## Collision Detection for Polygonal Objects

## given global location  and attitude  for all bodies at time t

## given  for all vertices on all bodies (constants)

## polygons must not be self-crossing

## polygons must not have holes

**Bounding Circle - quick check**























before the simulation, find maximum radius  over all vertices for each body (one constant scalar value per body that may be computed a priori)

at each time t, compute center distance  between all pairs of bodies

if  then no collision can occur between bodies i and j

if  then bodies i and j are candidates for collision detection

does not require calculation of ****** for any vertices on any bodies

works best for centroidal origins and objects with low aspect ratios















**Axis Aligned Bounding Box (AABB) - quick check**

at each time t, calculate ****** for all vertices on all bodies

find AABB for each body 

test between all pairs of bodies



if true, then AABBs intersect and bodies i and j are candidates for collision detection

if false, then AABBs do not intersect and collision cannot occur between bodies i and j

works best for objects with low aspect ratios













**Point in Polygon**

must check if any vertices on body j are inside body i AND check if any vertices on body i are inside body j

computational complexity O( ni x nj ) for convex objects and O( 2 ni x nj ) for ray casting

















## point in polygon does not always work for thin bodies (may need edge intersection)

**Point in Polygon – Convex Objects**

vertices must be ordered CW around polygon

at each time t, calculate ****** for all points on all bodies that are candidates for collision

select one point ****** on body j and test if it is to the right or to the left of edge ****** on body i

external normal  to the left of current edge

if , the point is to the left of the edge, is outside the polygon and the search across edges may be terminated for that point

if , the point is on the edge and may be inside the polygon

if , the point is to the right of the edge and may be inside the polygon

expand 

point ****** must be to the right of ALL edges to be inside body i

computational complexity O( ni x nj /2 ) because may terminate search early

does not work for bodies with concavities or holes









****











**Point in Polygon – Ray Casting**

at each time t, calculate ****** for all points on all bodies that are candidates for collision

select one point ****** on body j and test if it is inside body i

cast an infinite ray in any direction from ****** and compute intersections with all edges on body i

an infinite ray to the right from ******intersects edge  if 



if there are an even number of intersections, ****** is not inside body i

if there are an odd number of intersections, ****** is inside body i

computational complexity O( ni x nj )

works for concavities, holes and self-crossing and is independent of CW versus CCW boundary

special case when ****** is on an edge or vertex of body i

algorithm used by MATLAB “inpolygon”

**2**

**3**

**4**

**2**

**4**

**3**

**2**

**1**

**0**

% t\_inpolygon.m - test inpolygon

% HJSIII, 11.4.13

clear

% vertices for body i - bounded 0 to 1

ni = 5;

ri = rand(2,ni);

ri = [ ri ri(:,1) ]; % close the boundary

% points for j

nj = 500;

rj = rand(2,nj);

% which are inside?

% points j vertices i

in = inpolygon( rj(1,:),rj(2,:), ri(1,:),ri(2,:) );

figure( 1 )

 clf

 plot( ri(1,:),ri(2,:),'b', rj(1,in),rj(2,in),'.r', rj(1,~in),rj(2,~in),'.g' )

% bottom of t\_inpolygon



## Edge Intersection

## must check every edge on body i for intersection with every edge on body j

##













 intersect if  and 







if denominator = 0, segments are parallel

must check if they are coincident 

expand 

computational complexity O( ni x nj )

edge intersection does not always work if one body is completely inside the other (may need point in polygon)

% t\_polyxpoly - test polyxpoly

% HJSIII, 18.04.30

clear

% rectangle - CW closed curve

xbox = [ 3 3 13 13 3 ];

ybox = [ 2 8 8 2 2 ];

% two-part polyline - open curve with gap

xline = [ 0 6 4 8 8 10 14 10 14 NaN 4 4 6 9 15 ];

yline = [ 4 6 10 11 7 6 10 10 6 NaN 0 3 4 3 6 ];

% find edge intersections

[ xi, yi, ii ] = polyxpoly( xline, yline, xbox, ybox );

% list results

fprintf( 'line edge box edge\n' )

for i = 1 : length(ii),

 fprintf( '%10d', ii(i,1) )

 fprintf( '%10d\n', ii(i,2) )

end

% show results

figure( 1 )

 clf

 plot( xline,yline,'r', xbox,ybox,'g', xi,yi,'bo' )

% bottom of t\_polyxpoly

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

>> t\_polyxpoly

 line edge box edge

 1 1

 2 2

 4 2

 6 2

 8 2

 8 3

 14 3

 11 4



**Separating Axis Theorem (SAT)**

vertices must be ordered CW around polygon

before the simulation, select one edge  on body i and compute external normal 

compute the projection  for all points  onto the normal and store the minimum value  (one scalar constant per edge that may be computed a priori)



at each time t, calculate ****** for all points on all bodies that are candidates for collision

select edge ****** on body i and compute external normal  called the separating axis

select point ****** on body j and compute its projection  onto the separating axis



compute the projection  onto the separating axis for all other points on body j and store the maximum  and minimum  values

check for overlap of projections onto the separating axis



if false, the projections do not overlap, there can be no collision and the test may be terminated

if true, the projections overlap and must continue checking all other edges ******

does not work for bodies with concavities or holes

very similar to convex body point in polygon however do not need to check (i in j) and (j in i)

computational complexity O( ni x nj / 2) because may terminate search early AND do not need to check body i against body j

****





separating

axis





piMIN

pjMAX

**Classification of Contacts**

pjMIN

i

j

i

j

i

j

i

j

**vertex-edge**

**edge-edge**

**multiple vertex-edge**

**multiple edge-edge**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Body iverticesin j | Body iedgeintersections | Body jverticesin i | Body jedgeintersections | Contact points |
| vertex-edge | 0 | 1 | 1 | 2 | 3 |
| edge-edge | 1 | 2 | 1 | 2 | 4 |
| multiple vertex-edge | 0 | 1 | ≥2 | 2 | 5 |
| multiple edge-edge | 1 | 2 | ≥2 | 2 | 5 |

 **Interpolating Time of Collision (vertex-edge)**







given candidate vertex  on body j that collides with body i at time t

given candidate edge  on body i that has two edge intersections with body j at time t

must know position and velocity for all three points at time t-h (time step h)

**Constant velocity predictor**

predict position of all three points at local timeusing vales from t-h







when vertex  intersects edge  

substituting position equations and collecting similar terms











must check  and it must not be imaginary

if not true, then that vertex did not actually collide with that edge







relative motion of body j with respect to body i

**Constant acceleration predictor**

predict position of all three points at local time measured from t-h







substituting position equations and collecting similar terms









 (same as b above)

 (same as c above)

use root finding algorithm for  (e.g. MATLAB “roots”)

must check  and it must not be imaginary

**Three-dimensional Collision Detection**

**Quick checks**

Bounding spheres and AABB both work well

**Point in polyhedron – convex bodies**

at each time t, calculate ****** for all points on all bodies that are candidates for collision

select one point ****** on body j and test if it is outside or inside a facet on body i defined by external normal  and any vertex ****** on the facet

if , the point is outside the polyhedron and the search across facets may be terminated for that point

if , the point is on the facet and may be inside the polyhedron

if , the point is inside the facet and may be inside the polyhedron

point ****** must be inside of ALL facets to be inside body i

## point in polygon does not always work for thin bodies (may need edge intersection)

**Point in polyhedron – ray casting**

at each time t, calculate ****** for all points on all bodies that are candidates for collision

select one point ****** on body j and test if it is inside body i

cast an infinite ray in any direction from ****** and compute intersections with all facets on body i

if there are an even number of intersections, ****** is not inside body i

if there are an odd number of intersections, ****** is inside body i

works for concavities, holes and self-crossing and is independent of CW versus CCW boundary

ray-facet intersection is similar to edge-facet intersection described below

## point in polygon does not always work for thin bodies (may need edge intersection)

**Triangular facet intersection (MATLAB code available)**

vertices for facet on body i 

vertices for facet on body j 

edges on body i 

edges on body j 

unit normal for facet i 

unit normal for facet j 

unit vector for line of intersection of planes through facets 

equation for plane through facet i 

equation for ray on facet j along edge Q-P from P 

directed distance from P on body j to plane for facet i 

three directed distances on facet i that pierce plane for facet j

  

three directed distances on facet j that pierce plane for facet i

  

only use directed distances that are bounded 

there must be two valid directed distances on body i and two on body j for intersection

otherwise there is no intersection

use the ray equation to find the two piercing points for facet i and the two for body j



directed distances from first piercing point along line of intersection



sort piercing points by directed distances  along line of intersection

if first two piercing points belong to the same body, there is no intersection

otherwise the intersection segment runs from the second to third sorted piercing points

**General facet intersection**

## check if edges for a given facet on body j intersect facet on body i

establish local coordinate frame with origin at the first vertex for facet on i, local x axis along first edge and local y defined by second edge (CCW vertex sequence around facet i)

local z axis will be parallel to external normal

transform vertices for facet i and facet j into local coordinates and perform all following computations in local coordinates

test where edge  intersects facet i

use  and check 

if false, that edge does not intersect the facet and proceed to check next edge on j for same facet i

if true that edge intersects the plane of the facet at 

use planar point in polygon test if x,y components of  are inside x,y components of facet i

if false, the edge intersects the plane of the facet outside its boundary

if true, the edge intersects the facet inside its boundary

## must also check if edges for same facet on body i intersect same facet on body j

facet intersection does not always work if one body is completely inside the other (may need point in polygon)

## Handling Collision

Physical simulators differ in the way they react to a collision. Some use the softness of the material to calculate a force, which will resolve the collision in the following time steps like it is in reality. Due to the low softness of some materials this is very CPU intensive.

Other simulators estimate the time of collision by interpolation, roll back the simulation, and calculate the exact time of collision. This may require several iterations.

After an inelastic collision, special states of sliding are often imposed. For example, an open pin-in-slot constraint is added for a vertex-edge intersection to force that vertex to slide on the edge rather than penetrate it. The [Open Dynamics Engine](http://en.wikipedia.org/wiki/Open_Dynamics_Engine) uses this approach.