Building a physics engine - part 2: narrow phase of collision detection

Dr. Giuseppe Maggiore

NHTV University of Applied Sciences
Breda, Netherlands

Table of contents

Narrow phase

Narrow phase

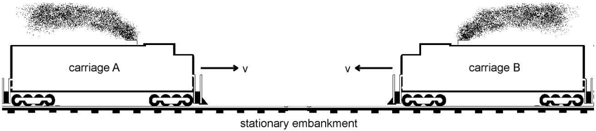
- Find intersections for each pair of rigid bodies
  - Whether they intersect
  - When they intersect
  - Where they intersect
- Acceptable precision
- Very high performance

Narrow phase

Response

- At all points of contact the *relative velocity* projected along the normal must be non-negative
- No pair of bodies in contact is getting closer
  - This would mean that they are penetrating each other

Relative Velocity

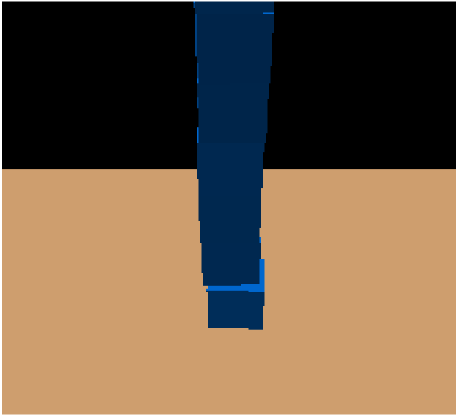


Narrow phase

Response

- *Relatively* easy to handle between pairs of bodies
- Multiple bodies in contact make it much harder

Relative Velocity

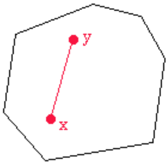


Convex polyhedra

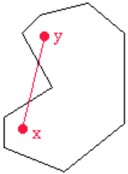
Convex polyhedra

- Also known as *polytopes*
- $\forall P, Q \in C . \forall \alpha \in [0..1] . \text{lerp}(\alpha, P, Q) \in C$
- Complex bodies can be treated as multiple polytopes with distance constraints

Convex polygon

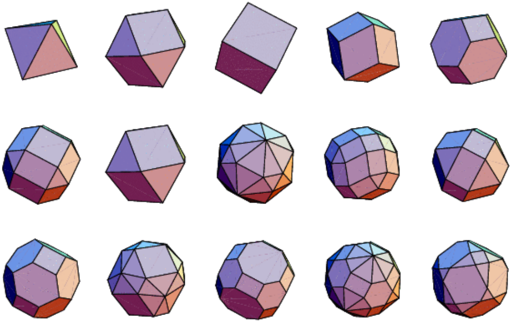


A convex polygon



A non–convex polygon

Convex polyhedra

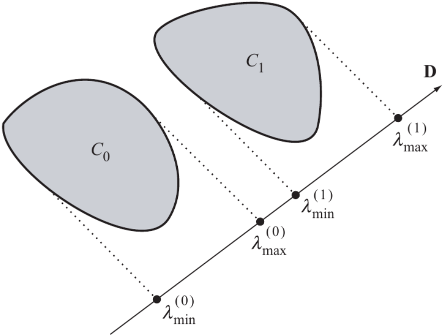


Separating axis

Method of separating axis

- We use a method that works in 2D and 3D
  - We look for an axis/plane that separates the bodies

Separating axis

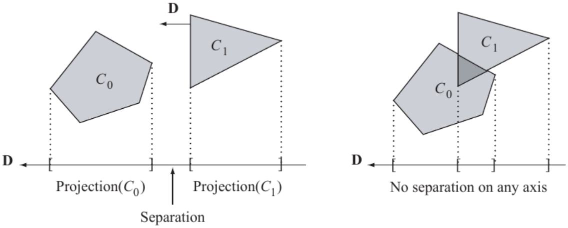


Separating axis

Method of separating axis

- Given a (not necessarily unit-length) direction D and two polytopes C_0 and C_1
- We project them over the direction D: $I_i = [\lambda^i_{min}, \lambda^i_{max}] = [\min_{x \in C_i}\{D \cdot (X - O)\}, \max_{x \in C_i}\{D \cdot (X - O)\}]$
- There is no collision if $\exists D . \lambda^0_{min} > \lambda^1_{max} \vee \lambda^1_{min} > \lambda^0_{max}$

Separating axis

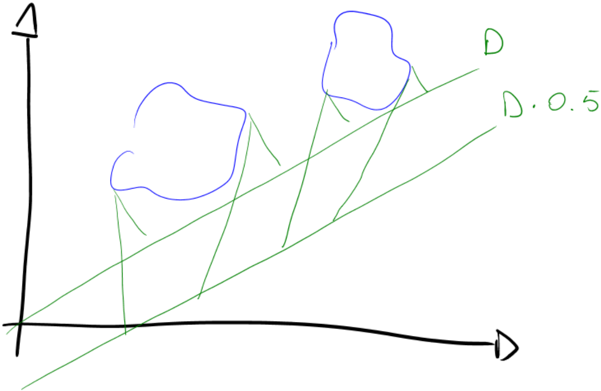


Separating axis

Method of separating axis

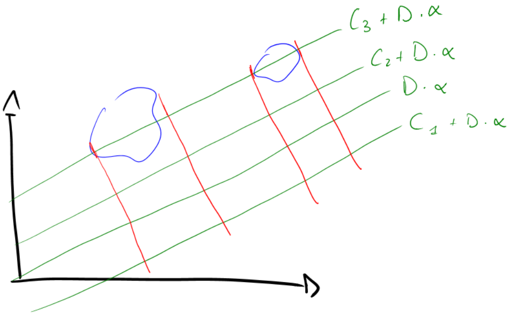
- D may be multiplied by a (non-zero) constant
- The projection is scaled uniformly, but no contact still translates into a separated projection
- The translation of a separating axis remains a separating axis, so we only deal with lines that go through the origin

Scale of separating axis



Translation of separating axis

29 August 2013 11:36



Separating axis

Method of separating axis

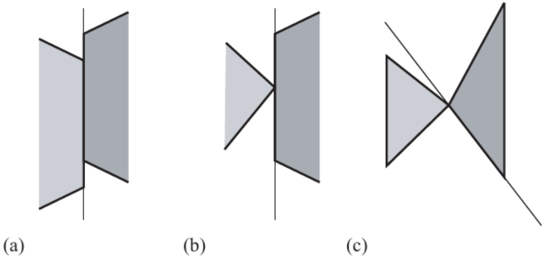
- We now consider C_j polytopes, each with
  - P_i^j vertices, *ordered counter-clockwise* and with wrapping indices
  - $E_i^j = P_{i+1}^j - P_i^j$ edges, *ordered counter-clockwise*
  - N_i^j normals, perpendicular to the edges
- Similarly we store the triangles in 3D

Separating axis

Method of separating axis

- There are infinite candidate directions D
- Fortunately we need only consider a finite set
- In 2D, the normals of each polygon

Separating axis



Separating axis algorithms

Algorithms

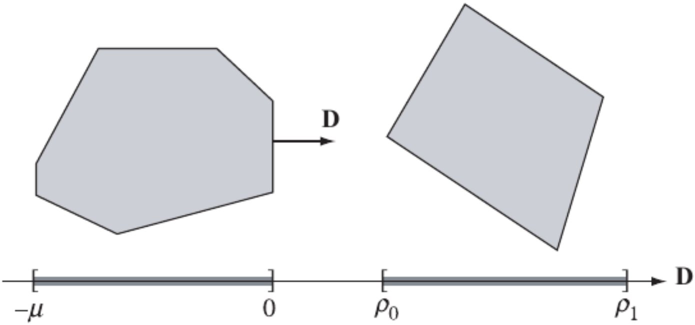
- Naïve algorithm
- For each normal:
  - Project all vertices of both polytopes
  - Compute the intervals
  - Check for intersection

Separating axis algorithms

Algorithms

- Smarter algorithm
  - For each body C_j
  - For each normal P_i^j, N_i^j
  - Check if the closest vertex P_k^l of the other body C_l is too close to the edge
  - $(P_k^l - P_i^j) \cdot N_i^j > \epsilon$

Separation against face



Pseudo-code

```
bool TestIntersection (ConvexPolygon C0, ConvexPolygon
    C1) {
  for (i0 = C0.GetN()-1, i1 = 0; i1 < C0.GetN(); i0 =
      i1++) {
    P = C0.GetVertex(i1);
    D = C0.GetNormal(i0);
    if (WhichSide(C1,P,D) > 0) {
      return false;
    }
  }
```

Pseudo-code

```
  for (i0 = C1.GetN()-1, i1 = 0; i1 < C1.GetN(); i0 =
      i1++) {
    P = C1.GetVertex(i1);
    D = C1.GetNormal(i0);
    if (WhichSide(C0,P,D) > 0) {
      return false;
    }
  }
  return true;
}
```

Pseudo-code

```
int WhichSide (ConvexPolygon C, Point P, Vector D) {
  posCount = 0;
  negCount = 0;
  zeroCount = 0;
  for (i = 0; i < C.GetN(); ++i) {
    t = Dot(D,C.GetVertex(i) - P);
    if (t > 0) { posCount++; }
    else if (t < 0) { negCount++; }
    else { zeroCount++; }
    if ((posCount > 0 and negCount > 0) or zeroCount >
         0) { return 0; }
  }
  return posCount ? 1 : -1;
}
```

Separating axis algorithms

Optimizations

- Bisection on the sorted vertices
- Find the vertex closest to an edge faster

Separating axis algorithms

Optimizations

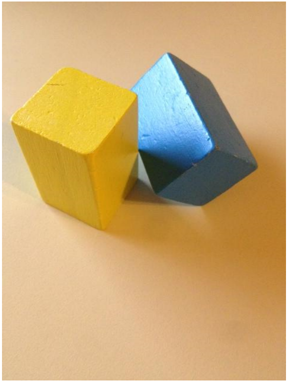
- 2D and 3D: BSP, Gauss map, hash table of the vertices sorted w.r.t. their direction
- Find the vertex closest to an edge much faster
- We will *maybe* see this algorithm, depending on how fast the course goes

Separating axis in 3D

Candidate axes

- In 2D the candidate axes are just the edge normals
- In 3D the face normals are not enough
- Edge-to-edge collisions are not covered

Separating axis

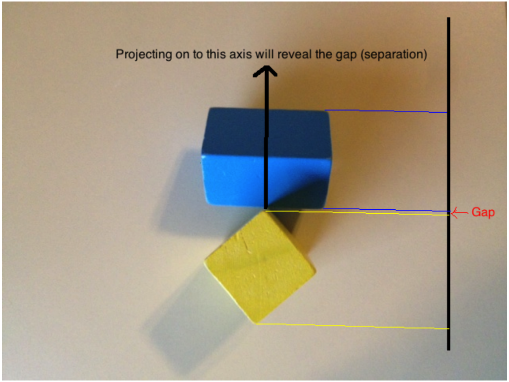


Separating axis in 3D

Candidate axes

- Also consider edge-to-edge cross products
- Exactly the same algorithm, but with more potential axes

Separating axis



Pseudo-code

```
bool TestIntersection (ConvexPolyhedron C0,
    ConvexPolyhedron C1) {
  for (i = 0; i < C0.GetFCount(); ++i) {
    D = C0.GetNormal(i);
    ComputeInterval(C0,D,min0,max0);
    ComputeInterval(C1,D,min1,max1);
    if (max1 < min0 || max0 < min1) { return false; }
  }
```

Pseudo-code

```
for (j = 0; j < C1.GetFCount(); ++j) {
  D = C1.GetNormal(j);
  ComputeInterval(C0,D,min0,max0);
  ComputeInterval(C1,D,min1,max1);
  if (max1 < min0 || max0 < min1) { return false; }
}
```

Pseudo-code

```
  for (i = 0; i < C0.GetECount(); ++i) {
    for (j = 0; j < C1.GetECount(); ++j) {
      D = Cross(C0.GetEdge(i),C1.Edge(j));
      ComputeInterval(C0,D,min0,max0);
      ComputeInterval(C1,D,min1,max1);
      if (max1 < min0 || max0 < min1){ return false; }
    }
  }
  return true;
}
```

Pseudo-code

```
void ComputeInterval (ConvexPolyhedron C, Vector D,
    double& min, double& max) {
  min = Dot(D,C.GetVertex(0));
  max = min;
  for (i = 1; i < C.GetVCount(); ++i) {
    value = Dot(D,C.GetVertex(i));
    if (value < min) {
      min = value;
    } else {
      max = value;
    }
  }
}
```

Moving objects

Moving objects

- Objects are usually moving in a game :)
- We must check for future intersections
- With some simplifying assumptions
  - Rotations can be ignored (phew!)
  - Interpenetration may happen a bit

Moving objects

Moving objects

- Use the *moving projection* on an axis
- Consider the *relative velocity* $V = V_2 - V_1$
- Speed of projection along D is $\sigma = V \cdot D$ when $|D| = 1$
- The distance of the minimum point must be bigger than σ
- $\Delta x = \sigma \Delta t \ \Delta t \frac{\Delta x}{d} =$ time of collision
- When the time of collision is outside the time of the current frame, then there is no collision

Moving objects

```
bool TestIntersection (ConvexPolygon C0, Vector V0,
    ConvexPolygon C1, Vector V1, double tmax, double&
    tfirst, double& tlast) {
  V = V1 - V0;
  tfirst = 0;
  tlast = INFINITY;
```

Pseudo-code

```
for (i0 = C0.GetN() - 1, i1 = 0; i1 < C0.GetN(); i0
    = i1++) {
  D = C0.GetNormal(i0);
  ComputeInterval(C0,D,min0,max0);
  ComputeInterval(C1,D,min1,max1);
  speed = Dot(D,V);
  if (NoIntersect(tmax,speed,min0,max0,min1,max1,
      tfirst, tlast)) { return false; }
}
```

Pseudo-code

```
  for (i0 = C1.N - 1, i1 = 0; i1 < C1.N; i0 = i1++) {
    D = C1.GetNormal(i0);
    ComputeInterval(C0,D,min0,max0);
    ComputeInterval(C1,D,min1,max1);
    speed = Dot(D,V);
    if (NoIntersect(tmax,speed,min0,max0,min1,max1,
        tfirst, tlast)) { return false; }
  }
  return true;
}
```

Moving objects

Moving objects

- if the polygons intersect at a first time t_{first}, then there is a separating axis $\forall t. t < t_{\text{first}}$
- if the polygons intersect at a last time t_{last}, then there is a separating axis $\forall t. t > t_{\text{last}}$
- if for all direction $t_{\text{first}} < t_{\text{last}}$, then the polygons intersect at time t_{first} (check against Δt)
- if for all direction $t_{\text{first}} > t_{\text{last}}$, then the polygons do not intersect (all axis must intersect at the same time!)

Pseudo-code

```
bool NoIntersect (double tmax, double speed, double
    min0, double max0, double min1, double max1,
    double& tfirst, double& tlast) {
  if (max1 < min0) {
    if (speed <= 0) { return true; }
    t = (min0 - max1)/speed;
    if (t > tfirst) { tfirst = t; }
    if (tfirst > tmax) { return true; }
    t = (max0 - min1)/speed;
    if (t < tlast) { tlast = t; }
    if (tfirst > tlast) { return true; }
```

Pseudo-code

```
} else if ( max0 < min1 ) {
  if (speed >= 0) { return true; }
  t = (max0 - min1)/speed;
  if (t > tfirst) { tfirst = t; }
  if (tfirst > tmax) { return true; }
  t = (min0 - max1)/speed;
  if (t < tlast) { tlast = t; }
  if (tfirst > tlast) { return true; }
```

Pseudo-code

```
  } else {
    if (speed > 0) {
      t = (max0 - min1)/speed;
      if (t < tlast) { tlast = t; }
      if (tfirst > tlast) { return true; }
    } else if (speed < 0) {
      t = (min0 - max1)/speed;
      if (t < tlast) { tlast = t; }
      if (tfirst > tlast) { return true; }
    }
  }
  return false;
}
```

Finding intersections

Contact manifold

- The collision response system needs the intersections
- So far we have built `TestIntersection`
- Huge numbers of possible algorithms for this (**GJK** is the most diffuse)
- We now *sketch* a description of one intuitive algorithm

Finding intersections

Finding intersections

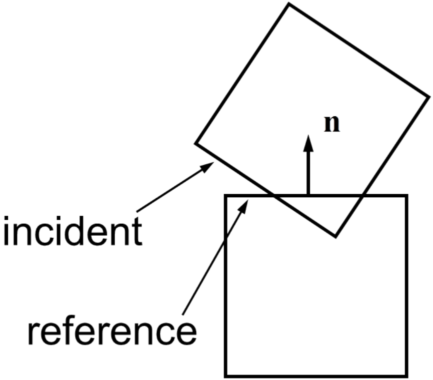
- When the intersection is found at time T, before returning
- We move the bodies forward with the respective velocities
- We compute and return (an approximation) of the contact set
- May be face-to-vertex or edge-to-edge

Finding intersections

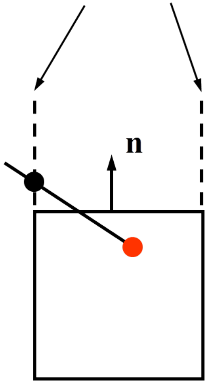
Finding intersections for SAT failures

- Identify *reference face* ($D = N$) and *incident face* (most anti-parallel to D)
- Clip incident face against edges of reference face
- Keep all resulting vertices *below or on* the reference face

Reference face



Reference face sideways



Finding intersections

Edge-to-edge SAT failures

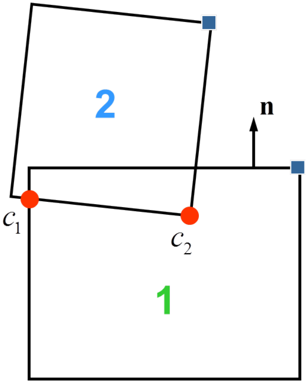
- Just intersect the edges

Finding intersections

Contact manifold

- Two vertices are enough in 2D
- Three vertices are enough in 3D
- Too many more vertices do not improve stability (a few might)
- We always choose the ones penetrating the most
- Deterministic as much as possible (contact caching)

Contact manifold



Finding intersections

Clipping and intersections

- Sutherland-Hogman algorithm for clipping
- Edge-to-edge intersection

Finding intersections

Clipping faces

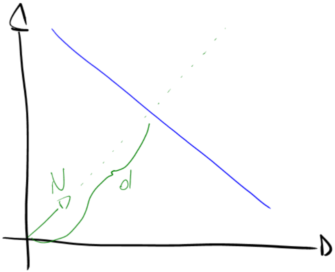
- Project planes from the edges of the reference face
- Vertices of the incident face *inside* all planes are contact points
- Reference plane with incident edge intersections are more contact points
- Keep a subset of the points: (most inside the reference face)

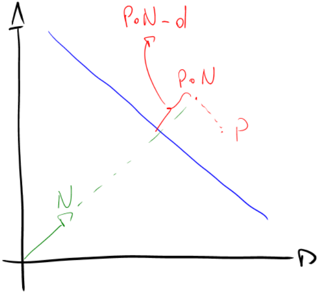
Finding intersections

Clipping a vertex

- Given a plane $\langle N, d\rangle$ where N is the normal of the plane and d is the minimum distance of the plane from the origin
- We determine the side of the plane on which a vertex V lies with the sign of $P \cdot N - d$

Clipping





Finding intersections

Clipping a vertex

- All signs of plane-vertex distance must be the same
- The planes must all be facing inward (or outward)

Finding intersections

Clipping an edge

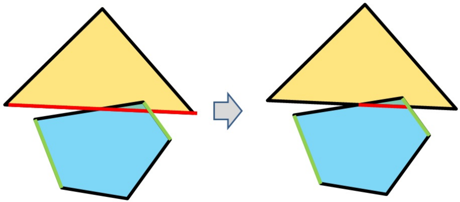
- Given a plane $\langle V \cdot N = d \rangle$ and an edge $V = A + \alpha B$ $(0 \leq \alpha \leq 1)$
- We put these equations together: $(A + \alpha B) \cdot N = d$ and solve for $\alpha = \frac{d - A \cdot N}{B \cdot N}$

Finding intersections

Clipping an edge

- $\alpha = \frac{d - A \cdot N}{B \cdot N}$
- Watch for lack of solutions
  - If $|B \cdot N| \leq \epsilon$ then edge and plane are parallel
  - No intersection possible

Clipping

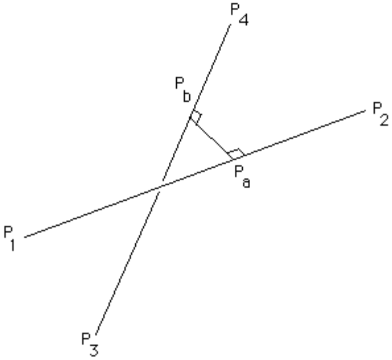


Finding intersections

Edge-to-edge intersections

- We consider two edges: $P_1 + \alpha D_1$ and $P_2 + \beta D_2$
- Intersection in 3D is rather rare
- We look for the shortest connecting axis: $P_a = P_1 + \mu_a D_1$ and $P_b = P_2 + \mu_b D_2$

Clipping



Finding intersections

Edge-to-edge intersections

- The connecting axis is perpendicular to the both edges, so

$$\begin{cases} (P_a - P_b) \cdot D_1 & = & 0 \\ (P_a - P_b) \cdot D_2 & = & 0 \end{cases}$$

Finding intersections

Edge-to-edge intersections

- The connecting axis is perpendicular to the both edges, so
$$\begin{cases}(P_a - P_b) \cdot D_1 & = & 0 \\ (P_a - P_b) \cdot D_2 & = & 0\end{cases}$$
- Expanding P_a and P_b results in
$$\begin{cases}(P_1 + \mu_a D_1 - P_2 - \mu_b D_2) \cdot D_1 & = & 0 \\ (P_1 + \mu_a D_1 - P_2 - \mu_b D_2) \cdot D_2 & = & 0\end{cases}$$

Finding intersections

Edge-to-edge intersections

- This is just a system with two unknowns and two equations; the dot products turn everything into numbers:
$$\begin{cases} (P_1 + \mu_a D_1 - P_2 - \mu_b D_2) \cdot D_1 & = & 0 \\ (P_1 + \mu_a D_1 - P_2 - \mu_b D_2) \cdot D_2 & = & 0 \end{cases}$$
- The full (tedious) derivation of the solutions is shown in `http://paulbourke.net/geometry/pointlineplane/`

Assignment

Assignment

- Before the end of next week
- Group-work archive/video on Natschool or uploaded somewhere else and linked in your report
- Individual report by each of you on Natschool
- Build a narrow phase collision detector that can compute the contact manifold

That's it

Thank you!