

will have no knowledge of the meaning of each part, and thereby be general and easy to replace.

What we have now is a physics simulator that actually works. As long as the speeds as well as the step lengths are small, it will behave pretty well even with Euler integration. I can apply a force and the object will translate and rotate. With better integrators (Baraff suggests fourth-order Runge-Kutta) it will be more stable and allow higher speeds.

But using better integration methods is only a part of the solution. Since we give the simulation the very strange task of integrating matrices, which is somewhat like interpolating between them, as a way to do rotation, we should ask if the representation is truly suitable.

The standard answer is “use quaternions”, and that is not a bad idea. A quaternion is an entity that consists of four numbers. When used to represent rotation, three of its components actually form the rotation axis. Just knowing that is enough to realize that quaternions are not magical at all. We will return to them in a later chapter.

There is one more answer. How about representing the rotation by a vector, just like the rotation speed? This is, surprisingly, an option that is ignored by most literature. (Not, however, by Parent [8].) You can fully represent a 3D rotation by not four but three numbers, a simple three-component vector, by making the magnitude the amount of rotation.

In both cases, it becomes easier to apply the rotation by a change of rotation instead of corrupting a rotation matrix.

7.3 Collision detection

In Volume 1, I covered the following collision detection methods:

- Spheres are tested using their radius.
- Polyhedra-polyhedra collisions are tested using containment test (vertex of one contained in the other) and edge intersection tests.
- Mixed situations, spheres to polyhedra, require some special tests.
- Spatial subdivision (hierarchical groupings, BSP, quadtrees) are used for limiting the number of tests.
- Simplified bounding shapes are used to simplify the tests.

The weakest part here is the polyhedra-polyhedra collisions. The containment and edge intersection tests will detect a collision of convex polyhedra, but it will not do it very efficiently, and it will not report a well-chosen point of impact. Every vertex in each shape is tested against every plane in the other, both ways, and the same is done for edges, which is even more expensive.

We can see some possibilities to optimize this. We may start the work by testing those vertices and surfaces that are closest to the center of the other, possibly by adding some data to the polyhedra. Such information can accelerate the process significantly, and it can be

saved and reused in the next frame, to provide a good starting point. Polyhedra are typically built from triangles, referring to vertices by indices. A simple improvement of the representation of a polyhedra is to add information to each vertex, referring to each triangle it is used by. Then we can iterate over the surface searching locally for the point we are looking for. This is indeed what is done in some algorithms.

In the following, I will discuss a number of specific methods:

- Intersection volume calculation.
- Closest point calculation with the GTK algorithm.
- Separating planes calculation using the Chung-Wang algorithm.

7.3.1 Intersection volume calculation

A very straight-forward collision detection is to calculate the intersection volume. This may sound complicated, but it is hardly more demanding than the containment/intersection algorithm from Volume 1.

In order to calculate the intersection volume, all vertices of one polyhedra A are tested against the faces of another polyhedra B. We do that one face at a time. Each face defines a plane, splitting 3D space in two halves, the “inside half-plane” and “outside half-plane”.

Using this information, all parts of A that are in the “outside half-plane” are cut away. Faces with all vertices outside are discarded, and those with some vertices outside are split.

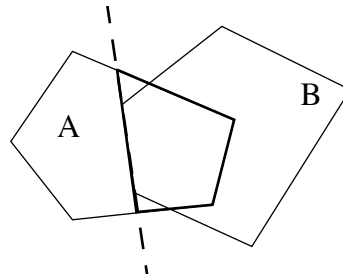


FIGURE 66. Intersection calculation. The polygon A is split over a plane defined by a face of B.

With some luck, or a good choice of first plane to split by, shape A will quickly be reduced to a much smaller shape than the original, containing little more than the vertices that are actually overlapping B, which will speed up the completion of the process. After all faces of B have been tested against the progressively smaller remains of A, we will produce the intersection volume.

This is not the most efficient method around, but it is fairly simple and produces the intersection volume, which in itself has a valuable information that can be used for high precision collision handling.

7.3.2 Closest point calculation with the GTK algorithm.

The Gilbert-Johnson-Keerthi algorithm, GTK for short, is an efficient and popular solution for collision detection [73]. It does not produce the intersection volume, but rather finds the closest points of two convex polyhedra. If the result of the last test between the two shapes is saved and used as starting point, it can be extremely fast.

Ericsson [4] gives a fairly mathematical description of GJK. Here, I will attempt to explain it from a more intuitive view.

GTK is based on *support mapping*, which is the task of finding the extreme point of a shape in a specified direction. In the figure below, the support mapping of the shape A along the vector \mathbf{v} , $S_A(\mathbf{v})$, results in the point \mathbf{p} . It should be obvious that this can be calculated by the dot product, so that for a shape where the vertices are p_k for a certain range of k ,

$$S_A(\mathbf{v}) = p_i \text{ where } i \text{ maximizes } \mathbf{v} \cdot \mathbf{p}_i$$

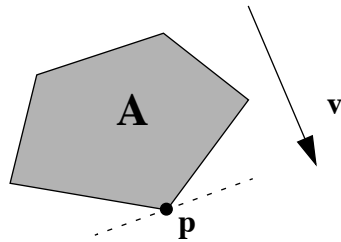


FIGURE 67. The support mapping of the shape A along the vector \mathbf{v} is the point \mathbf{p} .

The following figure is our example case. We wish to test two shapes, A and B, for collision. As you can see from the figure, they do not collide, so this is what the GJK algorithm should result in. But we will now see how it figures that out.

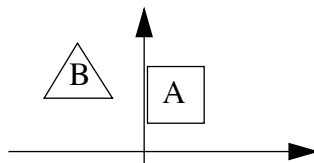


FIGURE 68. The two shapes for our GJK example.

The GTK algorithm implicitly uses the combined shape of two shapes A and B, the dilation of A by the negated (convoluted) B, the Minkowski sum $A \oplus -B$. See the next figure. We will now, in the figures, follow both the sum and the separate shapes through the search for the closest point.

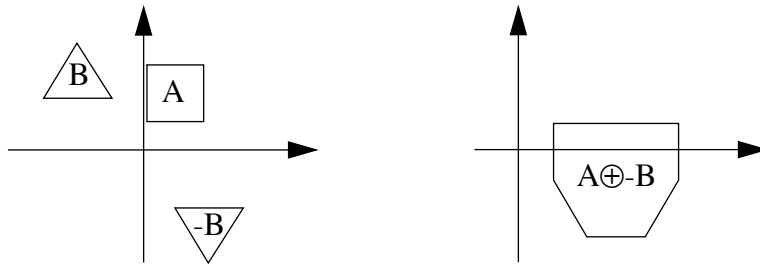


FIGURE 69. The negated shape $-B$ and the Minkowski sum $A \oplus (-B)$.

The GJK algorithm should start in some point on $A \oplus -B$. We call that point \mathbf{p}_0 . This point corresponds to one point in each of A and B , called \mathbf{p}_{0A} and \mathbf{p}_{0B} . From \mathbf{p}_0 , a support mapping is calculated towards origin. This corresponds to a support mapping along the line from \mathbf{p}_{0A} to \mathbf{p}_{0B} . The support mapping results in the point \mathbf{p}_1 , corresponding to \mathbf{p}_{1A} and \mathbf{p}_{1B} . The identity of these two operations can be written:

$$S_{A \oplus -B}(\mathbf{p}_0) = S_A(\mathbf{p}_{0A} - \mathbf{p}_{0B}) - S_B(\mathbf{p}_{0B} - \mathbf{p}_{0A}) = S_A(\mathbf{v}) - S_B(-\mathbf{v})$$

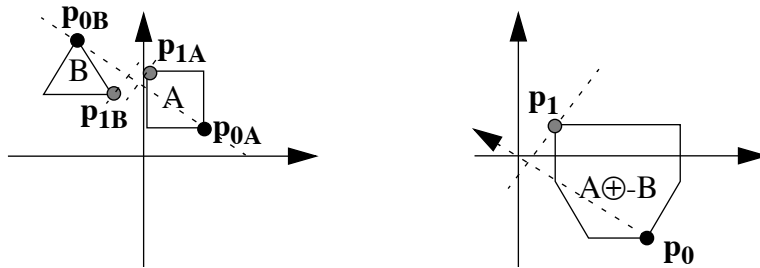


FIGURE 70. The first step of the GJK algorithm, on separate objects (left) and combined (right)

In the following iteration, we take the support mapping of the normal vector to $(\mathbf{p}_0 - \mathbf{p}_1)$ towards origin, resulting in \mathbf{p}_2 . Now \mathbf{p}_0 is farther away from origin than any of the other two and can be discarded.

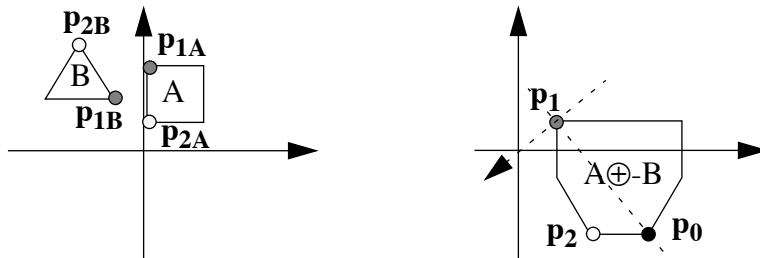
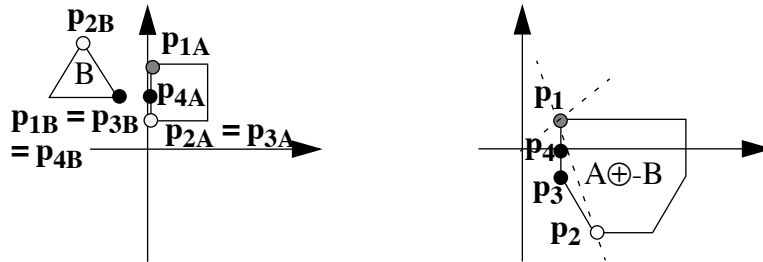


FIGURE 71. The second iteration finds a third point \mathbf{p}_2 .

One more iteration, using p_1 and p_2 , finds p_3 .



The we find the last vertex p_3 and from there the final result p_4 .

Since p_3 is no better than p_1 , we are on a face. As long as we make sure this isn't a face on the opposite side, the final result can be calculated as a point on this face, p_4 . The position of p_4 will tell whether A and B collide or not.

As you can see from the example above, the support mappings for $A \oplus -B$ corresponds to a walk among the vertices in A and B. We start in two arbitrary vertices, one in each shape, make a support mapping to find another pair. From there we use support mapping again to find new vertices that are good candidates, until we can detect that we have reached the minimum.

The support mapping calculation is a vital part of the algorithm and must be made efficiently. A naive implementation would search all vertices. Doing that kind of search for every iteration would make the algorithm slower than our earlier ones. Instead, it should be found using hill climbing among local neighbors. If the connections to the actual neighbors are complemented by well chosen "artificial neighbors" [4] then we do not only find the desired vertex faster, but we can also use that to get out of local minima on the backside.

Another important aspect is to re-use the result from last time. For the example above, let us consider the next frame, looking something like this:

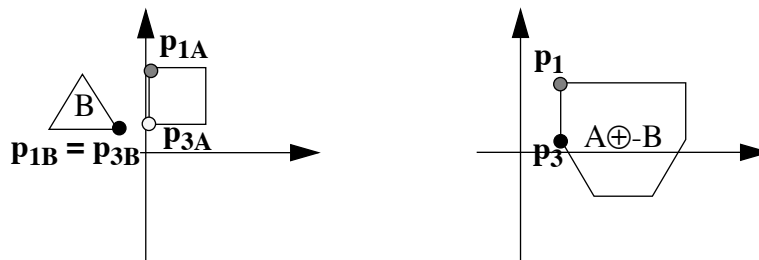


FIGURE 72. Next frame for A and B, where p_1 and p_3 are excellent starting points.

B has moved, so the closest point will change, but unless A or B rotates violently, \mathbf{p}_1 or \mathbf{p}_3 will provide very good starting points, often making the algorithm finish in a single iteration, that is constant time. This is called *vertex caching*.

GJK will make the collision detection very fast indeed, and provide one closest point. This point is still not necessarily well chosen as a representative for the whole collision when two objects meet face to face, but for other kinds of collisions it is sufficient. A modification that outputs a whole face in the case of planar collisions can help somewhat, but we may still need to extract better information.

It can be noted that GJK is easy to extend to handle moving objects. It is simply a matter of sweeping the volume covered by the movement. This holds for many other algorithms as well.

7.3.3 Separating planes calculation using the Chung-Wang algorithm.

There is one more approach that I want to mention: *separating planes* (a.k.a. *separating axis*). This is a test that can simplify the tests by avoiding complex tests when not needed, thus somewhat related to the simplified boundary shape tests.

For two convex objects in 3D space that do not collide, a separating plane always exists. This is known as the *separating axis theorem (SAT)*. (For 2D it becomes a separating line.) If we can find that plane, we know that the objects do not collide. We can save the separating plane and re-use it the next frame. Objects generally don't move extremely fast, so the knowledge of such a plane will either lead to a direct rejection in the next frame, since the plane still holds, or can be a good start for a new test.

When no separating plane can be found, the objects most likely collide (definitely if they are convex) and a more detailed collision detection is needed.

As with other methods, the brute-force solutions are simple but inefficient. An efficient algorithm for separating planes testing is the Chung-Wang algorithm [4][73].

7.3.4 Other methods

There are, of course, other advanced methods, like the full k-DOP method that breaks up the shapes in sub-shapes. Hierarchical subdivision of models has growing popularity and may become the winning solution. So even if I almost skip this problem completely here, it is well worth studying. Did I mention that you can create image-space collision detection algorithms? You can make collision test using the Z buffer. Etc...

7.3.5 Non-convex shapes

It should be noted that most collision detection algorithms work for convex shapes only. Non-convex shapes are a lot more complicated to handle. Three ways to handle them include:

- Calculate the convex hull of the shape and use only that, thus making it convex.
- Subdivide the shape into a number of convex shapes, and use the algorithms for convex shapes on the parts.
- Represent the shape by a progressive hierarchy of spheres, as outlined in Volume 1.

7.4 Collision response

Once we have detected a collision, we need to make a response. This is a topic where I was brief in Volume 1, so I will discuss this a bit more than collision detection.

There are many ways to deal with collisions, so I will try to list alternatives and discuss the implications. Many questions boil down to whether we allow objects in our simulation to *overlap* or not. Obviously rigid bodies should not overlap during extended time, and not even for short periods, but unnoticeable overlap can be acceptable. Allowing overlap is simple, but clearly less exact.

A related question is the *time of collision*. When allowing overlap, we can happily use *constant time steps*, and of objects overlap we do our best to separate them. Constant time steps may also disallow overlap, but then it has to separate objects immediately on overlap, and that will often cause other objects to overlap. In any event, constant time steps imply that the time of impact is the time when we detect the overlap.

Another option, for higher precision, is to *back up time* to the time of the first collision that occurred. This is often not possible to do exactly (due to complex shapes and/or complex movement) so it can be done by approximations and subdivision of the time step.

Perfection is hard to achieve, though. What happens if complex objects move fast? We can miss some collisions, and when we back up time, the missed collision causes an overlap in the denser time scale. Can that be detected, is it worth the price?

Now we have a time of impact. What about the actual response, change of positions and velocities? There are three options:

- Kinematic response
- The penalty method
- Impulse force calculation

Kinematic response is simple. It involves no forces. In its simplest form, we just separate objects. We can also change velocities in the way that I suggested in Volume 1: split the velocity vector in a parallel and perpendicular part, and play with the perpendicular part. While this is not at all incorrect for certain situations, it is not a full simulation that covers all cases.

The penalty method is a force-based method for simulations allowing overlap. When a point is found to penetrate another object, that point is “penalized” by a force that pushes