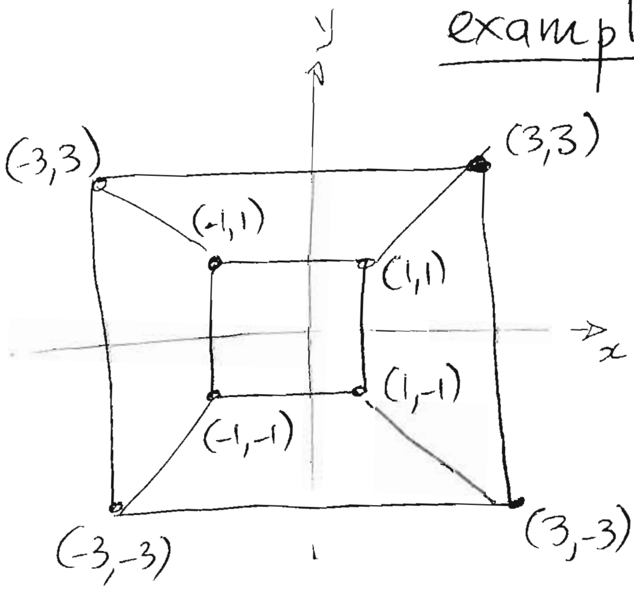
We shall use the term "meshed geometry" to denote the discrete data structure used to encode a geometric object, using a mesh.

A "meshed geometry" is a composite structure, containing 2 data structures:

"Meshed geometry" (or "geometry", for short) → "Particles"
→ "Mesh" (or <u>topology</u> or <u>connectivity</u>)

<u>example</u> (A quadrilateral mesh)



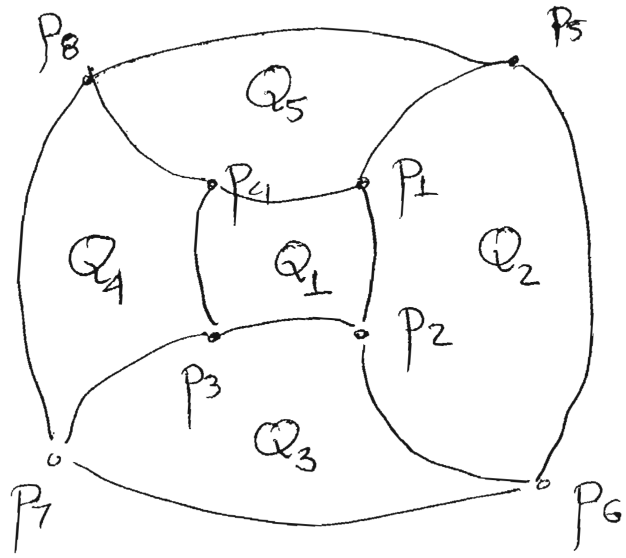
This geometric object is decomposed as follows:

Particle data structure

Particle ID	Position
P1	(1,1)
P2	(1,-1)
P3	(-1,-1)
P4	(-1,1)
P5	(3,3)
P6	(3,-3)
P7	(-3,-3)
P8	(-3,3)

Mesh data structure (essentially, a graph)

Symbolically:



On the computer

Element (Quad) ID	Vertices
Q_1	(p_1, p_2, p_3, p_4)
Q_2	(p_1, p_5, p_6, p_2)
Q_3	(p_2, p_6, p_7, p_3)
Q_4	(p_3, p_7, p_8, p_4)
Q_5	(p_4, p_8, p_5, p_1)

<u>Note</u>: We prefer encoding the vertices in a principled way, here, e.g. in clockwise order

<u>Why "particles" and not just "points" or "vertices"?</u>

A "<u>particle</u>" is a placeholder for more information than just a position coordinate. It may contain:

- → Position
- → Velocity
- → Acceleration, or force
- → Mass

} Physical attributes

- → Texture coordinates
- → Color, etc

} Secondary attributes.

Here, we used the term "attribute" to denote the constituent physical/practical quantities that comprise a particle.

Implementation/storage of particles

Approach 1:

```
struct Particle {
  float position[3];
  float velocity[3];
  float mass;
};
struct Particle particle_array[N];
```

Approach 2:

```
struct Particles {
  float positions[N][3];
  float velocities[N][3];
  float masses[N];
} particle_array;
```

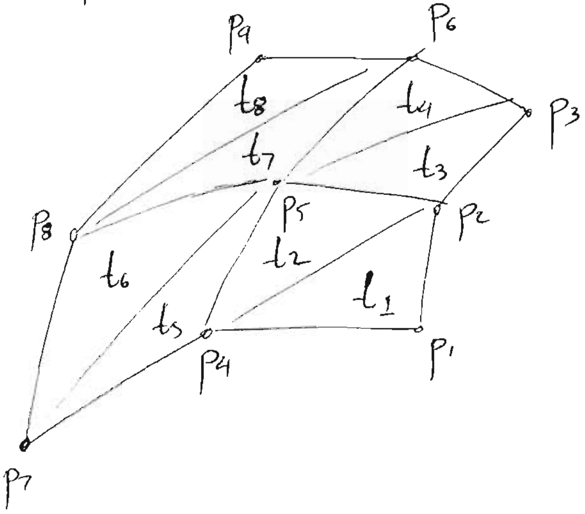
#1 Benefits

→ Particles are self-contained
→ Easy to construct subsets of particles
→ Can extend to accommodate particles with different attributes, on the same array

#2 Benefits

→ Simulation algorithms typically stream through the arrays of positions & velocities at different passess: keeping them separate exploits bandwidth better.
→ Easy to construct a subset of attributes
e.g. for visualization, we may only need positions, not velocities or masses.

Other examples:



A surface in 3D
(eg. cloth)

Particles:

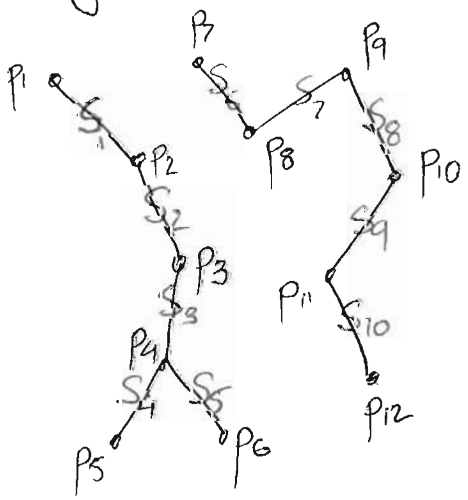
p_1	(x_1, y_1, z_1)
p_2	(x_2, y_2, z_2)
	$\vdots$
p_9	(x_9, y_9, z_9)

(Triangle) Mesh:

t_1	$(1,2,4)$
t_2	$(2,5,4)$
t_3	$(2,3,5)$
$\vdots$	$\vdots$
t_8	$(6,9,8)$

(CCW ordering)

A linear segmented curve (in 2D)
(e.g. hair, or a crack pattern)



Note: Geometry is disconnected!

Particles
p_1 (x_1, y_1)
p_2 (x_2, y_2)
$\vdots$
p_{12} (x_{12}, y_{12})

Segment Mesh
s_1 $(1,2)$
s_2 $(2,3)$
s_3 $(3,4)$
s_4 $(4,5)$
s_5 $(4,6)$
$\vdots$
s_{10} $(11,12)$

Naming conventions (not widely standardized, just for CS838b)

"Meshed" "object"

1st part (constituent element type)

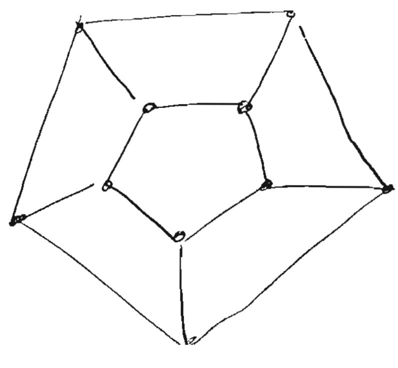
Tetrahedralized Triangulated Segmented Polyhedralized Polygonized Hexahedralized Quadrangulated ⋮	+	Volume Surface (3D) Area (2D) Curve (2D or 3D)	= Geometry name

e.g. Tetrahedralized volume ⇒ 3D object with interior structure, tessellated with tetrahedra.

Triangulated surface ⇒ 3D surface composed of triangles, without interior structure.

Polygonized area ⇒

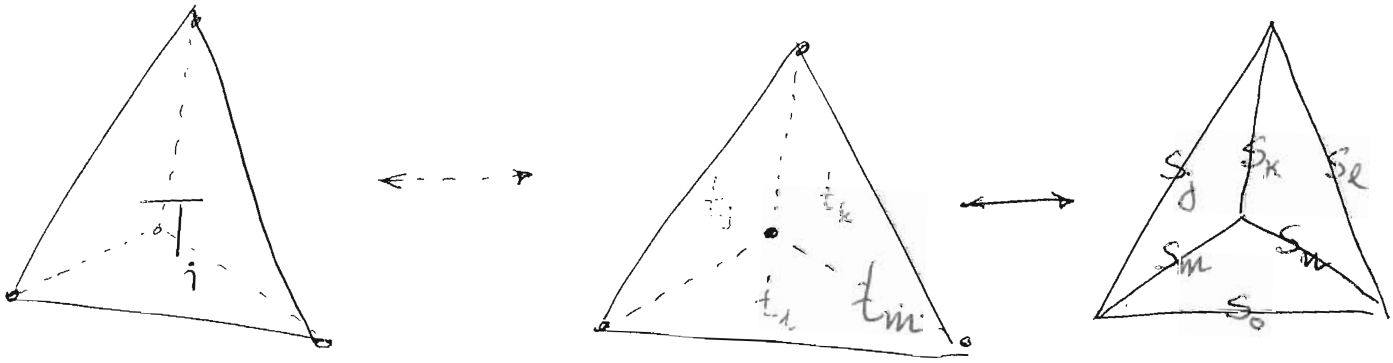
e.g.



etc.

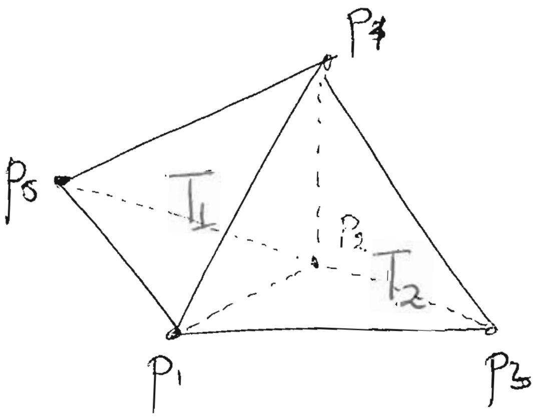
Derivative geometries

→ A triangle contains 3 segments (edges), and
→ A tetrahedron contains 4 triangles (faces) and 6 segments (edges)



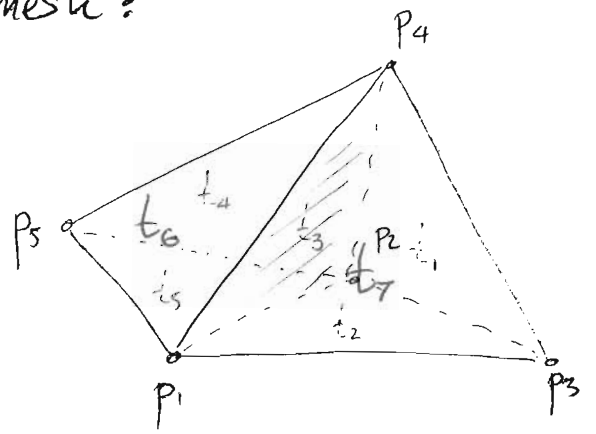
As a result, any triangle mesh implicity defines a segment mesh of its edges or, respectively, any triangulated area implicitly contains a segmented curve.

Likewise, in 3D, eg.



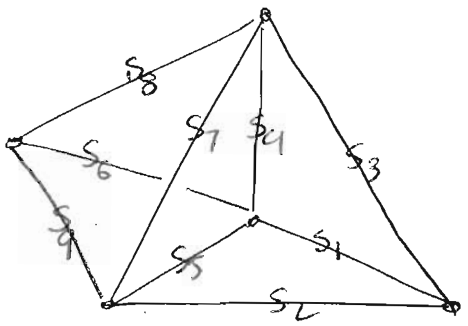
T_1	(1,2,3,4)
T_2	(1,2,4,5)

This volume also contains the following triangle mesh:



t_1	(2,3,4)
t_2	(1,3,2)
t_3	(1,2,4)
	⋮
t_6	(5,1,4)
t_7	(1,3,4)

or the segment mesh



Observations

→ The resulting derivative geometries share the exact same particle array

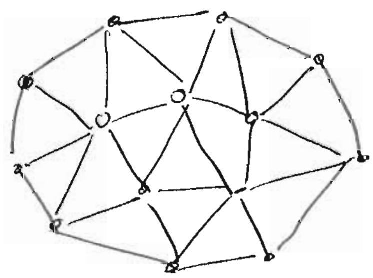
→ The derivative geometries are not requiring any user input for their construction, i.e. their information is fully dependent on the primary geometry (with the exception of the numbering choice of the derivative elements).

→ No need to store them (other than preserving the numbering). Can rebuild them on demand.

Topological queries & acceleration structures

When using a mesh-based geometry (with, or without derivative geometries) for a simulation algorithm, questions such as the following often arise:

→ Which tetrahedra contain p_i as one of their vertices?
→ Which triangles (on a triangulated surface) share a common edge with triangle t_i
→ Which segments lie on the boundary of a triangulated area?



red = boundary
black = interior

Observations

→ The answers to these queries are independent of the exact particle locations (i.e. queries are purely topological in nature).
→ The answers can be precomputed, from the information in the meshed geometry (and its derivatives), e.g.

int ** incident_tetrahedra → incident_tetrahedra[i][j] is the j-th of the tetrahedra that share vertex p_i

int (* adjacent_triangles)[3] → adjacent_triangles[i][j] is the j-th triangle sharing an edge with t_i