**[Collision detection (part 1): Overview](https://0fps.net/2015/01/07/collision-detection-part-1/)**

Posted on [January 7, 2015](https://0fps.net/2015/01/07/collision-detection-part-1/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/" \o "View all posts by mikolalysenko)

Collision, or intersection, detection is an important geometric operation with a large number of applications in graphics, CAD and virtual reality including: [map overlay operations](https://en.wikipedia.org/wiki/Map_Overlay_and_Statistical_System), [constructive solid geometry](https://en.wikipedia.org/wiki/Constructive_solid_geometry), [physics simulation](https://en.wikipedia.org/wiki/Collision_detection), and [label placement](https://en.wikipedia.org/wiki/Automatic_label_placement).  It is common to make a distinction between two types of collision detection:

* **Narrow phase**:  Test if **2** objects intersect
* **Broad phase**: Find all pairs of intersections in a set of **n** objects

In this series, I want to focus on the latter (broad phase), though first I want to spend a bit of time surveying the bigger picture and explaining the significance of the problem and some various approaches.

Narrow phase

The approach to narrow phase collision detection that one adopts depends on the types of shapes involved:

Constant complexity shapes

While it is true that for simple shapes (like triangles, boxes or spheres) pairwise intersection detection is a constant time operation, because it is frequently used in realtime applications (like VR, robotics or games) an enormous amount of work has been spent on optimizing.  The book “Realtime Collision Detection” by Christer Ericson has a large collection of carefully written subroutines for intersection tests between various shapes which exploit SIMD arithmetic,

C. Ericson, (2004) “[Realtime Collision Detection](http://www.amazon.com/exec/obidos/tg/detail/-/1558607323?tag=0fpsnet-20)“

There is also a [table of various collision detection predicates collected from different sources at realtimerendering.com](http://www.realtimerendering.com/intersections.html).

Convex polytopes

For more complicated shapes (that is shapes with a description length longer than O(1)) detecting intersections becomes algorithmically interesting.  One important class of objects are [convex polytopes](https://en.wikipedia.org/wiki/Convex_polytope), which have the property that between any pair of points in the shape the straight line segment connecting them is also contained in the shape.  There are two basic ways to describe a convex polytope:

* **V-Polytope:**As the set of all [convex combinations](https://en.wikipedia.org/wiki/Convex_combination) of a finite set of points and possibly infinite direction vectors (aka 1D cones)
* **H-Polytope:**As the intersection of a finite set of [closed linear half spaces](https://en.wikipedia.org/wiki/Half-space_%28geometry%29).

These two representations are equivalent in their descriptive power (though proving this is a bit tricky).  The process of converting a V-polytope into an H-polytope is called taking the “[convex hull](https://en.wikipedia.org/wiki/Convex_hull_algorithms)” of the points, and the dual algorithm of converting an H-polytope into a V-polytope is called “[vertex enumeration](https://en.wikipedia.org/wiki/Vertex_enumeration_problem).”

The problem of testing if two convex polytopes intersect is a special case of [linear programming feasibility](https://en.wikipedia.org/wiki/Linear_programming).  This is pretty easy to see for H-polytopes; suppose that:

S = \left \{ x \in \mathbb{R}^d : A x \leq b \right \}

T = \left \{ x \in \mathbb{R}^d : C x \leq d \right \}

Where,

* A is a n-by-d matrix
* b is a n-dimensional vector
* C is a m-by-d matrix
* d is a m-dimensional vector

Then the region S \cap T is equivalent to the [feasible region](https://en.wikipedia.org/wiki/Feasible_region) of a system of n + m linear inequalities in d variables:

A x \leq b

C x \leq d

If this system has a solution (that is it is feasible), then there is a common point x in the interior of both sets which satisfies the above equations.  The case for V-polytopes is similar, and leads to a transposed system of n + m variables in d constraints (that is, it is the [asymmetric dual of the above linear program](https://en.wikipedia.org/wiki/Linear_programming#Duality)).

Linear programs are a special case of [LP-type problems](https://en.wikipedia.org/wiki/LP-type_problem), and for low dimensions can be solved linear time in the number of half spaces or variables. (For those curious about the details, here are [some](https://www.youtube.com/watch?v=heCf2d0fBsc&list=PLESnaHRvLM-72xIXf8dL2EOqN8UgAZMj7&index=7) [lectures](https://www.youtube.com/watch?v=0Y0aRRjvRZs&list=PLESnaHRvLM-72xIXf8dL2EOqN8UgAZMj7&index=8)).  For example, using Seidel’s algorithm testing the feasibility of the above system takes O(d! (n + m)) time, which for fixed dimension is just O(n + m). The dependence on d can be improved using more advanced techniques like Clarkson’s algorithm.

Preprocessing

If we are willing to preprocess the polytopes, it is possible to do exponentially better than O(n + m).  In 2D, the problem of testing intersection between a pair of polygons reduces to calculating a pair of tangent lines between the polygons.  There is[a well known algorithm for solving this in [O(\log(n))](http://geomalgorithms.com/a15-_tangents.html) time by binary search](http://geomalgorithms.com/a15-_tangents.html) (assuming that the vertices/faces of the polygon are stored in an ordered array, which is almost always the case).

The 3D case is a bit trickier, but it can be solved in O(\log^3(n)) using a more sophisticated data structure,

B. Chazelle, D. Dobkin. (1988) “[Intersection of convex objects in two and three dimensions](http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=A8E46240279E48B7EB83BEC31D46B7BB?doi=10.1.1.126.7622&rep=rep1&type=pdf)”  Journal of the ACM

For interactive applications like physics simulations, another important technique is to reuse previous closest points in calculating distances (similar to using a warm restart in the simplex method for linear programming).  This ideas were first applied to collision detection in the now famous Lin-Canny method:

M. Lin, J. Canny, (1991) “[A fast algorithm for incremental distance calculation](http://web.stanford.edu/class/cs277/resources/papers/Lin1991.pdf)” ICRA

For “temporally coherent” collision tests (that is repeatedly solved problems where the shapes involved do not change much) the complexity of this method is practically constant.  However, because it relies on a good initial guess, the performance of the Lin-Canny method can be somewhat poor if the objects move rapidly.  More recent techniques like [H-walk](ftp://cs.stanford.edu/cs/robotics/dyhsu/papers/socg99.pdf) improve on this idea by combining it with fast data structures for linear programming queries, such as the Dobkin-Kirkpatrick hierarchy to get more robust performance:

L. Guibas, D. Hsu, L. Zhang. (1999) “[H-Walk: Hierarchical distance computation for moving bodies](ftp://cs.stanford.edu/cs/robotics/dyhsu/papers/socg99.pdf)” SoCG

Algebraic and semialgebraic sets

Outside of convex polytopes, the situation for resolving narrow phase collisions efficiently and exactly becomes pretty hopeless.  For [algebraic sets](https://en.wikipedia.org/wiki/Algebraic_variety) like [NURBS](https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline) or [subdivision surfaces](https://en.wikipedia.org/wiki/Subdivision_surface), the fastest known methods all reduce to some form of approximate root finding (usually via bisection or Newton’s method).  Exact techniques like [Grobner basis](https://en.wikipedia.org/wiki/Gr%C3%B6bner_basis) are typically impractical or prohibitively expensive.  In constructive solid geometry working with [semialgebraic sets](https://en.wikipedia.org/wiki/Semialgebraic_set), it is even worse where one must often resort to general nonlinear optimization, or in the most extreme cases fully symbolic [Tarski-Seidenberg quantifier elimination](https://en.wikipedia.org/wiki/Tarski%E2%80%93Seidenberg_theorem) (like the [cylindrical algebraic decomposition](https://en.wikipedia.org/wiki/Cylindrical_algebraic_decomposition)).

Measure theoretic methods

I guess I can say a few words about some of my own small contributions here.  An alternative to computing the distance between two shapes for testing separation is to compute the volume of their intersection, \mu(S \cap T).  If this volume is >0, then the shapes collide.  One way to compute this volume is to rewrite it as an integral.  Let 1_S denote the [indicator function of [S](https://en.wikipedia.org/wiki/Indicator_function)](https://en.wikipedia.org/wiki/Indicator_function), then

\mu(S \cap T) = \int \limits_{\mathbb{R}^d} 1_S(x) 1_T(x) dx

This integral is essentially a dot product.  If we perform an expansion of 1_S, 1_T in some basis, (for example, as Fourier waves), then we can use that to approximate this volume.  If we do this expansion carefully, then with enough work we can show that the resulting approximation preserves something of the original geometry.  For more details, here is a paper I wrote:

M. Lysenko, (2013) “[Fourier Collision Detection](http://intl-ijr.sagepub.com/content/early/2013/02/19/0278364913477165)” International Journal of Robotics Research

The advantage to this type of approach to collision detection is that it can support any sort of geometry, no matter how complicated. This is because the cost of the testing intersections scales with the accuracy of the collision test in a predictable, well-defined way.  The disadvantage though is that at high accuracies it is slower than other exact techniques.  Whether it is worthwhile or not depends on the desired accuracy, the types of shapes involved and if additional information like a separating axis is needed and so on.

Broad phase

Given a fast narrow phase collision test, we can solve the broad phase collision detection problem for n objects in O(n^2) calls to the underlying test.  As the number of reported collisions could be O(n^2) in the worst case, this would seem optimal.  However, we can get a sharper picture using a more detailed [output sensitive analysis](https://en.wikipedia.org/wiki/Output-sensitive_algorithm).  To do this, define k to be the number of reported intersections, and let us then analyze the time required to do the collision detection as a function of both n and k.  Using output sensitive analysis, there is also a lower bound of \Omega(n \log(n) + k) (for comparison based algorithms) by reduction to the [element uniqueness problem](http://en.wikipedia.org/wiki/Element_distinctness_problem).

Special cases

If we are only allowed to use pairwise intersection tests and know no other property of the shapes, then it is impossible to compute all pairwise intersections any faster than O(n^2 + k).  However, for special types of shapes in low dimensional spaces substantially faster algorithms are known:

Line segments

For line segments in the plane, it is possible to report all intersections in O((n+k) \log(n)) time using a [sweep line algorithm](https://en.wikipedia.org/wiki/Bentley%E2%80%93Ottmann_algorithm).  If k \in o( \frac{n^2}{\log(n)} ), this is a big improvement over naive brute force.  This technique can also be adapted to compute intersections in sets of general convex polygons by decomposing them into segments, and then building a secondary point location data structure to handle the case where a polygon is completely contained in another.

Uniformly sized and distributed balls

It is also possible to find all intersections in a collection of balls with constant radii in optimal O(n + k) time, assuming that the number of balls any single ball intersects is at most O(1).  The key to this idea is to use a grid, or hash table to detect collisions. This process is both simple to implement and has robust performance, and so it is used in many simulations and video games.

Axis aligned boxes

Finally, it is possible to detect all intersections in a collection of axis aligned boxes in O(n \log^d(n) + k) time, though we will postpone talking about this until next time.

General objects and bounding volumes

For general objects, no algorithms with running time substantially faster than O(n^2 + k) are known.  However, we can in practice still get a big speed up by using a simpler broad phase collision test to filter out intersections.  The main idea is to cover each object, S, with some simpler shape S' \supseteq S called a [bounding volume](https://en.wikipedia.org/wiki/Bounding_volume).  If a pair of bounding volumes do not intersect, then the shapes which they are covering can not intersect either.  In this way, bounding volumes can be used to prune down the set of collision tests which must be performed.

In practice, the most common choice for a bounding volume is either a box or a sphere.  The reason for this is that boxes and spheres support efficient broad phase intersection tests, and so they are relatively cheap.

Spheres tend to be more useful if all of the shapes are more or less the same size, but computing tight bounding spheres is slightly more expensive.  For example, if the objects being intersected consist of uniformly subdivided triangle meshes, then spheres can be a good choice of bounding volume.  However, spheres do have some weakness.  Because testing sphere intersection requires multiplication, it is harder to do it exactly than it is for boxes.  Additionally, for spheres of highly variable sizes it is harder to detect intersections efficiently.

Computing intersections in boxes on the other hand tends to be much cheaper, and it is simpler to exactly detect if a pair of boxes intersect.  Also for many shapes boxes tend to give better approximations than spheres, since they can have skewed aspect ratios.  Finally, broad phase box intersection has theoretically more robust performance than sphere intersection for highly variable box sizes.  Perhaps based on these observations, it seems that most modern high performance physics engines and intersection codes have converged on axis-aligned boxes as the preferred primitive for broad phase collision detection.  (See for example, [Bullet](http://bulletphysics.org/wordpress/), [Box2D](http://box2d.org/))

Bipartite vs complete

It is sometimes useful to separate objects for collision detection into different groups.  For example if we are intersecting water-tight meshes, it is useless to test for self intersections.  Or as another example, in a shooter game we only need to test the player’s bullets against all enemies.  These are both examples of **bipartite collision detection.**In bipartite collision detection, we have two groups of objects (conventionally colored red and blue), and the goal is to report all pairs of red and blue objects which intersect.

Range searching and more references

There is a large body of literature on intersection detection and the related problems of range searching.  Agarwal and Erickson give an excellent survey of these results in the following paper,

P.K. Agarwal, J. Erickson. (1997) “[Geometric range searching and its relatives](http://web.engr.illinois.edu/~jeffe/pubs/pdf/survey-tr.pdf)“

Next time

[In the next article](https://0fps.net/2015/01/18/collision-detection-part-2/), we will look at broad phase collision detection in more depth, focusing on boxes as a basic primitive.