Notes

- More optional reading on web for collision detection

Triangles

- Given x_1, x_2, x_3 the plane normal is

$$n = \frac{(x_2 - x_1) \times (x_3 - x_1)}{\left|(x_2 - x_1) \times (x_3 - x_1)\right|}$$

- Interference with a closed mesh
  - Cast a ray to infinity, parity of number of intersections gives inside/outside
- So intersection is more fundamental
  - The same problem as in ray-tracing

Triangle intersection

- The best approach: reduce to simple predicates
  - Spend the effort making them exact, accurate, or at least consistent
  - Then it's just some logic on top
  - Common idea in computational geometry
- In this case, predicate is sign of signed volume (is a tetrahedra inside-out?)

$$\mathrm{orient}(\mathrm{x}_0, \mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3) = \mathrm{sign}\,\det\begin{pmatrix} x_1 - x_0 & y_1 - y_0 & z_1 - z_0 \\ x_2 - x_0 & y_2 - y_0 & z_2 - z_0 \\ x_3 - x_0 & y_3 - y_0 & z_3 - z_0 \end{pmatrix}$$

Using orient()

- Line-triangle
  - If line includes x_4 and x_5 then intersection if orient(1,2,4,5)=orient(2,3,4,5)=orient(3,1,4,5)
  - I.e. does the line pass to the left (right) of each directed triangle edge?
  - If normalized, the values of the determinants give the **barycentric coordinates** of plane intersection point
- Segment-triangle
  - Before checking line as above, also check if orient(1,2,3,4) != orient(1,2,3,5)
  - I.e. are the two endpoints on different sides of the triangle?

Other Standard Approach

- Find where line intersects plane of triangle
- Check if it's on the segment
- Find if that point is inside the triangle
  - Use barycentric coordinates
- Slightly slower, but worse: less robust
  - round-off error in intermediate result: the intersection point
  - What happens for a triangle mesh?
- Note the predicate approach, even with floating-point, can handle meshes well
  - Consistent evaluation of predicates for neighbouring triangles

Distance to Triangle

- If surface is open, define interference in terms of distance to mesh
- Typical approach: find closest point on triangle, then distance to that point
  - Direction to closest point also parallel to natural normal
- First step: barycentric coordinates
  - Normalized signed volume determinants equivalent to solving least squares problem of closest point in plane
- If coordinates all in [0,1] we're done
- Otherwise negative coords identify possible closest edges
- Find closest points on edges

Testing Against Meshes

- Can check every triangle if only a few, but too slow usually
- Use an acceleration structure:
  - Spatial decomposition: background grid, hash grid, octree, kd-tree, BSP-tree, …
  - Bounding volume hierarchy: axis-aligned boxes, spheres, oriented boxes, …

Moving Triangles

- Collision detection: find a time at which particle lies inside triangle
- Need a model for what triangle looks like at intermediate times
  - Simplest: vertices move with constant velocity, triangle always just connects them up
- Solve for intermediate time when four points are coplanar (determinant is zero)
  - Gives a cubic equation to solve
- Then check barycentric coordinates at that time
  - See e.g. X. Provot, "Collision and self-collision handling in cloth model dedicated to design garment", Graphics Interface'97

For Later…

- We now can do all the basic particle vs. object tests for repulsions and collisions
- Once we get into simulating solid objects, we'll need to do object vs. object instead of just particle vs. object
- Core ideas remain the same

Elasticity

Elastic objects

- Simplest model: masses and springs
- Split up object into regions
- Integrate density in each region to get mass (if things are uniform enough, perhaps equal mass)
- Connect up neighbouring regions with springs
  - Careful: need chordal graph
- Now it's just a particle system
  - When you move a node, neighbours pulled along with it, etc.

Masses and springs

- But: how strong should the springs be? Is this good in general?
  - [anisotropic examples]
- General rule: we don't want to see the mesh in the output
  - Avoid "grid artifacts"
  - We of course will have numerical error, but let's avoid obvious patterns in the error

1D masses and springs

- Look at a homogeneous elastic rod, length 1, linear density ρ
- Parameterize by p (x(p)=p in rest state)
- Split up into intervals/springs
  - $0 = p_0 < p_1 < \ldots < p_n = 1$
  - Mass $m_i=\rho(p_{i+1}-p_{i-1})/2$ (+ special cases for ends)
  - Spring i+1/2 has rest length

    $$L_{i+1/2} = p_{i+1} - p_i$$

    and force

$$f_{i+1/2} = k_{i+1/2}\frac{x_{i+1} - x_i - L_{i+1/2}}{L_{i+1/2}}$$

Figuring out spring constants

- So net force on i is

$$F_i = k_{i+1/2}\frac{x_{i+1} - x_i - L_{i+1/2}}{L_{i+1/2}} - k_{i-1/2}\frac{x_i - x_{i-1} - L_{i-1/2}}{L_{i-1/2}}$$
$$= k_{i+1/2}\left(\frac{x_{i+1} - x_i}{p_{i+1} - p_i} - 1\right) - k_{i-1/2}\left(\frac{x_i - x_{i-1}}{p_i - p_{i-1}} - 1\right)$$

- We want mesh-independent response (roughly), e.g. for static equilibrium
  - Rod stretched the same everywhere: $x_i=\alpha p_i$
  - Then net force on each node should be zero (add in constraint force at ends…)

Young's modulus

- So each spring should have the same k
  - Note we divided by the rest length
  - Some people don't, so they have to make their constant scale with rest length
- The constant k is a material property (doesn't depend on our discretization) called the Young's modulus
  - Often written as E
- The one-dimensional Young's modulus is simply force per percentage deformation

The continuum limit

- Imagine Δp (or Δx) going to zero
  - Eventually can represent any kind of deformation
  - [note force and mass go to zero too]

$$\ddot{x}(p) = \frac{1}{\rho}\frac{\partial}{\partial p}\left(E(p)\left(\frac{\partial}{\partial a}x(p) - 1\right)\right)$$

  - If density and Young's modulus constant,

$$\frac{\partial^2 x}{\partial t^2} = \frac{E}{\rho}\frac{\partial^2 x}{\partial p^2}$$

Sound waves

- Try solution $x(p,t)=x_0(p-ct)$
- And $x(p,t)=x_0(p+ct)$
- So speed of "sound" in rod is $\sqrt{\frac{E}{\rho}}$
- Courant-Friedrichs-Levy (CFL) condition:
  - Numerical methods only will work if information transmitted numerically at least as fast as in reality (here: the speed of sound)
  - Usually the same as stability limit for good explicit methods [what are the eigenvalues here]
  - Implicit methods transmit information infinitely fast