Collision Handling with Variable-Step Integrators

Andrea Neumayr
Martin Otter
andrea.neumayr@dlr.de
martin.otter@dlr.de
DLR Institute of System Dynamics and Control
Oberpfaffenhofen, Germany

ABSTRACT

This paper deals with collision handling of many convex shapes in offline simulations of Modelica-like object-oriented models, using variable-step integrators with error control. Hereby, it is assumed that collisions appear only from time to time and/or that path planning algorithms shall utilize the provided information for collision avoidance. Improvements to the Minkowski Portal Refinement (MPR) algorithm are proposed, as well as enhancements so that zero-crossing functions can be provided for the integrator in order to trigger events when contact starts and ends. The algorithms are demonstrated with a prototype implemented with the Julia programming language.

KEYWORDS

Collision, convex shapes, ESP, GJK, Julia, Modia, Modelica, MPR, penetration depth, SimVis

ACM Reference Format:
Andrea Neumayr and Martin Otter. 2017. Collision Handling with Variable-Step Integrators. In *EOOLT'17: 8th International Workshop on Equation-Based Object-Oriented Languages and Tools, December 1, 2017, Wessling, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3158191.3158193

1 INTRODUCTION

The goal is to introduce collision handling in object-oriented modeling languages such as Modelica [1]. In particular this means that contact handling must be suited for variable-step integrators, as used in offline simulations of Modelica environments. Collision handling is a large area with a huge literature, especially in the field of computer games. A survey of collision detection methods for convex and concave rigid bodies as well as for deformable shapes is for example given in [2, 19]. In [24] an overview of response calculation methods is given based on impulses. Enhancing Modelica with contact handling is proposed in [3, 9, 17, 23]. Nearly all of the literature in this field is devoted to real-time simulation engines utilizing integrators with *fixed-step sizes* and *without error control*. On the other hand, off-line simulation engines, especially of object-oriented modeling languages, typically use *variable-step* integrators *with error control*. Whenever a collision occurs, the model response is drastically changed and variable-step integrators perform usually no longer well, if such a change is just discontinuously applied. Instead, both the efficiency and the precision of the result are improved if state events are generated at the start and end of a contact, provided contact situations only occur from time to time, see e.g. [23].

Below, proposals are made for how to utilize collision handling for elastic contacts with classical variable-step integrators. If there are many collisions, QSS integrators [12] might be more appropriate. A prototype implementation has been made with the Julia programming language [6] and the plan is to include this implementation into the equation based and object-oriented modeling system Modia [10, 11] – a domain-specific extension of Julia. Visualization of the 3D scenes is performed with the freeware version of SimVis [4, 14], a visualization system of DLR based on OpenSceneGraph[1].

2 MATHEMATICAL DESCRIPTION

In this paper Differential-Algebraic Equations (DAEs) of the following form shall be treated

$$\mathbf{0} = \begin{bmatrix} f_d(\dot{\boldsymbol{x}}, \boldsymbol{x}, t, z_i > 0) \\ f_c(\boldsymbol{x}, t, z_i > 0) \end{bmatrix} \quad (a) \qquad \boldsymbol{J} = \begin{bmatrix} \frac{\partial f_d}{\partial \dot{\boldsymbol{x}}} \\ \frac{\partial f_d}{\partial \boldsymbol{x}} \end{bmatrix} \text{ is regular, } (c) \qquad (1)$$
$$z = f_z(\boldsymbol{x}, t) \qquad (b)$$

where $\boldsymbol{x} = \boldsymbol{x}(t)$. The Jacobian (1c) is required to be regular. Therefore (1a) is an index 1 DAE. (1b) defines zero-crossing functions $\boldsymbol{z}(t)$. Whenever a $z_i(t)$ crosses zero the integration is halted, functions f_d, f_c (1a) might be changed (for example by providing elastic material laws at a contact) and afterwards integration is restarted.

With the algorithms proposed in [22] a large class of DAEs can be transformed to (1) *without solving algebraic equations and retaining the sparsity of the equations*[2]. This includes 3D mechanical systems with kinematic loops, electrical circuits, hydraulic circuits, thermo-fluid networks and others. In particular, with the Modia prototype [10, 11] physical systems of these types can be defined on a high level and are then *automatically* transformed to (1). A number of methods exist for solving (1) numerically. In particular, under mild conditions Backward Differentiation Formula methods with order k and fixed or variable-step size h converge with $O(h^k)$, see [7, pp.

EOOLT'17, December 1, 2017, Wessling, Germany

ACM ISBN 978-1-4503-6373-0/17/12…$15.00
https://doi.org/10.1145/3158191.3158193

[1] OpenSceneGraph: http://www.openscenegraph.org/
[2] Usually, Modelica tools transform DAEs to ODEs (Ordinary Differential Equations in state space form). Hereby, linear and/or nonlinear algebraic equations might need to be solved and the sparsity of the equations might get lost.

51–54]. This means that a DAE integrator such as Sundials IDA [15, 16] can solve such systems.

The objective of the remaining sections is to provide zero-crossing functions $\boldsymbol{z}$ (1b) for systems where many shapes may potentially collide. For simplicity, only *convex* shapes shall be treated and the collision response shall be computed with elastic material laws. State events shall be triggered whenever contact starts or ends using the *root finding* option of classic variable-step integrators. For collision handling, the *signed distances* between the shapes are used as zero-crossing functions z_i. Hereby the convention is used that the signed distance δ between two convex shapes is positive, when the shapes are not in contact, it is zero when they touch each other, and it is negative otherwise. In the following, one goal is therefore to compute the signed distances between convex shapes.

3 COLLISIONS OF CONVEX BODIES

It is standard to perform collision handling of n potentially colliding shapes with *fixed-step* integrators in the following way:

1. *Broad phase*
   The shapes are approximated by other shapes where collision can be very cheaply determined. Furthermore, the approximated shapes can be placed in a hierarchy so that all direct and indirect children of a node cannot penetrate, if a collision is not possible for the node. Typically, $O(n\,log(n))$ collision tests are being made in this phase, instead of $O(n^2)$ tests.
2. *Narrow phase*
   The signed distances are computed for the potentially colliding shape pairs that have been identified in the broad phase.
3. *Response calculation*
   If two shapes are penetrated, either a finite force and/or torque is applied at the contact point, such as a spring/damper force element, depending on the penetration depth. Alternatively, a force/torque impulse is applied that leads to discontinuous changes of the velocities/angular velocities of the shapes.

There are several algorithms to compute the signed distance between two convex shapes. In particular, the following approach seems to be widely used and it is claimed that it is *one of the fastest methods currently available* [5]:

1. *Describing convex shapes with support mappings*
   Convex shapes are defined with functions support$(A, \boldsymbol{n})$ that return a point on the boundary of shape A that is farthest away in direction of vector $\boldsymbol{n}$. Support functions for polytopes, spheres, cylinders, cones, ellipsoids, convex hulls, etc. are for example given in [5, 20].
2. *Transforming a shape-shape to a point-shape distance problem*
   The *Minkowski difference* $A \ominus B$ of two shapes A and B is defined as ($\boldsymbol{r}_A$, $\boldsymbol{r}_B$ are absolute position vectors of points in A and B respectively):
   $$A \ominus B = \left\{ \boldsymbol{r}_A - \boldsymbol{r}_B : \ \boldsymbol{r}_A \in A, \ \boldsymbol{r}_B \in B \right\}. \tag{2}$$
   The Minkowski difference of two convex shapes A and B is convex, its support point is support$(A, \boldsymbol{n})$ − support$(B, -\boldsymbol{n})$, and the distance between A and B is the distance of $A \ominus B$ from zero. This distance is unique (so exactly one point on the boundary of $A \ominus B$ is closest to zero). For proofs of these properties, see for example [5]. Computing the distance between *two* convex shapes can therefore be transformed to the much simpler problem of determining the distance of *one* convex shape (= $A \ominus B$) from the origin.
3. *Computing the closest distance with the GJK algorithm*
   The GJK (Gilbert-Johnson-Keerthi) algorithm and its improved versions [5, 13] compute the closest distance between a convex shape and the origin, provided the origin is outside of the shape (= distance of non-penetrating shapes). The approach is conceptually simple by using support mappings to construct in the 3D case appropriate 0,1,2,3-simplexes (= points, lines, triangles, tetrahedrons) where the vertices of the simplexes are on the boundary of the shape and the simplexes are constructed to move closer and closer to the origin. If no further progress is possible and the origin is outside of the shape, then the closest distance of the shape from the origin is the closest distance of the final simplex from the origin. For penetrating shapes, the GJK algorithm stops with a simplex that has the origin in its interior.
4. *Computing the penetration depth with EPA*
   The EPA (Expanding Polytope Algorithm) by v.d. Bergen [5, sec. 4.3.8] computes the closest distance of a convex shape from the origin, if the origin is in the interior of the shape. The algorithm starts from the final simplex of the GJK algorithm and then expands the simplex to a polytope where in every iteration a new (support point) vertex is added. If no further progress is possible, the penetration depth is the closest distance of the final polytope from the origin.
5. *Determine contact points on the shapes*
   The (unique) contact point found in the Minkowski difference is transformed back to the (potentially non-unique) contact points on shapes A and B using the barycentric coordinates of the contact point with respect to the final simplex or polytope.

The GJK and EPA algorithms are conceptually simple, but the implementation is non-trivial and elaborate due to the many special cases (e.g. handling 0,1,2,3-simplexes in all situations) and due to various numerical problems that might occur when selecting the next simplex or computing a termination condition. This might be the reason why no robust open source implementation with a permissive free license seems to be available for the 3D-case.[3]

Based on the Minkowski difference, Snethen [26] proposed the Minkowski Portal Refinement (MPR) algorithm to detect whether two convex shapes penetrate and if this is the case compute an approximation of the penetration depth. The MPR algorithm in 3D is much simpler than GJK/EPA because it operates basically with triangles. The drawback is that MPR may only compute an approximation of the penetration depth in some situations. In [20] GJK/EPA and MPR are compared with several millions randomized benchmarks with various shape types. In [18] a nice, compact pseudo-code for the MPR algorithm is given and it is shown that

[3] The GJK/EPA based software solid3 from v.d. Bergen, https://github.com/dtecta/solid3, is available under GPL license.

MPR can also be utilized to compute the closest distance of non-penetrating convex shapes.

The BSD-licensed open source C-library libccd[4] provides an implementation of the MPR algorithm. It is used for example in the open source Bullet Physics SDK[5]. However, libccd does not compute the distance of non-penetrating shapes and can therefore not be used as basis of the investigation of this paper.

As will be shown below, the MPR algorithm can be used to compute zero-crossing functions for variable-step integrators since the potential approximation of the exact distance does not matter in this case. Response calculations based on *penetration depths* can only give a reasonable approximation of reality, if the *penetration volume* is small enough. In such cases, MPR computes usually the exact penetration depth. For these reasons, and since MPR is a reasonable simple algorithm, it will be used in the following. Previous publications of the MPR algorithm [18, 20, 26] are incomplete, because the handling of special cases is not defined, the termination conditions are incomplete and the signed distance is unnecessarily approximated [18]. The improved version of the MPR algorithm below fixes these issues.

4 IMPROVED MPR ALGORITHM

In this section an improved version of the MPR algorithm is proposed and defined in form of a Julia/Matlab like pseudocode. The algorithm has the following interface:

$$(\delta, \boldsymbol{e}_A, \boldsymbol{r}_A, \boldsymbol{r}_B) = \text{mpr}(A, B, tol_{rel} = 10^{-4}, i_{max} = 100). \quad (3)$$

The input and output variables have the following meaning:

A, B	Convex or concave shapes A and B.
tol_{rel}	The relative tolerance used for the approximation of the distance (default = 10^{-4}).
i_{max}	Maximum number of iterations to avoid infinite looping (default = 100).
δ	Signed distance between A and B. $\delta > 0$ if the shapes are not in contact, $\delta = 0$ if the shapes are touching and $\delta < 0$ if the shapes are penetrating. If a shape is concave, the distance computation is performed with respect to its convex hull[6].
$\boldsymbol{r}_A, \boldsymbol{r}_B$	The absolute position vectors to a point on the boundary of shape A and to a point on the boundary of shape B that are closest to each other if the shapes are not in contact and that have the largest penetration depth along $\boldsymbol{e}_A$, if the shapes are in contact.
$\boldsymbol{e}_A$	A unit vector such that $\boldsymbol{r}_B - \boldsymbol{r}_A = \delta \boldsymbol{e}_A$. If the surface around $\boldsymbol{r}_A$ is differentiable, $\boldsymbol{e}_A$ is normal to A in $\boldsymbol{r}_A$.

The following utility functions must be provided for the shapes

$$\begin{aligned} \boldsymbol{r}_s &= \text{support}(A, \boldsymbol{e}) \\ \boldsymbol{r}_c &= \text{centroid}(A). \end{aligned} \quad (4)$$

Function support$(A, \boldsymbol{e})$ computes the position vector $\boldsymbol{r}_s$ to a point on the boundary of A that is farthest away in the direction of the unit vector $\boldsymbol{e}$. Function centroid(A) computes the position vector to the centroid of shape A or to an internal point that is close to it. The support point of the Minkowski difference $A \ominus B$ is calculated with (4), according to (2) as:

$$S = \text{support}(A, B, \boldsymbol{e}) \quad (5)$$

and

$$\begin{aligned} S.\boldsymbol{r}_A &= \text{support}(A, \boldsymbol{e}) \\ S.\boldsymbol{r}_B &= \text{support}(B, -\boldsymbol{e}) \\ S.\boldsymbol{r} &= S.\boldsymbol{r}_A - S.\boldsymbol{r}_B \\ S.\boldsymbol{e} &= \boldsymbol{e}. \end{aligned}$$

The (overloaded) function returns a structure S, and in particular $S.\boldsymbol{r}$ which is the absolute vector to the boundary of the Minkowski difference closest to the origin in direction of unit vector $\boldsymbol{e}$ (= the support point of the Minkowski difference).

Below, the parts of the algorithm that are displayed in light blue color are not present in publications about MPR [18, 20, 26]. These improvements involve the handling of unphysical collision situations and in some collision cases (e.g. two spheres are colliding) the signed distance δ can be computed directly (TC1). These enhancements are explained in sections 4.1 and 4.2.

4.1 Determining the initial portal

In the first phase, the goal is to construct a tetrahedron with a non-zero volume such that point $\boldsymbol{r}_0$ is in the interior of the Minkowski difference of the shapes A and B and the three points $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ and $\boldsymbol{r}_3$ are points on its boundary. The triangle constructed by these three points is called *portal* (figure 1). Furthermore, the ray from $\boldsymbol{r}_0$ through the origin shall go through the portal. Two typical examples are shown in the next figure:

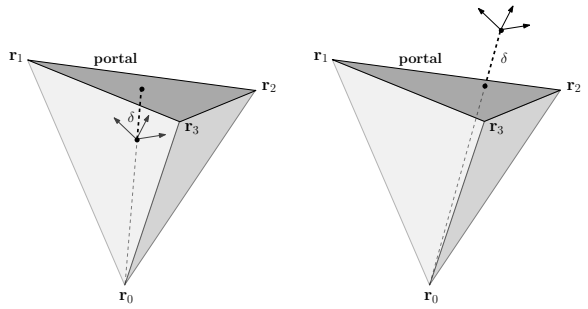

Figure 1: MPR portal for penetrating and non-penetrating shapes.

The first point $\boldsymbol{r}_0$ has to be in the interior of $A \ominus B$:

$$\begin{aligned} &\boldsymbol{r}_0 = \text{centroid}(A) - \text{centroid}(B) \\ &\textbf{if } |\boldsymbol{r}_0| \leq \varepsilon \\ &\quad \textbf{error}(\text{"Too large penetration."}) \\ &\textbf{end} \end{aligned} \quad (6)$$

If $|\boldsymbol{r}_0| = 0$, a division by zero will occur in (7). The reason is that the centre of the Minkowski difference is exactly in the origin, so shapes

[4] libccd: https://github.com/danfis/libccd
[5] bullet3: https://github.com/bulletphysics/bullet3
[6] Support mappings are also defined for concave shapes and result in the convex hull [5]. The distance between the convex hulls of two concave shapes can be useful for collision avoidance algorithms, as well as for cheap, precise tests whether contact is possible for deformable shapes, see for example [25].

A and B are largely overlapping. However, the MPR algorithm is based on the prerequisite that a ray from $\boldsymbol{r}_0$ to the origin passes the portal. If the portal and the origin are on different sides of $\boldsymbol{r}_0$, or if $\boldsymbol{r}_0 = \boldsymbol{0}$, the algorithm does no longer work. In order to guard against overflow and numerical inaccuracies, here and below a small number ε is used (say $\varepsilon \approx 10^{-12}$).

The first portal point $\boldsymbol{r}_1$ is found by searching from point $\boldsymbol{r}_0$ towards the origin:

$$\begin{aligned} \boldsymbol{e}_1 &= -\frac{\boldsymbol{r}_0}{|\boldsymbol{r}_0|} \\ S_1 &= \text{support}(A, B, \boldsymbol{e}_1) \\ \boldsymbol{r}_1 &= S_1.\boldsymbol{r} \end{aligned} \tag{7}$$

The second portal point $\boldsymbol{r}_2$ is found by searching in the direction of a vector that is perpendicular to $\boldsymbol{r}_0$ and $\boldsymbol{r}_1$:

$$\begin{aligned} &\boldsymbol{n}_2 = \boldsymbol{r}_0 \times \boldsymbol{r}_1 \\ &\textbf{if } |\boldsymbol{n}_2| \leq \varepsilon \\ &\quad \delta = -\boldsymbol{r}_1 \cdot \boldsymbol{e}_1 \\ &\quad \textbf{return } (\delta, \boldsymbol{e}_1, S_1.\boldsymbol{r}_A, S_1.\boldsymbol{r}_B) \\ &\textbf{end} \end{aligned} \tag{TC1}$$

$$\begin{aligned} \boldsymbol{e}_2 &= \frac{\boldsymbol{n}_2}{|\boldsymbol{n}_2|} \\ S_2 &= \text{support}(A, B, \boldsymbol{e}_2) \\ \boldsymbol{r}_2 &= S_2.\boldsymbol{r} \end{aligned} \tag{8}$$

If $|\boldsymbol{n}_2| = 0$ in (8), a division by zero would occur. The reason is that $\boldsymbol{r}_1$ is on the ray from $\boldsymbol{r}_0$ through the origin, as visualized in figure 2. Since $\boldsymbol{r}_1$ is already the closest point to zero in situation (TC1), the signed distance δ can be directly computed and the collision calculation can be terminated.

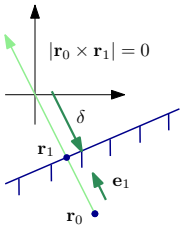

Figure 2: $\boldsymbol{r}_1$ is on the ray from $\boldsymbol{r}_0$ through the origin.

This situation occurs for example, whenever two spheres are colliding or if the closest points are on the line through the centroids of the two shapes.

The third portal point $\boldsymbol{r}_3$ is found by searching in the direction of a vector that is perpendicular to $\boldsymbol{r}_1 - \boldsymbol{r}_0$ and $\boldsymbol{r}_2 - \boldsymbol{r}_0$ and points towards the origin:

$$\begin{aligned} &\boldsymbol{n}_3 = (\boldsymbol{r}_1 - \boldsymbol{r}_0) \times (\boldsymbol{r}_2 - \boldsymbol{r}_0) \\ &\textbf{if } |\boldsymbol{n}_3| \leq \varepsilon \\ &\quad S_2 = \text{support}(A, B, -\boldsymbol{e}_2); \quad \boldsymbol{r}_2 = S_2.\boldsymbol{r} \\ &\quad \textbf{if } |(\boldsymbol{r}_2 - \boldsymbol{r}_1) \cdot \boldsymbol{e}_2| \leq \varepsilon \\ &\quad\quad \textbf{error}(\text{"Shapes are too thin."}) \\ &\quad \textbf{end} \\ &\quad \boldsymbol{n}_3 = (\boldsymbol{r}_1 - \boldsymbol{r}_0) \times (\boldsymbol{r}_2 - \boldsymbol{r}_0) \\ &\textbf{end} \\ &\boldsymbol{e}_3 = \frac{\boldsymbol{n}_3}{|\boldsymbol{n}_3|} \qquad \#\ |\boldsymbol{n}_3| = 0 \text{ is not possible} \\ &\boldsymbol{e}_3 = -\boldsymbol{e}_3 \ \textbf{if } \boldsymbol{e}_3 \cdot \boldsymbol{r}_0 > 0 \\ &S_3 = \text{support}(A, B, \boldsymbol{e}_3) \\ &\boldsymbol{r}_3 = S_3.\boldsymbol{r} \\ &\boldsymbol{n}_{3c} = (\boldsymbol{r}_2 - \boldsymbol{r}_1) \times (\boldsymbol{r}_3 - \boldsymbol{r}_1) \\ &\textbf{if } |\boldsymbol{n}_{3c}| \leq \varepsilon \\ &\quad S_3 = \text{support}(A, B, -\boldsymbol{e}_3) \\ &\quad \boldsymbol{r}_3 = S_3.\boldsymbol{r} \\ &\quad \textbf{if } |(\boldsymbol{r}_3 - \boldsymbol{r}_1) \cdot \boldsymbol{e}_3| \leq \varepsilon \\ &\quad\quad \textbf{error}(\text{"Shapes are too thin."}) \\ &\quad \textbf{end} \\ &\textbf{end} \end{aligned} \tag{9}$$

If $|\boldsymbol{n}_3| = 0$ in the second line of (9), a division by zero would occur. The reason is that $\boldsymbol{r}_2$ is on the ray from $\boldsymbol{r}_0$ to $\boldsymbol{r}_1$. This situation is depicted in figure 3.

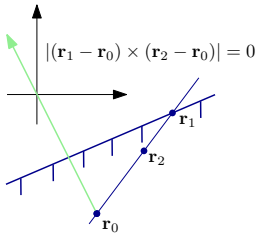

Figure 3: $\boldsymbol{r}_2$ is on the ray from $\boldsymbol{r}_0$ through $\boldsymbol{r}_1$.

In order to proceed, it is tried to find another portal point $\boldsymbol{r}_2$ by searching in the opposite direction of $\boldsymbol{e}_2$. If the newly found $\boldsymbol{r}_2$ is in the plane or close to it that is perpendicular to $\boldsymbol{e}_2$ and contains $\boldsymbol{r}_1$, then the Minkowski difference of the two shapes is planar (or very thin). Collision detection would then require to proceed with a 2D version of the MPR algorithm (using a line and not a triangle for the portal). However, in 3D it makes no sense to compute a penetration depth of a shape with zero volume. For this reason, an error is triggered.

In case the new $\boldsymbol{r}_2$ is *not* in this plane, $\boldsymbol{n}_3$ can be re-computed with the new $\boldsymbol{r}_2$ and it is then guaranteed that $|\boldsymbol{n}_3|$ cannot be zero, so a division with $|\boldsymbol{n}_3|$ can be performed.

At the end of (9), $\boldsymbol{n}_{3c}$ is computed as a vector that is perpendicular to $\boldsymbol{r}_2 - \boldsymbol{r}_1$ and to $\boldsymbol{r}_3 - \boldsymbol{r}_1$. If this vector is a zero vector, $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ and $\boldsymbol{r}_3$ are on one line and the triangle of these three points has a zero area. To proceed, $\boldsymbol{r}_3$ is newly computed with an opposite search direction $-\boldsymbol{e}_3$. If the newly found $\boldsymbol{r}_3$ is in the plane that is perpendicular to $\boldsymbol{e}_3$ and contains $\boldsymbol{r}_1$, then the Minkowski difference of the two shapes is planar or very thin and again an error message is triggered. Otherwise, it is guaranteed that the newly computed $\boldsymbol{r}_3$, together with $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ form a triangle that has a non-zero area.

The portal points $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ and $\boldsymbol{r}_3$ are now modified so that the ray from $\boldsymbol{r}_0$ through the origin passes through the newly constructed

portal. This is achieved by searching for new support points if the goal is not yet reached:

```
success = false
for i = 1 : i_max
  n_4 = (r_1 − r_0) × (r_3 − r_0)
  if n_4 · r_0 < −ε
    S_2 = S_3
    r_2 = S_2.r
    S_3 = support(A, B, n_4/|n_4|)
    r_3 = S_3.r
  else
    n_4 = (r_3 − r_0) × (r_2 − r_0)
    if n_4 · r_0 < −ε
      S_1 = S_3;
      r_1 = S_1.r
      S_3 = support(A, B, n_4/|n_4|)
      r_3 = S_3.r
    else
      n_4 = (r_2 − r_1) × (r_3 − r_1)
      if |n_4| ≤ ε
        error("Should not occur (reason is unclear).")
      end
      e_4 = n_4/|n_4|
      e_4 = −e_4 if e_4 · r_0 > 0
      success = true; break
    end
  end
end
if not success
  error("Max. number of iterations reached.")
end
```

(10)

The for-loop iterates until the ray goes through the portal or the maximum number of iterations is reached. In the latter case the algorithm is terminated with an error. With every iteration a new support point is computed and one of the previous support points is replaced by the new one. When computing the support structure S_3, a division by zero cannot take place, because this structure is only computed if $\boldsymbol{n}_4 \cdot \boldsymbol{r}_0 < -\varepsilon$.

Once the ray goes through the portal, the normal $\boldsymbol{n}_4$ to this portal is computed. If $|\boldsymbol{n}_4| \leq \varepsilon$, the portal points $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ and $\boldsymbol{r}_3$ are located on one line or in one point and the area of the portal triangle is zero (and the ray goes through this line or point). It is then no longer possible to continue with the algorithm and an error is triggered. On the other hand, if $|\boldsymbol{n}_4| > \varepsilon$, it is guaranteed that the portal triangle has a non-zero area and the algorithm continues.

4.2 Determining the portal closest to the origin

The goal of the next phase is to reduce the size of the portal and position it closer to the origin until no further progress is possible. This is performed in an iterative way by computing in every iteration one new support point $\boldsymbol{r}_4$ and then determining through which of the three triangles $(\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_4)$, $(\boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4)$ and $(\boldsymbol{r}_3, \boldsymbol{r}_1, \boldsymbol{r}_4)$ the ray from $\boldsymbol{r}_0$ through the origin is passing. This triangle is then used as a (smaller) new portal that is closer to the origin. A typical example is shown in figure 4. Note, termination conditions are marked in (8) and below as (TCx). The following algorithm performs the desired actions:

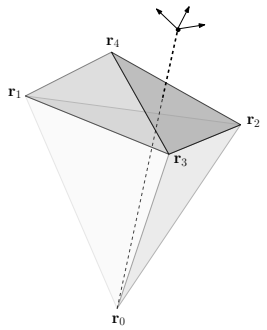

Figure 4: The next portal is $(\boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4)$, because the ray from $\boldsymbol{r}_0$ through the origin passes this triangle.

```
for i = 1 : i_max
  S_4 = support(A, B, e_4)                                   (11)
  r_4 = S_4.r
  if |(r_4 − r_1) · e_4| ≤ tol_rel                           (TC2)
    (δ, r_4, e_4b) = distanceToPortal(r_1, r_2, r_3, e_4)
    (r_4A, r_4B) = barycentric(S_1, S_2, S_3, r_4, e_4)
    return (δ, e_4b, r_4A, r_4B)
  end
  if isNextPortal(r_0, r_1, r_2, r_4)                        (12)
    S_3 = S_4;  r_3 = S_3.r
  elseif isNextPortal(r_0, r_2, r_3, r_4)
    S_1 = S_4;  r_1 = S_1.r
  elseif isNextPortal(r_0, r_3, r_1, r_4)
    S_2 = S_4;  r_2 = S_2.r
  else
    error("Should not occur (reason is unclear).")
  end
  n_4 = (r_2 − r_1) × (r_3 − r_1)
  e_4 = n_4/|n_4|        # |n_4| = 0 is not possible
end
error("Max. number of iterations reached.")
```

The function distanceToPortal(..) determines the point $\boldsymbol{r}_4$ in the interior or on the boundary of the triangle $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3$ that is closest

to the origin and $\delta = \boldsymbol{r}_4 \cdot \boldsymbol{e}_4 \leq 0 \ ? \ |\boldsymbol{r}_4| : -|\boldsymbol{r}_4|$. If $\boldsymbol{r}_4$ is in the interior of the triangle $\boldsymbol{e}_{4b} = \boldsymbol{e}_4$. Otherwise, $\boldsymbol{e}_{4b} = |\boldsymbol{r}_4| > \epsilon \ ? \ -\boldsymbol{r}_4/|\boldsymbol{r}_4| : \boldsymbol{e}_4$, so that $\boldsymbol{r}_{4B} - \boldsymbol{r}_{4A} = \delta \boldsymbol{e}_{4b}$.

The function barycentric(..) expresses $\boldsymbol{r}_4$ with barycentric coordinates of $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ and $\boldsymbol{r}_3$ and then uses the same barycentric coordinates on shapes A and B utilizing the corresponding support points, see for example [5].

The function isNextPortal(..) returns *true* if the ray of $\boldsymbol{r}_0$ through the origin passes the selected triangle:

$$\begin{aligned} \text{isNextPortal}(\boldsymbol{r}_0, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_4) = \\ ((\boldsymbol{r}_2 - \boldsymbol{r}_0) \times (\boldsymbol{r}_4 - \boldsymbol{r}_0)) \cdot \boldsymbol{r}_0 \leq 0 \ \textbf{and} \\ ((\boldsymbol{r}_4 - \boldsymbol{r}_0) \times (\boldsymbol{r}_1 - \boldsymbol{r}_0)) \cdot \boldsymbol{r}_0 \leq 0. \end{aligned} \tag{13}$$

In this final phase, the for-loop iterates until the portal closest to the origin is found or the maximum number of iterations is reached. In the latter case an error is triggered. The algorithm is terminated if the new support point $\boldsymbol{r}_4$ is on the plane of the portal (so $|(\boldsymbol{r}_4 - \boldsymbol{r}_1) \cdot \boldsymbol{e}_4| = 0$), that is, if no further progress is possible and the portal is the closest one to the origin (to which the ray passes). This situation is depicted in figure 5. As can be seen, the signed

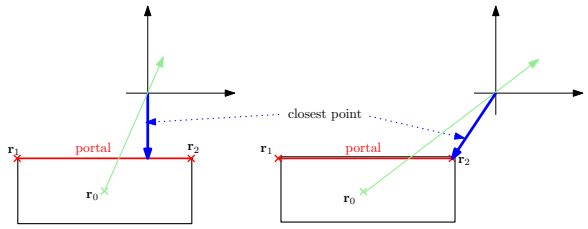

Figure 5: Closest point to zero of polyhedra.

distance is then either the perpendicular distance of the portal to the origin (see left part of figure 5) or it is the closest distance of one of the vertices or one of the edges of the triangle (see right part of figure 5). Therefore, the function distanceToPortal(..) is called to determine the shortest distance of the portal triangle to the origin. The returned point $\boldsymbol{r}_4$ is on the portal. This point is mapped to a point on shape A and shape B with function barycentric(..).

If the termination condition is not fulfilled, $\boldsymbol{r}_4$ cannot be on the plane of the portal. Therefore, it is possible to select either $\boldsymbol{r}_1, \boldsymbol{r}_2$, or $\boldsymbol{r}2, \boldsymbol{r}3$ or $\boldsymbol{r}_3, \boldsymbol{r}_1$ that form together with $\boldsymbol{r}_4$ a new portal that is smaller and closer to the origin and the ray from $\boldsymbol{r}_0$ to the origin passes it. Since it is guaranteed that the area of the new triangle is not zero, $|\boldsymbol{n}_4| > 0$ and a division by $|\boldsymbol{n}_4|$ is allowed.

4.3 Properties of improved MPR

The following properties of the improved MPR algorithm are used for the collision handling with variable-step integrators. These properties are summarized in the following new theorems. Hereby, A and B are convex shapes and δ_{mpr} is the signed distance returned by function mpr(A, B) under the assumption that computations are performed with infinite precision (and $tol_{rel} = 0, i_{max} = \infty$).

THEOREM 4.1. ***(Contact detection).***
The following holds:

1. $\delta_{mpr} > 0$*: A and B are not in contact to each other.*
2. $\delta_{mpr} = 0$*: A and B are touching each other.*
3. $\delta_{mpr} < 0$*: A and B are penetrating each other.*

PROOF. If shapes A and B are not in contact, the origin is located *outside* of the Minkowski difference $A \ominus B$. If they are penetrating, the origin is *inside* $A \ominus B$, and if they are touching, it is on the *boundary* of $A \ominus B$. If function mpr(A, B) is terminating successfully, the relationship between the ray from $\boldsymbol{r}_0$ to the origin through the final portal is always well-defined and the contact situation follows automatically:

- *Assume function mpr(..) terminates due to* (TC1)*:*
  In such a case $\boldsymbol{r}_1$ is located on the ray and no shape point is beyond the plane (in direction of the ray) that is perpendicular to the ray through point $\boldsymbol{r}_1$. Therefore, if the origin is between $\boldsymbol{r}_0$ and $\boldsymbol{r}_1$, it is inside $A \ominus B$ (so $\delta_{mpr} < 0$ and shapes A and B are penetrating). If $\boldsymbol{r}_1 = \boldsymbol{0}$ the shapes are touching ($\delta_{mpr} = 0$). If $\boldsymbol{r}_1$ is between $\boldsymbol{r}_0$ and the origin, then the origin is outside of $A \ominus B$ (so $\delta_{mpr} > 0$ and shapes A and B are not in contact).
- *Assume function mpr(..) terminates due to* (TC2)*:*
  In such a case, no shape point is beyond the portal plane (in direction of the ray). If the origin is on the same side of the plane as $\boldsymbol{r}_0$, then the ray from $\boldsymbol{r}_0$ passes first the origin and then the portal. Therefore, the origin is inside $A \ominus B$ (so $\delta_{mpr} < 0$ and shapes A and B are penetrating). If the origin is on the plane, then the ray from $\boldsymbol{r}_0$ passes the origin and this is only possible if the origin is on the portal. Since (TC2) implies that the portal is on the boundary, the origin is on the boundary of $A \ominus B$ and the shapes are touching ($\delta_{mpr} = 0$). If the origin is on the opposite side of the plane as $\boldsymbol{r}_0$, it is outside of $A \ominus B$ (so $\delta_{mpr} > 0$ and shapes A and B are not in contact).

□

Theorem 4.1 is sufficient in order that δ_{mpr} can be used as zero-crossing function for a variable-step integrator: It is standard for variable-step integrators with root-finding support to evaluate a zero-crossing function $z_i(t)$ at time instant $t_j + h$ of a completed integrator step with step-size h. If $z_i(t_j) \cdot z_i(t_j + h) \leq 0$ an interval $[t_j, t_j + h]$ is determined in which z_i crosses zero. Several algorithms with *guaranteed convergence* are known to reduce the interval in which the zero-crossing takes place until a prescribed tolerance for the final interval is met. For example, the integrators of the Sundials suite use a modified secant method [16, sec. 2.3], whereas the integrator DASSL [7] uses the method of Brent [8, pp. 58–59]. In both cases the interval is successively reduced in every iteration. Hereby, the only needed property is the sign of z_i (which is provided by the MPR algorithm according to theorem 4.1: $z_i = \delta_{mpr,i}$). If $z_i(t)$ is additionally smooth, convergence is faster (for example, in case of the method of Brent [8], convergence is super-linear).

The distance $\delta_{exact} > 0$ between two shapes A and B that are *not in contact* to each other is most naturally defined as the shortest Euclidean distance (see e.g. [21]):

$$\delta_{exact} = \min\{|\boldsymbol{r}_A - \boldsymbol{r}_B| : \ \boldsymbol{r}_A \in A, \ \boldsymbol{r}_B \in B\}.$$

The following theorem defines how δ_{mpr} is related to δ_{exact}.

THEOREM 4.2. ***(Closest distance).***
If $\delta_{mpr} > 0$, the following holds:

1. *If termination occurred via* (TC2) *and $\boldsymbol{r}_4$ (returned by function distanceToPortal(..)) is not located on one of the edges or vertices of the portal triangle, or termination occurred via* (TC1)*, then*
$$\delta_{mpr} = \delta_{exact}.$$
2. *If termination occurred via* (TC2) *and $\boldsymbol{r}_4$ is located on one of the edges or vertices of the portal triangle, then*
$$0 < -\boldsymbol{r}_4 \cdot \boldsymbol{e}_4 \leq \delta_{exact} \leq \delta_{mpr}.$$

PROOF. Assume 1. holds: The closest point to zero is $\boldsymbol{r}_1$ in case of ($TC1$) and $\boldsymbol{r}_4$ in case of ($TC2$) due to the proof of theorem 4.1. Since these vectors are in parallel to their search direction, their absolute value is the closest distance, so $\delta_{mpr} = \delta_{exact}$.
Assume 2. holds: No shape point is beyond the portal plane (in direction of the ray). Therefore, a lower bound for the closest distance is the projection of $\boldsymbol{r}_4$ on the plane normal: $-\boldsymbol{r}_4 \cdot \boldsymbol{e}_4 \leq \delta_{exact}$. On the other hand, the closest distance δ_{exact} cannot be larger than $|\boldsymbol{r}_4| = \delta_{mpr}$. □

Theorem 4.2 states that the MPR algorithm either returns the closest distance of two non-penetrating shapes or an upper bound.

Contrary to non-penetrating shapes, there are various useful definitions for the (exact) signed distance if shapes A and B are in contact to each other, in particular:

Definition 4.3. **(Penetration depth).** [5, p. 36]
The penetration depth of an intersecting pair A and B is the length of the shortest vector over which one of the shapes needs to be *translated* in order to bring the pair in touching contact:
$$\delta_{exact}(A, B) = \inf\{\|\boldsymbol{r}\| : \boldsymbol{r} \notin A \ominus B\}.$$

There are also definitions which take shape rotations into account. δ_{exact} is the simplest definition that is based on pure geometric properties at the actual time instant. The drawback of all pure geometric definitions is that a dynamic movement of shapes may lead to unphysical discontinuities in the contact points, see figure 6.

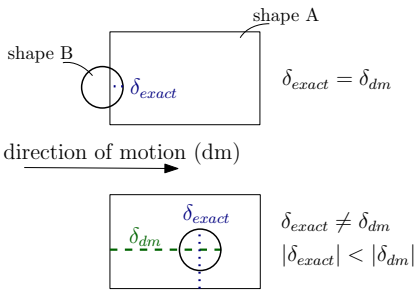

Figure 6: Discontinuities in penetration depth computation.

Here, shape B moves from left to right and penetrates box shape A. If the penetration δ_{dm} is large enough, the (translational) penetration depth δ_{exact} is physically wrong. There are proposals to fix this issue, by taking the movement of the contact points (so past geometric properties) into account, see for example [27]. In figure 7 this issue is analyzed in more detail:

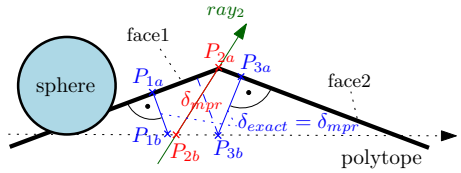

Figure 7: Penetration depth computation sphere - polytope.

A sphere is sliding on a polytope from left to right along the dotted arrow. If the lower point of the sphere arrives point P_{1b}, the penetration depth is the orthogonal distance to face1, so $|\delta_{exact}| = \overline{P_{1a}P_{1b}} = |\delta_{mpr}|$, which in turn is also the distance computed by the MPR algorithm. Once the lower point of the sphere arrives a point where the distance to face2 becomes smaller as the distance to face1 (here: point P_{3b}) the contact point on the polytope jumps *discontinuously* from face1 to face2 and $|\delta_{exact}| = \overline{P_{3a}P_{3b}} = |\delta_{mpr}|$. This means that the response calculation will be unphysical. The MPR algorithm introduces a slightly different discontinuity: Once the polytope and the sphere centroids are on a line (= ray2) that passes vertex P_{2a}, the MPR algorithm terminates with condition (TC1) and therefore $|\delta_{mpr}| = \overline{P_{2a}P_{2b}} > |\delta_{exact}|$. Therefore, the discontinuous change from face1 to face2 occurs earlier, and not only the contact point at the boundary of the polytope, but also δ_{mpr} changes discontinuously when the lower point of the sphere arrives at P_{2b}. This leads again to an unphysical response calculation.

This issue can be improved by *smoothening* the shapes, for example, by moving the middle point of a tiny sphere with radius μ over the shapes boundary and using the convex hull of this shape for the contact handling, see for example [5, pp. 166-167]. This smoothening can be implemented computationally by just adapting the support point computation:
$$\text{support}(\tilde{A}, \boldsymbol{e}) \;=\; \text{support}(A, \boldsymbol{e}) + \mu_A \boldsymbol{e},$$
where $\tilde{A}$ is shape A smoothened with a tiny sphere of radius μ_A. In such a case, two penetrating and continuously moving shapes A, B will lead to a *continuous* penetration depth $\delta_{exact} = \delta_{mpr}$ as long as $\delta_{exact} < \mu_A + \mu_B$.

5 ZERO-CROSSING FUNCTIONS

The distances $\delta_{mpr,i}$ shall be used as zero-crossing functions z_i (1b) for systems where many convex shapes may potentially collide. A brute force method would be to use the distances between any two shapes as zero-crossing functions. However, this approach is not practical for a larger number of shapes n_s, because $O(\dim(\boldsymbol{z})) = O(n_s^2)$: The number of zero-crossing functions, as well as the number of distance computations would grow quadratically with the number of shapes.

To avoid these drawbacks, a new approach is proposed: Basically, the user defines a fixed number n_z of zero-crossing functions, which shall be used by the integrator. This means that at most n_z shape pairs can be in contact at the same time instant. If more shapes get in contact, the simulation is halted with an error (alternatively, the simulation is halted and restarted with an enlarged $\boldsymbol{z}$ vector). Below, the following dimension information is used:

n_s Number of shapes.

n_z Number of zero-crossing functions (for example $n_z = 100$).
n_n Number of distance computations in the narrow phase (usually $n_n \ll O(n_s^2)$).

The following two functions shall be provided.[7] Hereby, `C` is a structure that holds information about the actual contact situation (e.g. all contact shapes), in particular $z = C.z$:

1. selectContactPairs!(`C`)
   This function performs (a) a broad phase to determine which shapes are potentially in contact to each other, (b) computes the distances of these shapes in a narrow phase and (c) *selects the n_z shape pairs with the smallest distances and orders them according to their distances* in $O(n_n log(n_z))$ operations.[8] This function is called before every integrator step.
2. getDistances!(`C`)
   This function performs (a) a broad phase to determine which shapes are potentially in contact to each other, (b) computes the distances of these shapes in a narrow phase and (c) *stores the distances of the contact pairs selected by the last call of function* selectContactPairs!(`C`) in z in $O(n_n log(n_z))$ operations.[9] This function is called whenever the integrator requests a new zero-crossing function evaluation.

With function selectContactPairs!(`C`) new contact pairs are selected before every new step of an integrator, and this selection is kept until the end of the step. This is uncritical, at the initial step after initialization and at the first step after an event occurred, say at time instant t_{ev}: n_z contact pairs are selected. For this selection the zero-crossing functions z are computed at t_{ev} and one or more times during the first integrator step of length h.

At the second and further steps after initialization or event-restart the z_i may characterize different contact pairs. However, since no zero-crossing occurred in the previous step, the number of negative and the number of positive z_i are not changing. As explained in section 4.3, the time instant of the zero-crossing functions is determined with algorithms that need only the sign of the zero-crossing functions. Therefore, even if the meaning of the zero-crossing functions might be changed before the beginning of a new step, this does not matter, because the zero-crossing functions of the new selection have the same signs as the zero-crossing functions evaluated with the selection from the previous step. This is demonstrated at hand of the small example of table 1 (column z contains the values of the zero-crossing functions at the respective time instant; column ID contains integer values that characterizes uniquely shape pairs that are potentially in contact to each other). Column "ID's from selection t_{i-1}" contains shape pair ID's and associated z-values at the end of the previous step. The ID's are identical to the ID's in column t_{i-1}, but the z-values are (potentially) different because the shapes are moving. However, the number of negative and positive z-values are *identical* at both time instants, because no zero-crossings took place in this step. Column "ID's from selection t_i" contains shape pair ID's and associated z-values at the beginning of the next step (so the z-values are sorted according to their distances). Here, no new distance computations are performed, since during every step two data structures are utilized to hold (a) the n_z shape pairs ordered according to their actual distance (= $dict1$ below) and (b) the n_z shape pairs selected at the beginning of the step (= $dict2$ below). Therefore, at beginning of step t_i, just the data structure of (b) is newly constructed from (a), so (potentially) different shape pairs are utilized for z, but sign(z_i) is not changing.

t_{i-1}		t_i			
		ID's from selection t_{i-1}		ID's from selection t_i	
z	ID	z	ID	z	ID
-0.09	10	-0.08	10	-0.08	10
-0.003	5	-0.015	5	-0.002	3
-0.001	3	-0.002	3	-0.015	5
0.02	7	0.04	7	0.04	7
0.3	8	0.55	8	0.4	12
0.4	2	0.53	2	0.5	4

Table 1: Zero-crossing functions z and associated shape pairs (ID's) at two time instants.

The details of the two functions from above are presented below. Furthermore, utility functions operating on *balanced binary trees* are utilized. The following notation is used (based on Julia package DataStructures[10]). Hereby, $n \le n_z$ is the current number of items in the container:

1. $dict$ is an instance of a balanced binary tree data structure with (key,value) pairs ordered according to the keys.
2. insert!($dict, key, value$) inserts the key and the $value$ into $dict$ in $O(\log(n))$ operations. If the key is already present, this overwrites the old value.
3. delete!($dict, key$) deletes the item whose key is key in $O(\log(n))$ operations.
4. $(key, value)$ = last($dict$) returns the item of $dict$ with the largest key in $O(log(n))$ operations.

The balanced binary trees $dict1, dict2$ are stored in structure `C`:

$dict1$ Uses the n_z smallest distances as keys. The values are unique IDs, that identify the collision pairs. $dict1$ is updated every time when computeDistances(`C`) is called.
$dict2$ Uses the n_z IDs as keys that have the smallest distances. The values are the indices of the collision pairs in the z vector. $dict2$ is newly constructed every time when selectContactPairs!(`C`) is called.

The flag `distancesComputed` (stored in structure `C`) is `true` after function computeDistances(`C`) was called, and is set to `false`, after shapes have changed their positions.

```
function selectContactPairs!(C)
  if not distancesComputed
    computeDistances!(C, false)
    distancesComputed = true
  end
  for i=1:n_z
```

[7] As usual in Julia, function names with a ! at the end indicate that one or more of the input arguments are changed by the function call.
[8] If the distances of more than n_z shape pairs are negative, an error is triggered; if the number of shape pairs that are potentially in contact is less than n_z, a positive dummy value is provided for the corresponding z_i.
[9] An error is triggered, if one of the contact pairs has a negative distance and is not selected by selectContactPairs!(`C`).
[10] http://datastructuresjl.readthedocs.io/en/latest/sorted_containers.html

```
    z[i] = keys(dict1)[i]
    dict2[values(dict1)[i]] = i
end; end

function getDistances!(C)
  if not distancesComputed
    computeDistances!(C, true)
    distancesComputed = true
end; end

function computeDistances!(C, getDist)
  for i=1:length(C.contactShapes)-1
    for j=i+1:length(C.contactShapes)
      A = C.contactShapes[i], B = C.contactShapes[j]
      if potentialCollision(A,B)  # broad phase
        δ = mpr(A,B)              # narrow phase
        ID = uniqueID(i,j)
        if length(dict1) < n_z
          insert!(dict1,δ,ID)
        else
          (k,v) = last(dict1)
          if δ < k && k ≤ 0.0
            error("n_z must be enlarged.")
          elseif δ < k && k ≥ 0.0
            delete!(dict1,k)
            insert!(dict1,δ,ID)
        end; end

        if getDist  # called by getDistances!()
          # if ID of shape pair is in dict2,
          # distance is stored in z
          (isInside, zindex) = findID(dict2, ID)
          if isInside
            C.z[zindex] = δ
          elseif δ < 0.0
              error("n_z must be enlarged.")
end; end; end; end; end; end
```

6 EXAMPLES

The algorithms proposed in the previous sections have been implemented and evaluated in a prototype using the Julia programming language [6].

The prototype consists of the following parts:

1. A higher level Julia interface to the DLR visualization program SimVis [4, 14] to define and visualize simple shapes such as spheres, boxes, cylinders, ..., as well as complex shapes defined in various file formats.
2. An implementation of the improved MPR algorithm of section 4 with support point computations for some of the simple shapes of SimVis, as well as for polyhedrons defined with Wavefront (*.obj) files. In the latter case distance computations with the MPR algorithm are either performed for the *convex hull* of a polyhedron, or a concave polyhedron is approximately transformed to a *set of convex polyhedrons* with program V-HACD[11] and distance computation is performed for these convex polyhedrons. Additionally, zero-crossing functions for the integrator are provided.
3. A small 3D multi-body program for tree-structured systems with a few joint types has been implemented in Julia that uses (1) and (2). Elastic collision forces are implemented along the approach described in [23].
4. The multi-body program is utilized as model in the simulation engine under development for Modia [10, 11] that is based on the Sundials IDA integrator [15, 16]. Simulation results are visualized (a) with one of the plot packages available for Julia, especially PyPlot, and (b) with SimVis for 3D animations.

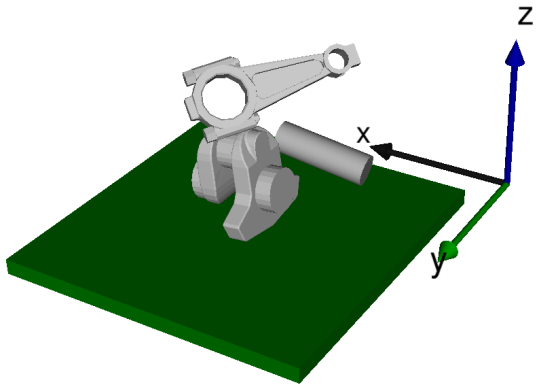

Figure 8: Parts (rod, crank shaft, cylinder) falling on ground.

In figure 8, a typical animation of one of the examples is shown. Here three parts, a piston rod, a part of a crank shaft (provided as Wavefront-files) and a cylinder are falling on the ground, described by the green box. The four parts collide with each other. Distance computations and response calculations with the penetration depths computed by the MPR algorithm are performed as expected (for the rod and the crank shaft, distance computation is performed for the convex hulls).

In figure 9, five objects are colliding: three convex shapes (cylinder, box and sphere) and two concave shapes. A concave shape is approximated by a set of convex shapes (that are visualized with different colors). Since the convex parts of one concave shape are rigidly attached together, no collisions are possible between them. This results in 79 shapes that can potentially collide with each other, and in 651 potential shape pairs where the MPR algorithm might be applied.

[11] V-HACD: https://github.com/kmammou/v-hacd

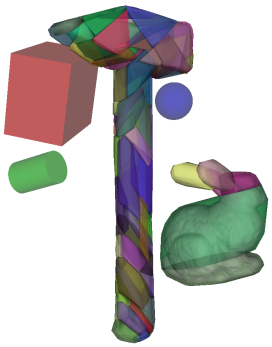

Figure 9: Colliding convex and concave shapes.

7 CONCLUSION

In this paper improvements to the MPR algorithm are proposed to compute the signed distance (= closest distance if not in contact and penetration depth if in contact) between two convex shapes. Especially, special collision situations are treated properly and newly introduced termination conditions speed up the algorithm. It is shown that the computed signed distances can be used as zero-crossing functions for variable-step integrators which are typically used in object-oriented modeling languages in offline simulations. Furthermore, a new method is proposed to use a small number of zero-crossing functions, even if many shapes are present that can potentially collide. This method uses standard broad and narrow phases for the distance computation, but utilizes only the most relevant distances for the zero-crossing functions. The developments have been evaluated with a prototype implemented in Julia.

Currently, only a very simple broad phase has been implemented. It is planned to make this implementation more efficient by using octrees, along the approach proposed in [9]. Additionally, the prototype will be made more complete to handle all shape types supported by SimVis and by including it into Modia [10, 11] after more involved tests have been performed.

ACKNOWLEDGEMENTS

We would like to thank our colleagues Tobias Bellmann and Matthias Hellerer for their support and input.

REFERENCES

[1] Modelica Association. 2017. *Modelica, A Unified Object-Oriented Language for Systems Modeling. Language Specification, Version 3.4.* Technical Report. Modelica Association. https://www.modelica.org/documents/ModelicaSpec34.pdf

[2] Q. Avril, V. Gouranton, and B. Arnaldi. 2009. *New trends in collision detection performance.* Technical Report. INRIA. https://hal.archives-ouvertes.fr/hal-00412870

[3] G. Bardaro, L. Bascetta, F. Casella, and M. Matteucci. 2017. Using Modelica for advanced Multi-Body modelling in 3D graphical robotic simulators. In *Proc. of the 12th International Modelica Conference*, J. Kofranek and F. Casella (Eds.). LiU Electronic Press. http://www.ep.liu.se/ecp/132/097/ecp17132887.pdf

[4] T. Bellmann. 2009. Interactive Simulations and advanced Visualization with Modelica. In *Proc. of the 7th International Modelica Conference*, Francesco Casella (Ed.). LiU Electronic Press. http://www.ep.liu.se/ecp/043/062/ecp09430056.pdf

[5] G.v.d. Bergen. 2003. *Collision Detection in Interactive 3D Environments.* Morgan Kaufmann Publishers.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98.

[7] K. E. Brenan, S. L. Campbell, and L. R. Petzold. 1996. *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations.* SIAM.

[8] R.P. Brent. 1973. *Algorithms for Minimization without derivatives.* Prentice Hall. http://maths-people.anu.edu.au/~brent/pub/pub011.html

[9] H. Elmqvist, A. Goteman, V. Roxling, and T. Ghandriz. 2015. Generic Modelica Framework for MultiBody Contacts and Discrete Element Method. In *Proc. of the 11th International Modelica Conference*, Peter Fritzson and Hilding Elmqvist (Eds.). LiU Electronic Press. http://www.ep.liu.se/ecp/118/046/ecp15118427.pdf

[10] H. Elmqvist, T. Henningsson, and M. Otter. 2016. Systems Modeling and Programming in a Unified Environment based on Julia. In *Proc. of ISoLA Conference.* Springer. https://doi.org/10.1007/978-3-319-47169-3_15

[11] H. Elmqvist, T. Henningsson, and M. Otter. 2017. Innovations for Future Modelica. In *Proc. of the 12th International Modelica Conference*, J. Kofranek and F. Casella (Eds.). LiU Electronic Press. http://www.ep.liu.se/ecp/132/076/ecp17132693.pdf

[12] X. Floros, F. Bergero, F. E. Cellier, and E. Kofman. 2011. Automated Simulation of Modelica Models with QSS Methods: The Discontinuous Case. In *Proc. of the 8th International Modelica Conference.* LiU Electronic Press. http://www.ep.liu.se/ecp/063/073/ecp11063073.pdf

[13] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. 1988. A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space. *IEEE Journal of Robotics and Automation* 4, 2 (1988), 193–203. https://graphics.stanford.edu/courses/cs448b-00-winter/papers/gilbert.pdf

[14] M. Hellerer, T. Bellmann, and F. Schlegel. 2014. The DLR Visualization Library - Recent development and applications. In *Proc. of the 10th International Modelica Conference*, Hubertus Tummescheit and Karl-Erik Arzen (Eds.). LiU Electronic Press. http://www.ep.liu.se/ecp/096/094/ecp14096094.pdf

[15] A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. 2005. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Trans. Math. Software* 31, 3 (Sept. 2005), 363–396.

[16] A.C. Hindmarsh, R. Serban, and A. Collier. 2015. *User Documentation for IDA v2.8.2.* Technical Report UCRL-SM-208112. Lawrence Livermore National Laboratory.

[17] A. Hofmann, L. Mikelsons, I. Gubsch, and C. Schubert. 2014. Simulating Collisions within the Modelica MultiBody Library. In *Proc. of the 10th International Modelica Conference*, Hubertus Tummescheit and Karl-Erik Arzen (Eds.). LiU Electronic Press. http://www.ep.liu.se/ecp/096/099/ecp14096099.pdf

[18] B. Kenwright. 2015. *Generic Convex Collision Detection using Support Mapping.* Technical Report. http://www.xbdev.net/misc_demos/demos/minkowski_difference_collisions/paper.pdf

[19] David Mainzer. 2015. *New Geometric Algorithms and Data Structures for Collision Detection of Dynamically Deforming Objects.* Ph.D. Dissertation. Clausthal University of Technology. http://cgvr.cs.uni-bremen.de/papers/diss_weller2012/DissertationWeller.pdf

[20] L. Olvang. 2010. *Real-time Collision Detection with Implicit Objects.* Technical Report IT 10 009. Department of Information Technology, Uppsala Universitet, Sweden. http://uu.diva-portal.org/smash/get/diva2:343820/FULLTEXT01.pdf

[21] C.J. Ong and E.G. Gilbert. 1996. Growth Distances: New Measures for Object Separation and Penetration. *IEEE Transactions on Robotics and Automation* 12, 6 (1996), 888–903.

[22] M. Otter and H. Elmqvist. 2017. Transformation of Differential Algebraic Array Equations to Index One Form. In *Proc. of the 12th International Modelica Conference*, J. Kofranek and F. Casella (Eds.). http://www.ep.liu.se/ecp/132/064/ecp17132565.pdf

[23] M. Otter, H. Elmqvist, and J. Diaz Lopez. 2005. Collision Handling for the Modelica MultiBody Library. In *Proc. of the 4th International Modelica Conference*, Gerhard Schmitz (Ed.). https://modelica.org/events/Conference2005/online_proceedings/Session1/Session1a4.pdf

[24] Friedrich Pfeiffer. 2012. On non-smooth multibody dynamics. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics* 226, 2 (2012), 147–177.

[25] H. Pungotra. 2010. *Collision Detection and Merging of Deformable BSpline Surfaces in Virtual Reality Environment.* Ph.D. Dissertation. The University of Western Ontario. http://ir.lib.uwo.ca/cgi/viewcontent.cgi?article=1086&context=etd

[26] G. Snethen. 2008. Xenocollide: Complex collision made simple. In *Game Programming Gems 7*, Scott Jacobs (Ed.). Charles River Media, 165–178.

[27] X. Zhang, Y.J. Kim, and D. Manocha. 2014. Continuous Penetration Depth. *Computer-Aided Design* 46 (2014), 3–13.