1, 0, 0)$, $v_2 = (0.5, 2, 0.5)$

```
glColor3f(1.0, 1.0, 0.0);
glBegin(GL_TRIANGLES);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(1.0, 0.0, 0.0);
  glVertex3f(0.5, 2.0, 0.5);
glEnd();
```

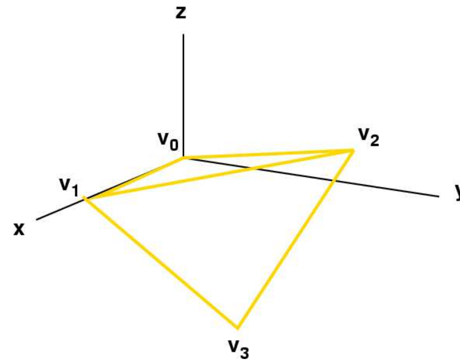


What you see on the screen depends on where you put the camera

OpenGL, two triangles

These two triangles can be drawn as separate triangles ($v_3 = (2, 2, 0)$):

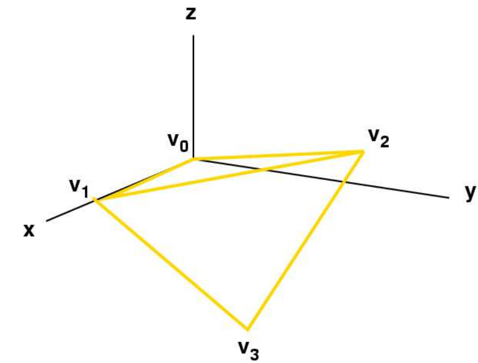
```
glColor3f(1.0, 1.0, 0.0);
glBegin(GL_TRIANGLES);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(1.0, 0.0, 0.0);
  glVertex3f(0.5, 2.0, 0.5);

  glVertex3f(0.5, 2.0, 0.5);
  glVertex3f(1.0, 0.0, 0.0);
  glVertex3f(2.0, 2.0, 0.0);
glEnd();
```



OpenGL triangle strip

... or as two "joined" triangles (a **triangle strip**):

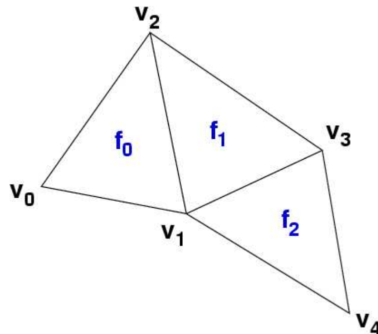
```
glColor3f(1.0, 1.0, 0.0);
glBegin(GL_TRIANGLE_STRIP);
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(1.0, 0.0, 0.0);
  glVertex3f(0.5, 2.0, 0.5);
  glVertex3f(2.0, 2.0, 0.0);
glEnd();
```



This saves 2 vertices for each additional triangle!
$\rightarrow$ applications in mesh compression

A simple mesh data structure

$\rightarrow$ list the vertices, and for each face have a list of indices into the vertex array
$v_0 = (1, 1, 0), v_1 = (2, 0.9, 0), v_2 = (1.8, 2, 0), \ldots$
$f_0 = \{v_0, v_1, v_2\}, f_1 = \{v_2, v_1, v_3\}, f_2 = \{v_1, v_4, v_3\}$
(Note the counter-clockwise ordering of the vertices)



This data structure is sufficient for rendering (`GL_TRIANGLES` or `GL_POLYGONS`)

A simple mesh file format: `OFF`

`OFF` = Object File Format

```
OFF
nr_vertices nr_faces nr_edges
x1 y1 z1
...
xn yn zn
nr_vertices_of_face_1 v_11 v_12 ... v_1k
...
nr_vertices_of_face_m v_m1 v_m2 ... v_mk
```

- `nr_edges` is always 0

A simple mesh file format: OFF

Example: OFF file of a cube:

```
OFF
8 6 0
-0.50 -0.50 0.50
0.50 -0.50 0.50
-0.50 0.50 0.50
0.50 0.50 0.50
-0.50 0.50 -0.50
0.50 0.50 -0.50
-0.50 -0.50 -0.50
0.50 -0.50 -0.50
4 0 1 3 2
4 2 3 5 4
4 4 5 7 6
4 6 7 1 0
4 1 7 5 3
4 6 0 2 4
```

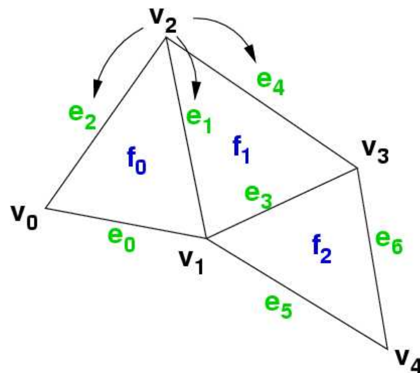

Operations on meshes: edge paths

- With the simple data structure, all operations are possible
- But some are harder than others
- What should be added if we want to traverse the mesh along its edges?
  (we may want to cut the mesh into parts)

(in the following, assume that if there are faces then we have face to vertex pointers (indices), and that if there are edges then we have edge to vertex pointers (indices))

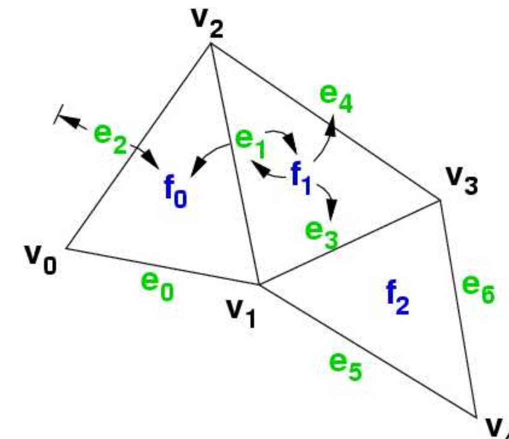
Operations on meshes: edge paths

- $e_0 = (v_0, v_1), e_1 = (v_1, v_2), e_2 = (v_2, v_0), e_3 = (v_1, v_3), e_4 = (v_3, v_2), e_5 = (v_1, v_4), e_6 = (v_4, v_3)$
- add pointers from a vertex to all its edges (need a dynamic array/ linked list for this)



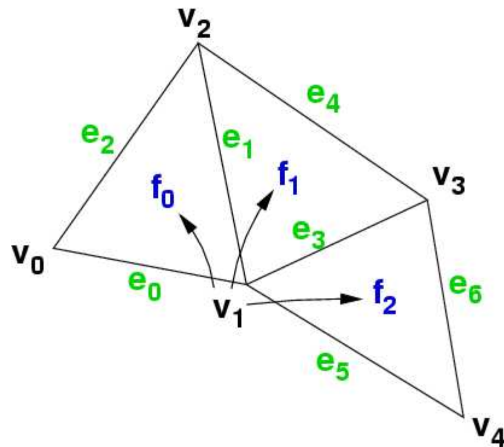
Operations on meshes: shortest paths

- Now we want to find a shortest path from a vertex v_s to a vertex v_g, not limited to edges only
- one option: face-to-edge pointers (always three if we're using triangles only) and edge-to-face pointers (always two)



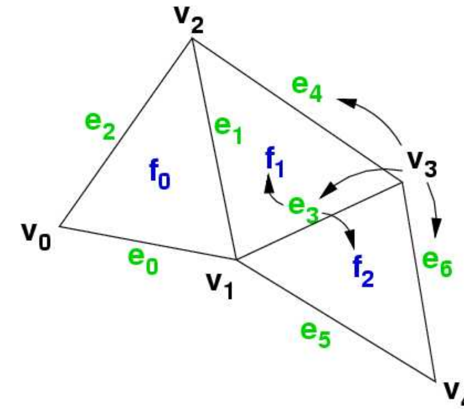
Operations on meshes: computing a vertex normal

- we want to compute an approximate vertex normal, given the normals of its surrounding faces
- so we need vertex to face pointers

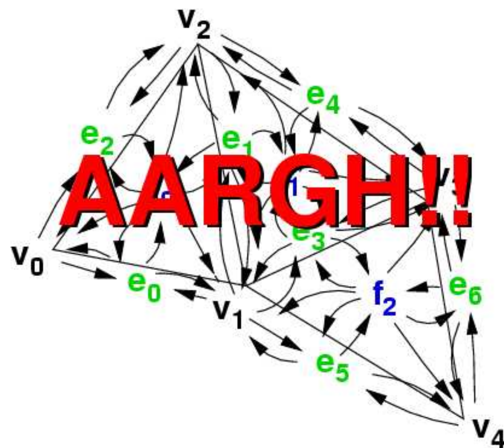


Operations on meshes: edge collapse

- we want to simplify a mesh using edge collapses
- so we need edge to face pointers (because we have to know which faces must be deleted) and vertex to edge pointers (because we have to know which edges must be combined)



- keep track of everything...
- plus the pointers we already have...



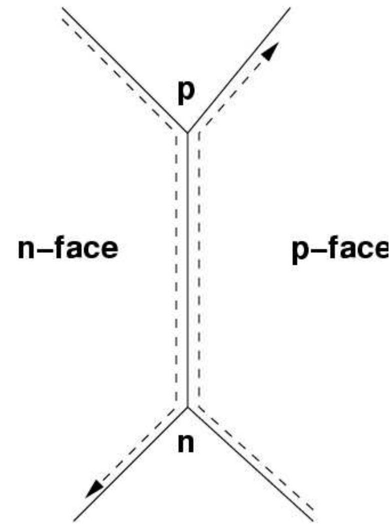
The Winged-Edge Data Structure

- developed by B. Baumgart, Stanford, 1972
- with the goal to make certain mesh operations cheaper
- for example: which two faces share a particular edge?

The Winged-Edge Data Structure

An **edge** points to:

- its vertices **n** and **p**
- its two adjacent faces:
  - when travelling from **n** to **p**, the face to the *right* is the **p-face**
  - the other face is the **n-face**
- *four* additional edges:
  - at the **n** vertex: the next (clockwise) edge of the **n-face**, and the previous (counter-clockwise) edge of the **p-face**
  - the opposite at the **p** vertex



The Winged-Edge Data Structure

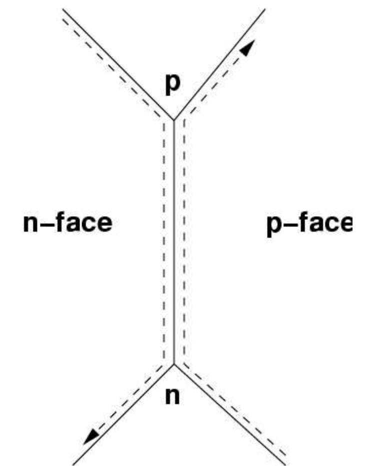
A **vertex** points to:

- **one** of its incident edges

A **face** points to:

- **one** of its surrounding edges

Now it is easy to answer adjacency relationship questions of the form: which {vertices, edges, faces} are adjacent to which {vertices, edges, faces}? (9 combinations)

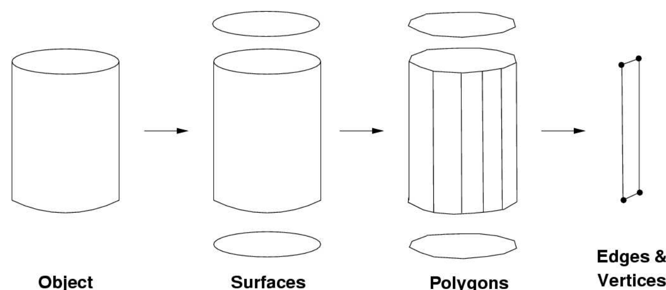


(Note: the original winged-edge data structure described on `http://www.baumgart.org/winged-edge/winged-edge.html` contains more pointers)

Another example data structure

from Watt & Policarpo (2.2):

- An object consists of multiple surfaces,
- each surface is represented by a polygon,
- each polygon consists of multiple edges,
- each edge has two vertices.



Reading

- Watt, 2.2