Algorithms for Component-Based 3D Modeling

Andrea Neumayr[1] Martin Otter[1]

[1]DLR, Institute of System Dynamics and Control, Germany, {andrea.neumayr,martin.otter}@dlr.de

Abstract

The experimental modeling environment Modia3D is used to test and evaluate ideas to model and simulate larger and more complex 3-dimensional systems than is possible with a pure equation-based modeling system such as current Modelica. The goal is to closely combine equation-based modeling with component-based 3D modeling as used in modern game engines. In this article some key algorithms are discussed that have been developed for Modia3D. The overall objective is to utilize the results for the design of the next Modelica language generation.
Keywords: Modelica, Modia, Modia3D, Julia, DAE, equation-based modeling, component-based modeling, multi-body, collision handling

1 Introduction

The Modelica standard library[1] supports the modeling of 3-dimensional multi-body systems with its sub-library Modelica.Mechanics.MultiBody (Otter et al., 2003). There have been several attempts to improve this library with regards to visualization, collision handling or support of larger models, for example (Otter et al., 2005; Höger et al., 2012; Hofmann et al., 2014; Elmqvist et al., 2015; Bardaro et al., 2017). Over the years it was recognized that this is hard because the technology of current Modelica has some natural limitations:

- No modern data structures, like dictionaries or trees, or objects with member functions are supported in Modelica, but they are standard in high level programming languages and are needed to model for example 3D meshes or collision detection algorithms. Then, the only choice is to interface external programs with Modelica models: Developing such algorithms from scratch in, say, C++, and then interface to Modelica is too much effort.
  Using existing code is hard either, because only partial, incompatible solutions are available. For example, it would be nice to interface the Bullet Physics SDK[2] to Modelica to get a state-of-the-art collision handling package. However, this engine determines only the penetration depth of colliding bodies, but for variable-step solvers in offline simulation also zero-crossing functions for DAE-solvers are needed that require the Euclidean distance between non-colliding shapes as well (Neumayr and Otter, 2017). Visualization, collision handling, mass properties calculations require geometric information. Integrating such different description forms in Modelica is hard due to the missing modern data structures. Whenever such packages are integrated, shapes need a unique identification, but this feature is hard to provide in Modelica.
- Modelica tools typically support only generic symbolic transformation algorithms. It is hard or impossible to utilize algorithms which are specialized for a particular model class, for example to remove redundant equations of nonlinear-equation systems due to kinematic loops, to compute a common mass and center of mass of rigidly connected bodies and use it in the simulation, or to use an O(n) multi-body algorithm. In Modelica, a user would have to use a pre-processor that generates Modelica code, see e.g. (Elmqvist et al., 2009).
- Since Modelica compilers typically expand the models for the symbolic engine, the same equation is analyzed many times. For example if a mechanical system has 100 bodies, then the equations of a body are present 100 times in the generated code. C or C++ compilers are not designed to handle huge code parts in a good way. Therefore, there are natural limitations on the model size. For fluid models there are some solutions available, where code for a component is generated only once and reused many times, e.g. (Sahlin and Grozman, 2003). For multi-body systems such solutions might be possible, but yet need to be developed.

The article *Component-Based 3D Modeling of Dynamic Systems* (Neumayr and Otter, 2018) starts an approach to cope with the underlying inherent issues. The basic idea is to combine 3D modeling techniques closely with equation-based modeling à la Modelica within one high level programming environment. Modia[3] (Elmqvist et al., 2016, 2017) is used for the equation-based modeling. It is implemented with the Julia programming language[4] (Bezanson et al., 2017). Julia allows to program numerical algorithms conveniently on a high level. It supports modern data structures, multiple dispatch, metaprogramming, has a just-in-time-compiler and has excellent performance benchmarks relative to C.

[1]https://github.com/modelica/ModelicaStandardLibrary
[2]https://github.com/bulletphysics/bullet3
[3]https://github.com/ModiaSim/Modia.jl
[4]https://julialang.org

Modia utilizes Julias metaprogramming features to integrate an equation-based modeling language with a programming language (e.g. a Modia model can be stored in a dictionary that in turn is inquired in another Modia model to select and use a submodel from this dictionary). Modia3D[5] is designed to model 3D systems, initially only mechanical systems, but it shall be expanded into other domains in the future. It is implemented in Julia and utilizes ideas of multi-body programs and game engines. In the near future, Modia and Modia3D shall be closely integrated, e.g. using a Modia3D model in Modia or using Modia models in Modia3D. Up to now, Modia3D is implemented for functionality and not tuned for efficiency. Therefore, there are no benchmarks yet and in particular no comparison with Modelica models. For animation the free community edition as well as the professional edition[6] of the *DLR Visualization* library[7] (Bellmann, 2009; Hellerer et al., 2014) are used. The overall goal is to apply the results of the Modia/Modia3D prototyping into the design of the next Modelica language generation.

The user's view of Modia3D was introduced in (Neumayr and Otter, 2018) to show the very flexible definition of 3D systems. In this article, several key algorithms are discussed which have been developed for the Modia3D prototype.

2 Component-Based 3D Modeling

Modia3D has two design patterns: the *component-based* and the *hierarchical structure*. The ideas for component-based structuring are from modern game engines, such as Unity or Unreal Engine, which have a component-based design. In the context of game engines a coordinate system is located in 3D and has a container with optional components (in Unity such an object is called *GameObject*[8], in Unreal Engine it is named *Actor*[9], and in Three.js it is called *Object3D*[10]). Each of these components has optional properties such as geometry, visualization, dynamics, collision properties, light, camera, sound, etc., see for example (Nystrom, 2014)[11]. This is a very flexible way to define many optional components and variants and treat them in a modular way. In this paper, this very special design of the generic *component-based design pattern* is called *component-based 3D modeling*. The Julia programming language is particularly suited for this programming pattern. In Section 2 a brief overview to component-based 3D modeling and the features used in this paper is given.

Hierarchical structuring for grouping, aggregating and defining 3D objects is performed with the Modia3D macro `@assembly`. A Julia macro is a metaprogramming[12] language element and starts with `@`. It generates an abstract syntax tree (AST) of Julia code which is automatically compiled and executed at the line where the macro is called. For further information, see (Neumayr and Otter, 2018).

Object3D

In Modia3D, component-based 3D modeling is performed with so-called *Object3D* objects. An Object3D consists of a 3D coordinate system that has optional associated properties collected in the `data` container. Furthermore, an Object3D stores connections to other Object3Ds, via joint, force, or sensor elements (see Figures 2, 1). The code-snippet[13] of the following Julia constructor call[14] creates a new Object3D object `obj`:

```
1  obj = Object3D(parent,data,r=[0,0,0],
2                 R=eye(3),fixed=true)
```

Each `obj` can be defined relative to a `parent` Object3D, with the position vector `r` and the rotation matrix `R`. It is *rigidly connected* to its `parent` if `fixed=true`, and it can move freely if `fixed=false`. The initial position and rotation matrix is defined with `r`, `R`. An Object3D is said to be a reference Object3D, if no `parent` Object3D is given. The 14 Object3Ds of Figure 2 demonstrate different properties and are used below to explain a core algorithm.

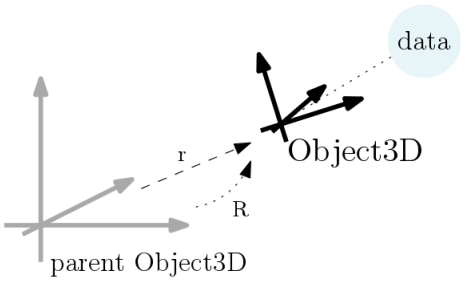

Figure 1. Object3D defined relatively to its parent.

Joint Object

Two Object3Ds can be connected via a joint. In Figure 2 there are several joints, one joint is e.g. between `obj2` and `obj4`.

```
3  joint1 = Revolute(obj2,obj4;axis=3)
```

A reference to the revolute joint is stored in `obj4`. In case the joint introduces a kinematic loop, it is replaced internally by a cut-joint (in Figure 2 this happens e.g. with the joint between `obj5` and `obj13`). A cut-joint is referenced by the two Object3Ds that are constrained by it.

[5] https://github.com/ModiaSim/Modia3D.jl
[6] https://visualization.ltx.de/
[7] http://www.systemcontrolinnovationlab.de/the-dlr-visualization-library/
[8] https://docs.unity3d.com/Manual/GameObjects.html
[9] https://docs.unrealengine.com/en-us/Engine/Components
[10] https://threejs.org/docs/index.html#api/core/Object3D
[11] http://gameprogrammingpatterns.com/component.html
[12] https://docs.julialang.org/en/stable/manual/metaprogramming/
[13] For better reference every code-snippet is marked with a unique line number on the left-hand side.
[14] When calling a Julia function, all optional keyword arguments (name-value pairs) can be given in any order. They are set after the positional arguments (here: `parent` and `data`).

DOI 10.3384/ecp19157383

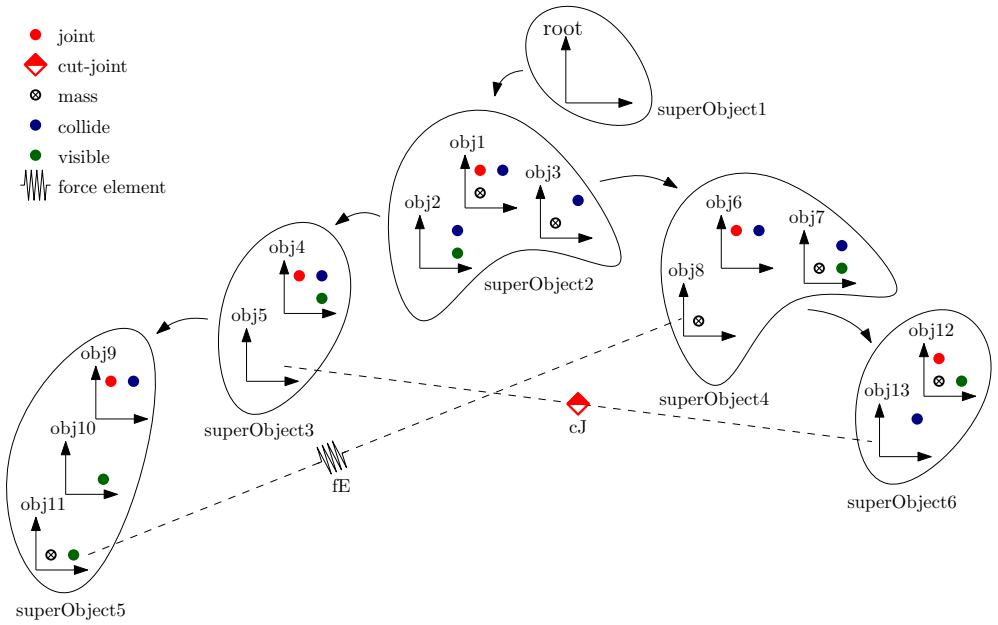

Figure 2. 14 Object3Ds with different properties like they are allowed to collide, can have a mass, are visible and/or can have a force element, are grouped into six rigidly attached general super-objects disjunct via joints and cut-joints.

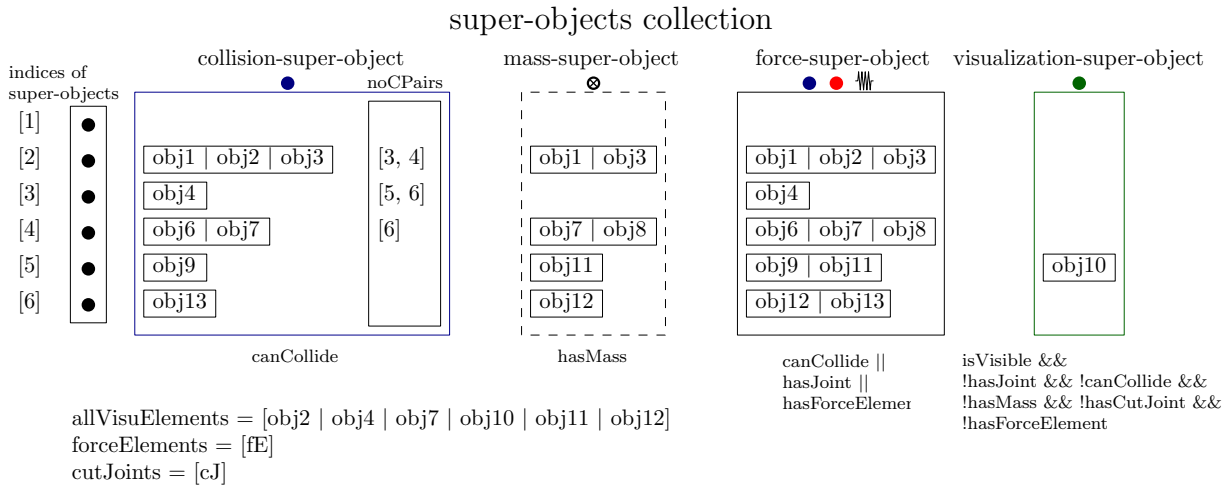

Figure 3. A super-object collection holds four different super-objects types for collision, mass, force computation, and visualization.

Force Object

Two Object3Ds can be connected via a force element. In Figure 2 there is a force element between `obj11` and `obj8`.

```
4  spring = Spring(obj11,obj8;c = 1e3)
```

A force element is referenced by the two Object3Ds on which it is acting.

Geometry Object

A geometry, such as a sphere, box, cylinder, or a mesh can be defined and associated with an Object3D. For example, a sphere is associated with `obj4` in Figure 2.

```
5  sphere1 = Object3D(obj4,Sphere(0.9),
6                     r=[0,0,0.8])
```

Visualization Object

Visualization objects are an interface to the DLR Visualization library (Bellmann, 2009; Hellerer et al., 2014). For defining the visualization properties a `Material` object has to be associated to a geometry object, or special visualization objects can be used (for example a `CoordinateSystem`). The following constructor call generates a new `Material` object and associates it to a sphere geometry.

```
7  vmat = Material(color=[0,0,255],
8           wireframe=false,transparency=0.5,
9           shininess=0.7,reflectslights=true)
10 sphere2 = Object3D(obj12,
11              Sphere(0.9,material=vmat),
12              r=[0,0,0.8])
```

A geometry object is only visualized if a visualization material is defined for the object.

MassProperties Object

A `MassProperties` object can be associated with an Object3D to define mass, center-of-mass and inertia tensor with respect to this Object3D. There are various options to define mass properties, for example defining them explicitly (with or without a geometry) or computing them from the volume of a geometry object and of a given density.

Collision Object

The geometry of the associated Object3D takes place in collision handling if a contact response characteristics is defined via an `AbstractContactMaterial` object. For example, an elastic response characteristic with a linear spring and damper is defined with `ContactMaterialElastic()`.

3 Collision Handling

Collision detection in Modia3D is based on the Minkowski Portal Refinement algorithm (MPR-algorithm) (Snethen, 2008), which computes the shortest penetration depth of two convex shapes/convex hulls. The MPR-algorithm is much simpler to implement and has less numerical problems than the often used GJK/EPA-standard algorithms (Gilbert et al., 1988; Bergen, 2003), because it only works with triangles and not with tetrahedrons.

A Modia3D model is mathematically defined as a Differential-Algebraic-Equation system (DAE) with $\boldsymbol{x} = \boldsymbol{x}(t)$ and a regular Jacobian $\boldsymbol{J}$ (1c):

$$\begin{aligned} \mathbf{0} &= \begin{bmatrix} \boldsymbol{f}_d(\dot{\boldsymbol{x}}, \boldsymbol{x}, t, z_i > 0) \\ \boldsymbol{f}_c(\boldsymbol{x}, t, z_i > 0) \end{bmatrix} \quad (a) \\ \boldsymbol{z} &= \boldsymbol{f}_z(\boldsymbol{x}, t) \qquad\qquad\quad (b) \end{aligned} \qquad \boldsymbol{J} = \begin{bmatrix} \dfrac{\partial \boldsymbol{f}_d}{\partial \dot{\boldsymbol{x}}} \\ \dfrac{\partial \boldsymbol{f}_c}{\partial \boldsymbol{x}} \end{bmatrix} \quad (c) \qquad (1)$$

Therefore, (1a) is an index 1 DAE and (1b) defines zero-crossing functions $\boldsymbol{z}(t)$. To speed up the simulation and to improve the robustness of the integration, Modia3D uses the distances between convex shapes as zero-crossing functions $z_i(t)$ (1b).

In the original version of the MPR-algorithm (Snethen, 2008) only penetration depths are determined. In Modia3D improvements of the MPR-algorithm are utilized that have been proposed in (Kenwright, 2015; Neumayr and Otter, 2017), in particular to compute the distances of shapes that are not in contact and treating special collision situations properly.

In Modia3D collision handling of n potentially colliding shapes is performed in the following (mostly standard) way:

1. *Broad Phase*
   The shapes are approximated by bounding volumes where potential collisions can be very cheaply determined resulting in $O(n^2)$ cheap tests. When using special data structures (such as octrees or kd-trees), it is possible to reduce the number of cheap tests to $O(n\log(n))$.

2. *Narrow Phase*
   For the potentially colliding shape pairs as identified in the broad phase, the signed distances are computed with the improved MPR-algorithm (Neumayr and Otter, 2017).

3. *Response Calculation*
   If two shapes are penetrated, a force and/or torque is applied at the contact point, such as a spring - damper force element, depending on the penetration depth.

The broad phase in Modia3D uses *AABB*s (= Axis Aligned Bounding Boxes) (see e.g. (Bergen, 2003)). Each AABB approximates one shape and only if the AABBs are intersecting, the distance between these two possibly colliding shape pairs is calculated in the narrow phase. A preprocessing of the tree-structure is executed to reduce the number of possible collision pairs to n_{pp} before the broad phase is processed. This leads to $n_{pp} \leq O(n^2)$ tests. There are two preprocessing rules:

1. Rigidly attached shapes cannot collide with each other.

2. Shapes connected by a joint cannot collide with each other if the joint specific option `canCollide` is set to `false` (the default setting).

4 Object Preprocessing

In this section a central preprocessing step of Modia3D is explained. The goal is to evaluate efficiently many objects of different kinds during integration.

For example, the position and orientation of a visualization object should only be computed when needed (at communication points), and not in every model evaluation. Furthermore, if mass properties are associated with rigidly connected Object3Ds (two or more), then the resultant mass properties of all these objects is computed once in the preprocessing step (note, such an operation is hard to automatically perform with a Modelica multi-body model).

Figure 2 presents an example of a Modia3D model, and the connected objects are given. The goal is to generate the data structure that is shown in Figure 3. Afterwards, the usage of this data structure for an efficient evaluation of the model during integration is explained.

4.1 Super-Objects

Rigidly connected Object3Ds are grouped together into so-called *super-objects*. Super-objects are disjunct via joints. Without any further assumptions, the grouping of the 14 Object3Ds of Figure 2 leads to six general super-objects (Figure 4). Figure 5 also shows these six super-objects connected via joints/cut-joints. The super-objects $2, 3, 4$ and 6 are forming a kinematic loop. This kinematic loop is detected and the joint between super-object 3 and 6 is

DOI 10.3384/ecp19157383

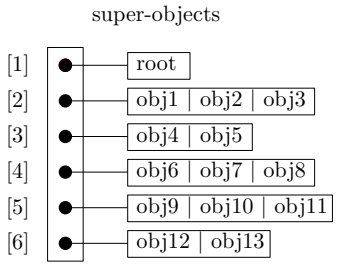

Figure 4. Six general super-objects.

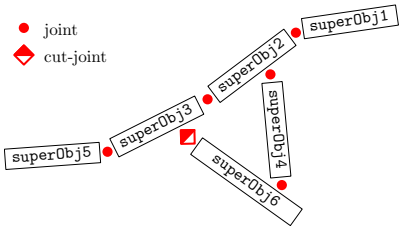

Figure 5. Six super-objects are connected via joints/cut-joints.

internally replaced by a cut-joint. Object3Ds can have several properties that are collected in different super-object data structures.

4.1.1 Super-Objects Collection

In the *super-objects collection* all information about different super-object types is stored, for example super-objects for collision handling, mass and force computation as well as visualization. The super-objects collection of the above mentioned example (Figure 2) with its different super-objects is shown in Figure 3. For each of the super-object types, there is a function `assignObj(obj,superObjType)` to store a reference of the object in the corresponding data structure identified by `superObjType`. A super-object collection has 3 additional containers: `allVisuElements` stores every Object3D which has visualization properties, all force elements and all cut-joints are stored respectively in `forceElements` and `cutJoints` (see Figure 3).

4.1.2 Super-Objects for Collision Handling

A geometry associated with an Object3D takes place in collision handling, in case a contact material is defined (see Section 2) and hence it gets assigned to a *collision-super-object* (lines 13 - 17).

```
13 function assignObj(obj::Object3D,
14              superObjType::SuperObjCollision)
15   if canCollide(obj)
16     push!(superObjType.superObj, obj)
17 end; end
```

All Object3Ds within one collision-super-object are rigidly connected, so they cannot collide with respect to each other and therefore they already fulfill the first preprocessing rule (see Section 3). To fulfill also the second preprocessing rule, all collision-super-objects which are disjunct by a joint/cut-joint are not allowed to collide either (if option `canCollide = false`). Therefore, the indices of collision-super-objects which are not allowed to collide, are stored in a collision-list called *no collision pairs* (= `noCPairs`). For example `superObject2` is not allowed to collide with `superObject3` and vice versa (see Figures 2, 3, 5). It is sufficient to store this relation only once for the first executed super-object. This leads to the corresponding `noCPairs` container for collision-super-objects (see Figure 3).

4.1.3 Super-Objects for Mass Computation

In case mass properties are defined for an Object3D it gets assigned to a *mass-super-object* (lines 18 - 22). In a later step, a resultant mass, center-of-mass and inertia tensor is computed for all mass properties of one mass-super-object.

```
18 function assignObj(obj::Object3D,
19              superObjType::SuperObjMass)
20   if hasMass(obj)
21     push!(superObjType.superObj, obj)
22 end; end
```

4.1.4 Super-Objects for Force Computation

All Object3Ds which are allowed to collide, or have a joint, or a force element are stored in a *force-super-object* (lines 23 - 28). For these Object3Ds kinematic laws (positions, translation matrices, etc.) and especially forces need to be re-calculated in each solver step.

```
23 function assignObj(obj::Object3D,
24              superObjType::SuperObjForce)
25   if (canCollide(obj) || hasJoint(obj) ||
26       hasForceElement(obj))
27     push!(superObjType.superObj, obj)
28 end; end
```

4.1.5 Super-Objects for Visualization

Object3Ds within a *visualization-super-object* are exclusively for visualization (lines 29 - 36). The positions and rotation matrices only need to be calculated at communication points with the visualization engine.

```
29 function assignObj(obj::Object3D,
30              superObjType::SuperObjVisu)
31   if (isVisible(obj) && !hasJoint(obj) &&
32       !hasMass(obj) && !canCollide(obj) &&
33       !hasForceElement(obj) &&
34       !hasCutJoint(obj))
35     push!(superObjType.superObj, obj)
36 end; end
```

4.2 Algorithm for Constructing Super-Objects

The goal of this section is to introduce an algorithm to detect and group rigidly attached super-objects. This algorithm is based on a *depth-first search algorithm* (DFS) (Tarjan, 1972; Hopcroft and Tarjan, 1974). The depth-first search as well as the augmented version takes $O(n)$ time.

4.2.1 Depth-First Search Algorithm

The depth-first search algorithm explores each branch as far as possible to its leaves-level, afterwards it is stepping back (see Figure 6). This procedure uses a stack and is

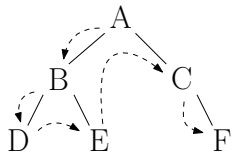

Figure 6. Example of a DFS.

executed until the stack is empty and hence every node has been visited exactly once. Below a Julia pseudo code is shown (lines 37 - 45). The DFS-algorithm works with a Last In First Out - stack (LIFO). Therefore, `append!` (line 44) inserts all children of an `obj` at the end of the stack and `pop!` (line 42) returns the last item. All nodes are stored in the above described order and a depth-first search of the example in Figure 6 would lead to `result = [A,B,D,E,C,F]`.

```
37 stack  = []
38 result = []
39 function DFS(root)
40    push!(stack,root)
41    while length(stack) > 0
42       obj = pop!(stack)
43       push!(result,obj)
44       append!(stack,obj.children)
45 end; end
```

4.2.2 Augmented Depth-First Search Algorithm

The augmented depth-first search is based on the idea of the depth-first search algorithm (Section 4.2.1). Below, a Julia pseudo code of the augmented depth-first search is presented (lines 54 - 79). It creates a super-object collection which holds all described super-object types. The assignment of each Object3D takes place in function `assignAll(...)` (lines 66, 71).

Every Object3D, except one, has a parent object and all Object3Ds and their associated properties are connected together and build up a tree, see Figure 2. The Object3D without a parent is treated as the world object and it is the root of the tree. The world object is not allowed to have any additional properties, except for visualization. In case any other Object3D has no parent an error occurs. Since the Object3Ds form a tree, the root of every super-object is an Object3D that is connected via a joint or it can move freely with respect to its parent. This parent is located on a different super-object.

The augmented DFS-algorithm works with a stack-like buffer and a stack.

buffer Whenever the root of a new super-object is found, it is pushed on the buffer. The index of an element in the buffer is also the index of the super-object. Therefore, the maximally reached length of the buffer is equal to the total amount of general super-objects. Additionally, variable `actPos` holds the index of the super-object that is currently processed. When the processing of a super-object is finished, this variable is incremented by one as long as there are elements on the buffer.

stack Starting from the root of a super-object, all Object3Ds for this super-object are inspected with the help of this stack. Whenever a boundary (an Object3D with a joint or a freely moving Object3D) is reached, this Object3D is pushed on the buffer (and not on the stack). All Object3Ds of the super-object have been inspected in depth-first order (see Figure 4), if the stack is empty.

The properties of a super-object are stored in the following structure:

```
46 mutable struct SuperObjs
47  superObjCollision::SuperObjCollision
48  superObjMass::SuperObjMass
49  superObjForce::SuperObjForce
50  superObjVisu::SuperObjVisu
51  noCPair::Array{Int64,1}
52  ...
53 end
```

Hereby, every essential property of an Object3D is an element of this struct (such as `superObjMass` which holds all Object3Ds that have a mass) of the corresponding super-objects.

The top-level part of the algorithm:

```
54 stack  = []
55 buffer = []
56 coll   = SuperObjCollection()
57 augmentDFS!(root_obj)
```

initializes the stack, the buffer and the super-object collection and then calls function `augmentedDFS!` with the root of the Object3D tree (= the topmost parent Object3D of the root level assembly) as input argument. The details of function `augmentedDFS!` are given below:

```
58 function augmentedDFS!(root::Object3D)
59   push!(buffer, root)
60   actPos = 1
61   nPos   = 1
62   while actPos <= nPos
63     superObj = SuperObjs()
64     obj = buffer[actPos]
65     if obj != root
66       assignAll(superObj,obj)
67     end
68     fillStackOrBuffer!(superobj,obj)
69     while length(stack) > 0
70       objChild = pop!(stack)
71       assignAll(superObj,objChild)
72       fillStackOrBuffer!(superObj,objChild)
73     end
74     safeSuperObjsToCollection(coll,superObj)
75     nPos = length(buffer)
76     actPos += 1
77   end
78   addIndicesOfCutJointsToSuperObj(coll)
79 end
```

First, the root Object3D is pushed on the buffer and the current element of the buffer `actPos` is set to one. Afterwards all elements of the buffer are inspected. For every element of the buffer a depth-first search is performed. All

DOI 10.3384/ecp19157383

Object3Ds are pushed on the stack that are rigidly connected with their parents. Otherwise, it is pushed on the buffer. This decision is made with function `fillStackOrBuffer!`:

```
80 function fillStackOrBuffer!(superObj,obj)
81  for child in obj.children
82   if isNotRoot(child)
83    if isNotFixed(child)
84     push!(buffer,child)
85     if !child.joint.canCollide
86      push!(superObj.noCPair,length(buffer))
87     end
88    else
89     push!(stack,child)
90 end; end; end; end
```

For each element of the `SuperObjs` data structure (lines 46 - 53) the `assignAll` function:

```
91 function assignAll(superObj,obj)
92   for val in fieldnames(typeof(superObj))
93     assignObj(getfield(superObj,val), obj)
94   end
95 end
```

calls the `assignObj` function to store the Object3D in the particular specialized super-object. The right `assignObj` function is chosen via multiple dispatch of the Julia programming language, see Section 4.1.

4.3 Algorithms for Using Super-Objects

In this section, the usage of the generated data structure is shortly sketched for an efficient evaluation during integration.

4.3.1 Usage of Collision-Super-Objects

The MPR-algorithm computes the distance between two shapes in the narrow phase `narrowPhase_MPR` (line 111) if their AABBs are intersecting in the broad phase `broadPhase_checkAABB` (line 110). All Object3Ds (line 104) of the actual super-object (line 102) are allowed to collide with all Object3Ds (line 109) of the subsequent super-object (line 107). In case the actual super-object is not allowed to collide with the subsequent super-object, the index of the subsequent super-object is stored in `noCPairs` (see Figure 3 and Section 4.1.2).

```
96 # counter
97 # is: actual super-object
98 # js: subsequent super-object
99 # i: Object3D of is_th super-object
100 # j: Object3D of js_th super-object

101 for is = 1:length(collSuperObjs)
102  actSuperObj = collSuperObjs[is]
103  for i = 1:length(actSuperObj)
104   actObj = actSuperObj[i]
105   for js = is+1:length(collSuperObjs)
106    if !(js in noCPairs[is])
107     nextSuperObj = collSuperObjs[js]
108     for j = 1:length(nextSuperObj)
109      nextObj = nextSuperObj[j]
110      if broadPhase_checkAABB(actObj,nextObj)
111       narrowPhase_MPR(actObj,nextObj)
112 end; end; end; end; end; end
```

4.3.2 Usage of Mass-Super-Objects

If there are two or more Object3Ds with mass-properties in a super-object, the resultant mass, inertia tensor and center of mass is computed and a new Object3D is constructed at the center-of-mass location. The previous mass-properties objects are removed.

4.3.3 Usage of Force- and Visualization-Super-Objects

In general, the Object3Ds on a super-object form a tree. This tree is reconstructed so that every Object3D has the root of the super-object as parent, in order to avoid unnecessary coordinate transformations during integration. All Object3Ds with exception of the Object3Ds that are only used for visualization, are stored in an Object3D vector `Object3DEvaluation` in depth-first order. During integration, the absolute position, rotation, velocity, angular velocity, acceleration, angular acceleration of all Object3Ds are computed at every model evaluation by traversing this vector from index 1 up to its last index and computing the absolute quantities of an Object3D with its relative quantities and the absolute quantities of its parent Object3D. Furthermore, the forces and torques due to the acceleration/angular acceleration of the mass properties Object3Ds from section 4.3.3 are computed and stored in the respective Object3D, as well as the forces and torques of all `forceElements` and of all contact force elements. Afterwards, vector `Object3DEvaluation` is traversed from its last index down to index 1 and the resultant force and torque at an Object3D is transformed and summed to the force and torque of its parent Object3D. Finally, the projection of the forces/torques at all joints into the non-constrained motion of the respective joint results in the residues $\bar{f}_2$ of equation (7).

The handling of the cut-joints is a bit more involved (to compute the residues $\bar{f}_3, \bar{f}_4$) and is not further elaborated here. The absolute position and rotation of the Object3Ds that have only visualization-objects need to be computed only at communication points. This calculation is a simple extension of the approach sketched above.

5 Simulation

Once the preprocessing steps are finished, the model is transformed to DAE form (1) as sketched in section 4.3.3 and solved with Sundials IDA (Hindmarsh et al., 2005, 2015) that uses a variable-step, variable-order BDF-integration (Backward Differentiation Formula) method. The transformation of a multi-body system with kinematic loops (for an example see figure 7) to the form (1) is sketched in (Otter and Elmqvist, 2017) and shortly repeated here:

Starting point are the equations of motion of a multi-body system, see, e.g. (Arnold, 2016):

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{v} \\ \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{G}^T(\boldsymbol{q},t)\boldsymbol{\lambda} &= \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) \\ \boldsymbol{0} &= \boldsymbol{g}(\boldsymbol{q},t) \end{aligned} \tag{2}$$

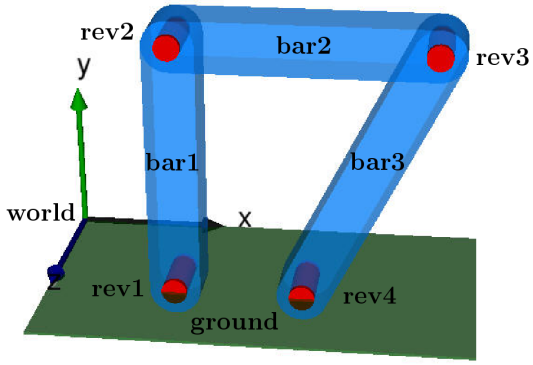

Figure 7. Modia3D model of a multi-body system with a kinematic loop.

where $\boldsymbol{q}$ are the generalized coordinates (here: joint coordinates, such as a revolute angle), $\boldsymbol{v}$ are the generalized velocities, $\boldsymbol{\lambda}$ are the generalized forces/torques in the cut-joints, $\boldsymbol{M} = \boldsymbol{M}^T$ is the positive definite mass matrix, $\boldsymbol{g}$ are the kinematic constraint equations of the cut-joints on position level, $\boldsymbol{G} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{q}}$ are the partial derivatives of the constraints equations with respect to $\boldsymbol{q}$ and has full row rank. This DAE has index 3 and gives rise to numerical problems when integrating it directly. Instead, with the method of (Gear et al., 1985; Gear, 1988) it can be transformed to the following index 1 form, see (Otter and Elmqvist, 2017):

$$\begin{aligned} \mathbf{0} &= \dot{\boldsymbol{q}} - \boldsymbol{v} + \boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\mu}}_{int} \\ \mathbf{0} &= \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\lambda}}_{int} - \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) \\ \mathbf{0} &= \boldsymbol{g}(\boldsymbol{q},t) \\ \mathbf{0} &= \boldsymbol{G}(\boldsymbol{q},t)\boldsymbol{v} + \boldsymbol{g}^{(1)}(\boldsymbol{q},t) \end{aligned} \tag{3}$$

where (a) the derivative of the constraint equations $\mathbf{0} = \boldsymbol{g}(\boldsymbol{q},t)$ are added as new equations, (b) new unknowns $\dot{\boldsymbol{\mu}}_{int}$ are introduced that are used for stabilizing the DAE and (c) the generalized constraint forces $\boldsymbol{\lambda}$ are replaced by $\dot{\boldsymbol{\lambda}}_{int}$ the derivatives of its integral. The question is how these equations can be constructed by Modia3D:

IDA and other DAE integrators assume that the DAE is mathematically described as:

$$\mathbf{0} = \boldsymbol{f}(\dot{\boldsymbol{y}}(t), \boldsymbol{y}(t), t) \tag{4}$$

For the solution, the following Jacobian is computed numerically (c_h is a step-size dependent variable that is provided by the integrator):

$$\boldsymbol{J} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}} + c_h \cdot \frac{\partial \boldsymbol{f}}{\partial \dot{\boldsymbol{y}}} \tag{5}$$

This Jacobian can be automatically generated by IDA, but is optionally also provided by Modia3D (to experiment with sparse Jacobians).

One problem is that the new term $\boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\mu}}_{int}$ does not appear in the equations of motions and should be "somehow" constructed. The DAE variables $\boldsymbol{y}$ of the IDA interface are defined as:

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \boldsymbol{y}_3 \\ \boldsymbol{y}_4 \end{bmatrix} = \begin{bmatrix} \boldsymbol{q} \\ \boldsymbol{v} \\ \boldsymbol{\lambda}_{int} \\ \boldsymbol{\mu}_{int} \end{bmatrix} \tag{6}$$

Hereby $\boldsymbol{q}$ are the generalized coordinates of the joints in the tree of the super-objects (for example an angle of a revolute joint), $\boldsymbol{v}$ are the generalized velocities of these joints (for example the angular velocity of a revolute joint), $\boldsymbol{\lambda}_{int}$ is the integral of the generalized cut-forces in the cut-joints (for example the cut-forces of a spherical cut-joint) and $\boldsymbol{\mu}_{int}$ does not appear in the model.

In a first step the residues of the model equations are computed in the following form (note that $\boldsymbol{y}, \dot{\boldsymbol{y}}$ are provided by the IDA integrator as input arguments to the model):

$$\bar{\boldsymbol{f}} = \begin{bmatrix} \bar{\boldsymbol{f}}_1 \\ \bar{\boldsymbol{f}}_2 \\ \bar{\boldsymbol{f}}_3 \\ \bar{\boldsymbol{f}}_4 \end{bmatrix} = \begin{bmatrix} \dot{\boldsymbol{y}}_1 - \boldsymbol{y}_2 \\ \boldsymbol{M}\dot{\boldsymbol{y}}_2 + \boldsymbol{G}^T\dot{\boldsymbol{y}}_3 - \boldsymbol{h}(\boldsymbol{y}_1, \boldsymbol{y}_2, t) \\ \boldsymbol{g}(\boldsymbol{y}_1, t) \\ \boldsymbol{G}(\boldsymbol{y}_1, t)\boldsymbol{y}_2 + \boldsymbol{g}^{(1)}(\boldsymbol{y}_1, t) \end{bmatrix} \tag{7}$$

Hereby, $\bar{\boldsymbol{f}}_1$ is directly computed from the input arguments, $\bar{\boldsymbol{f}}_2$ are the generalized forces of the joints in the super-object tree (for example the projection of the cut-torque in a revolute joint to its axis of rotation, see section 4.3.3), $\bar{\boldsymbol{f}}_3$ are the generalized kinematic closure conditions of the cut-joints on position level and $\bar{\boldsymbol{f}}_4$ are the generalized kinematic closure conditions of the cut-joints on velocity level.

From time to time (so not in every step) the integrator requires a Jacobian (5). First, the part of the Jacobian is computed numerically where $\dot{\boldsymbol{\mu}}_{int}$ is not yet taken into account (using (7)):

$$\bar{\boldsymbol{J}} = \frac{\partial \bar{\boldsymbol{f}}}{\partial \boldsymbol{y}} + c_h \cdot \frac{\partial \bar{\boldsymbol{f}}}{\partial \dot{\boldsymbol{y}}} \tag{8}$$

It can be noted that matrix $\boldsymbol{G}$ is part of this Jacobian (the rows of this Jacobian with respect to $\bar{\boldsymbol{f}}_4$ and the columns with respect to $\boldsymbol{y}_2$):

$$\boldsymbol{G} = \frac{\partial \bar{\boldsymbol{f}}_4}{\partial \boldsymbol{y}_2} = \bar{\boldsymbol{J}}_{42} \tag{9}$$

It is now possible to compute the Jacobian (5) as required by IDA:

$$\boldsymbol{J} = \bar{\boldsymbol{J}} + \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & c_h \cdot \bar{\boldsymbol{J}}_{42}^T \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{10}$$

Furthermore, in every model evaluation also the residues (4) can be calculated as required by IDA:

$$\boldsymbol{f} = \bar{\boldsymbol{f}} + \begin{bmatrix} \bar{\boldsymbol{J}}_{42}^T \cdot \boldsymbol{y}_4 \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{11}$$

DOI 10.3384/ecp19157383

Since the Jacobian (5) is not computed in every integrator step, $\bar{\boldsymbol{J}}_{42}^T \cdot \boldsymbol{y}_4$ need not be identical to $\boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\mu}}_{int}$ because $\bar{\boldsymbol{J}}_{42}^T$ is potentially computed at a previous time instant. However, this is uncritical because the method of Gear to stabilize the DAE only requires that matrix $\boldsymbol{G} \cdot \bar{\boldsymbol{J}}_{42}^T$ must be regular (see for example (Otter and Elmqvist, 2017): the derivation after eq. (13) shows that $\dot{\boldsymbol{\mu}}_{int} = \boldsymbol{0}$). In the unlikely situation that this approximation of the stabilizing term looses rank, the integrator will most likely detect a problem with its variable step-size control and will force a new computation of the Jacobian, that will solve the issue.

6 Relation to other Work

Modern games use physics engines, like Havok or PhysX for collision detection and rigid body simulations (Gregory, 2014). Physics engines of games work with fixed-step size solvers and are interactive real-time simulations. Modia3D simulates the system with variable-step size solver because the target of its initial version is offline simulation. Therefore, there are natural differences between the implementation approaches, for example in a physics engine the position and orientation of visual objects need to be computed in every model evaluation, whereas in Modia3D this is only needed at communication points (if the visual object does not take place in collision handling). To improve efficiency, this is specially handled in Modia3D.

Game engines typically use a scene graph[15] (Gregory, 2014) to describe the representation of the 3D objects. Often this is a tree data structure where all operations applied on a node effect all children nodes. Changes to nodes might be material data, such as the color of objects, or 3D transformations. Usually, closed kinematic loops are not supported by game engines or are approximated with various techniques. Therefore, a scene graph with a tree data structure is sufficient.

In Modia3D kinematic loops are inherently supported and therefore a pure tree data structure does not reflect the system. From a users point of view a 3D system is a graph with loops. It seems therefore not useful to apply, say, a color to a node and define that this color holds for all children, because the children might be part of a loop that includes the node. Instead in Modia3D, an object such as a material object might be defined once and then references to this object might be used in the various nodes. For practical reasons, the graph is represented internally as a tree with additional information for the closing conditions of kinematic loops. However, this is hidden from the user and the user should not know in which way an internal tree is constructed (this might even change with a new version).

The Modelica MultiBody library (Otter et al., 2003) was implemented in 2003 and since this time only minor improvements have been made. The design is made in a rigid way by defining a few part types, such as `BodyShape` or `BodyBox`, to represent a fixed setup for a part with a geometry, mass properties computed from this geometry and a fixed set of frame connectors. This design is far away from the flexibility of the Modia3D library where various geometries, including base shapes and meshes, can be defined and used in various ways for visualization, mass properties computation, collision handling. Systems with kinematic loops can be defined with the Modelica MultiBody library, but it was not possible to make this fully automatic so that the user just defines the system as it is. Instead, practically the user has to define somehow the cut-joints or assembly-joints for a kinematic loop and also has usually to explicitly define the states in a loop with the `StateSelect` attribute, because otherwise the simulation becomes much too slow due to the dynamic state selection.

[15] https://en.wikipedia.org/wiki/Scene_graph

7 Conclusion

In this article some newly developed algorithms have been described that are used by the Modia3D prototype to construct a model that can be efficiently evaluated in a simulation with a variable-step solver. Due to its architecture that is inspired by game engines, Modia3D allows a very flexible way to build-up dynamic models of 3D-mechanical systems and to model collisions. Contrary to games, the main target of the package design are variable-step solvers with step-size control. Modia3D is still an early prototype and several important parts are under development, especially the integration with Modia is missing. Furthermore, the code was currently mainly developed for its functionality and is not yet tuned for efficiency. For these reasons, benchmarks about the simulation efficiency have not yet been performed.

References

M. Arnold. DAE aspects of multibody systems. Technical report, Martin-Luther-Universität Halle-Wittenberg, Institut für Mathematik, April 2016. URL http://sim.mathematik.uni-halle.de/reports/sources/2016/01-2016.pdf.

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

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

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

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59 (1):65–98, 2017.

H. Elmqvist, S. E. Mattsson, and C. Chapuis. Redundancies in Multibody Systems and Automatic Coupling of CATIA and Modelica. In *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*, pages 551–560. Linköping University Electronic Press, 2009. URL http://www.ep.liu.se/ecp/043/063/ecp09430113.pdf.

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

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

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

C. W. Gear. Differential-algebraic equation index transformations. *SIAM J. Sci. Stat. Comput.*, 9(1):39 – 47, 1988.

C. W. Gear, G.K. Gupta, and B. Leimkuhler. Automatic integration of euler–lagrange equations with constraints. *J. Comp. Appl. Math.*, 12-13:77 – 90, 1985.

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

J. Gregory. *Game engine architecture*. AK Peters/CRC Press, 2014.

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

C. Höger, A. Mehlhase, C. Nytsch-Geusen, K. Isakovic, and R. Kubiak. Modelica3D - Platform Independent Simulation Visualization. In M. Otter and D. Zimmer, editors, *Proc. of the 9th International Modelica Conference*, Sept. 2012. URL http://www.ep.liu.se/ecp/076/049/ecp12076049.pdf.

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

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

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

J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

B. Kenwright. Generic Convex Collision Detection using Support Mapping. Technical report, 2015. URL https://www.semanticscholar.org/paper/Generic-Convex-Collision-Detection-using-Support-Kenwright/4f0f2d95375db7cfdbfaa345847418789d8cb970.

A. Neumayr and M. Otter. Collision Handling with Variable-step Integrators. In *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT'17, pages 9–18. ACM, 2017. URL https://modiasim.github.io/Modia3D.jl/resources/documentation/CollisionHandling_Neumayr_Otter_2017.pdf.

A. Neumayr and M. Otter. Component-Based 3D Modeling of Dynamic Systems. In M. Tiller, H. Tummescheit, and L. Vanfretti, editors, *Proceedings of the American Modelica Conference*, Oct. 2018. URL https://elib.dlr.de/124126/1/2018_Modelica_Modia3D.pdf.

R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014. URL http://gameprogrammingpatterns.com/.

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

M. Otter, H. Elmqvist, and S. E. Mattsson. The New Modelica MultiBody Library. In P. Fritzson, editor, *Proc. of the 3rd International Modelica Conference*, Nov. 2003. URL https://www.modelica.org/events/Conference2003/papers/h37_Otter_multibody.pdf.

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

P. Sahlin and P. Grozman. IDA Simulation Environment - a tool for Modelica based end-user application deployment. In P. Fritzson, editor, *Proc. of the 3rd International Modelica Conference*, Nov. 2003. URL https://www.modelica.org/events/Conference2003/papers/h33_Sahlin.pdf.

G. Snethen. Xenocollide: Complex collision made simple. In Scott Jacobs, editor, *Game Programming Gems 7*, pages 165–178. Charles River Media, 2008.

R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.