Physics based animation

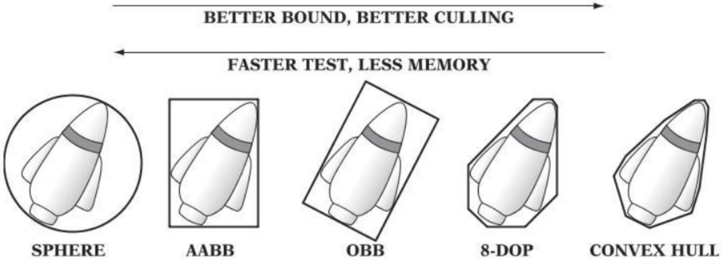
Lecture 10 - Collision detection - Part 3

Grégory Leplâtre

g.leplatre@napier.ac.uk, room D32
School of Computing
Edinburgh Napier University

Semester 1 - 2016/2017

Reminder


BETTER BOUND, BETTER CULLING
FASTER TEST, LESS MEMORY
SPHERE
AABB
OBB
8-DOP
CONVEX HULL

Plan

- Closest point computations (Monday)

Plan

- Closest point computations (Monday)
- Primitive collision tests (Wednesday)

Outline

Closest point computations

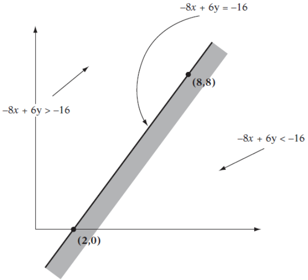
- Plane to point
- Point to segment
- Point to AABB
- Point to OBB
- Point to triangle
- Point to tetahedron
- Point to convex polyhedron

- Closest points of two lines
- Closest points of two segments
- Closest points of segment and triangle
- Closest points of two triangles

Reminder: Planes, hyperplanes and halfspaces

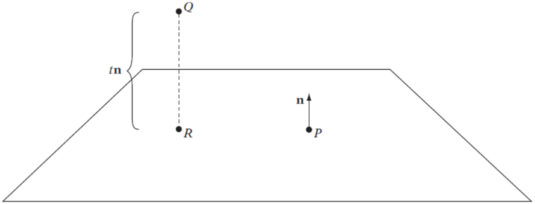
- **Plane equation**:
  - Three points forming a triangle:
    $$P(\boldsymbol{u}, \boldsymbol{v}) = A + \boldsymbol{u}(B - A) + \boldsymbol{v}(C - A)$$
  - A normal and a point on a plane
    $$\boldsymbol{n} \bullet (X - P) = 0$$
  - A normal and a distance from the origin
    $$\boldsymbol{n} \bullet X = d$$

Reminder: Planes, hyperplanes and halfspaces

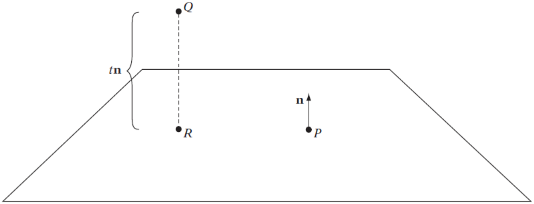


- **Hyperplane**:
  - planes with one fewer dimension than the space they are in.
  - In 3D, planes are hyperplanes
  - in 2D, lines are hyperplanes
- **halfspace**
  - A hyperplane divides space into two halfspaces

Closest point to plane

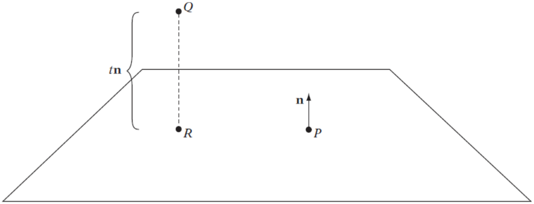


Closest point to plane



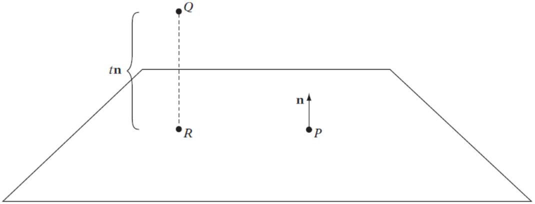
Plane equation: $\boldsymbol{n} \bullet X = d$

Closest point to plane



Plane equation: $\boldsymbol{n} \bullet X = d$
Point on plane: $R = Q - t\boldsymbol{n}$

Closest point to plane

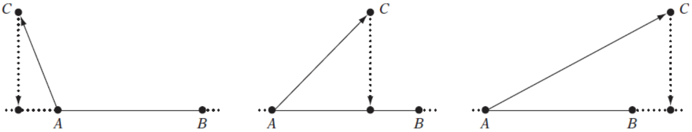


Plane equation: $\boldsymbol{n} \bullet X = d$
Point on plane: $R = Q - t\boldsymbol{n}$

$$t = \frac{\boldsymbol{n} \bullet Q - d}{\boldsymbol{n} \bullet \boldsymbol{n}}$$

Closest point to segment

Closest point to segment

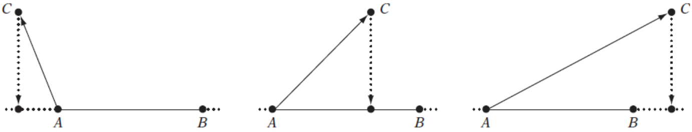


Point on plane: $t = \dfrac{(C - A) \bullet (B - A)}{(B - A) \bullet (B - A)}$

if t < 0 then t = 0
if t > 1 then t = 1

$$D = A + t * (B - A)$$

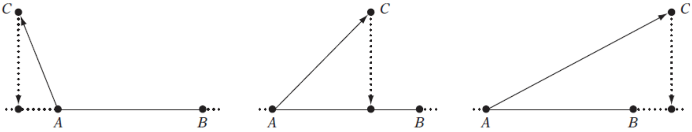
Closest point to segment



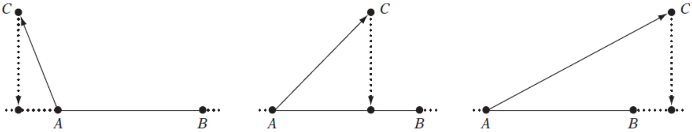

```
void closestPtPointSegment(Point c, Point b, Point a,
    Point &d, float &t){
Vector ab = b - a;
// project c onto ab, computing position d(t)=a+t*(b-a)
t = dot(c - a, ab)/dot(ab, ab);
if (t < 0.0f) t = 0.0f;
if (t > 1.0f) t = 1.0f;
// compute projected position
d = a + t * ab;
```

distance from point to segment

distance from point to segment

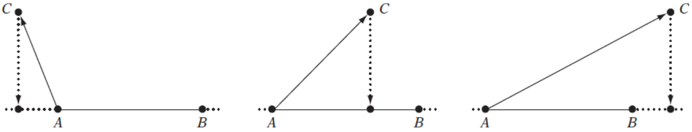


distance from point to segment



$e = (C - A) \bullet (B - A)$
if e <= 0 then $d^2 = (C - A) \bullet (C - A)$
$f = (B - A) \bullet (B - A)$
if e >= f then $d^2 = (C - B) \bullet (C - B)$
else $d^2 = (C - A) \bullet (C - A) - e * e/f$

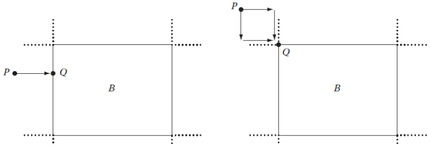
distance from point to segment



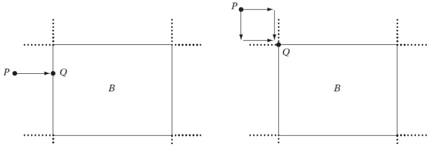

```
float sqDistPointSegment(Point c, Point b, Point a){
Vector ab = b - a, ac = c - a,  bc = c - b;
float e = dot(ac, ab);
if (e <= 0.0f) return dot(ac, ac);
float f = dot(ab, ab);
if (e >= f) return dot(bc, bc);
// else, c is projected inside segment
return dot(ac, ac) - e* e / f;
```

closest point on AABB to point

closest point on AABB to point



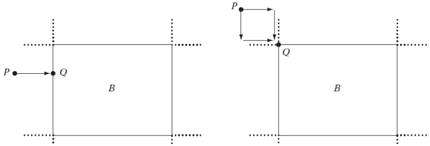
closest point on AABB to point



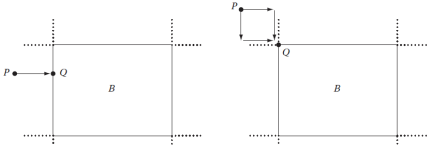

```
void closestPointAABB(Point p, AABB, b, Point &q){
// for each coordinate axis, if the point coordinate is
    outside box, clamp it to box
for (int i = 0; i < 3; i++){
float v = p[i];
if (v < b.min[i]) v = b.min[i];
if (v > b.max[i]) v = b.max[i];
q[i] = v;
}
```

distance from point to AABB

distance from point to AABB



distance from point to AABB



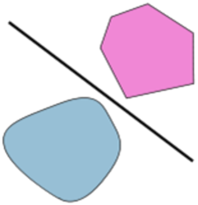

```
// return the squared distance betwen point and AABB
float sqDistPointAABB(Point p, AABB, b){
float sqDist = 0.0f;
// for each coordinate axis, count any excess distance outside box
for (int i = 0; i < 3; i++){
float v = p[i];
if (v < b.min[i]) sqDist += (b.min[i] - v) * (b.min[i] - v);
if (v > b.max[i]) sqDist += (v - b.max[i]) * (v - b.max[i]);
return sqDist
}
```

Outline

Basic primitive tests

- Separating-axis test
- Sphere against plane
- Box against plane
- Cone against plane
- Sphere against AABB
- Sphere against OBB
- Sphere against triangle
- Sphere against polygon
- AABB against polygon
- Triangle against triangle

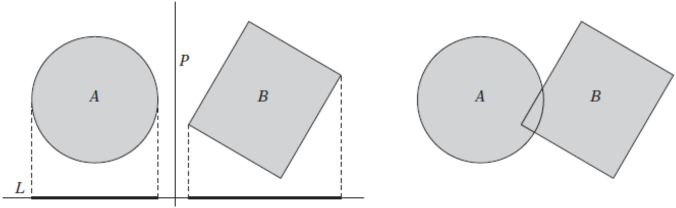
Separating axes



Separating plane theorem

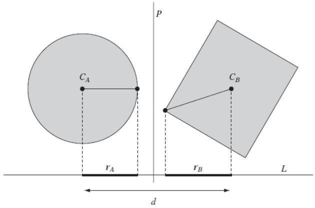
Two convex sets A and B are either intersecting or there exists a hyperplane P such that A is on one side of P and B is on the other side.

Separating axes



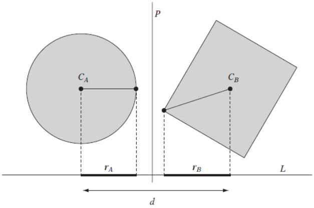
- A separating axis is a line which is perpendicular to some separating hyperplane
- A test for a separating axis is cheaper than separating plane.

Separating axes - symmetrical primitives



- For symmetrical primitives with a defined centre point (e.g. AABBs, OBBs, spheres, etc.), the centre point will always project onto the middle of the projection interval along the tested axis.

Separating axes - symmetrical primitives



- An efficient separation test of two symmetrical objects A and B is to compute the halfwidth, or radii, of their projection intervals and compare the sum of them against the distance between their centre projections.
- If the sum is less than the distance between the centre projections, the objects must be disjoint.

Separating axes - arbitrary primitives

- Possible intersections between two convex hulls:
  - face-face
  - face-edge
  - edge-edge
  - face-vertex
  - edge-vertex
  - vertex-vertex

Separating axes - arbitrary primitives

- Possible intersections between two convex hulls:
  - face-face
  - face-edge
  - edge-edge
  - face-vertex
  - edge-vertex
  - vertex-vertex
- Vertex tests can be considered as special cases of edge contacts.

Separating axes - arbitrary primitives

- Possible intersections between two convex hulls:
  - face-face
  - face-edge
  - edge-edge
  - face-vertex
  - edge-vertex
  - vertex-vertex
- Vertex tests can be considered as special cases of edge contacts.
- For edge-edge
  - **cross product** of the two edges is potential separating axis.

Separating axes - arbitrary primitives

- **General case** separation axes:
  - Axes parallel to face normals of object A
  - Axes parallel to face normals of object B
  - Axes parallel to cross products of all edges in A with all edges in B
- **How many tests**?

Separating axes - arbitrary primitives

- **General case** separation axes:
  - Axes parallel to face normals of object A
  - Axes parallel to face normals of object B
  - Axes parallel to cross products of all edges in A with all edges in B
- **How many tests**?
  - 2 OBBs?

Separating axes - arbitrary primitives

- **General case** separation axes:
  - Axes parallel to face normals of object A
  - Axes parallel to face normals of object B
  - Axes parallel to cross products of all edges in A with all edges in B
- **How many tests**?
  - 2 OBBs? $3 + 3 + 3^2$

Separating axes - arbitrary primitives

- **General case** separation axes:
  - Axes parallel to face normals of object A
  - Axes parallel to face normals of object B
  - Axes parallel to cross products of all edges in A with all edges in B
- **How many tests**?
  - 2 OBBs? $3 + 3 + 3^2$
  - 2 convex hulls?

Separating axes - arbitrary primitives

- **General case** separation axes:
  - Axes parallel to face normals of object A
  - Axes parallel to face normals of object B
  - Axes parallel to cross products of all edges in A with all edges in B
- **How many tests**?
  - 2 OBBs? $3 + 3 + 3^2$
  - 2 convex hulls? $F + E^2$

Separating axes - arbitrary primitives

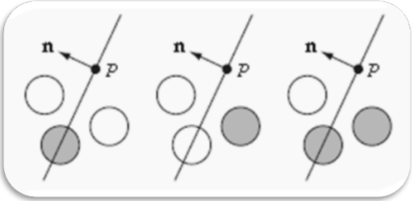
- **Algorithm**
  - As soon as a separating axis is found, exit routine
  - If none of the axes are separating axes, then the objects are intersecting

Separating axes - arbitrary primitives

- **Algorithm**
  - As soon as a separating axis is found, exit routine
  - If none of the axes are separating axes, then the objects are intersecting
- **Contact information**
  - Instead of exiting when a separating axis is detected, find out **all separating axes**
  - The axis with the least overlap can be used as the **contact normal**
  - The overlap can be used to estimate **penetration** along that axis

Outline

Sphere-plane intersection



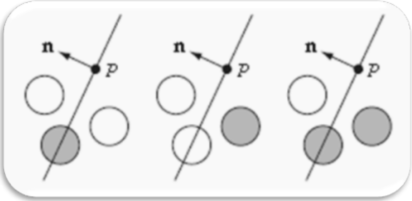
- Plane P defined by:

$$(\boldsymbol{n}.X) = d$$

(normalised *i.e.*, $|\boldsymbol{n}| = 1$)

- Sphere: Centre C and radius r

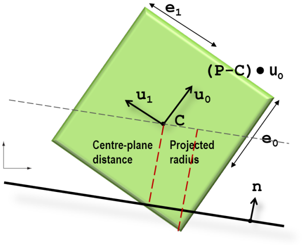
Sphere-plane intersection



- Test if sphere intersect with plane

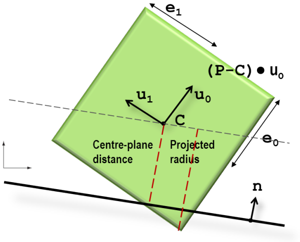
```
// determine whether sphere intersects with plane
bool testSpherePane(Sphere sphere, Plane plane){
   float dist = dot(sphere.centre, plane.normal) - plane.
    distance;
   return Abs(dist) <= sphere.radius;
}
```

Box-plane intersection



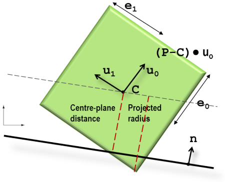
- Observation:
  - Box vertices are furthest points from box centre
  - they are also closest points to the plane

Box-plane intersection

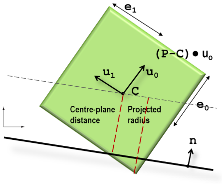


- Observation:
  - Box vertices are furthest points from box centre
  - they are also closest points to the plane
- Algorithm:
  - Find the largest projection on the plane normal of centre to vertex distances
  - If it is greater than the distance between the box centre to the plane, then there is an intersection

Box-plane intersection



Box-plane intersection



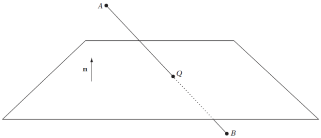

```
// determine whether box intersects with plane
bool testBoxPlane(OBB box, Plane plane){
    float radius =
        box.e[0] * abs( dot( plane.n, box.u[0] ) ) +
        box.e[1] * abs( dot( plane.n, box.u[1] ) ) +
        box.e[2] * abs( dot( plane.n, box.u[2] ) );

    float distance = dot( plane.n, box.c ) - plane.d;

    return abs(distance) <= radius;
}
```

Outline

line, ray and segment intersections

- segment against plane
- ray or segment against sphere
- ray or segment against box
- line against triangle
- line against quadrilateral
- ray or segment against triangle
- ray or segment against convex polyhedron

Ray/segment-plane



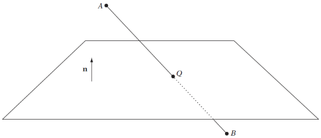
- Plane P defined by:

$$(\boldsymbol{n}.X) = d$$

- Segment:

$$S(t) = A + t(B - A), \textit{for} \quad 0 \leq t \leq 1$$

Ray/segment-plane



- Plane P defined by:

$$(\mathbf{n}.X) = d$$

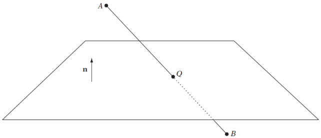
- Segment:

$$S(t) = A + t(B - A), for \quad 0 \leq t \leq 1$$

- Intersection point Q:

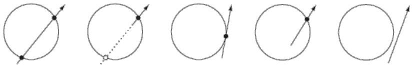
$$Q = A + \frac{(d - \mathbf{n} \bullet A)}{(\mathbf{n} \bullet (B - A))}(B - A)$$

Ray/segment-plane




```
bool intersectSegmentPlane(Point a, Point b, Plane plane,
     float &t, float &q){
// compute t value for the intersection between line ab
     and plane
Vector ab = b - a;
t = (plane.d - dot(plane.n, a)) / dot(plane.n, ab);

// If t in [0..1], compute intersection point
if (t >= 0.0f && t <= 1.0f){
q = a + t * ab;
return true;
}
// Else no intersection
return 0;
}
```

Ray/segment-sphere



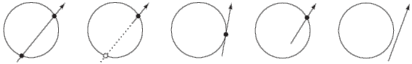
- Ray defined as:

$$R(t) = P + t\boldsymbol{d}, t \geq 0$$

Where:

  - P is the ray origin
  - $\boldsymbol{d}$ is the normalised direction vector
  - $0 \leq t \leq t_{max}$ for vector instead or ray

Ray/segment-sphere



- Ray defined as:

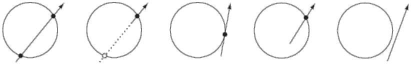
$$R(t) = P + t\boldsymbol{d}, t \geq 0$$

Where:

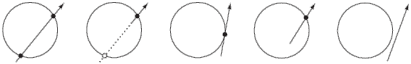
  - P is the ray origin
  - $\boldsymbol{d}$ is the normalised direction vector
  - $0 \leq t \leq t_{max}$ for vector instead or ray
- Sphere defined as:

$$(X - C) \bullet (X - C) = r^2$$

Ray/segment-sphere



Ray/segment-sphere



- Intersection points, solve quadratic equation:

$$(P + t\boldsymbol{d} - C) \bullet (P + t\boldsymbol{d} - C) = r^2$$

$$t^2 + 2(\boldsymbol{m} \bullet \boldsymbol{d})t + \boldsymbol{m} \bullet \boldsymbol{m} - r^2 = 0$$

Where:

- $\boldsymbol{m} = P - C$

Ray/segment-sphere

```
bool testRaySphere(Point p, Vector d, Sphere s, float &t,
      Point &q)
{
    Vector m = p - s.c;
    float b = dot(m, d);
    float c = dot(m, m) - s.r * s.r;
```

Ray/segment-sphere

```
bool testRaySphere(Point p, Vector d, Sphere s, float &t,
      Point &q)
{
    Vector m = p - s.c;
    float b = dot(m, d);
    float c = dot(m, m) - s.r * s.r;
    // exit if p outside sphere (c>0) and r pointing away
     from s (b>0)
    if (c > 0.0f && b > 0.0f) return false;
```

Ray/segment-sphere

```
bool testRaySphere(Point p, Vector d, Sphere s, float &t, Point &q)
{
    Vector m = p - s.c;
    float b = dot(m, d);
    float c = dot(m, m) - s.r * s.r;
    // exit if p outside sphere (c>0) and r pointing away from s (b>0)
    if (c > 0.0f && b > 0.0f) return false;
    // compute discriminant
    float discr = b*b - c;
```

Ray/segment-sphere

```
bool testRaySphere(Point p, Vector d, Sphere s, float &t,
     Point &q)
{
   Vector m = p - s.c;
   float b = dot(m, d);
   float c = dot(m, m) - s.r * s.r;
   // exit if p outside sphere (c>0) and r pointing away
    from s (b>0)
   if (c > 0.0f && b > 0.0f) return false;
   // compute discriminant
   float discr = b*b - c;
   //negative discriminant => ray misses sphere
   if (discr < 0.0f) return false;
```

Ray/segment-sphere

```
bool testRaySphere(Point p, Vector d, Sphere s, float &t,
     Point &q)
{
   Vector m = p - s.c;
   float b = dot(m, d);
   float c = dot(m, m) - s.r * s.r;
   // exit if p outside sphere (c>0) and r pointing away
    from s (b>0)
   if (c > 0.0f && b > 0.0f) return false;
   // compute discriminant
   float discr = b*b - c;
   //negative discriminant => ray misses sphere
   if (discr < 0.0f) return false;
   // compute smallest root
   t = -b - sqrt(discr);
```

Ray/segment-sphere

```
bool testRaySphere(Point p, Vector d, Sphere s, float &t,
     Point &q)
{
   Vector m = p - s.c;
   float b = dot(m, d);
   float c = dot(m, m) - s.r * s.r;
   // exit if p outside sphere (c>0) and r pointing away
    from s (b>0)
   if (c > 0.0f && b > 0.0f) return false;
   // compute discriminant
   float discr = b*b - c;
   //negative discriminant => ray misses sphere
   if (discr < 0.0f) return false;
   // compute smallest root
   t = -b - sqrt(discr);
   // if smallest root negative, then ray starts within
    sphere
   if (t < 0.0f) t = 0.0f;
```

Ray/segment-sphere

```
bool testRaySphere(Point p, Vector d, Sphere s, float &t, Point &q)
{
    Vector m = p - s.c;
    float b = dot(m, d);
    float c = dot(m, m) - s.r * s.r;
    // exit if p outside sphere (c>0) and r pointing away from s (b>0)
    if (c > 0.0f && b > 0.0f) return false;
    // compute discriminant
    float discr = b*b - c;
    //negative discriminant => ray misses sphere
    if (discr < 0.0f) return false;
    // compute smallest root
    t = -b - sqrt(discr);
    // if smallest root negative, then ray starts within sphere
    if (t < 0.0f) t = 0.0f;
    // return p (if inside sphere) or first intersection point (otherwise)
    q = p + t * d;
    return true;
}
```

Outline

Summary

- Most simple proximity/distance/collision tests covered.
- Others covered in textbook.

Coming up

- Next week: broad phase techniques ...

References

- Ericson, C. (2004). Real-time collision detection. CRC Press.