Minkowski sums and differences

posted by harrison on September 28, 2012

There are many ways to do collision detection, but a fairly general one is Minkowski differences. The idea is that you do a binary operation on two shapes to get a new shape, and if the origin (the zero vector) is inside that shape, then they are colliding. The minkowski sum lets you define some interesting shapes that are both easy to draw and easy to do collision detection on.

The Minkowski sum

What does it mean to add shapes?

One representation of a shape is a (possibly infinite) set of points. so, a point is just a set with one element, and a circle is the set $\{v| ||v - c|| \leq r\}$, or the set of all points within radius r of a centre point c.

The minkowski sum of A and B is the set of all points that are the sum of any point in A and B. The formula is:

$$A \oplus B = \{a + b | a \in A, b \in B\}$$

Some examples

To instill you with intuition of what a minkowski sum looks like, here are a few examples:

The sum of any shape and a point is that shape translated by that point.

The sum of any shape and two points is two translated (possibly overlapping) copies of that shape.

The sum of two circles is a larger circle (sum the radii) with its centre at the sum of the centres of the smaller circles.

The sum of any shape and a line is that shape swept through that line. Think of placing your shape in sand, and dragging it along the line.

Similarly, the sum of a shape and any curve is what you'd get by sweeping the shape through the curve.

The sum of two parallel lines is a longer line.

For perpendicular lines, you get a square.

Minkowski difference

The minkowski difference is what you get by taking the minkowski sum of a shape and the mirror of another shape. So, your second shape gets flipped about the origin (all of its points are negated).

$$A \ominus B = \{a - b | a \in A, b \in B\}$$

And if we ask “are these objects colliding?”, we’re really asking if they have any points in common. If they do have a point p in common, then $p - p = 0$ must be in the Minkowski difference. If they do not share a point, then $a - b$ is never 0, because $a \neq b$. So, we have reduced the collision detection problem to a Minkowski sum problem.

Associativity and Commutativity

The Minkowski sum is associative and commutative, so$(A \oplus B) \oplus C = A \oplus (B \oplus C)$ and $A \oplus B = B \oplus A$. This means that we can rearrange sums to make our calculations easier.

So, let’s say you have two circles C_k of radius r_k with centre at point c_k and a square S of width w at centre s and at angle θ, and you want to find $C_0 \oplus S \oplus C_1$. that not so fun, but you can think about it more easily if you rearrange it as $(c_0 \oplus c_1 \oplus s) \oplus ((C'_0 \oplus C'_1) \oplus S')$, where C'_k is a circle at the origin with radius r_k, and S' is the square centred at the origin. also, you can further decompose S' into the sum of two line segments, so then you’re just sweeping a circle through a line, and then sweeping the resultant capsule through another line, and then you can translate the whole thing.

The Minkowski difference is not commutative, because subtraction is not commutative. It is anticommutative, though, which is just about as good. Any time you flip the order of a difference, you have to negate the result. It’s the same as regular subtraction that way.

Convex polygons and lines

This comes up frequently because you will often want to extrude your shapes along their direction of motion, so that if you’re checking for collisions at regular intervals, fast-moving objects won’t be able to jump over other objects in a single step.

The sum of a polygon and a line is an interesting case, both because it’s simple to compute, and because it comes up naturally in collision detection. Suppose you have a polygon represented by line segments, and each line segment has a normal point out. If you take a new line segment L, you can split the polygon’s edges into three groups: those parallel to L, those with their normal pointing the same direction as L , or $n \cdot (L_{end} - L_{start}) > 0$, and those with their normal pointing the opposite direction. If there are no parallel edges, then 2 of the points between edges on the polygon will be between an opposite and a same direction edge. You can insert the new line segment on those points, or extend the existing parallel lines, and then you’ll have a new polygon.

This doesn’t work quite right with non-convex polygons, because after the operation, the new polygon can intersect itself. You could modify the method to fix the polygon afterward, but it’s gross, and not something you want to be doing in your physics calulations.

Higher dimensions

The Minkowski difference method will work in any number of dimensions. There’s also a technique to improve on the sweep-test method described above.

Say you’re working in 2D, and you have fast-moving objects. Sometimes, they will cross paths in a single time step, but they shouldn’t actually collide. if you did a sweep test, they would get marked as having collided, though. One way you can solve this is by going up a dimension to 3D, with time as the third dimension, so instead of extruding in the direction of the velocity, you can extrude in the direction of the velocity with the time component set to 1, instead of 0.

You can cheat a little bit here and bring it back down to a 2D problem by noticing that you only have to worry about the slice of the sum where time = 0. You can rearrange the sum $(S_0 \oplus V_o) \ominus (S_1 \oplus V_0)$ into $(S_0 \ominus S_1) \oplus (V_0 \ominus V_1)$, and then we can look at $(V_0 \ominus V_1)$, which is a parallelogram, and find the line segment in it where time = 0.

Support functions and line segments

A support function is a function on a set such that, given a direction, you get the point in the set that is furthest in the direction. if there is more than one furthest point, any one of them will do. So, for a circle, you get $f(d) = d * r + c$, and for a rectangle, you get a function that returns one of the corners.

a useful property of the support function is that the support function of a Minkowski sum of shapes is the sum of the support functions of those shapes.

If you have a convex shape with a support function, and you're working in 2D, you can easily find the sum of that shape and a line segment:

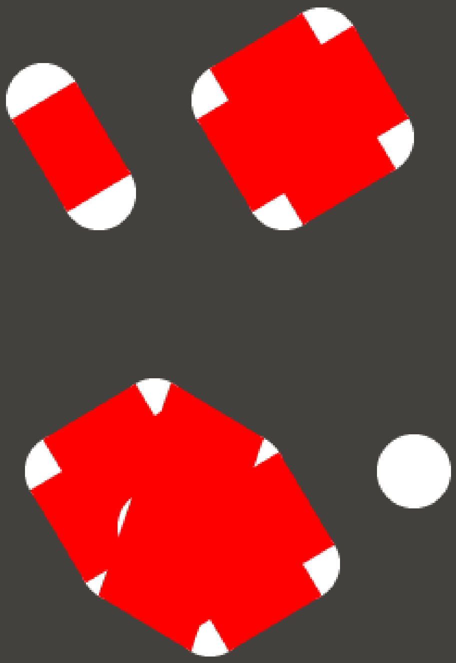
find the support points in the directions perpendicular to the line segment (there are two). Then, create a parallelogram from the line segment, and the segment between the two support points. The new shape is the union of the parallelogram, and a copy of the shape at each end of the line segment.

you can also do pretty cool things with Minkowski sums of shapes with support functions and polygons, but i'll leave alone for today.

here's some javascript code as an example of what i mean about line segments: starting with a circle, and adding 1, 2, then 3 lines.

```
var example = document.getElementById('example');
var context = example.getContext('2d');
function draw_parallelogram(l1, l2,x,y) {
    context.beginPath();
    context.moveTo(x+l1.start.x + l2.start.x, y+l1.start.y + l2.start.y);
    context.lineTo(x+l1.start.x + l2.end.x, y+l1.start.y + l2.end.y);
    context.lineTo(x+l1.end.x + l2.end.x, y+l1.end.y + l2.end.y);
    context.lineTo(x+l1.end.x + l2.start.x, y+l1.end.y + l2.start.y);
    context.fill();
}
function draw_circle(c, x, y) {
    context.beginPath();
    context.arc(x+c.x, y + c.y, c.r, 0, 2*3.141592653589793);
    context.fill();
}
function add_line(shape, line) {
    return {
        draw: function(p) {
            shape.draw({x: line.start.x + p.x, y: line.start.y+p.y});
            shape.draw({x: line.end.x + p.x, y: line.end.y + p.y});
            a = shape.support({x:-line.end.y + line.start.y, y:line.end.x - line.start.x});
            b = shape.support({x: line.end.y - line.start.y,y: -line.end.x + line.start.y});
            context.fillStyle = 'red';
            draw_parallelogram({start: a, end: b},line, p.x, p.y);
        },
        support: function(d) {
            s = shape.support(d);
            x = d.x; y = d.y;
            dot = x*(line.end.x-line.start.x) + y*(line.end.y-line.start.y);
            p = dot > 0 ? line.end : line.start;
            return {x: s.x+p.x, y:s.y+p.y };
        }
    }
}
function make_circle(r, x, y) {
    c = {x:x,y:y,r:r};
    return {
        draw: function(p) { context.fillStyle = 'white'; draw_circle(c, p.x, p.y); },
        support: function(d) {
            l = Math.sqrt(d.x*d.x+d.y*d.y);
            return {x:d.x/l * r+x, y: d.y/l * r + y};
        }
    }
}
circ = make_circle(20, 0,0);
capsule = add_line(circ, {start: {x:0,y:0}, end:{x:30,y:50}});
round_square = add_line(capsule, {start: {x:0,y:0}, end:{x:50,y:-30}});
hex = add_line(round_square, {start:{x:10,y:0}, end:{x:60,y:30}});
circ.draw({x:300,y:300});
capsule.draw({x:100,y:100});
round_square.draw({x:200,y:100});
hex.draw({x:100,y:300});
```

and the output:



Comments are closed.

Twisted Oak Studios offers consulting and development on high-tech interactive projects. Check out our portfolio, or Give us a shout if you have anything you think some really rad engineers should help you with.

Archive