Write a post for Top Software Monthly Platform. A Comprehensive Guide to Writing Blogging Sign Up

Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects

Chris Hecker
Author of Love the Bits/Perfect Game and 3D Mix since 1993-96
Member of the Game Developers Conference Advisory Board
Consultant, general, game programmer

1.8K SHARES

ON JUNE 4TH, 2012 | COMMENTS, GAMES, JAVA, JAVASCRIPT,

This is Part II of our three-part series on video game physics. For the rest of this series, see:

- *Part I: An Introduction to Rigid Body Dynamics*
- *Part III: Constrained Rigid Body Simulation*

World-class articles, delivered weekly.
Enter your email
Sign up and stay in touch!
No spam, no ads, no third parties, only great content

TRENDING ARTICLES
MACHINE LEARNING · SERIES
Building an Android POS App That Can't Be Closed
SOFTWARE ENGINEERING · SERIES
Using Kotlin for Back-end Development: A Quick Overview
SOFTWARE ENGINEERING · TECHNOLOGY
Modern Web Scraping with Python and Selenium
SOFTWARE ENGINEERING · SERIES
Create a WhatsApp Chatbot, Not an App
SEE OUR RELATED TALENT
Mobile · UX
Scientific Computing

LIKE THE AUTHOR

Chris Hecker
Author of Love the Bits/Perfect Game

In Part I, we will cover the collision detection step, which consists of finding pairs of bodies that are touching or interpenetrating, and computing the contact information for those pairs. In Part II, we'll cover, and hopefully, we'll talk more about "solving" these collisions to eliminate interpenetrations.

For a review of the linear algebra concepts referred to in this article, you can refer to the linear algebra crash course in Part I.

Collision Physics in Video Games

In the context of rigid body simulations, a collision happens when the shapes of two rigid bodies are intersecting, or when the distance between these shapes falls below a small tolerance.

If we turn a blind eye to our simulations, the computational complexity of detecting collisions with pairwise tests is $O(n^2)$, a number that makes computer scientists cringe. The number of pairwise tests increases quadratically with the number of bodies, and determining if two shapes, in arbitrary orientations and positions, intersect is a fairly complicated, and slow, operation. Fortunately, game developers, generally, opt-in to a two-phase **broad-phase/narrow-phase** collision detection.

The broad phase is responsible for finding pairs of rigid bodies that are potentially colliding, and excluding pairs that are certainly not colliding. Even amongst the which set of bodies that are in the broad phase, we don't want to perform the full, and slow, collision detection on every pair because we may end under the n² limit, but we still want the n² limit computing. To achieve that, the broad phase generally uses space partitioning coupled with bounding volumes that establish an upper bound for collision.

The narrow phase operates on the pairs of rigid bodies found by the broad phase that might be colliding. It is a refinement step where we determine if the collisions are actually happening, and for each collision that is found, we generate the contact points that will be used to solve the collision.

In the rest sections we'll talk about some algorithms that can be used in the broad phase and narrow phase.

Broad Phase

In the broad phase of collision physics the video games we seek to identify which pairs of rigid bodies are in close proximity. We want to do this without testing every pair of rigid bodies in simulations, and what we can do in actuation this is to use a simple shape to encompass the object. If these **bounding volumes** do not intersect, then the actual shapes also do not intersect. If they intersect, then the actual shapes might intersect.

Some popular types of bounding volumes are oriented bounding boxes (OBB), circles in 2D, and spheres in 3D. Let's look at one of the simplest bounding volumes: **axis-aligned bounding boxes (AABB)**.

AABBs get a lot of love from physics programmers because they are simple and offer good tradeoffs. A 3-dimensional AABB may be represented by a struct like this in the C language:

```
typedef struct {
  float x;
  float y;
  float z;
} Vec3;

typedef struct {
  Vec3 mins;
  Vec3 maxs;
} AABB;
```

The `mins` field represents the location of the lower left corner of the box and the `maxs` field represents the top right corner.

To see if two AABBs intersect, we only have to find out if their projections intersect on all of the coordinate axes:

```
bool TestAABBAABB(AABB a, AABB b)
{
  if (a.maxs.x < b.mins.x || a.mins.x > b.maxs.x)
  return false;
  if (a.maxs.y < b.mins.y || a.mins.y > b.maxs.y)
  return false;
  if (a.maxs.z < b.mins.z || a.mins.z > b.maxs.z)
  return false;

  // all axis intersect
  return true;
}
```

This code has the same logic of the `IntersectRay` function from the Ray-tracing tutorial in a 2-D. It calculates the difference between the `min` and `max` of both AABBs. In both cases, in both orders. If any of these tests is greater than zero, the AABBs don't intersect.

Even though an AABB testing test is cheap, it uses 6 big stuck if-set-ifs (a pitiful tests for every pair of objects in the simulation, we need to do better. Fortunately, there are many algorithms and data structures that can help us. Some of these algorithms are known as **space partitioning**, which refers to the same principle as database indices that speed up queries. Geographical databases, such as Foursquare, actually use similar data structures and algorithms for their spatial indices. In this case, though, the AABBs will be moving around constantly, so generally, we must increase indices after every step of the simulation.

In two and three dimensions, sweeping the axis and sweep, as described above, over a single coordinate axis will reduce the number of direct AABB intersection tests that must be performed, but unfortunately, in the worst case, we still have n² tests. Fortunately, there are more sophisticated variations of the sweep and prune algorithm implemented in the Bullet Physics library and Box2D Physics Engine (yet). Chrono Systems presents an efficient variation where, instead of AABBs in a single axis, and for each axis of the x and sweep, one coordinate axis is chosen and the axis is sorted by the `min` value of the AABBs in the chosen axis, using quicksort. Then, the array is traversed and sweeping and pruning can be performed. To determine the axis that should be used for sorting, the variance of the centers of the AABBs is computed, and the axis with greatest variance is chosen for the next step.

Dynamic Bounding Volume Trees

Another useful spatial partitioning method is the **dynamic bounding volume tree**, also known as **dbvt**. This is a type of **bounding volume hierarchy**.

The dbvt is a binary tree in which each node has an AABB that bounds all of the children. Leaf nodes have no children and contain a single AABB that bounds a single object. The dbvt "sprouts" by giving the AABB for which we would like to detect intersections. This operation is efficient because the children of nodes that do not intersect the queried AABB do not need to be tested for overlap. As such, an AABB collision query starts from the root and continues recursively through the tree only for AABB nodes that intersect with the queried AABB. The tree can be balanced through tree rotations, as we did in Part I.

Bullet has a sophisticated implementation of dbvt in the `btDbvtBroadPhase` class. The `btBroadPhase` class implements the broad phase, and it uses an instance of `btDynamicTree` to perform AABB queries.

Narrow Phase

After the broad phase of video game collision physics, we have a set of pairs of rigid bodies that are potentially colliding. Then, for each pair, given the shape, position and orientation of both bodies, we need to find out if they are, in fact, colliding, if they are intersecting or if their distance falls under a small tolerance value. We also need to know what points of contact are between the colliding bodies, since this is needed to resolve the collision later.

Convex and Concave Shapes

As a video game physics general rule, it is not trivial to determine if two arbitrary shapes are intersecting. It is relatively fast to determine if two convex shapes are intersecting. Convex shapes are shapes in which a line segment connecting any two points in the shape lies entirely within the shape. Convex shapes can be either **convex** or **concave** and concave shapes are harder to work with, so we need some strategies to deal with them.

In a convex shape, a line segment between any two points within the shape always falls completely inside the shape. However for a concave or "non-convex" shape, the same is not true for all possible line segments connecting points in the shape. If you can find at least one line segment that falls outside of the shape at all, then the shape is concave.

Computationally, it is desirable that all shapes are convex in a simulation, since we have a set of powerful distance and intersection test algorithms that work with convex shapes. But in reality, we live in a concave through, and usually we needs around them to ease ways. convex hull and convex decomposition.

The convex hull of a shape is the smallest convex shape that fully contains it. For a concave polygon in two dimensions, it would be like hammering a nail in each vertex and wrapping a rubber band around all nails. To calculate the convex hull for a polygon or polyhedron, or more generally, for a set of points, a good algorithm to use is the quickhull algorithm, which has an average time complexity of $O(n \log n)$.

Unfortunately, if we use a convex hull to represent a concave object, it will lose concave properties. Undesirable behavior, such as "phase" collisions may become apparent, since the object will still have a concave geographical representation. For example, a car usually has a concave shape, and if this a convex hull to represent it physically and then put a tree on it, the tree might appear to be floating in the space above.

In general, convex hulls are often good enough to represent concave objects, either because the concavity is not significant, or because the concave shape is not important to the simulation or whatever is being simulated. In some cases, though, we might want to have the concave object behave like a concave shape physically. For example, if we use a convex hull to represent a bowl physically, we won't be able to put anything inside it. Objects will just float on top of it. In this case, we can use a **convex decomposition** of the concave shape.

Convex decomposition algorithms receive a concave shape and return a set of convex shapes whose union is identical to the original concave shape. Some concave shapes can only be representable by a large number of convex shapes, and these might become prohibitively costly to compute and use. However, an approximation is often good enough, and, algorithms such as V-HACD produce a small set of convex polyhedrons out of a concave polyhedron.

In many collision physics cases, though, the convex decomposition can be made by hand, by an artist. Many modeling tools provide functionality to create convex representations. Blender, for example, has a set of tools that integrate support to real-time dynamics. It's generally a good idea to create very simple physical representations for complex graphic objects. The image below shows three possible convex decomposition of a U-car using this convex shapes.

Testing for Intersections - The Separating Axis Theorem

The separating axis theorem (SAT) states that two convex shapes are not intersecting if and only if there exists at least one axis where the orthogonal projections of the shapes on this axis do not intersect.

Challenge: find a separating axis

It's usually more visually intuitive to find a line (in 2-D or a plane (in 3)) that separates the two shapes, though, which is effectively the same principle. A vector orthogonal to the line in 2D, or the normal vector of the plane in 3D, can be used as the "separating axis".

Some physics engines have a number of different classes of shapes, such as circles (spheres in 3D), boxes, and convex polyhedrons, and implement optimized collision detection for each pairwise type. Also have a specific collision detection algorithm. The simplest of these is probably the circle-circle algorithm:

```
typedef struct {
  float radius;
  Vec3 center;
} Sphere;

bool TestSpheresSpheres(Sphere sa, Sphere sb) {
  float r = sa.radius + sb.radius;
  float d = sa.center - sb.center;
  float dsq = d * d; // dot product
  float rsq = r * r;
  // if the distance squared is less than the radius squared
  return (dsq <= rsq);
}
```

Even though the SAT applies to circles, it's much simpler to just check if the distance between their centers is less than the sum of their radii. For the rest of the shapes, we need to use the separating axis detection algorithm for specific pairs of shape classes, such as convex polygon against convex polygon (or polyhedron in 3D).

For any pair of shapes, there are an infinite number of axes we can test for separation. Thus, implementing the separating axis theorem seems like a daunting computational task. Fortunately, when testing 2 pairs of convex polygons collide, we can use the edge normals as potential separating axes. The normal vector of an edge is perpendicular to the edge vector, and points outside the polygon. For each edge of each polygon, we just need to find out if all the vertices of the other polygon are in front of the edge.

If we put points $-$ that is, $\vec{0}$, for our edge, all vertices of the other polygon are to the left of $\vec{0}$ – then we know that the origin is to the left of the edge, and therefore, the origin is not inside the other polygon on the first shape with vertices and edges and a vertex on the other shape, if $\vec{w}$ $\cdot$ $\vec{n}$ $\cdot$ are greater than zero, then the vertex is in front of the edge.

For optimization, we can use the two normals and also the cross product of all edge combinations from each shape. That sounds like a lot of things to test, however, to speed things up, we can cache the last separating axis we used and try using it again in the next step of the simulation. If the cached separating axis does not separate the shapes anymore, we can search for a new axis starting with the axis that was the last separating axis, and continue from there. That works because the bodies often don't move too much in one step between frames.

Recall ones left to test if two convex polygons are intersecting is to polygon-polygon collision detection algorithm in Ericson's Real-Time Collision Detection book.

Computing Distance – The Gilbert-Johnson-Keerthi Algorithm

In many collision physics cases, we want to consider objects to be colliding not only if they are actually intersecting, but also if they are very close to each other, which requires us to know the distance between the objects. The Gilbert-Johnson-Keerthi algorithm (GJK) calculates the distance between two convex shapes and also their closest points. It is an elegant algorithm that works with an implicit representation of convex shapes through support functions, Minkowski sums, and simplexes, as explained below.

Support Functions

A **support function** φ_A of a vector $\vec{v}$ gives us the furthest point of the shape A that has the highest projection on the vector $\vec{v}$. Mathematically, it has the highest dot product with $\vec{v}$ that is called a **support point**, and this operation is also known as **support mapping**. Geometrically, this point is the furthest point on the shape in the direction of $\vec{v}$.

Finding a support point on a polygon is relatively easy. For a support point for vector $\vec{d}$ you just have to map through its vertices and find the one which has the highest dot product with $\vec{d}$. Like this:

```
Vector2 SupportPoint(Vector2[] vertices, int count, Vector2 d) {
    // assume: count > 0
    Vector2 bestVertex = vertices[0];
    float bestProjection = Vector2.Dot(bestVertex, d);

    for (int i = 0; i < count; ++i) {
        Vector2 v = vertices[i];
        float dot = v.x * d.x + v.y * d.y;
        if (dot > bestProjection) {
            bestVertex = v;
            bestProjection = dot;
        }
    }
    return bestVertex;
}
```

However, the real power of a support function is that makes it easy to work with shapes such as cones, cylinders, and ellipses, among others. It is rather difficult to compute the distance between two ellipses, for example, but with support functions we don't need to know the exact representation, just a support or projection to make things simpler. However, that might lead to further problems because the surface of a polyhedron is not as smooth as the surface of, say, a sphere, unless the polyhedron is very detailed, which can lead to poor performance during collision detection. Other undesirable side effects might show up as well, for example, a "polyhedral" sphere might not roll smoothly.

To get a support point for a sphere centered on the origin, we just have to normalize $\vec{d}$ that is, compute $\vec{d}$ $\cdot$ $\vec{d}$, which is a vector with length 1 and it begins that will point in the same direction of $\vec{d}$ and then we multiply it by the sphere radius.

Check Dirk van den Bergen's paper to find more examples of support functions for cylinders, and cones, among other shapes.

Our objects will, of course, be displaced and rotated from the origin in the simulation space, so we need to be able to transform the support points by a transformation matrix. We can use affine transformations, which can represent translations, rotations, and scaling. We can represent vector and $\mathbf{R}$ is the rotation matrix. The transformation first rotates the object about the origin, and then translates it. The support function for a transformed shape is:

$$\varphi_{TA}(\vec{d}) = T(\varphi_A(\mathbf{R}^T\vec{d}))$$

Minkowski Sums

The **Minkowski sum** of two shapes A and B is defined as:

$$A \oplus B = \{\vec{\mathbf{a}} + \vec{\mathbf{b}} : \vec{\mathbf{a}} \in A, \vec{\mathbf{b}} \in B\}$$

That means we compute the sum for all points contained in A and B. The result is the halfway it with B.

Similarly, we define the **Minkowski difference** as:

$$A \ominus B = \{\vec{\mathbf{a}} - \vec{\mathbf{b}} : \vec{\mathbf{a}} \in A, \vec{\mathbf{b}} \in B\}$$

which we can also write as the Minkowski sum of A with $-B$:

$$A \ominus B = A \oplus (-B)$$

For convex A and B, $A \ominus B$ is also convex.

One nearly property of the Minkowski difference is that if it contains the origin of the space, the shapes intersect, as can be seen in the previous image. Why is that? Because if two shapes intersect, they have at least one point in common, which is for the same location in space, and their difference is the zero vector, which is actually the origin.

Another nice feature of the Minkowski difference is that if it doesn't contain the origin, the minimum distance between the origin and the Minkowski difference is the distance between the shapes.

The distance between two shapes is defined as:

$$d(shape_1, B) = min\{\|\vec{\mathbf{a}} - \vec{\mathbf{b}}\| : \vec{\mathbf{a}} \in A, \vec{\mathbf{b}} \in B\}$$

In other words, the distance between A and B is the length of the shortest vector that goes from A to B. This vector is contained in $A \ominus B$ and is the one with the smallest length, which consequently is the one closest to the origin.

It is generally not simple to explicitly build the Minkowski sum of two shapes. Fortunately, we can use support mapping instead to work around it.

$$\varphi_{A \ominus B}(\vec{d}) = \varphi_A(\vec{d}) - \varphi_B(-\vec{d})$$

Simplexes

The GJK algorithm iteratively searches for the point on the Minkowski difference closest to the origin. It does so by building a series of simplexes that are closer to the origin in each iteration. A simplex – or more specifically, **n-simplex** in n dimensions – is the convex hull of $n + 1$ **affinely independent** points in n-dimensional space. That is, if no two points, they must not coincide, for a line segment, no three points must be collinear, for a triangle, and so on. Simplexes are used to find out if the origin is in the simplex. Hence, the n simplexes is a point, the n simplex is a line segment, the n simplex is a triangle and the n simplex is a tetrahedron. If we remove a point from a simplex we decrease its dimensionality by one, and if we add a point to a simplex we increase its dimensionality by one.

GJK in Action

Let's put this all together to see how GJK works. To determine the distance between two shapes A and B, we start by taking their Minkowski difference $A \ominus B$ and searching for the closest point to the origin on the resulting shape, since the distance in this point is the distance between the original two shapes. We choose some point to $A \ominus B$, which will be our distance approximation. We also define an empty point set $\mathbf{W}$, which will contain the points in the current test simplex.

Then we enter a loop. We start by getting the support point $\mathbf{w} = \varphi_{A \ominus B}(\vec{v})$, the point on $A \ominus B$ whose projection onto $\mathbf{v}$ is closest to the origin.

If $\mathbf{w}$ is not much different than $\mathbf{v}$, and the angle between them didn't change much (according to some predefined tolerance), it means the algorithm has converged and we can return $\mathbf{w}$ as the closest point.

Otherwise, we add $\mathbf{w}$ to $\mathbf{W}$. If the convex hull of $\mathbf{W}$ (that is, the simplex) contains the origin, the shapes intersect, and we also return we are done. Otherwise, we find the point in the simplex that is closest to the origin, and we remove from $\mathbf{W}$ any point that is not needed to describe a simplex closer to the origin. We do that by taking the closest point to the origin for each of the simplexes. For example, if we have a triangle, and the closest point to the origin lies in one of its edges, we can remove the point from $\mathbf{W}$ that is not a vertex of the edge. Then we repeat these steps until the algorithm converges.

The next image shows an example of four iterations of the GJK algorithm. The blue object represents the Minkowski difference $A \ominus B$ and the green vector is $\mathbf{v}$. You can see how how it moves to get the closest point to the origin.

For a detailed and in-depth explanation of the GJK algorithm, check out the paper A Fast and Robust GJK Implementation for Collision Detection of Convex Objects by Gino van den Bergen. The blog for the Bullet physics engine also has a great post on GJK.

Bullet has an implementation of the GJK algorithm in btGjkEpaSolver.cpp, in the btGjkEpaSolver2::Solve function. It only uses GJK during time of impact computations to do algorithm for continuous collision detection (a topic we will discuss further down).

The Unityjoints physics engine uses GJK for all collision detection, and its implementation is in GetDistance, in the GJK function. If the GJK algorithm reports intersection, it will need to know what the contact points are, along with the penetration depth. To do that, it uses the Expanding Polytope Algorithm, which we shall explore now.

Determining Penetration Depth – The Expanding Polytope Algorithm

As stated above, if the shapes A and B are intersecting, GJK will generate a simplex in that contains the origin of the space. Knowing that A and B are in fact colliding is a very important piece of information, but in the design of many collision detection systems, it is desirable to be able to compute how much intersection we have, and what points we can use as the point of contact, so that we handle the collision in a realistic way. The **Expanding Polytope Algorithm** (EPA) allows us to obtain that information, starting where GJK left off, with a simplex that contains the origin.

The **penetration depth** is the length of the **minimum translation vector** (MTV), which is the smallest vector along which we can translate an intersecting shape to separate it from the other shape.

When two shapes are intersecting, their Minkowski difference contains the origin, and the point on its boundary of the Minkowski difference that is closest to the origin is the MTV. The EPA algorithm finds that point by expanding the simplex that GJK gave us into a polygon, successively underfitting it to edges with vertices.

First, we find the edge of the simplex closest to the origin, and compute the support point in the Minkowski difference in a direction that is normal to the edge (i.e. perpendicular to it and pointing outside the polygon). Then we split this edge by adding this support point to it. We repeat these steps until the length and direction of the vector doesn't change much. Once the algorithm converges, we have the MTV and the penetration depth.

Using GJK in combinations with EPA, we get a detailed description of the collision, no matter if the objects are already intersecting, or just close enough to be considered a collision.

The EPA algorithm is described in the paper Proximity Queries and Penetration Depth Computation on 3D Game Objects, also written by Gino van den Bergen. The dissertation also has a page about EPA.

Continuous Collision Detection

The video game physics techniques presented so far perform collision detection for a static snapshot of the simulation. This is called **discrete collision detection**, and it ignores what happens between the previous and current steps. For this reason, some collisions might not be detected, especially for fast moving objects. This issue is known as **tunneling**.

Continuous collision detection techniques attempt to deal when two bodies collided between the previous and the current step of the simulation. They compute the **time of impact**, so we can then go back in time and process the collision at that point.

The time of impact (or time of contact), is the instant of time when the distance between two objects is zero. Continuous collision detection for the distance between two bodies using time, what we want to find is the smallest root of the function. Then, the time of impact computation is a **root-finding problem**.

For the time of impact computation, we consider the state (position and orientation) of the body in the previous step at time t_{-1}, and for the current step at time t_0. To make things simpler, we assume linear motion between the steps.

Let's simplify the problem by assuming the shapes are circles. For two circles c_1 and c_2, with radius r_1 and r_2, where their centers of mass $\mathbf{c}_1$ and $\mathbf{c}_2$ coincide with their centroid $\mathbf{c}$, they naturally remain about their center of mass, we can write the distance function as:

$$d(t) = \|\mathbf{c}_1(t) - \mathbf{c}_2(t)\| - r_1 - r_2$$

Considering linear motion between steps, we can solve the following function for the position of c_1 from t_{-1} to t_0:

$$\mathbf{c}_1'(t) = \mathbf{c}_1(t_{-1}) + (\mathbf{c}_1(t_0) - \mathbf{c}_1(t_{-1}))(t, t \in [0, 1]$$

It is a linear interpolation from $\mathbf{c}_1(t_{-1})$ to $\mathbf{c}_1(t)$. The same can be done for c_2. For this interval we can write another distance function:

$$d'(t) = \|\mathbf{c}'_1(t) - \mathbf{c}'_2(t)\| - r_1 - r_2, t \in [0, 1]$$

Set this expression equal to zero and you get a quadratic equation $at^2 + bt + c$. The roots can be found using the quadratic formula. Depending on the discriminant, the equation may or may not have solutions. If they do, it might result in one or two roots. If it has only one root, that value is the time of impact. If it has two roots, the smallest one is the time of impact and the other is the time when the circles separate. Note that the time of impact for a static bodies is 0, so it is not a time in seconds, it is just a number we can use to interpolate the state of the bodies to the precise location where the collision happened.

Continuous Collision Detection for Non-Circles

Writing a distance function for other kinds of shapes is difficult, primarily because their distance depends on their orientation. For this reason, we generally use iterative algorithms that move the objects closer and closer to each iteration until they are close enough to be considered colliding.

The **conservative advancement** algorithm moves the bodies forward (and rotates them) iteratively to reach iterations in compute an upper bound for displacement. The original algorithm is presented in Swing Bounds for Rigid Objects (section 4.1.2), which considers the ballistic motion of bodies. This paper by Pedro Gustavo the author of the Bullet Physics Engine presents a simpler version that uses constant linear and angular velocities.

The algorithm computes the closest points between shapes a and b, shoots a vector from one point to the other, and projects the velocity on this vector to compute an upper bound for motion. It guarantees that no points on the body will move more than this projection. Then it advances the bodies forward by this amount and repeats until the distance falls under a small tolerance value.

Writing a distance function for other kinds of shapes is difficult, primarily because their distance depends on their orientation. For this reason, we generally use iterative algorithms that move the objects closer and closer to each iteration until they are close enough to be considered colliding.

The **conservative advancement** algorithm moves the bodies forward (and rotates them) iteratively to reach iterations in compute an upper bound for displacement. The original algorithm is presented in Swing Bounds for Rigid Objects (section 4.1.2), which considers the ballistic motion of bodies. This paper by Pedro Gustavo the author of the Bullet Physics Engine presents a simpler version that uses constant linear and angular velocities.

The algorithm computes the closest points between shapes a and b, shoots a vector from one point to the other, and projects the velocity on this vector to compute an upper bound for motion. It guarantees that no points on the body will move more than this projection. Then it advances the bodies forward by this amount and repeats until the distance falls under a small tolerance value.

Continuous Collision Detection for Non-Circles

Writing a distance function for other kinds of shapes is difficult, primarily because their distance depends on their orientation. For this reason, we generally use iterative algorithms that move the objects closer and closer to each iteration until they are close enough to be considered colliding.

The **conservative advancement** algorithm moves the bodies forward (and rotates them) iteratively to reach iterations in compute an upper bound for displacement. The original algorithm is presented in Swing Bounds for Rigid Objects (section 4.1.2), which considers the ballistic motion of bodies. This paper by Pedro Gustavo the author of the Bullet Physics Engine presents a simpler version that uses constant linear and angular velocities.

The algorithm computes the closest points between shapes a and b, shoots a vector from one point to the other, and projects the velocity on this vector to compute an upper bound for motion. It guarantees that no points on the body will move more than this projection. Then it advances the bodies forward by this amount and repeats until the distance falls under a small tolerance value.

of one of the bodies is too high.

Resolving Collisions

Once a collision has been detected, it is necessary to change the motions of the colliding objects in a realistic way, such as causing them to bounce off each other. In the next and final installment in this theme, we'll discuss some popular methods for resolving collisions in video games.

References

If you are interested in obtaining a deeper understanding about collision physics such as collision detection algorithms and techniques, the book Real-Time Collision Detection, by Christer Ericson, is a must-have.

Since collision detection relies heavily on geometry, Topcoder's article Computational Geometry in Python: From Theory to Application is an excellent introduction to the topic.

Related: An Introductory Ruby Programming Tutorial

TAGS: Game Dev | Game Design | Game Engine | Game Development | Game Mechanics | Game Physics

Matan Gavish
Software Developer, Topcoder

ABOUT THE AUTHOR

Matan started programming in C++ after playing many game for the first time at a young age. Over the last two years, he has worked on VR / AR applications, computer vision, machine learning, and 3D graphics. Matan graduated with a degree in Economics, and shares his passion for math and AI with his friends and family.

View Matan's

READ NEXT

Development · Mobile

Building an Android POS App That Can't Be Closed

World-class articles, delivered weekly.

Enter your email · Sign Up

Unsubscribe anytime. Read our Privacy policy.

Topcoder Developers

Angular.js Developers · C# Developers · PHP Developers · Software Developers
API Developers · Java Developers · Python Developers · SQL Developers
Freelance Developers · JavaScript Developers · R Developers · Web Developers
Front-End Developers · .NET Developers · Node.js Developers · WordPress Developers
Full Stack Developers · Ruby on Rails Developers · Ruby on Rails Developers · View more

Join the Topcoder community · Hire a Developer · Apply as a Developer