Collision detection

- Collision detection is about finding where and when two objects intersect
- Main problems:
  - An accurate collision detection requires checking each polygon against all other polygons: $O(N_p^2)$, where N_p can be tens of thousands of polygons, every frame!
  - Computing the actual time of collision, not just whether or not two objects are currently intersecting
  - Finding the contact point, where two objects first hit each other

Collision detection (cont.)

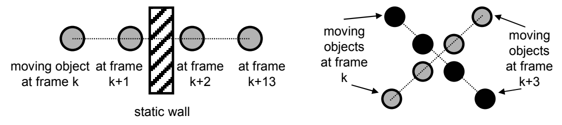
- Solutions to the complexity problem:
  - Broad phase + narrow phase
    - Broad phase does culling to quickly get rid of many obviously non-colliding objects
    - Narrow phase checks for intersection of remaining objects
  - Single phase
    - Use subdivision so that search space for any object is limited to its neighbourhood
  - Spatial and temporal coherence
    - Amount of change between frames is small, so reuse results from previous frames instead of starting from scratch every time
    - Speed of objects is often limited, which sets a minimum time two objects are guaranteed not to collide

Collision detection (cont.)

- Broad phase detection
  - Like visibility culling: use a simple test to quickly remove object pairs which definitely will not intersect
  - Test with simple bounding volumes
  - Don't test objects which are more than some number of grid squares away from each other
  - Only test objects which matter in the current state of the game, such as only objects (almost) visible to the player

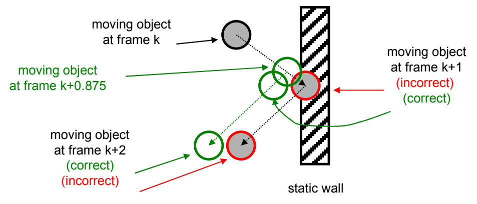
Collision detection (cont.)

- The fixed timestep problem:
  - If intersection calculations are performed only at each frame time update, collisions may be missed. An object with high velocity may have moved completely through another object within the time step.



Collision detection (cont.)

- Must compute an accurate collision time, generally better than frame time step resolution
  - Wrong collision time can produce different response



moving object at frame k
moving object at frame k+0.875
moving object at frame k+1 (incorrect) (correct)
moving object at frame k+2 (correct) (incorrect)
static wall

Collision detection (cont.)

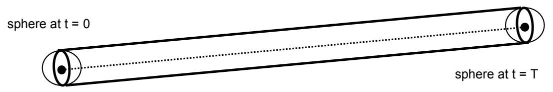
- Solutions to the time step problem
  - Swept volume: as an object moves over a time step, it sweeps out a volume. Check intersection between pairs of such swept volumes
    - Possibly use simple bounding volumes around the swept volume as a broad phase step before doing more accurate intersection computation
    - If there are multiple intersection points, pick the one with smallest `t` (time) value

Collision detection (cont.)

- Swept volume example
  - Sphere moving in a straight line in a time step T sweeps out a cylinder capped at both ends with a hemisphere

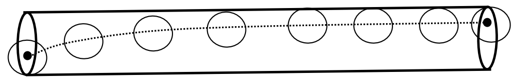
    Start point of cylinder is initial sphere position $P(0) = (x_0, y_0, z_0)$

    End point is $P(t) = (x_0, y_0, z_0) + (vx_0, vy_0, vz_0)\ t$



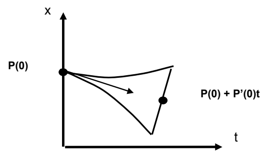
Collision detection (cont.)

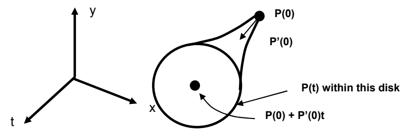
- Swept volume example
  - Sphere moving in a curved path (e.g. gravity). Use a bounding volume, such as a cylinder or oriented box
    - Could use (approximate) tangent at the halfway point of the path to orient the bounding volume
  - If path is simple, such as a quadratic, can often compute a tighter bounding volume by using its mathematical properties



Collision detection (cont.)

- If path is not known exactly (e.g. predicting collision ahead of time), limits on the object’s motion create a bounded volume
  - For example in point in 2D
    - Initial position $P(0) = (x_0, y_0)$
    - Initial velocity $P'(0) = (vx_0, vy_0)$
    - Say acceleration is bounded in magnitude over the time step $|P''(t)| \le f,\ 0 \le t \le T$
    - Path is defined by the equation of motion with acceleration $P(t) = P(0) + P'(0)\,t + 1/2\,P''(t)\,t^2$
    - Then change in position over time step is bounded by $|P(t) - (P(0) + P'(0)\,t)| \le (f/2)t^2,\ 0 \le t \le T$





Collision detection (cont.)

- Collision detection with axis-aligned bounding boxes (AABB)
  - Simple to test for intersection of two AABBs
    - Given two AABBs `A` and `B`, they do not intersect if:

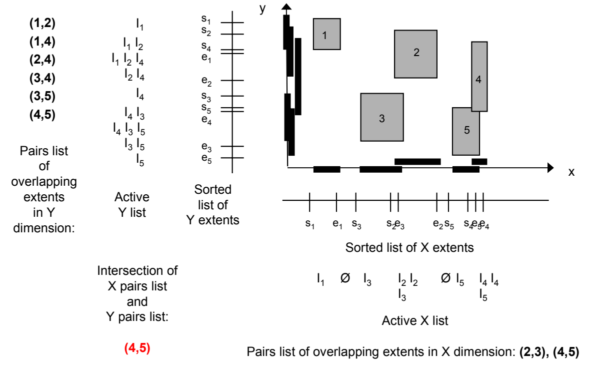
```
(A_Xmax < B_Xmin) || // A entirely left of B
(A_Xmin > B_Xmax) || // A entirely right of B
(A_Ymax < B_Ymin) || // A entirely below B
(A_Ymin > B_Ymax) || // A entirely above B
(A_Zmax < B_Zmin) || // A entirely in front of B
(A_Zmin > B_Zmax)    // A entirely behind B
```

    - This implies that two AABBs intersect if and only if they overlap in all 3 dimensions

Collision detection (cont.)

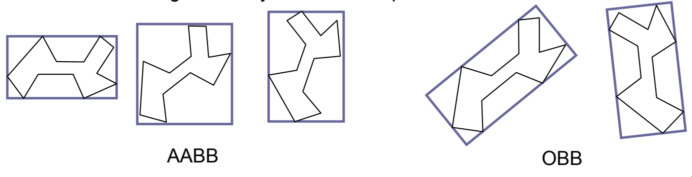
- Can reduce the number of AABB pairs to check by using a sort and sweep approach
  1. Maintain a sorted list in each dimension of AABB start and end positions.
  2. Sweep through each sorted list and build a second “active” list for each dimension. Whenever a “start” value is encountered, add that interval to the active list. Whenever an “end” value is encountered, remove that interval.
  3. When an interval is added to the active list add all the object pairs to a “pair list”.
  4. Compare the object pairs in each dimension’s pair list. Pairs that exist in the pair list for all 3 dimensions intersect. Those that don’t, do not intersect.
  5. As objects move, each extent list must be resorted. Due to spatial coherence changes will be small, so fast sorting methods which work on almost-sorted lists can be used.

Collision detection (cont.)


(1,2)
(1,4)
(2,4)
(3,4)
(3,5)
(4,5)
Pairs list of overlapping extents in Y dimension:
I1
I1 I2
I1 I2 I4
I2 I4
I4
I4 I3
I4 I3 I5
I3 I5
I5
Active Y list
s1 s2 s4 e1 e2 s3 s5 e4 e3 e5
Sorted list of Y extents
y
1
2
3
4
5
x
s1 e1 s3 s2 e3 e2 s5 s4 e5 e4
Sorted list of X extents
I1 Ø I3 I2 I2 I3 Ø I5 I4 I4 I5
Active X list
Intersection of X pairs list and Y pairs list:
(4,5)
Pairs list of overlapping extents in X dimension: (2,3), (4,5)

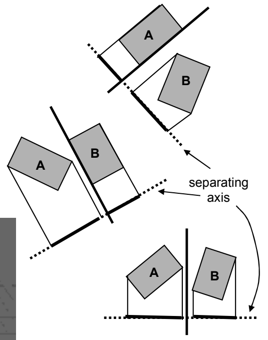
Collision detection (cont.)

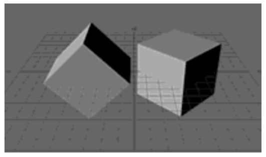
- AABB are fast to compute and check, but can be a poor fit
  - May need to recompute AABB when object rotates or animates
  - Collision detection with AABB is simple
- Oriented bounding boxes (OBB) give a better fit, but are computationally more complex
  - OBB remains constant with rotation
  - OBB for non-animated objects can be computed offline
  - OBB collision detection less simple
- Choice is a balance between extra cost of doing OBB compared with AABB, and the savings made by fewer narrow phase checks



Collision detection (cont.)

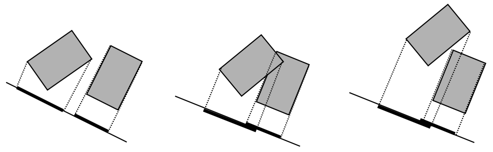
- Broad Phase collision detection with OBBs: Separating Axis Theorem (SAT)
- For 2 arbitrary, convex, disjoint polyhedra, A and B, there exists a separating axis where the projections of the polyhedra, which form intervals on the axis, are also disjoint
- If A and B are disjoint, then they can be separated by an axis that is orthogonal to either:
  1. a face of A
  2. a face of B
  3. an edge from each polyhedron





Collision detection (cont.)

- In 3D there are maximum of 15 separating axes that may need testing.
  - 3 axes orthogonal to the 3 orthogonal faces of A
  - 3 axes orthogonal to the 3 orthogonal faces of B
  - 9 axes orthogonal to an edge of A and an edge of B
- Project OBBs onto separating axis and check if their projections overlap
  - If no overlap for any separating axis, OBBs don't overlap
  - Optimisation for first 6 cases: projection of OBB with orthogonal face is that orthogonal face
  - Last 9 cases occur relatively rarely, so could consider not doing them and rely on narrow phase detection for final decision



Collision detection (cont.)

- Broad phase collision detection with spatial partitioning
  - Assume one or both of potentially colliding objects have a spatial partitioning
    - Regular grid
    - Bounding volume tree (spheres, AABB, OBB)
    - Octree (bounding volume tree with tightly packed AABBs)
    - BSP tree
  - Spatial partitioning and subdivision reduces the number of objects and polygons that need to be checked for collision by rejecting large groups of them early

Collision detection (cont.)

- Regular grid
  - If objects are no larger than one grid square, two objects will only ever collide if they are no more than one grid square away from each other

```
for all objects O
  for y = O.grid_y-1 to O.grid_y+1
    for x = O.grid_x-1 to O.grid_x+1
      for all objects O’ in grid(x,y)
        check collision O <-> O’
```

Collision detection (cont.)

- Bounding volume trees (BV trees)
  - Assume bounding volume at any node in tree bounds the union of bounding volumes of its children
    1. Check object/polygon/point against top node of BV tree
    2. If intersection with BV node, recurse into children until certain collision or no collision found
  - Can check two BV trees A and B for collision:
    1. Check intersection between node A and node B bounding volume
    2. If no intersection
       1. return no collision
    3. Else if A and B are leaves
       1. return narrow phase intersection test between objects in leaves
    4. Else if A is a leaf
       1. return intersection test between objects in leaf A and child nodes of B
    5. Else if B is a leaf
       1. return intersection test between objects in leaf B and child nodes of A
    6. Else
       1. For each pair of child nodes from A, B
          1. If BV of child nodes intersect, recurse using those two nodes
  - Can modify to check trees breadth-first
    - If collision detection is running out of time, can terminate recursion at the current level and assume that there is a collision

Collision detection (cont.)

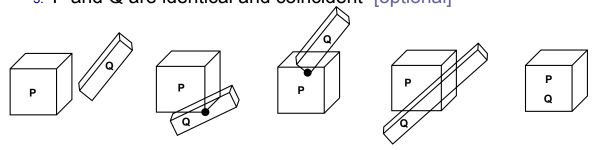
- Collision detection with BSP trees:
  1. At a node, classify object/BV/poly/point as in front, behind, or across partition plane
  2. If across
     1. Test against polygons in node
     2. If intersection found, return true
  3. If in front or across
     1. Recurse into front node
     2. If intersection found, return true
  4. If behind or across
     1. Recurse into back node
     2. If intersection found, return true
  5. Return false

Collision detection (cont.)

- Narrow phase collision detection
  - Determine accurate intersection between:
    - Points
    - Lines and edges
    - Planes and polygons
    - Polyhedra, very often convex only

Collision detection (cont.)

- Collision of two of convex polyhedra
  - Given 2 convex polyhedra, P and Q, determine if they intersect
  - 5 possible cases for intersection:
    1. No intersection
    2. 1 or more vertices of P falls within Q
    3. 1 or more vertices of Q falls within P
    4. 2 or more edges of P intersect a face of Q [optional]
       (note: 2 or more edges of Q intersect a face of P is a redundant test)
    5. P and Q are identical and coincident [optional]



Collision detection (cont.)

- Test for a vertex within a convex polyhedron (cases 2, 3, possibly 5)
  - Similar to testing a point within the view frustum
  - To test for a vertex v of Q falling within P:
    1. For all faces of P
       1. Compute which side of face v is on
       2. If v is on the outside of face, v is outside P
    2. If v is inside all faces, v is inside P
  - Test each vertex of Q. If any is inside P, Q is intersecting P
  - If all vertices of Q are outside P, repeat with P and Q reversed

Collision detection (cont.)

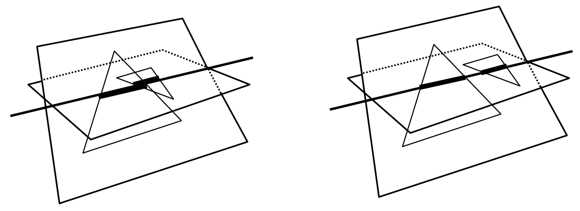
- Test for a vertex in a concave polyhedron
  - Shoot a ray from vertex to infinity
    - Use axis-aligned ray for ease
  - Count the number of times the ray intersects the polyhedron
  - If count is even, vertex is outside
  - If count is off, vertex is inside
  - Must be very careful with ray (almost) intersecting vertex, or going through face edge-on

Collision detection (cont.)

- Test for an edge penetrating a face of a polyhedron
  - For each edge e of Q
  - For each face f of P
    1. Test if edge intersects infinite plane of a face f
       - compute distance to plane for each edge vertex using plane equation
       - If sign differs, edge intersects plane
    2. Find ray parameter t at intersection point
    3. If 0 <= t <= 1, add t to list of face intersection points for edge
  - If list length > 1:
    - Sort list of t values (possibly create list sorted using insertion sort)
    - In order for an edge to penetrate P, there must exist a pair of consecutive face intersections that form an “entry” into P followed by an “exit” from P, which implies any point in between them must be inside P.
    - For each pair of t values, compute a midpoint and apply the vertex within a polyhedron test. If true, edge penetrates P, else no penetration.

Collision detection (cont.)

- Triangle-triangle intersection
  - Special case of polygon/polygon that takes advantage of geometry
  - Interval Overlap Method
    - If the planes of the two triangles intersect
      - Find the line where the two planes intersect
      - Find the extents along that line of each triangle
      - Test if extents overlap



Collision detection (cont.)

- Finding collision time:
  - For given object pair, can do a binary search over the frame interval
    - Take earliest time when they intersect
    - Assumes at most one collision between pair during the frame interval
- Responding to a collision:
  - Boom, ouch, you're dead
  - Bounce off by changing direction and re-running movement from collision time to end of frame interval
    - May want to re-check for collisions given new movement
  - Return to previous "safe" position where there was no collision
    - But previous position may no longer be safe, as a 3rd object may have moved into that space