Collision Detection

Prof. Doug James
Most sounds are produced by contact interactions

The need for collision detection

NVIDIA FleX
https://developer.nvidia.com/flex

Distributed Ray Tracing [Cook et al. 1984]
Collision Detection & Proximity Queries
Collisions

Two parts:

1. Collision detection:
   ○ Geometric determination of collisions (what, where, when)

2. Collision resolution:
   ○ Physical application of forces or impulses to modify motion
Collision Detection & Proximity Queries

Fast computer methods for
- **Proximity** and **distance** queries
- Fast **neighbor** finding
- **Overlap** tests, e.g., triangle-triangle test
- Discrete and continuous **collision** tests
- **Ray intersection** tests (rendering!)

Many interesting data structures:
- Geometric bounds
- Bounding volume hierarchies
- Spatial subdivisions (uniform and adaptive grids)
- Sorting and hashing
Proximity & Distance Queries

Finding closest objects and features (e.g., vertex, edge, face)

Evaluating minimum distances
- Between primitives
- Between objects

Object representations:
- Mathematical domains
  - Implicit functions, e.g., distance fields
- Polyhedral meshes
- Spline or subdivision surfaces
- Swept motions

Fast methods for most cases.
Distance Fields: Encode distance from a point, \( x \), to an object, \( O \):
\[
d(x) = \min \|x-y\|_2 \quad \text{over all } y \text{ on } O.
\]

Signed Distance Fields: Sign indicates if inside/outside object:
\[
d(x) > 0 : \text{Outside object} \\
d(x) = 0 : \text{On object} \\
d(x) < 0 : \text{Inside object}
\]

Gradient of \( d(x) \) is direction to closest surface.

Analytical formulas exist only for simple shapes, E.g., [https://www.shadertoy.com/view/XsXSz4](https://www.shadertoy.com/view/XsXSz4)
Distance Fields for Complex Shapes

Interpolate distances previously sampled on a regular (or adaptive) grid

Pros:
- Fast \( d(x) \) evaluation
- Complex shapes
- Parallel precompute

Cons:
- High memory overhead
  - Adaptive helps
- Precomputation
- Rigid geometry
  - Can’t deform
- Only point-object distance

http://barbic.usc.edu/signedDistanceField/
Fast neighbor finding

Often want to know neighbors within some distance cutoff, R.

Example: Particle-based fluid simulation
- Particle-particle interactions only occur within some small distance, R.
- Each particle needs to find neighboring particles within distance, R.
Motivation: Particle-based Fluid Simulation

Fast neighbor finding
Overlap and Intersection Tests

Determine if shapes or primitives (triangles, rays, etc.) overlap

Different queries possible:

- Determine if overlap occurs
  - Return true or false
  - E.g., boolean collision check
- Determine all overlaps
  - Enumerate all overlapping pairs for meshes
- Determine where primitives overlap
  - E.g., to estimate “contact points” for collision processing
  - E.g., to find “intersection point” for ray-triangle intersection
Problem: Missed collisions

Discrete (sampled) vs continuous time collision checks

- Avoids missed collisions
  - Common in game physics libraries, e.g., Bullet
- Useful for accurate time-of-collision (ToC) estimates
  - Ensures proper ordering of collision events
- Important for simulation of cloth and thin objects
  - Less common for volumetric simulations
Fundamental Problem:
Pairwise calculations are $O(N^2)$ work.
**Fundamental Problem:**

**Pairwise calculations are O(N^2) work**

Double-loop calculations don’t scale well for large numbers of things, N

**Naive all-pairs loop over N^2 entries:**

```plaintext
for(i = 1 .. N)
    for (j = 1 .. N)
        processPair(i,j);
```

**Better:** Only the \((N^2-N)/2\) unique pairs:

```plaintext
for(i = 1 .. N)
    for (j = i+1 .. N)
        processPair(i,j); // unique pair, i<j
```

- N diagonal (i=j)
- \(N^2-N\) off-diagonals (i\(\neq\)j)
- \((N^2-N)/2\) unique pairs
Pairwise calculations are $O(N^2)$ work

Double-loop calculations don’t scale to large number of particles, $N$

Sometimes there are $O(N^2)$ non-zero interactions:

- E.g., N-body gravitational forces
- Deal with it using different strategies:
  - Parallel computing
  - Fast-summation algorithms (Barnes-Hut, fast multipole)

But in many problems, most interactions are inactive and can be “culled”

  - E.g., expect #contacts proportional to #bodies in typical cases
  - $O(1)$ contacts per body $\rightarrow O(N)$ contacts for $N$ bodies

Strategy: Cull inactive contributions, e.g., distant bodies in collision detection. How? Many approaches.

Complexity of Broad Phase Collision Detection:

- REBOUND
Bounding Volumes

Simple convex bounds.
Fast & simple overlap tests:
- False positives possible
- False negatives impossible

Fast methods to fit bounds exist, e.g., O(N) for sphere, AABB, 8-DOP.

Avoiding all-pairs checks

Ericsson 2004
Avoiding all-pairs checks

Bounding Volume Overlap Tests

Bounding Sphere

Overlaps if \[ \text{dist}_{12} < R_1 + R_2 \]

Avoid the square-root using:

\[ (x_1 - x_2)^2 + (y_1 - y_2)^2 < (R_1 + R_2)^2 \]
Avoiding all-pairs checks

Bounding Volume Hierarchies

Common BVHs in graphics:

- AABB Trees [van den Bergen 1998]: good for deformable models
- OBB Trees [Gottschalk et al. 1996]: tighter fitting
- Sphere Trees [Hubbard 1996]

Many strategies for BVH construction
Terminology

Broad- and Narrow-Phase Checks

Collision Detection Module

Geometry → Broad Phase → Narrow Phase → Collision Response
Collision Detection Module

Geometry → Broad Phase → Narrow Phase → Collision Response

A

B

C
Collision Detection Module

Geometry → Broad Phase → Narrow Phase → Collision Response

A → B → C

VectorStock.com images
Collision Detection Module

Geometry → Broad Phase → Narrow Phase → Collision Response
Collision Detection Module

Geometry → Broad Phase → Narrow Phase → Collision Response

VectorStock.com images
**Broad- and Narrow-Phase Checks**

**Broad Phase:**
- Coarse object-level view of simulation, e.g., as bounding volumes
- Determines object pairs that *may* collide, and culls other pairs
- Usually fast
- Acceleration schemes:
  - Trees:
    - Bounding Volume Hierarchy (BVH)
  - Sorting
    - Sweep & Prune
  - Spatial subdivisions
    - Uniform grids; hashing
    - Adaptive grids: quadtree (2D) and octree (3D), kd-trees

**Narrow Phase:**
- Processes object-object pairs
- Detailed collision checks
  - Triangle-triangle
  - Ray-triangle
- May require its own broad- & narrow- phase layer
  - E.g., BVH on triangle meshes to determine triangle-triangle tests.
- Usually slower
  - Unless objects separated
- Worst case is still $O(N^2)$ work!
  - Why?
Speeding up the broad phase:

Culling pairwise checks for distant pairs
Collision Detection in 1D

Objects are intervals, \([x_{\text{min}}, x_{\text{max}}]\) for \(x_{\text{min}} \leq x_{\text{max}}\).

Collision detection amounts to detecting overlapping intervals.

**1D Overlap Test:** Two intervals \([a,A]\) & \([b,B]\) overlap iff 
\[
\max(a,b) \leq \min(A,B).
\]

OpenProcessing Tip: Use `Math.min` & `Math.max` whenever possible.

Generalization to higher dimensions?
Separating Axis Theorem (SAT)

“Two convex objects do not overlap if there exists a line (called axis) onto which the two objects' projections do not overlap.”

https://en.wikipedia.org/wiki/Hyperplane_separation_theorem
Convex objects

Separating Axis Theorem (SAT)

Useful for overlap tests between convex shapes:
- Lines, triangles, boxes (AABB & OBB), balls, convex hulls, etc.

Approach for polygons:
- Determine list of axes to test, \( \{n_i\} \)
  - E.g., all face normals
- For each direction:
  - Project vertices onto axis
  - Find max/min values, \([a,A]\) & \([b,B]\)
  - Perform overlap tests
    - Return false ASAP if a non-overlapping interval is found

Optimizations: Use bounds and/or center-of-mass axis first; test incrementally and early exit

https://en.wikipedia.org/wiki/Hyperplane_separation_theorem
Testing Oriented Bounding Boxes (OBBs)

Separating Axis Theorem (SAT)

Projection of box shapes onto a potential separating axis (the arrow)

Tracking axis and overlap distance for each test

Repeated for all potential separating axes...

Resolve by moving objects in the axis of least overlap, with a total distance equal to the overlap amount.

https://www.atoft.dev/posts/2020/04/12/implementing-3d-collision-resolution/
N-Particles Overlap Tests
With 1D & 2D Uniform Subdivision

https://www.openprocessing.org/sketch/973557
**Broad-Phase Tests**

**1D Uniform Spatial Subdivision**

**Construction:**

- Divide space into N bins of equal width, h
- Add each object to each bin that its bounding volume overlaps:
  - Use 1D overlap test

**Cell Index, i:** Given coordinate x, find containing cell index(x) using Math.floor(x/h) clamped to [0,N-1].
1D Uniform Spatial Subdivision

Overlap Testing:
- Given test bound
- Find overlapping cells, and for each bound
  - Do overlap test
- Return overlapping results as a set.

Q: Can duplicate overlaps occur?

Weakness of 1D subdivision?

Broad-Phase Tests
2D Uniform Spatial Subdivision

Construction:
- Divide space into \( Nx \)-by-\( Ny \) bins of constant width, \( h \) (or \( hx \) & \( hy \))
- Add each object to each bin that its bounding volume overlaps:
  - Use 1D overlap tests

Cell Index \((i,j)\): Given coords \( x \) & \( y \),
\[
i = \text{floor}(x/h_x) \text{ clamped to } [0,Nx-1],
\]
\[
J = \text{floor}(y/h_y) \text{ clamped to } [0,Ny-1].
\]
Overlap Testing:
- Given test bound
- Find overlapping cells, and for each bound
  - Do overlap test
- Return overlapping results as a set.

Q: Can duplicate overlaps occur?

Weakness of 2D subdivision?
Broad-Phase Tests

2D Uniform Spatial Subdivision

GridCheck2D:

- Simplified implementation. Just drop in your sketch.
- See Sketch: [https://www.openprocessing.org/sketch/973557](https://www.openprocessing.org/sketch/973557)

```java
// A simple uniform 2D subdivision for interval-overlap tests on [0,width]x[0,height].
// Given an object and rectangular interval, [xmin,xmax]x[ymin,ymax], the object is stored in cells it overlaps.
// A test rectangle can be used to find objects with overlapping intervals.

class GridCheck2D {

constructor(nXCells, width, nYCells, height) {
    this.nXCells = nXCells;
    this.nYCells = nYCells;
    this.nXCellsMinus1 = nXCells-1; // used often
    this.nYCellsMinus1 = nYCells-1; // used often
    this.dx = width /this.nXCells;
    this.dy = height /this.nYCells;
    this.invDX = 1/this.dx; // used often
    this.invDY = 1/this.dy; // used often
    this.width = width;
    this.height = height;
}
```
Other spatial subdivisions

- **Spatial Hashing**: Virtual grid of *unbounded* size. Use a hashtable to map cells to a finite size storage container.
  - Basic Idea:
    - Map cell (i,j) to a hashcode
      - e.g., key = $N*(i\%N) + j\%N$ (or something more random)
    - Insert/retrieve contents using hashtable key.
      - Watch for “hashcode collisions,” e.g., due to wraparound. OK, since testing bounds.
  - GPU acceleration very effective. Many schemes.

  - OpenVDB: Sparse volume data structure
From “Fast Fluid Simulation with Sparse Volumes on the GPU” [Wu et al. 2018]