UC San Diego
UC San Diego Previously Published Works

Title
Collision detection with relative screw motion

Permalink
https://escholarship.org/uc/item/11b1w47r

Journal
Visual Computer, 21(1-2)

ISSN
0178-2789

Author
Buss, Samuel R

Publication Date
2005-02-01

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Collision Detection with Glide Rotations

Samuel R. Buss*
Department of Mathematics
Univ. of California, San Diego
La Jolla, CA 92093-0112

July 15, 2004

Abstract

We introduce a framework for collision detection between a pair of rigid polyhedra. Given the initial and final positions and orientations of two objects, the algorithm determines whether they collide, and if so, when and where. All collisions, including collisions at different times, are computed at once along with "properness" values that measure the extent to which the collisions are between parts of the objects that properly face each other. This is useful for handling multiple (nearly) concurrent collisions and contact points.

The relative motions of the rigid body is limited to glide rotation (i.e., screw motion). This limitation is not always completely accurate, but we can estimate the error introduced by the assumption.

Our implementation uses rasterization to approximate the position and time of the collisions. This allows level-of-detail techniques that provide a tradeoff between accuracy and computational expense.

The collision detection algorithms are only approximate, both because of the glide rotation limitation and because of the rasterization. However, they can be made robust so as to give consistent information about collisions and to avoid sensitivity to roundoff errors.

1 Introduction

Collision detection is important for physical simulations and motion planning. It is also a difficult problem, especially since collision detection must be fast, robust, and tolerant of roundoff errors. This paper presents an

*Supported in part by NSF grants DMS-0100589 and DMS-04XXXXX. Email: sbuss@ucsd.edu

algorithm for collision detection between two moving, polygonally modeled objects. Our original motivations were detecting collisions in computer game (in particular racing games) where collision detection must be performed on objects moving at high speed. In this situation, objects frequently are found to be significantly interpenetrating and may even completely pass through each other in a single simulation step. Thus the algorithm is designed to correctly detection collisions in these situations. In addition, the algorithm discovers multiple collisions, not just the first collisions. This allows handling multiple collisions at high speeds and maintaining multiple contact points between statically contacting objects.

The algorithm takes as input the initial position[1] and a final position for each object, and estimates the intermediate relative motions of the objects as a screw motion. The algorithm computes whether a collision occurs during the motion from the initial position to the final position. A single pass of the algorithm determines the times and positions of the collisions: there is no need to "back up" time or do a binary search to find the time where the objects first interpenetrate.

The algorithm computes simultaneously all the collisions between the initial and final positions, not just the first collision(s). Collisions are classified as to how "proper" they are, where the "proper-ness" depends on how well the objects are facing each other locally at the collision. This data can be used by subsequent application-dependent algorithms to recognize concurrent collisions.

For detecting collisions at intermediate times, the objects' relative motion is assumed to be screw motion, also called a "glide rotation." A screw motion consists of rigid body rotation around a fixed axis combined with a constant rate translation parallel to the axis. This assumption of screw motion is not always exactly correct, but is no less reasonable than other common assumptions, such as rotation combined with arbitrary translation. In addition, the error introduced by assuming screw motion can be quantified as being quite small (see Section 4.3).

Several other authors have used glide rotations for collision detection: Rossignac and J. Kim [49] advocate the use of screw motions for intermediate relative motion, and Kim-Rossignac [25] and Redon et al. [45] use screw motions for collision detection.

The principal idea behind our algorithm is that the glide rotations allow us to view the two rigid objects as moving towards each along a fixed track, with no rotation beyond the curvature induced by the curvature of

[1] By "position," we mean both position and orientation.

the “track.” This allows us to set up a curved “collision screen,” and project the two objects onto the collision screen (similar in spirit to the way that objects can be orthogonally projected onto a flat plane). Lines and edges are drawn onto the collision screen, and collisions are found by examining where the two objects project to common points on the screen.

A second idea behind our algorithm is that every collision is given a *properness rating*: Since the algorithm reports multiple collisions during a time step it is useful to find the true “first collisions.” The properness values help identify true collisions. A nonnegative properness value indicates the facets are correctly facing each other at the instant of collision, as would occur for a true first collision. Negative values of properness indicate the facets are not correctly facing each other and that it is not a first collision. Negative properness values close to zero indicate collisions that are close to proper, and we have found it useful to use slightly improper collisions as true collisions for handling high-speed collisions. Sections 3 and 5 discuss properness in more detail.

The implemented version of our algorithm uses solid, rigid objects defined by polygonal surfaces. It allows arbitrary such objects, including nonconvex objects. The implemented form of the algorithm works well with objects that are not too far from convex, but it could be extended to handle highly nonconvex objects (e.g., a screw in a screw hole).

The algorithm can readily be adapted to work with non-closed objects or with objects that are not manifolds. As long as the local adjacency information for polygons is known, it is possible to calculate properness of collisions, even for non-manifolds.

We describe an implementation of the collision algorithm that uses rasterization rather than the analytic methods used by Kim-Rossignac [25] and Redon et al. [45]. Our algorithm differs from those in that it looks for collisions between the entire two polyhedra at once, rather than treating the facets of the polyhedra individually. This allows our algorithm to be robust and tolerant of roundoff errors. Furthermore, adjusting the resolution of the rasterization allows a tradeoff of accuracy against computational cost.

Our collision detection algorithm is intended as only one component of a large scale simulation of collisions and contacts between multiple moving bodies. Once there are very many moving objects, it is too time-consuming to check for collisions between all pairs of objects. Instead, space partitioning algorithms (e.g., octrees, BSP trees, k-d trees) can be used to prune the set of possible collisions, and bounding volumes (swept spheres, OBBs, k-DOPs, etc.) can be used to quickly decide that many objects to do *not* collide (c.f., [14, 24, 21, 31, 47, 44, 57]). It is intended that our algorithm

would be used after these preliminary tests to check all remaining potentially colliding pairs of objects. Subsequently, the data from our collision detection algorithm would be used by application-dependent algorithms to decide how to handle the collisions, e.g., to calculate the forces, or changes in motion, that result from the collisions. It is beyond the scope of this paper to consider space partitioning and hierarchical methods, or how to calculate collision responses. Rather, we only examine the question of determining whether and how a single pair of rigid objects has collided.

2 Prior Work and Motivations

Early papers on collision detection include [6, 7, 10], and the field has since grown too large for us to survey here. (See [37] for a survey.) Instead, we discuss only some of the more relevant work in order to contrast it with our algorithm — our primary focus is the collision of rigid polyhedral objects.

A number of collision detection algorithms work with convex objects. These include algorithms by Lin and Canny [35, 36] exploiting the Voronoi diagram of the space around convex objects and the linear algebra based methods of Gilbert et al. [19], as well as many enhancements of these algorithms (cf. [9, 39]). These algorithms compute the distance between two non-overlapping convex bodies; if the convex bodies are interpenetrating, this is detected, but the algorithms do not reliably decide how they collided. Instead, when two moving bodies are discovered to be interpenetrated, then time is "backed up" and a binary search procedure (bisection) is used to find the approximate time and position of the first collision between the convex bodies. These algorithms may, however, miss collisions where the bodies pass through each other in a single simulation step.

Another class of algorithms attempts to deal directly with interpenetrating objects, particularly convex objects. [12, 1, 26] give algorithms for finding the minimum interpenetration distance of two overlapping convex bodies. [27] extend these algorithms to apply to nonconvex objects using decomposition into convex pieces. All these algorithms, like our own, attempt to deal directly with bodies that may be interpenetrating and can be used to estimate the collision times and positions. Unlike our algorithm, they restrict to purely translational motion.

Various techniques are used to speed up the collision detection algorithms. First, coherence is often used to greatly increase the speed of collision detection. Second, Dobkin and Kirkpatrick [13] gave a method which, after a one-time preprocessing, can perform distance calculations be-

tween two non-overlapping convex bodies in arbitrary orientations in only $O(\log^2 n)$ time, where n is the number of edges or faces of the bodies.

There are also impressive algorithms for tracking multiple moving polygonal planar objects and detecting collisions, based on kinetic data structures (see [17, 29, 30]). In addition the SWIFT systems [15, 16] can track large number of polyhedra with a multilevel hierarchy. The former algorithms do not work in three space (yet), and the latter work with only with objects that are convex or can be decomposed into convex objects.

The above-discussed algorithms only detect collisions at discrete points in time; there are several prior algorithms that directly calculate collisions at intermediate times without needing to use bisection. Boyse [6] gave an algorithm to find collisions between moving polygonal objects moving purely translationally or purely rotationally. Cameron [7, 8] describes a four dimensional approach which handled (piece-wise) linear motion of convex polyhedra. Canny [10] extended the methods of Boyse to find collisions between polygonal faces that can be both moving translationally and rotating at an approximately uniform rate. Eckstein and Schömer[14] used regula falsi in place of bisection to search for the first collision. Schömer and Thiele [50] gave a subquadratic algorithm to find the first point of intersection between two polyhedra, one of which either is moving translationally or is rotating on a fixed axis. Redon et al. [43] present algorithms for polygonal objects moving either translationally, or rotationally, or with a non-classical screw motion. Snyder et al. [55], Von Herzen et al. [57], Redon et al. [45] and others have used interval arithmetic to find collisions between moving objects. [57] allowed general motions with objects that are allowed to deform. [45] used interval arithmetic for colliding polygons in a "polygon soup": they allowed the polygons' relative motion to be screw motion. Kim and Rossignac [49, 25] also used screw motions and discussed analytic methods of colliding polygons whose relative motion is a screw motion; they use Newton iterations to approximate the collision time.

Our algorithm also uses screw motions, but collides the objects in a "holistic" fashion, rather than just colliding pairs of faces.

A side effect of the use of screw motion is that it allows a particularly sophisticated form of culling back-facing faces. As is already discussed in [49], and is described again in Section 4.2 below, when using glide rotations, back-face culling can cut a polygonal face into two sub-polygons, one part front-facing and one part back-facing. For collision detection, the back-facing portions of the face can then be discarded. This improves on the back face culling methods of Vaněček [56] and Redon et al. [45] which discard a face only if all its vertices are back-facing.

Another line of prior work has been our earlier unpublished software for racing car games for PC's and game consoles published by Angel Studios. Those algorithms also handle arbitrary polygonal objects, take motion into account, are capable of detecting some collisions where objects have completely passed through each, and attempt to identify multiple nearly-concurrent collisions. Those earlier algorithms do not handle rotation well and are much more ad-hoc than the present one and are susceptible to occasionally yielding erroneous results.

Our implemented algorithm uses rasterization to find approximate collision points and times. Rasterization has been used in image-based algorithms, including [48, 53, 41, 32, 2, 22], to find interpenetration of static objects. However these algorithms do not work with moving objects and do not identify the positions or directions of collisions.

3 Intermediate motion and multiple collisions

In this section, we argue that it is helpful to consider the intermediate motion of the potentially colliding objects, and to find all collisions that occur, not just the first ones. By find "all collisions", is meant to find all instances at which portions of the objects contact under the assumption that the motion continues through collisions without any effect on the motion.

Our motivations arose from collision detection in real-time computer games, particularly, high-speed car racing games. In such games, speeds in excess of 150 mph are not uncommon and at simulation rates of 30 frames per second, the vehicle moves a rather substantial 2.2 meters per time step. Even at 60 frames per second, the distance is still a substantial 1.1 meters per time step. This allows for considerable interpenetration: a vehicle may penetrate a stationary object a distance of 2.2 (resp., 1.1) meters during a single simulation step. Even worse, vehicles colliding head-on have potentially twice as much penetration. Without taking motion into account, the collision detection could give very wrong results, such as cars appearing to pass completely though each other (colliding out the "wrong way"), or cars that hit an object head-on, but the collision occurs sideways.

Figures 1 and 2 show some simple two-dimensional examples of how objects' motions are used to determine collisions. If the two squares of Figure 1 have collided by purely horizontal movement, there would be a vertex-vertex (VV) collision. However, as Section 5 discusses, vertex-vertex collisions are not desirable, and it is preferable to find vertex-face collisions.[2]

[2]In our two-dimensional example, we use "face" as a synonym for "edge", since the

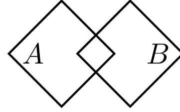
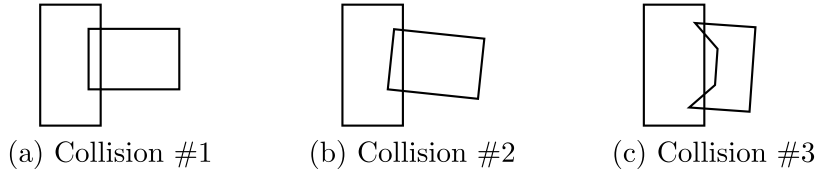


Figure 1: Two interpenetrating squares. In the absence of information about the objects' movement, the collision direction is ambiguous.



(a) Collision #1 (b) Collision #2 (c) Collision #3

Figure 2: (a),(b). Two rectangles A and B have collided. Rectangle A on the left is stationary. Rectangle B on the right is translating horizontally leftward. (c) A polygon has collided with A while translating horizontally.

For instance, one might decide that the lower left face of B has collided with the upper right face of A; if so, the algorithm reports two vertex-face collisions, namely the left vertex of B with the upper right face of A plus the right vertex of A with the lower left face of B.

Figure 2(a,b) shows two examples of two interpenetrating rectangles, colliding horizontally. The first collision is entirely unproblematic: there are two simultaneous collisions of the left vertices of object B against the right face of object A. In the second collision, B is tilted slightly so that its lower left vertex contacts A before its upper left vertex. Nonetheless, our algorithm again reports two collisions: the first collision is the lower left vertex of B against the right face of A, the second collision occurs slightly later and is the upper left vertex of B against the same face of A. The two collisions are also assigned a properness rating. The first collision is entirely proper, the second one is slightly improper. The improperness is due to the fact that the left face of B is already inside A when the upper left vertex of B collides with A: the improperness indicates that the collision is not a first collision point, not even locally. Part (c) of Figure 2 shows a differently shaped object B: it also is translating horizontally. In this case, there are two proper vertex-face collisions which occur at slightly different times.

The treatment of multiple, non-concurrent collisions and improper collisions depends on the application. In real-time video game applications, we have obtained good results by using all the collisions that not too im-

analogous collisions in three dimensions are vertex-face collisions.

proper and calculating collision impulses (and thereby the collision response) under the assumption that the collisions all occurred simultaneously. (See [3, 34, 46, 18, 11] for methods of calculating impulses or forces in response to multiple collisions.) One consequence of this is that the two collisions depicted in parts (a) and (b) of Figure 2 would give rise to very similar collision responses. On the other hand, if only first collision shown in (b) were handled, then the objects A and B would begin to tumble, possibly leading immediately to more collisions and to ultimately a very different physical simulation. This points out another advantage to treating the multiple collisions simultaneously; namely, since the bodies are modeled as being completely rigid, if we were to treat the collisions singly and model the physical responses as impulses, then a huge number of collisions could occur in a very short period of time.

There are several possibilities for deciding which collisions should be handled. For a more accurate simulation, not necessarily real-time, the first occurring collision could be used; this would require backing up time to the time of the first collision, and restarting the simulation from that point in time. Mirtich [40] describes this kind of approach. In real-time applications however, is often useful to treat all collisions occurring occur within a fixed arbitrary time of each other as being simultaneous.

4 Glide rotations

The collision detection algorithm is given the initial positions and the final positions of two objects, and estimates the intermediate movement from these positions. It assumes that the *relative* movement of the objects is a screw motion, also called a "glide rotation." The initial positions of the two objects are given by rigid, orientation preserving, affine transformations M_i and N_i, and the final positions are given by M_f and N_f. If $\mathbf{x}$ is a point in object A's local coordinates, $M_i\mathbf{x}$ and $M_f\mathbf{x}$ represent the initial and final positions of the point $\mathbf{x}$ as expressed in global coordinates. The transformations N_i and N_f similarly describe the movement of object B.

To express the motion of B from the point of view of A (in A's local coordinate system) suppose a point is moving rigidly with B and is initially at position $\mathbf{x}$ in A's local coordinate system. The final position of the point, still expressed in A's local coordinate system, is given by

$$\Omega\mathbf{x} \;=\; M_f^{-1} N_f N_i^{-1} M_i \mathbf{x}. \qquad (1)$$

Of course, Ω is a rigid, orientation-preserving, affine transformation. It is

well-known that any such transformation is either (a) a translation or (b) a screw motion. (See [51], for instance.)

A screw motion, or glide rotation, is a rigid, orientation-preserving affine transformation defined by the following: (a) A rotation axis that is specified by a point $\mathbf{u}$ and a unit vector $\mathbf{v}$, so that the axis contains $\mathbf{u}$ and is parallel to $\mathbf{v}$. Note the rotation axis may not pass through the origin and that $\mathbf{u}$ and the direction of $\mathbf{v}$ are not uniquely determined. (b) A glide angle θ. (c) A glide distance g. The glide rotation acts on a point $\mathbf{x}$, by rotating it an angle θ around the rotation axis, with the direction of rotation determined by the right-hand rule, and then translating distance g in the direction of $\mathbf{v}$.

For example, if the rotation axis vector $\mathbf{v}$ is $\langle 0,1,0\rangle$ and the point $\mathbf{u}$ is the origin, then the glide rotation consists of rotating angle θ around the y-axis while translating upwards distance g. This glide rotation is given by

$$\Omega \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} 0 \\ g \\ 0 \end{pmatrix}. \qquad (2)$$

Glide rotations have the property that they can be generated by a smooth flow. There is a velocity field $\boldsymbol{\omega}$, such that for each point $\mathbf{x}$, $\boldsymbol{\omega}(\mathbf{x})$ specifies the velocity of the point at position $\mathbf{x}$ and such that the velocity field generates the glide rotation in a unit time period. The flow generating the glide rotation (2) is

$$\boldsymbol{\omega}(\mathbf{x}) = \theta\mathbf{j} \times \mathbf{x} + g\mathbf{j}, \qquad (3)$$

or, equivalently,

$$\boldsymbol{\omega}(\langle x_1, x_2, x_3\rangle) = \langle \theta x_3, g, -\theta x_1\rangle. \qquad (4)$$

An object initially at position $\mathbf{x}$ is carried by the velocity field $\boldsymbol{\omega}$ to $\Omega\mathbf{x}$ in a unit time period. The flow $\boldsymbol{\omega}$ is best expressed using the cross product, but can equivalently be obtained from the expression (2) for Ω, by replacing θ with θt and g with gt and taking the derivative with respect to t.

4.1 Determination of the glide rotation

We now discuss how to calculate the glide rotation parameters $\mathbf{u}$, $\mathbf{v}$ and g; a similar construction is given by [49]. The transformation Ω is given as a 3×3 rotation matrix A plus a translation amount $\mathbf{t}$, so that $\Omega\mathbf{x} = A\mathbf{x} + \mathbf{t}$. To express Ω as a glide rotation, first express the rotation matrix A as a rotation of θ radians around an axis $\mathbf{v}$, where $\mathbf{v}$ is a unit vector. (See [54] or [52] for algorithms for determining θ and $\mathbf{v}$.) This vector $\mathbf{v}$ is the direction

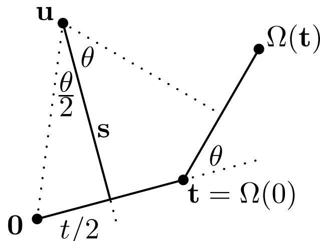


Figure 3: Determination of the point $\mathbf{u}$ on the rotation axis. The rotation axis $\mathbf{v}$ is pointing straight up out of the figure. $\mathbf{s} = \mathbf{v} \times \mathbf{t}/(2\tan(\theta/2))$ is perpendicular to the rotation axis.

of the glide rotation axis. The point $\mathbf{u}$ on the rotation axis that is closest to the origin is equal to

$$\mathbf{u} \;=\; \mathbf{t}/2 \;+\; (\mathbf{v} \times \mathbf{t})/(2\tan(\theta/2)).$$

This equation for $\mathbf{u}$ can be verified by examination of Figure 3; the calculation can be simplified using the half-angle formula $\tan(\theta/2) = (1 - \cos\theta)/\sin\theta$.

After determining the glide rotation axis and angle, we do a change of coordinates so that the glide rotation axis is the y-axis. In these coordinates, $\mathbf{u}$ is the origin $\mathbf{0}$ and $\mathbf{v} = \langle 0, 1, 0\rangle$.

4.2 The forward facing surfaces

A crucial property of the glide rotations is that, as a rigid object is transformed by a glide rotation flow $\boldsymbol{\omega}$, the surface of the object can be consistently divided into "forward-facing" regions and "backward-facing" regions. The intuition is that the forward facing regions are the areas of the surface where the outward surface normal is looking forward in the direction of motion. More precisely, if $\mathbf{x}$ is the position of a point on the surface and $\mathbf{n}$ is the surface normal, then the point $\mathbf{x}$ is *forward facing* provided that $\mathbf{n} \cdot \boldsymbol{\omega}(\mathbf{x}) \geq 0$. Points where $\mathbf{n} \cdot \boldsymbol{\omega}(\mathbf{x}) \leq 0$ are *backward facing.*

The motion of the glide rotation changes the values of $\mathbf{n}$ and $\mathbf{x}$, but the value of $\mathbf{n} \cdot \boldsymbol{\omega}(\mathbf{x})$ remains constant. Intuitively this is clear from the symmetry of the glide rotation. More formally, it is because the glide rotation acts on $\mathbf{n}$ and $\boldsymbol{\omega}(\mathbf{x})$ by rotating them about the rotation axis at a constant rate. Therefore, points do not switch between forward and backward facing.

Before performing collision detection, we compute the forward facing faces of B and the *backward* facing faces of A under the same glide rotation. The backward facing faces of A are used since A can be viewed as

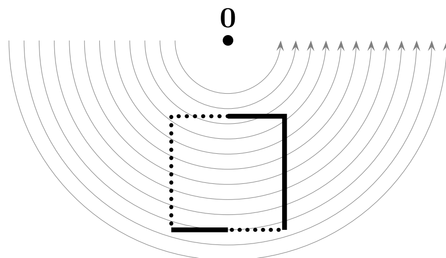


Figure 4: The forward facing surfaces (solid lines) and backward facing surfaces (dotted lines) of a square rotating around the origin.

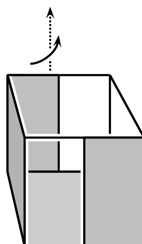


Figure 5: A cube is spiraling upward with a glide rotation around the vertical axis behind the cube. The backward facing faces are drawn gray and are the left face, the bottom face, and half of the near and far faces.

being transformed by the inverse of $\boldsymbol{\omega}$ from the viewpoint of B. Under the assumption that A and B are not interpenetrating at the initial configurations as given by M_i and N_i, the first set of collisions between A and B must involve forward facing faces of B and backward facing faces of A. Restricting attention to the appropriate forward/backward facing edges reduces the number of faces that are collision candidates.

Figure 4 shows a simple two dimensional example of a square rotating counterclockwise around the origin. The forward facing faces are shown as solid lines, the backward facing faces are shown as dotted lines. Note that two faces (edges) have both forward and backward facing components. Figure 5 shows a three dimensional example of backward facing faces on a cube. The cube is drawn in wire frame, and the backward facing faces are drawn as solid grey rectangles. The rest of the surface is forward facing.

It is simple to compute the forward and backward facing parts of a polygonally modeled surface. Consider a polygonal patch π lying in a plane P which has normal $\mathbf{n}$ so $P = \{\mathbf{x} : \mathbf{n} \cdot \mathbf{x} + d = 0\}$. The forward facing parts

of P satisfy $\boldsymbol{\omega}(\mathbf{x}) \cdot \mathbf{n} \geq 0$. Using (3), we have

$$\boldsymbol{\omega}(\mathbf{x}) \cdot \mathbf{n} \;=\; (\theta \mathbf{j} \times \mathbf{x}) \cdot \mathbf{n} \;+\; g\mathbf{j} \cdot \mathbf{n} \;=\; (\theta \mathbf{n} \times \mathbf{j}) \cdot \mathbf{x} \;+\; (g\mathbf{j} \cdot \mathbf{n}).$$

If $\theta\mathbf{n}\times\mathbf{j}$ is zero, $\boldsymbol{\omega}(\mathbf{x})\cdot\mathbf{n}$ is constant, equal to $g\mathbf{j}\cdot\mathbf{n}$, and, depending on its sign, the plane P is either everywhere forward facing or everywhere backward facing. Otherwise $\boldsymbol{\omega}(\mathbf{x}) \cdot \mathbf{n} = 0$ defines a plane Q, with normal direction $\theta\mathbf{n} \times \mathbf{j}$. Since $\mathbf{n} \cdot (\mathbf{n} \times \mathbf{j}) = 0$, Q is perpendicular to P, so Q and P intersect in a line L which splits P into forward facing and backward facing parts. The patch π, if it is intersected by L, splits into two polygonal subpatches, one forward facing and the other backward. The subpatches are computed by clipping the patch against the plane Q.

When computing forward and backward facing faces, there is the additional complication that some polygons have to be “doppeled” (i.e., projected to the collision screen twice) in order to not miss any collisions. Doppeling will be discussed in Section 8. It is logically distinct from the process of culling back faces, but is implemented by our algorithm at the same time.

4.3 An error estimate

In many cases, the assumption that the objects’ relative motion is a glide rotation is not completely correct. We can justify the use of glide rotations by three arguments. (See [49] for more arguments for using glide rotations for intermediate motion.) First, the assumption that the motion is a glide rotation is a priori no worse than many other assumptions. One might think that a better assumption would be to let each object rotate on its own axis, but this assumption is also often incorrect, since physically moving bodies do not in general rotate on a fixed axis (c.f. [20]). Second, glide rotations have very nice properties such as the constancy of forward facing surfaces, and the fact that the forward facing portion of a polygonal face is a polygon. For these reasons, the algorithm described in Sections 6-9 requires the use of glide rotations. Third, the error introduced by glide rotations turns out to be quite small. The actual error depends on the motions of the two bodies of course, but it can be adequately estimated by finding the error under the assumption that one body is stationary and the other is rotating at a constant rate on a fixed axis. In this case the error is essentially the difference between “lerping” and “slerping” (see [54]) since the only error is in the position of the center of masses. To quantify this, we compare the motion of a point moving on a straight line to its motion when rotating around the origin. (For instance, the center of mass would be moved in a circular arc by a glide rotation.) Figure 6 shows the two paths from $\mathbf{x}$ to $\mathbf{y}$.

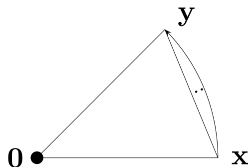


Figure 6: The length of the short dotted line is the estimated error bound (5) for rotation angle θ.

The total rotation angle is θ and the maximum distance from the circular arc to the line segment is $(1 - \cos(\theta/2))r$ where r is the distance from the rotation center. Since the arc length is θr, we estimate the error in terms of the ratio of the maximum distance to the arc length. Thus, the percentage error is estimated by

$$\frac{1 - \cos(\theta/2)}{\theta}. \tag{5}$$

Some sample percentage error values are given in the table. (Equation (5) applies for θ measured in radians, but the table uses degrees.)

Angle θ	% Error	Angle θ	% Error
60°	0.173	20°	0.044
45°	0.097	10°	0.022
30°	0.065	5°	0.011

A rotation of 45° gives an estimated error of less than 10%, which may be quite acceptable for low-accuracy applications such as computer games. (In any event, a rotation of 45° is quite large for a single step of a simulation, and frequently rotations need to clamped to be less than 45° per simulation step to have stable physical simulations.) A 5° rotation gives an error estimate of just over 1%. Thus glide rotations can give quite acceptable accuracy for many applications, but usually the total rotation in a single simulation step should be less than 45°. For higher levels of accuracy, note that the absolute error decreases quadratically with the simulation step size.

5 Types of collisions, Properness

When working with polygonally modeled surfaces, collisions can be categorized as "face to face" (FF), "face to edge" (FE), "face to vertex" (FV), "edge to edge" (EE), "edge to vert" (EV), "vert to vert" (VV), etc. Different applications may want to use different types of collisions. For instance, an edge colliding parallel to the interior of a face could be viewed either as an

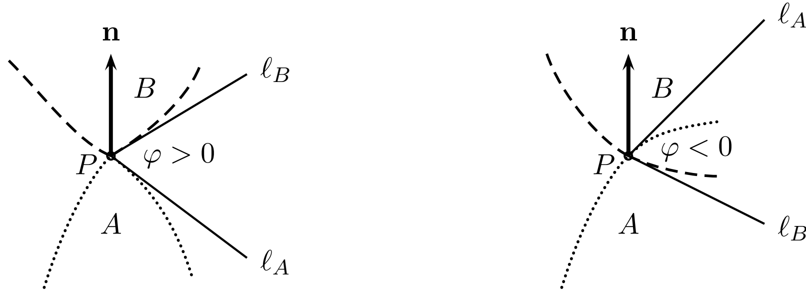


Figure 7: The surfaces of A and B are shown in the vicinity of a collision at a point P. Object A is bounded by and lies below the surface represented by the dotted curve. Object B is bounded by and lies above the surface represented by the dashed curve. The lines ℓ_A and ℓ_B are tangents to the surfaces in a given direction. φ is the signed angle between these lines. We have $\varphi > 0$ on the left and $\varphi < 0$ on the right. The collision on the right is therefore improper.

EF collision or as two VF collisions with the vertices being the endpoints of the edge. As another example, physical simulations cannot effectively work with VV collisions, since the normal vector describing the direction of the collision is usually required to determine the collision response and a VV collision does not have a uniquely determined collision normal.

Our algorithm finds collisions of many types, but resolves them into collisions between vertices and faces (types VF and FV) and between two edges (type EE).[3] For each collision, it computes (a) the time of the collision, (b) the spatial position, (c) the vertex, edges or face involved in the collision, (d) a normal vector which is the collision direction, and (e) the properness of the collision. The properness is a local condition that measures the extent to which the collision involves facets of A and B that are locally facing each other.

A formal definition of properness for collisions of arbitrary surfaces is illustrated in Figure 7. There is a collision at P with collision normal $\mathbf{n}$. For any direction out from P orthogonal to $\mathbf{n}$, let the lines ℓ_A and ℓ_B be tangent to the surfaces of A and B in that direction. Let φ be the angle between these surfaces measured from ℓ_A to ℓ_B with the direction of measurement being counterclockwise in the figure. The *properness* of the collision is defined to be the minimum value of $\sin\varphi$ obtained in this way. If the properness value

[3] It is common for these three kinds of collisions to be viewed as fundamental; see, for instance, [4].

is nonnegative, then the objects are not interpenetrating in the intermediate vicinity of the collision point P. Negative values, between -1 and 0, indicate that A and B are interpenetrating in the intermediate vicinity of P; in this case the collision is *improper*. A negative properness value close to zero indicates the collision is only slightly improper. For objects with smooth surfaces, the maximum properness is zero since the surfaces will be tangent at a proper collision of smooth surfaces.

Because of the constancy of the values $\mathbf{n} \cdot \boldsymbol{\omega}(\mathbf{x})$ under glide rotations, the properness of a collision does not depend on the time of the collision. Instead the properness can be calculated from just the geometry of the two bodies in the neighborhood of the colliding points.

Properness is defined using the sine function since this allows using dot products to efficiently calculate the properness for polygonal surfaces. The use of the sine function is not particularly important since the only use made of properness values is to decide, based on a fixed threshold, whether to discard a potential collision. For example, we could use φ instead of $\sin\varphi$ and this would be equivalent if the threshold were changed appropriately.

For an FV collision (face f of body A against vertex v of body B), the normal is the unit normal $\mathbf{n}$ pointing outward from f. Let the vertex have degree d, and let $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_d$ be the unit vectors pointing from v along its d adjacent edges. In order for the collision to be proper, it is necessary that $\mathbf{e}_i \cdot \mathbf{n} \geq 0$. The *properness* of the collision is $\min\{\mathbf{e}_i \cdot \mathbf{n}\}$.

For an EE collision, let $\mathbf{e}_A$ and $\mathbf{e}_B$ be unit vectors along the two colliding edges, which are assumed to not be parallel. The collision normal $\mathbf{n}$ is a unit vector in the direction $\pm\mathbf{e}_A \times \mathbf{e}_B$. The sign of $\mathbf{n}$ is chosen so as to point from body A towards body B; this done by choosing the sign to maximize the value (7) below. Let $\mathbf{n}_{A,1}$ be the normal of the face on the left side of $\mathbf{e}_A$ and $\mathbf{n}_{A,2}$ be the normal of the face on the right side (by “left” and “right” we mean from the point of view of someone outside A traversing the edge). Define $\mathbf{n}_{B,1}$ and $\mathbf{n}_{B,2}$ similarly. The properness value of a proper EE collision can be shown to equal

$$||\mathbf{e}_A \times \mathbf{e}_B|| \cdot \min\{(\mathbf{n}\times\mathbf{e}_A)\cdot\mathbf{n}_{A,1}, -(\mathbf{n}\times\mathbf{e}_A)\cdot\mathbf{n}_{A,2}, -(\mathbf{n}\times\mathbf{e}_B)\cdot\mathbf{n}_{B,1}, (\mathbf{n}\times\mathbf{e}_B)\cdot\mathbf{n}_{B,2}\}. \tag{6}$$

But in practice, our algorithm uses

$$\min\{(\mathbf{n}\times\mathbf{e}_A)\cdot\mathbf{n}_{A,1}, -(\mathbf{n}\times\mathbf{e}_A)\cdot\mathbf{n}_{A,2}, -(\mathbf{n}\times\mathbf{e}_B)\cdot\mathbf{n}_{B,1}, (\mathbf{n}\times\mathbf{e}_B)\cdot\mathbf{n}_{B,2}\} \tag{7}$$

instead of the true properness — this makes essentially no difference to the operation of the algorithm since the properness of EE edges is never used for any algorithmic purpose. We will not prove the correctness of the

formula (6) except to note that it is easy to check that both (6) and (7) are equal to the properness when the edges are perpindicular. It is also easy to check that (6) and (7) are positive for proper collisions and negative for improper collisions. We exclude the degenerate case of parallel edges colliding: although these kinds of collisions could be proper, we never return them as collisions; instead they generate EV and VE collisions.

Our collision detection algorithm first identifies VV, EV, VE, EE, VF, and FV collisions. These are then resolved into VF, FV, and EE collisions. EE collisions are remain unchanged. The EV (resp., VE) collisions are resolved into FV (resp., VF) collisions by choosing one of the two faces adjacent to the edge as the collision face, namely the one that maximizes the properness value. Each VV collision is resolved into an FV or VF collision: the algorithm considers each possible collision of one of the vertices against one of the faces adjacent to the other vertex, and chooses a vertex-face pair that maximizes the properness.

6 Overview of algorithms

We now give a high-level description of the collision detection algorithms based on glide rotation. The input to the algorithm is two rigid objects specified by polygonally modeled surfaces, and their initial and final positions. The polygonally modeled surfaces should be stored in a winged edge structure [5] or other similar data structure that permits easy access to adjacency information such as the list of edges adjacent to a vertex and the pair of faces adjacent to an edge. We compute the glide rotation Ω and the associated flow $\boldsymbol{\omega}$ for the relative motion (1) of the bodies. Then backward facing faces of A and forward facing faces of B are found. These faces, and their edges and vertices, are candidates for collisions between A and B.

The collision detection projects the facets of A and B onto a “collision screen” that is perpendicular to the glide rotation flow. For points on A or B, we calculate the elapsed time before the points hit the screen. If a point on A and a point on B project to the same point on the screen, then they potentially collide; the difference in the elapsed times until they hit the collision screen gives twice the collision time. The collision screen for general glide rotations is difficult to visualize and its definition is postponed until the next section. The idea of the collision screen is illustrated in Figure 8 for the much simpler case of purely translational motion.

When projecting to the collision screen, the facets of A and B are drawn superimposed on the screen. Vertices project to points on the screen; edges

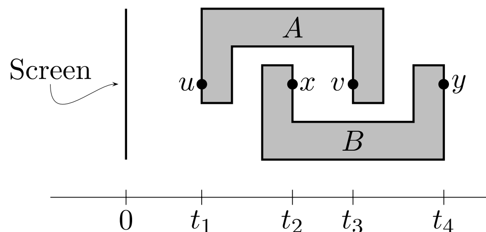


Figure 8: Two nonconvex objects A and B are moving horizontally. A is moving leftward and B rightward, both at the same speed. A vertical collision screen is on the left and all four labeled points project orthogonally to the same point on the screen. The horizontal axis measures the time for a point on A or B to hit the screen. Subtracting the times for two points of A and B gives twice the time until they collide. For instance, v will hit the screen in time t_3, and x hit the screen at t_2 time units in the past. The two points will collide at time $(t_3 - t_2)/2$ in the future. On the other hand, $t_1 < t_2$, so u and x appear to have collided at $(t_2 - t_1)/2$ time in the past.

project to curves. Faces project to regions bounded by curves. Since edges project only to curves, not to straight lines, the algorithm approximates the projected edges by a series of straight line segments that closely match the curve. Wherever two projections of edges of A and B intersect, there is a potential EE collision. Likewise, wherever there is a projection of a vertex of A (resp. B) that lies in a projection of a face of B (resp. A), there is a potential VF (resp. FV) collision. When a point of A and a point of B project to a common point on the screen, there is a potential VV collision. Finally, if a vertex from one body projects onto the projection of an edge from the other body, there is a potential EV or VE collision.

For each potential collision, we find the preimages of the collision on the objects A and B. We view B as being transformed by the glide rotation and A as being transformed by the negative of the glide rotation flow, and calculate the times required for the point on A and the point on B to reach the screen. Then one half the difference in times gives the collision time for this potential collision. If a collision time is not in the interval $[0, 1]$, the potential collision is discarded and not considered further. The algorithm then computes the properness of the remaining potential collisions. All collisions which are not too improper are reported as collisions.

The version of the algorithm we have implemented uses a screen that consists of pixels that hold information about all the facets that project onto the pixel. In this approach, each backward (resp., forward) facing

vertex of body A (resp. B) is projected to a pixel on the screen, and each appropriately facing edge is drawn as one or more pixels on the screen. Each pixel may hold information from multiple vertices and edges. The pixels holding edge information also hold information about the adjacent faces, and this information is used to detect when vertex pixels potentially collide with faces.

The pixel-based approach is not the only possibility. An alternate method would treat the projection of the vertices as actual points, and the projections of edges as curves (perhaps approximated by straight line segments), and then use techniques of [28] to determine what faces are intersected by vertices of the other body, and what edges from one body intersect edges from the other body. The method could be made quite fast, although it is a little more prone to roundoff errors causing discrepancies in the interpenetration status from one simulation step to the next. It would be interesting to see it fleshed-out and implemented, but this has not been done yet.

7 Projection to the collision screen

The collision screen spirals around the glide rotation axis in a complicated way. The pixels on the collision screen are indexed with pairs $\langle r, h\rangle$. The r value is the radius, namely, the distance from the glide rotation axis. For a fixed value of r, the h value ("h" stands for "height") measures distances along a spiral which is perpendicular to the velocity flow of points at radius r. Points on objects A and B can collide only if they project to the same $\langle h, r\rangle$ pixel on the screen.

Figure 9 illustrates the collision screen for a glide rotation with parameters g and θ and rotation axis the y-axis. The cylinder in the figure has radius r and is centered on the glide rotation axis. The thick lines that spiral upward are the velocity flow of the glide rotation. The dotted spiraling line is the intersection of the collision screen and the cylinder: this line is at right angles to the flow lines. This line is the h axis and one of the flow lines is chosen as the t axis. (Our convention has been to choose the t and h axes so that they intersect at the positive z axis.)

A point C at $\langle x, y, z\rangle$ is assigned coordinates $\langle r, h\rangle$ on the collision screen and a time value t. The t value is the time at which the point impacts the collision screen and $\langle r, h\rangle$ is the position on the collision screen. Since the flow lines wrap around the cylinder, the choice of h and t is not unique. For this reason, some points need to be projected to more than a single point on the projection screen; this is called "doppeling".

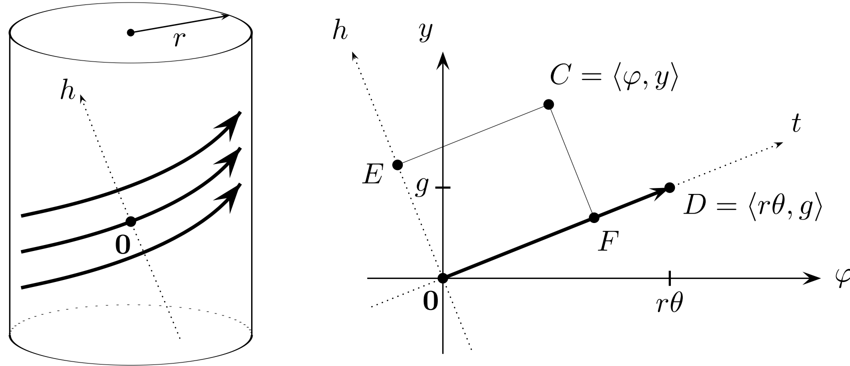


Figure 9: The thick lines in both figures show the velocity flow. The dotted line in the left figure is the intersection of the collision screen with the points at distance r from the rotation axis. The figure on the right shows the plane that wraps around the cylinder. The y-axis points up as usual. The φ axis measures azimuth and points rightward in the plane. The h and t values for a point C is found by projecting to E and F on the h- and t-axes.

The second part of Figure 9 shows the geometry for the calculation of r, h, and t. The cylinder for $r = \sqrt{x^2+y^2}$ has been unwrapped into a flat plane. Then $\varphi = \arctan(x/z)$ is the angle of the point C relative to the yz-plane. Thus $\langle r, \varphi, y\rangle$ are cylindrical coordinates for C. The vector $\overrightarrow{\mathbf{0}D}$ shows the motion under the glide rotation in a unit period of time. The time axis (t-axis) is coordinatized so that $\mathbf{0}$ is at $t = 0$ and D at $t = 1$. The h-axis is perpendicular to this flow. (Note that the slope of the h-axis, like the slope of the t-axis, depends on r.) The point C, at $\langle \varphi, y\rangle$, is projected orthogonally to the h- and t-axes. The h-coordinate of C is equal the distance $||\mathbf{0}E||$, and the t-coordinate of C is equal to $||\mathbf{0}F||/||\mathbf{0}D||$. By similar triangles, the h and t-coordinates of C are

$$h \;=\; \frac{y\cdot r\theta - \varphi\cdot g}{\sqrt{g^2+r^2\theta^2}} \qquad \text{and} \qquad t \;=\; \frac{y\cdot g + \varphi\cdot r\theta}{g^2+r^2\theta^2}.$$

The time t for the point E on the screen to reach C can be positive or negative. The value of φ was set with the multivalued arctangent function; by default, this puts φ in the interval $[-\pi, \pi]$ and thereby uniquely determines h and t. However, when doppeling, φ will get values outside the range $[-\pi, \pi]$.

For a point that lies on the glide rotation axis, $r = 0$ and the φ value is undefined. However, we can still use the above formulas for h and t, with $\varphi = 0$, so that $h = 0$ and $t = y/g$ for such points.

Projection when the glide rotation is just a translation with $\theta = 0$. Although the above construction still works, it is easier to just orthogonally project to the collision screen. This case is further simplified by the fact that straight edges project to straight lines on the screen.

8 Doppeling

Recall that θ denotes the total rotation of the glide rotation, whereas φ is used for cylindrical coordinates of points, with $\varphi = \arctan(x/z)$. As an example, suppose $\theta = 20^\circ$, and consider a point u on object B which has φ value equal to 10°. As B glides, the value of φ of the point u increases up to $10^\circ + \theta = 30^\circ$. Thus, the point u can potentially collide with a point v on the stationary object A only if the φ value for v is between 10° and 30°.

A more problematic case is when u has $\varphi = 170^\circ$. Then u may collide with points v on A that have φ value between 170° and 190°. The problem is that a point v with φ value greater than 180° (190°, for example) is by default projected to the collision screen using $\varphi = -170^\circ$ instead of $\varphi = 190^\circ$. These two choices for φ will, in general, project the point to different points on the collision screen and give it different collision times t. On the other hand, we cannot just use $\varphi = 190^\circ$ for such points v. A point v with $\varphi = -170^\circ$ can potentially collide with points u on B that have $\varphi \in [-180^\circ, -170^\circ]$ or have $\varphi \in [170^\circ, 180^\circ]$. Thus, v must be projected twice to the collision screen, with both $\varphi = -170^\circ$ and $\varphi = 190^\circ$.

Thus, for general values of θ (and returning to using radians), for any point v of object A with φ value in the interval $[-\pi, -\pi + \theta]$, the point must be projected twice to the collision screen, once with φ and once with $\varphi + 2\pi$. This works well as long as $\theta < \pi$. (In any event, usually $\theta \leq \pi/4$ in order to control the glide rotation error, c.f. Section 4.3.) When a point is twice projected, we call it a "doppeled" point.

As discussed in the next section, the collision detection algorithm works by projecting, one-by-one, each polygonal face to the collision screen. A face is projected as a series of edges. When projecting a face of A we must check whether it has any points that need to be doppeled. If so, the face's edges must be modified appropriately to properly doppel portions of the edges. There are several cases to consider, but the main ones are shown in Figures 10 and 11. In Figure 10, a polygonal face contains points that need to be doppeled, but does not intersect the glide axis. The figure shows a top view, looking down the glide axis, and the polygon is seen to intersect both the $\varphi = \pm\pi$ line and the $\varphi = \theta - \pi$ line. The polygon is split by adding new

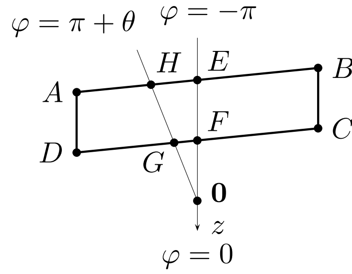


Figure 10: The polygon $ABCD$ is replaced by the two polygons $AEFD$ and $BCGH$.

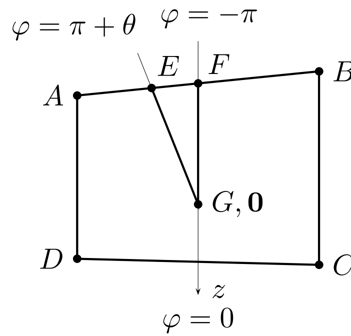


Figure 11: The polygon $ABCD$ is replaced by the "polygon" $GEBCDAF$.

vertices E, F, G and H. Vertices G and H use the φ value $\pi + \theta$, whereas vertices E and F use $\varphi = -\pi$. The single polygon is split into two polygons $AEFD$ and $BCGH$, and both are projected to the collision screen.

Figure 11 shows a polygon $ABCD$ that intersects the glide rotation axis and thus contains points that need to be doppeled. For this polygon, we create new vertices E, F, and G, with G the point where the polygon intersects the glide rotation axis (usually, G is not the same as $\mathbf{0}$). Then, the polygon $ABCD$ is replaced by the "polygon" $GEBCDAF$: the edges around this polygon bound a non-self-overlapping region on the collision screen since the φ starts off at $\pi + \theta$ at E and monotonically decreases down to $\varphi = -\pi$ at F.

The situation is similar for body B, but is simpler since doppeling is not needed. However, we still need to take care of edges that cross over the $\varphi = \pm\pi$ boundary. Projecting from body B uses exactly the same algorithm as for projecting from body A except that now θ is set equal to zero. For example, in the situation of Figure 10, when we work with body B and have θ replaced by zero, the points E and H correspond to the same point on the edge of B, except that E has $\varphi = -\pi$ and H has $\varphi = +\pi$.

9 Pixel-based algorithm implementation

We now outline the pixel-based algorithm that we have implemented. The input to the algorithm includes the initial and final positions (and orientations) as given by N_i, N_f, M_i and M_h. The resolution of the rasterized collision screen also must be specified. The resolution is specified in terms of a distance between pixels (the r and h values are discretized to this resolution), as well as a resolution in time.

The algorithm first determines the glide rotation parameters g and θ (see Section 4.1). The coordinate system is chosen so that the y-axis is the rotation axis and so that the (bounding spheres of the) two objects lie as close to the xz-plane as possible. Based on the bounding spheres, we attempt to place the z-axis so that no doppeling will be needed. We also do a bounding spheres test by bounding the maximum and minimum r, h and t values for all points on A and on B and checking whether any collision is possible. If not, the algorithm halts, reporting no collisions. Otherwise, all initial vertex positions, edge directions and face normals are computed in this coordinate system to save later re-computation.

The second step clips edges so that only forward (resp., backward) faces of B (resp., A) will be considered. The clipping is done separately for each face of A and B. Whenever a face is clipped in a non-trivial way, two of its edges are also clipped. Since each edge abuts two faces, this means each edge can be potentially split into three segments. In the same step, doppeling information is also recorded for each edge. Doppeling can further split an edge into up to three segments. In addition, for doppeling of the type as shown in Figure 10, a face may also need to be doppeled. Note that doppeling creates new edges; these are called *virtual edges* (e.g., edges GH and EF in Figure 10 and GE and FG in Figure 11). The result of the second step is a list of clipped edges, i.e., original object edges and (sub)edges generated by the clipping and doppeling. With each (sub)edge, we store information about (a) which original edge (if any) it is a subedge of, (b) whether the edge is virtual, (c) the identity of its left and right faces if they are forward facing. (Sub)edges from B (resp., A) that do not have any adjacent forward (resp., backwards) facing edge can be discarded.

The third step projects the (sub)edges to pixels in the collision screen space, generating a list of *pixel records*. Each edge is considered separately. We find the r and h values of the edges' endpoints, and the time t at which the endpoints contact the collision screen (as described in Section 7). Then, a divide and conquer scheme projects the whole edge to a curve formed of pixels on the collision screen. This is done by projecting the midpoint of the

edge to the collision screen and recursively considering the first and second halves of the edge. The divide-and-conquer assumes that the positional accuracy in projecting to the pixel screen needs to be no better than the pixel size, and likewise that the time value need be no more accurate than the time resolution input value.[4] Once these accuracies are achieved, or once the line segments are only one pixel long, the rest of the projected edge is filled in with a Bresenham algorithm. If successive pixels from an edge project to the same r value, they are combined into a single pixel record; this is done to reduce the number of pixel records that are needed — later stages of the algorithm will always scan all pixels with the same r value at once. Each projected pixel record contains its r value, its maximum and minimum h values, h_{min} and h_{max}, and its time value t. In addition, the pixel record holds information about which edge it comes from, about the left and right adjacent (appropriately facing) faces, about the time-duration of the pixel (this is important for edges that hit the collision screen somewhat perpendicularly since the range of t values for the pixel could be substantial), and whether it is the projection of a vertex of the body. To aid later steps, we also store some redundant information with each pixel, including information about the slope and direction of the projection of the edge through that point.

In the fourth step, the pixels are sorted lexicographically, first bucket-sorted by their r value, and second shell-sorted by their h_{min} value. We then make a copy of the pixel arrays, and re-sort so that the copy is sorted lexicographically by r and then h_{max}. The pixels for body A and for body B are kept separate. We also create a list of pixels from vertices of A and pixels from vertices of B, still sorted by r and h.

The fifth step finds vertices of body A that project into the interior of a face of body B. For each appropriate value of r, we scan the following in order of increasing h values: (i) pixels from vertices of A and (ii) pixel records from B sorted by h_{min}, and (iii) pixel records from B sorted by h_{max}. Pixels of type (ii) tell us when the projection of a face of B is entered; pixels of type (iii) tell us when the projection of a face of B is exited. For pixels of type (i), we record that the vertex is potentially colliding with all the currently entered faces of B. This process is repeated with the roles of A and B reversed.

The sixth step finds places where the projection of an edge of A crosses or intersects the projection of an edge of B. For this, pixel records are scanned lexicographically. Every time a pixel is found where two non-virtual edges

[4]This is the only place where the time resolution value is used.

cross, we store information about the potential intersection in a hash table.[5] The edges' pixels have time information, and if a pair of edges intersects at multiple pixels (say if the edges coincide for a series of pixels), then we use the pixel where the time until the potential collision is minimized. Some of these EE collisions involve endpoints. These are stored as EV, VE or VV collisions in another hash table.

The fifth and sixth steps extracted all the potential EE, VV, EV, VE, FV and VF collisions. The final steps resolve these into EE, FV and VF collisions. Collisions with properness value below a given threshold are discarded (the application has to determine the threshold; -0.1 works well for driving game applications). For collisions between two edges, the information about the r, h and t values for the collision are obtained from the pixel records without further ado. For collisions between vertices and faces, the r and h position of the vertex is of course known. If the collision was resolved from a VE, EV, or EE collision, then the pixel records give the collision time. However, if the collision is a VF or FV with the vertex colliding with the interior of the face on the pixel screen, then the collision time cannot be obtained from just the pixel record. Instead, an iterative Newton method is used to find the point on the face that projects to the $\langle r, h \rangle$ position on the collision screen where the vertex is projected, and from this, the collision time t is easily calculated. Convexity considerations prove that the Newton method has fast guaranteed convergence.

The algorithm returns a list of potential EE, FV and VF collisions. Each collision record contains the following information: (a) the edges, or the vertex and face, that have collided, (b) the time of the collision, (c) the spatial position and normal vector of the collision, and (d) its properness.

9.1 Run times

We have tested the algorithm described above with several shapes, including with cubes, with the simple polygonally modeled car shown in Figure 12, and with a torus composed of 300 rectangular faces and thus 600 edges. Although runtime comparisons are very sensitive to details of implementations and of the execution environment, it is useful to report some typical runtimes. We report runtimes at three resolutions for the collision screen pixels: at the lowest resolution, the length of a typical short edge that is projected sideways onto the collision screen is approximately 6 pixels. For the medium resolution, the length is approximately 12 pixels. For the high resolution

[5] Virtual edges are never part of a reported collision; instead, they serve only to mark the boundaries of faces.

the length is approximately 24 pixels. (The runtime is independent of the overall size of the collision screen since pixel data is stored sparsely; instead, the runtime depends on the number of pixels occupied by features.) The approximate runtimes in microseconds are reported in the table below; tests were performed on a 2.4GHz Pentium IV. The first line is for a cube colliding with a cube, the second line for a car versus car, and the third line for two colliding complex tori.

	low res	medium res	high res
Cubes:	56 μs	63 μs	77 μs
Cars:	109 μs	128 μs	170 μs
Tori:	2040 μs	2560 μs	3480 μs

Clearly there is a direct tradeoff between the runtime of the algorithm, and the resolution (and thereby the accuracy). Even the low resolution collisions work qualitatively quite well, although a close look will show that the collisions are only approximate. The choice of resolution is application dependent.

In general, for a fixed resolution, the runtime is expected to be approximately $O(n \log n)$ for the part of the algorithm that projects edges and vertices to pixels on the collision screen and sorts the pixels by their r and h values, since sorting $O(n)$ records takes time $O(n \log n)$. The rest of the algorithm is expected to take time approximately $O(m)$ where m is the number of pairs of facets from A and B that project to the same pixel on the screen. In the worst case, $m = O(n^2)$; indeed this is unavoidable, since there could be $\Omega(n^2)$ many collisions that need to be reported. However, if $m \approx n^2$, the runtime potentially could be reduced significantly if we modified the algorithm so that it sorted pixels by time t in addition to by r and h. This change would allow the algorithm to work well with highly non-convex objects, for instance screws threads in a screw hole.

It is difficult to make runtime comparisons with other published algorithms, since the bulk of these use extensive hierarchical methods to cull collision testing, whereas our numbers above are for a full collision test.

We can make a fair comparison of runtime with the $O(n^2)$ runtime algorithms of Angel Studios. We have only been able to make very crude, approximate comparisons, but our algorithm appears to be close in speed to Angel Studios's algorithms for a simple polygonal car, or perhaps slightly slower. We should add that the Angel Studios's algorithms have been greatly optimized over a period of years, whereas our own implementation is fairly preliminary, thus it is likely that better implementations can improve our

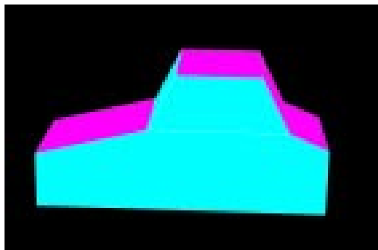


Figure 12: A simple polygonal model of a car, with 16 vertices, 26 edges, and 12 faces.

algorithm's speed to be the faster.[6] More importantly, our new algorithm handles rotation properly, gives much higher quality results and more robust results, and works well in many more situations.

We can also compare runtimes with the algorithms of [27]. Their algorithms solve the penetration depth problem for non-convex bodies under the assumption that there is an unknown purely translational motion. For colliding tori, they report times of 0.3 seconds to 3.7 seconds, on a 1.6GHz Pentium with the use of GeForce3 hardware acceleration, depending on whether the tori are interlocked. They do not specify how their tori were designed, but their tori apparently have polygonal complexity similar to our own, with several hundred faces: our own runtimes are nearly two to three orders of magnitude faster and are not particularly sensitive to whether the tori are interlocked. This comparison is not completely fair however, since [27] are seeking an unknown translational amount, whereas our algorithm works with a known motion (a given glide rotation).

9.2 Some examples

Figures 13 and 14 show screen shots of collision detection between two tori. The initial and final positions of the tori are shown in part (a) and (e) of Figure 13. From the point of view of the torus in the middle (object B), the other torus (object A) has moved completely though it in a glide rotation along the screw motion shown by the black curves. In (a), torus A is below B; in (e) it has moved up to well above B. Although it is difficult to see in the figures, A has also moved considerable distance away from the viewer. The first collision between A and B is shown in (b) of the figure. Here the top front of A has collided with the bottom front of B. (By the "front" of A

[6] If nothing else, we use double precision floating point, even though single precision would suffice.

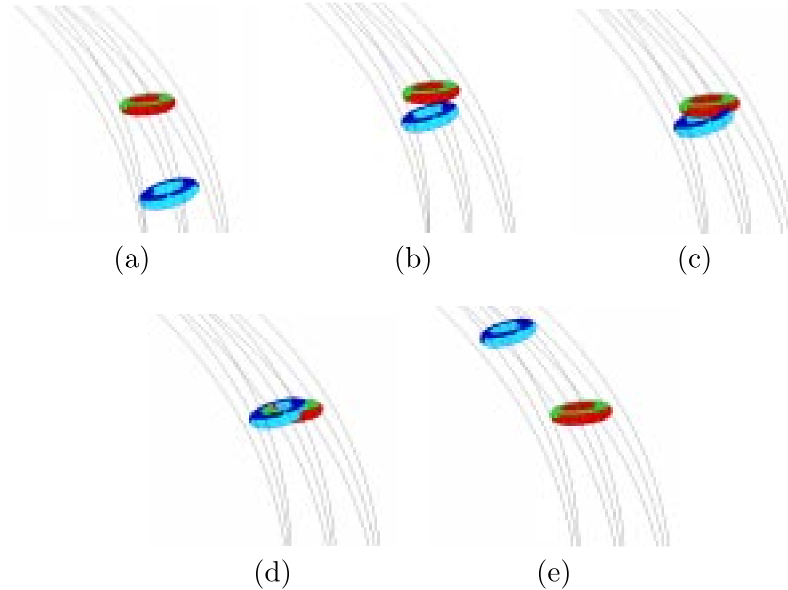


Figure 13: Two tori colliding. Torus A is moving away and upward. Torus B is stationary.

is meant the part most distant from the viewer; but the "front" of B is the part closest to the viewer.) In (c), the second collision is shown. Here the left top part of A has collided against the bottom part of B. In (d), the third collision is the right portion of the front of A colliding with the back inside of B. These three collisions in Figure 13 are all the proper collisions that occur between A and B. Figure 14 shows the tori A and B drawn on the rasterized collision screen. The collision algorithm, in effect, finds points of intersection from the overlaid drawings.

10 Conclusions

We have described a robust algorithm for collision detection which gives good approximate results. The algorithm reports, all at once, all the potential collisions between two bodies over a period of time.

The robustness means that it is unlikely that the algorithm can become confused about whether two bodies are interpenetrating. The approximateness does mean that sometimes the algorithm may report a collision when, in fact, the two bodies have only approached each other closely, and that,

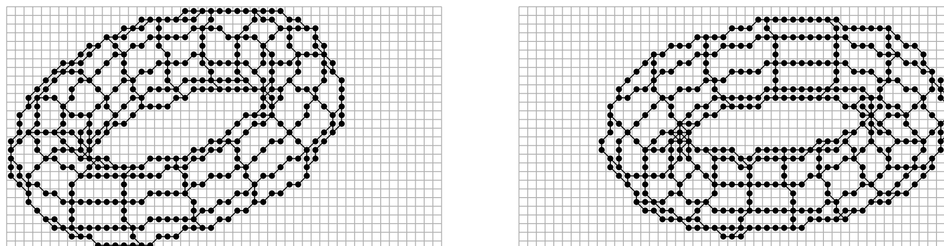


Figure 14: The rasterized projections of tori A and B onto the collision screen. The (barely visible) short lines between pixels show the connectivity of projected edges. The dimensions of the collision screen is automatically determined by the algorithm based on the r, h resolution levels and the extents of the objects; in this case, it has dimensions 28×50.

much more rarely, valid collisions where the bodies just barely interpenetrate can be missed, However, with care, the algorithm can be designed so that, during the course of multiple simulation steps, two rigid bodies never end up interpenetrating without a reported collision.[7] One difficulty with robustness is, in part, that two bodies might approach each other closely or even interpenetrate slightly without generating a collision, and then, in the next simulation step, as the glide rotation motion may have completely changed, the bodies may appear to be already interpenetrated and to be moving in such a way that no collision is occurring or has recently occurred. To make the collision detection fully robust, the algorithm could be modified to find all collisions where they approach to within a pixel's resolution of each other. This would require a relatively simple change to (only) step six of the algorithm and would increase the runtime only modestly; however, it might mean that the pixel resolution would need to be finer so as to not trigger too many collisions when the bodies have not actually collided. A second difficulty with robustness arises from the assumption of relative screw motion. It could happen that the glide rotation for the relative motion of A and B shows no intersection, but A has a collision halfway through the time step with a third object. Then, if A and B are moved to their positions halfway through the time step, it may happen that they actually interpenetrate due to their relative motion not actually being a glide rotation. (Similar problems arise in other scenarios, such as when interpenetrated objects are displaced to remove the interpenetration.) One way to

[7]This kind of robustness is very important for many applications, in particular, when two objects are in static contact, they frequently are almost touching or are barely interpenetrated.

reliably avoid this kind of problem is to keep track of the last time objects were known not to be interpenetrating, and check for collisions from these last known 'good' positions.

We expect that our algorithms can be improved or extended in several ways. First, there are several alternative implementations of the algorithm described above, that might give a faster implementation. One such possibility was already mentioned at the end of Section 6. To mention another, we could have used a hash table to hold all pixels projected to the screen and thereby detect places where edges intersect or cross (instead of sorting and sequentially scanning). Second, as we noted above, the algorithm could be extended to handle highly non-convex objects, such a screw fitting into a screw hole, by sorting pixels by time. Third, it would be interesting to extend our algorithm to other rigid shapes, say spheres or cylinders or, more generally, quadrics. Fourth, to handle highly complex objects, the algorithm should be extended with hierarchical methods. Since the runtime is approximately $O(n \log n)$, it may be too slow for objects with many thousands of faces. Of course, a large speedup can be obtained with space partitioning methods applied to (swept) bounding volumes to prune intersection testing. Another major speedup could also be obtained for complex objects by using a lower resolution polygonal bounding volume that approximates the shape of the object. The lower resolution bounding volumes could be intersected with the glide rotation algorithm and, where they indicate potential collisions, the higher resolution surfaces in just the regions of the potential collisions could be collided, again with the glide rotation method, for accurate collision detection.

A final natural question is whether the rasterized algorithm lends itself well to implementation on graphics hardware. Indeed, graphics hardware acceleration has been successfully used for motion planning [33], for computation of Voronoi diagrams [23], and for hardware-based detection of interpenetration [2, 22, 32]; see also [38]. The recent development (c.f. [42]) of sophisticated algorithms on graphics chips also gives hope that graphics hardware could accelerate our collision detection algorithm.

References

[1] P. K. Agarwal, L. J. Guibas, S. Har-Peled, A. Rabinovitch, and M. Sharir, *Penetration depth of two convex polytopes in 3D*, Nordic J. Computing, 7 (2000), pp. 227–240.

[2] G. Baciu and W. S. K. Wong, *Image-based techniques in a hybrid collision detector*, IEEE Transactions on Visualization and Computer Graphics, 9 (2003), pp. 254–271.

[3] D. Baraff, *Fast contact force computation for nonpenetrating rigid bodies*, Computer Graphics, 28 (1994), pp. 23–34. Proc. SIGGRAPH'94.

[4] ——, *Rigid body simulation II—nonpenetration*, 1997. Siggraph'97 Course Notes.

[5] B. G. Baumgart, *A polyhedral representation for computer vision*, in Proc. AFIPS Natl. Computer Conf., Vol. 44, 1975, pp. 589–596.

[6] J. W. Boyse, *Interference detection among solids and surfaces*, Communications of the ACM, 22 (1979), pp. 3–9.

[7] S. Cameron, *A study of the clash detection problem in robotics*, in Proc. IEEE International Conf. on Robotics and Automation, 1985, pp. 488–493.

[8] ——, *Collision detection by four-dimensional intersection testing*, IEEE Transaction on Robotics and Animation, 6 (1990), pp. 291–302.

[9] ——, *Enhancing GJK: Computing minimum and penetration distances between convex polyhedra*, in Proc. IEEE International Conf. on Robotics and Automation, 1997, pp. 3112–3117.

[10] J. Canny, *Collision detection for moving polyhedra*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 8 (1986), pp. 200–209.

[11] A. Chatterjee and A. Ruina, *A new algebraic rigid-body collision law based on impulse space considerations*, Journal of Applied Mechanics, 65 (1998), pp. 939–951.

[12] D. P. Dobkin, J. Hershberger, D. G. Kirkpatrick, and S. Suri, *Computing the intersection-depth of polyhedra*, Algorithmica, (1993), pp. 518–533.

[13] D. P. Dobkin and D. G. Kirkpatrick, *Determining the separation of preprocessed polyhedra — a unified approach*, in Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science #443, Springer Verlag, 1990, pp. 400–413.

[14] J. Eckstein and E. Schömer, *Dynamic collision detection in virtual reality applications*, in Proc. 7th Intl. Conf. on Computer Graphics and Visualization and Interactive Digital Media, WSCG'9, 1999, pp. 71–78.

[15] S. Ehmann and M. C. Lin, *Accelerated proximity queries between convex polyhedra by multi-level voronoi marching*, in Proc. International Conf. on Intelligent Robots and Systems, 2000, pp. 2101–2106.

[16] ——, *Accurate and fast proximity queries between polyhedra using convex surface decomposition*, Computer Graphics Forum, 20 (2001), pp. 500–510. Proc. Eurographics 2001.

[17] J. Erikson, L. J. Guibas, J. Stolfi, and L. Zhang, *Separation-sensitive collision detection for convex objects*, in Proc. ACM-SIAM Symp. on Discrete Algorithms (SODA), 1999, pp. 327–336.

[18] T. Giang, G. Bradshaw, and C. O'Sullivan, *Complementarity based multiple point collision resolution*, in Prof. Fourth Irish Workshop on Computer Graphics (Eurographics, Ireland), 2003, pp. 1–8.

[19] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, *A fast procedure for computing the distance between objects in three-dimensional space*, IEEE J. Robotics and Automation, RA-4 (1988), pp. 193–203.

[20] H. Goldstein, *Classical Mechanics*, Addison-Wesley, 1950.

[21] S. Gottschalk, M. C. Lin, and D. Manocha, *OBBTree: A hierarchical structure for rapid interference detection*, in Proc. ACM SIGGRAPH'96, New York, 1996, ACM Press, pp. 171–180.

[22] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, *CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware*, in Proc. ACM SIGGRAPH-Eurographics Workshop on Graphics Hardware, 2003, pp. 25–32.

[23] K. E. Hoff III, T. Culver, J. Keyser, M. Lin, and D. Manocha, *Fast computation of generalized Voronoi diagrams usings graphics hardware*, Computer Graphics, 33 (1999), pp. 277–286. Siggraph'99.

[24] P. M. Hubbard, *Collision detection for interactive graphics applications*, IEEE Transactions on Visualization and Computer Graphics, 1 (1995), pp. 218–230.

[25] B. Kim and J. Rossignac, *Collision prediction for polyhedra under screw motions*, in Proc. 8th ACM Symp. on Solid Modeling and Applications, 2003, pp. 4–10.

[26] Y. J. Kim, M. C. Lin, and D. Manocha, *DEEP: Dual space expansion for estimating penetration depth between convex polytopes*, in Proc. IEEE International Conf. Robotics and Automation, 2002.

[27] Y. J. Kim, M. A. Otaduy, M. C. Lin, and D. Monocha, *Fast penetration depth computation for physically-based animation*, ACM Transactions on Graphics, (2002), pp. 23–31. SIGGRAPH 2002.

[28] D. G. Kirkpatrick, *Optimal search in planar subdivision*, SIAM J. Computing, 12 (1983), pp. 28–35.

[29] D. G. Kirkpatrick, J. Snoeyink, and B. Speckmann, *Kinetic collision detection for simple polygons*, International J. Computational Geometry, (2002), pp. 3–27.

[30] D. G. Kirkpatrick and B. Speckmann, *Kinetic maintenance of context-sensitive hierarchial representations for disjoint simple polygons*, in Proc. 1st ACM Symp. on Computational Geometry, 2002, pp. 179–188.

[31] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, *Efficient collision detection using bounding volumes of k-DOPs*, IEEE Transactions on Visualization and Computer Graphics, 4 (1998), pp. 21–36.

[32] D. Knott and D. K. Pai, *CInDeR: Collision and interference detection in real-time using graphics hardware*, in Proc. Graphics Interface, 2003, pp. 73–80.

[33] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg, *Real-time robot motion planning using rasterizing computer graphics hardware*, Computer Graphics, 24 (1990), pp. 327–335. SIGGRAPH'90.

[34] C. Lennerz, E. Schömer, and T. Warken, *A framework for collision detection and response*, in Proc. 11th European Simulation Symposium and Exhibition (ESS'99), 1999, pp. 309–314.

[35] M. Lin, *Collision Detection for Animation and Robotics*, PhD thesis, U.C. Berkeley, 1993.

[36] M. C. Lin and J. F. Canny, *Efficient algorithms for incremental distance computation*, in Proc. IEEE International Conference on Robotics and Automation, 1991, pp. 1008–1014.

[37] M. C. Lin and S. Gottschalk, *Collision detection between geometric models: A survey*, in Proc. of IMA Conf. on Mathematics of Surfaces, 1998, pp. 37–56.

[38] D. Manocha et al., *Interactive geometric computations using graphics hardware*, 2002. SIGGRAPH Course Notes #31.

[39] B. Mirtich, *V-Clip: Fast and robust polyhedral collision detection*, ACM Transactions on Graphics, 17 (1998), pp. 177–208.

[40] ——, *Timewarp rigid body simulation*, in Proc. ACM SIGGRAPH 2000, 2000, pp. 193–200.

[41] K. Myszkowski, O. G. Okunev, and T. L. Kunii, *Fast collision detection between complex solids using rasterizing graphics hardware*, The Visual Computer, 11 (1995), pp. 497–512.

[42] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, *Ray tracing on programmable graphics hardware*, ACM Transactions on Graphics, 21 (2002), pp. 703–712. SIGGRAPH 2002.

[43] S. Redon, A. Kheddar, and S. Coquillart, *An algebraic solution to the problem of collision detection for rigid polyhedral objects*, in Proc. IEEE International Conference on Robotics and Automation, vol. 4, 2000, pp. 3733–3783.

[44] ——, *CONTACT: Arbitrary in-between motions for collision detection*, in Proc. 10th IEEE International Workshop on RObot and Human Interactive Communication (ROMAN'2001), 2001, pp. 106–111.

[45] ——, *Fast continuous collision detection between rigid bodies*, Computer Graphics Forum, 21 (2002). Proc. Eurographics 2002.

[46] ——, *Gauss's least constraints principle and rigid body simulation*, in Proc. IEEE International Conference on Robotics and Automation, vol. 1, 2002, pp. 517–522.

[47] ——, *Hierarchial back-face culling for collision detection*, in Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, 2002, pp. 3036–3041.

[48] J. Rossignac, A. Megahed, and B. D. Schneider, *Interactive inspection of solids: Cross-sections and interferences*, in SIGGRAPH'92, 1992, pp. 353–360.

[49] J. J. Rossignac and J. J. Kim, *Computing and visualizing pose-interpolating 3D motions*, Computer Aided Design, 33 (2001), pp. 279–291.

[50] E. Schömer and C. Thiel, *Efficient collision detection for moving polyhedra*, in Proc. 11th Annual ACM Symp. on Computational Geometry, 1995, pp. 51–60.

[51] J. M. Selig, *Geometrical Methods in Robotics*, Springer Verlag, New York, 1996.

[52] S. W. Shepperd, *Quaternion from rotation matrix*, Journal of Guidance and Control, 1 (1978), pp. 223–224.

[53] M. Shinya and M. C. Forgue, *Interference detection through rasterization*, Journal of Visualization and Computer Animation, 2 (1991), pp. 131–134.

[54] K. Shoemake, *Animating rotation with quaternion curves*, Computer Graphics, 19 (1985), pp. 245–254. SIGGRAPH'85.

[55] J. M. Snyder, A. R. Woodbury, K. Fleischer, B. Currin, and A. H. Barr, *Interval methods for multi-point collisions between time-dependent curved surfaces*, Computer Graphics, 27 (1993), pp. 321–334. SIGGRAPH'93.

[56] G. Vaněček Jr., *Back-face culling applied to collision detection of polyhedra*, Journal of Visualization and Computer Animation, 5 (1994), pp. 55–63.

[57] B. von Herzen, A. H. Barr, and H. R. Zatz, *Geometric collisions for time-dependent parametric surfaces*, Computer Graphics, 24 (1990), pp. 39–48. SIGGRAPH'90.