Part 10: Collision Detection

Overview

- Bounding Volumes
  - Separating Axis Theorem
- Using Bounding Volume Hierarchies
- Efficient Collision Detection for Several Hundreds of Objects

Further Reading

  - Chapters 13 + 14
- Free collision detection software
  - http://www.cs.unc.edu/~geom/V_COLLIDE/

Prof. Bernhard Jung
Collisions: In General

- Without collision detection (CD), it is practically impossible to e.g., games, movie production tools
- Three major parts
  - Collision detection: do objects collide?
  - Collision determination: where?
  - Collision response
- This lecture: collision detection + determination
  - Collision response involves physically-based animation
- Use rays for simple applications
- Use Bounding Volume Hierarchies (BVHs) to test two complex objects against each other
- But what if several hundreds of objects?

Typical Bounding Volumes

- Sphere
- Box
  - Axis-aligned Bounding Box (AABB)
  - Oriented (OBB)
- k-DOP
  - "discrete oriented polytype"
  - generalizes AABB
    - (in 3D): AABB = 6-DOP
- Different tightness
  - … and different costs for intersection testing
k-DOPs

source: Unreal developer network
http://udn.epicgames.com/Two/CollisionTutorial

Simple Tests:
Ray/Box Intersection

- Boxes and spheres often used as bounding volumes
  - also: "hit-boxes" in games

- A slab is the volume between two parallel planes:

- A 3D box is the logical intersection of three slabs (2 in 2D):
Simple Tests: Ray/Box Intersection (2)

- Intersect the 2 planes of each slab with the ray
- Ray: \( r(t) = o + td \) (\( o \) origin, \( d \) direction)

- Keep max of \( t_{min} \) and min of \( t_{max} \)
- If \( t_{min} \neq t_{max} \) then we got an intersection
- Special case when ray parallel to slab

Simple Tests: Sphere-Sphere

- trivial:
  - if distance of centers > sum of radii then disjoint

```cpp
bool Sphere_intersect(c1, r1, c2, r2)
returns({OVERLAP, DISJOINT});
1: l = c1 - c2
2: \( l^2 = l \cdot l \)
3: if(\( l^2 > (r_1 + r_2)^2 \)) return (DISJOINT);
4: return (OVERLAP);
```
Simple Tests: AABB-AABB

- trivial:
  - project boxes to axes

- if no overlap in any of \(\{x,y,z\}\)-axes then disjoint ("separating axis")

```c
bool AABB_intersect(A, B) {
  return (OVERLAP, DISJOINT);
  for each i \(\in\{x,y,z\}\)  
  if (\(a_i^{\min} < b_i^{\max}\) or \(b_i^{\min} > a_i^{\max}\))
  return (DISJOINT);
  return (OVERLAP);
}
```

Simple Tests: k-DOP – k-DOP

- generalizes AABB-AABB intersection test
- project bounding volumes to \(k/2\) axes
  - disjoint if not overlapping on at least one ("separating") axis
- for moderate values of \(k\), e.g. \(k=18\), an order of magnitude faster than OBB-OBB intersection testing

```c
kDOP_intersect(d_1^{\min}, \ldots, d_k^{\min}, d_1^{\max}, \ldots, d_k^{\max}, d_{k/2}^{\min}, \ldots, d_{k/2}^{\max}) {
  return (OVERLAP, DISJOINT);
  for each i \(\in\{1,\ldots,k/2\}\)  
  if (\(d_i^{B_{\min}} > d_i^{A_{\max}}\) or \(d_i^{A_{\min}} > d_i^{B_{\max}}\))
  return (DISJOINT);
  return (OVERLAP);
}
```
Separating Axis Theorem (SAT)

- **Summary:** Given two convex shapes, if we can find an axis along which the projection of the two shapes does **not** overlap, then the shapes don't overlap.
- **Concretely:** Two convex polyhedrons, A and B, are disjoint if any of the following axes separate the objects:
  - An axis orthogonal to a face of A
  - An axis orthogonal to a face of B
  - An axis formed from the cross product of one edge from each of A and B

**SAT example: Triangle-AABB (2D case)**

- **Box is axis-aligned**
  - → test x, y axes
- **Triangle:**
  - two edges parallel to x, y
  - → test axis perpendicular to the one non-aligned edge

http://www.harveycartel.org/
SAT example: Triangle-Box

- Box is axis-aligned
  1) test the axes that are orthogonal to the faces of the box
- That is, x, y, and z

Assume that they overlapped on x, y, z
Must continue testing
2) Axis orthogonal to face of triangle
Triangle-Box with SAT (3)

- If still no separating axis has been found…
  3) Test axis: \( t = e_{bo} \times e_{tri} \)

- Example:
  - x-axis from box: \( e_{bo} = (1,0,0) \)
  - \( e_{tri} = v_1 - v_0 \)

- Test all such combinations
- If there is at least one separating axis, then the objects do not collide
- Else they do overlap

OBB – OBB with SAT

- Project to 15 potential separating axes:
  - 3 from object-1
    - \( x_1, y_1, z_1 \)
  - 3 from object-2
    - \( x_2, y_2, z_2 \)
  - 9 from combinations of edges from two objects
    - \( x_1 \times x_2 \)
    - \( x_1 \times y_2 \)
    - \( x_1 \times z_2 \)
    - \( y_1 \times x_2 \)
    - ...

- can stop testing when first separating axis is found
Rules of Thumb for Intersection Testing

- Acceptance and rejection test
  - Try them early on to make a fast exit
- Postpone expensive calculations if possible
- Use dimension reduction
  - E.g. 3 one-dimensional tests instead of one complex 3D test, or 2D instead of 3D
- Share computations between objects if possible
- Do timing comparisons

Dynamic Separating Axis Theorem
(advanced topic)

- SAT: tests one axis at a time for overlap

- Same with DSAT, but:
  - Need to adjust the projection on the axis so that the interval moves on the axis as well
- Need to test same axes as with SAT
- Same criteria for overlap/disjoint:
  - If no overlap on axis => disjoint
  - If overlap on all axes => objects overlap
For many, many objects...

- Test BV of each object against BV of other object
- Works for small sets, but not very clever
- Reason …
  - assume moving \( n \) objects
  - number of tests in naïve approach: \( \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2) \)
  - if \( m \) static objects, then: \( nm + \binom{n}{2} \)
  - there are smarter ways, e.g. sweep-and-prune (last part of lecture)

Collision detection with rays

- Imagine a car is driving on a road sloping upwards
- Could test all triangles of all wheels against road geometry
- For certain applications, we can approximate, and still get a good result
- Idea: approximate a complex object with a set of rays
CD with rays, cont’d

- Put a ray at each wheel
- Compute the closest intersection distance, t, between ray and road geometry
- If t=0, then car is on the road
- If t>0, then car is flying above road
- If t<0, then car is plowing deep in the road
- Use values of t to compute a simple collision response

Another simplification

- Sometimes 3D can be turned into 2D operations
- Example: maze
  - A human walking in maze, can be approximated by a circle
  - Test circle against lines of maze
  - Or even better, move walls outwards with circle radius
    - test center of circle against moved walls
A collision detection system for accurate detection and for many objects

- covered here: "pruning" and "exact CD"
- "Simulation" is how objects move

General Hierarchical Collision Detection

- Model is represented as a hierarchy of some kind of bounding volumes
- A hierarchy that is commonly used is a data structure called a k-ary tree
  - typical BVHs: binary trees
- At each internal node, there is a BV that encloses all of its children in its volume
- At each leaf there are one or more primitives (e.g. triangles)
Hierarchy Building

Three ways:

- **Bottom-up**
  - by combining a number of primitives and finding a BV for them

- **Top-down**
  - by finding a BV for all primitives of the model, then divide-and-conquer is applied

- **Incremental tree-insertion**
  - starts with an empty tree and then all other primitives and their BVs are added one at a time to this tree
  - Insertion point should be selected so that the total tree volume increase is minimized

BVH building

- Can split on triangle level as well

Use split plane

Sort using plane, w.r.t triangle centroids

Find minimal boxes

...and so on.
OBBTree Hierarchy Building

- OBBTree: binary tree with each internal node holding an OBB and each external node holding only one triangle
- top-down hierarchy building approach is divided into
  - finding a tight-fitting OBB
  - and splitting this along one axis of the OBB
  - details see (Akenine-Möller & Haines, 2002)

Pseudo code for BVH against BVH

```plaintext
FindFirstHitCD(A, B)
returns {[TRUE, FALSE]};
1: if (isLeaf(A) and isLeaf(B))
2:   for each triangle pair T_A ∈ A_0 and T_B ∈ B_0
3:     if (overlap(T_A, T_B)) return TRUE;
4:   else if (isNotLeaf(A) and isNotLeaf(B))
5:     if (overlap(A_SV, B_SV))
6:       if (Volume(A) > Volume(B))
7:         for each child C_A ∈ A
8:           if (isLeaf(C_A) and isNotLeaf(B))
9:             FindFirstHitCD(C_A, B)
10:            else
11:               for each child C_B ∈ B
12:                 if (overlap(T_A, B_SV))
13:                   FindFirstHitCD(C_B, A)
14:                 else
15:                   FindFirstHitCD(C_B, B)
16:             end
17:          else
18:            for each child C_B ∈ B
19:              if (isLeaf(C_B) and isNotLeaf(A))
20:                FindFirstHitCD(A, C_B)
21:          end
22:     end
23:   end
24: return FALSE;
```

Pseudocode deals with 4 cases:

1) Leaf against leaf node
2) Internal node against internal node
3) Internal against leaf
4) Leaf against internal
Comments on pseudocode

- The code terminates when it finds the first triangle pair that collided
  - collision detection
- Simple to modify code to continue traversal and put each pair in a list
  - collision determination
- Reasonably simple to include rotations for objects as well
  - Note that if we use AABB for both BVHs, then the AABB-AABB test becomes a AABB-OBB test

---

Tradeoffs

- Choice of BV
  - Sphere, AABB, OBB, k-DOP
- In general, the tighter BV, the slower test
- Less tight BV, gives more triangle-triangle tests in the end
- Model rotation may require updates of AABB, k-DOP
  - not: OBB, sphere
- Cost function: \[ t = n_v c_v + n_p c_p + n_u c_u \]
Multiple Objects CD: Pruning

- Why needed?
- Consider several hundreds of rocks tumbling down a slope...
- Include a first rough but simple test on whether objects can potentially collide
- This system is often called "First-Level CD"
- We execute this system because we want to execute the 2nd system less frequently
- Assume high frame-to-frame coherency
  - Means that object is close to where it was previous frame
  - Reasonable

Sweep-and-prune algorithm
[by Ming Lin]

- Assume objects may translate and rotate
- Then we can find a minimal cube (AABB), which is guaranteed to contain object for all rotations
  - alternatively, may use bounding spheres
- Do collision overlap three times
  - One for x, y, and z-axes

- Let’s concentrate on one axis at a time
- Each cube on this axis is an interval, from $s_i$ to $e_i$, where $i$ is cube number
Sweep-and-prune algorithm

- Sort all \( s_i \) and \( e_i \) into a list
- Traverse list from start to end
- When an \( s \) is encountered, mark corresponding interval as active in an active_interval_list
- When an \( e \) is encountered, delete the interval in active_interval_list
- All intervals in active_interval_list are overlapping!

Sorting is expensive: \( O(n\log n) \)

But, exploit frame-to-frame coherency!
- a.k.a. "temporal" coherence
- the list is not expected to change much
- Therefore, "re-sort" with bubble-sort, or insertion-sort
- Expected time complexity for sorting: \( O(n) \)
Sweep-and-prune algorithm

- Keep a Boolean for each pair of intervals
- Invert when sort order changes (in later frame)
- If all Booleans for all three axes are true $\Rightarrow$ objects overlap
- Expected time complexity of complete sweep-and-prune algorithm:
  - $O(n+k)$ where $k$ is the number of pairwise overlaps

Collision Detection Conclusions

- Very important part of games!
- Many different algorithms to choose from
- Trend: hardware-acceleration (GPU)
  - e.g. CULLIDE, UNC, 2003