Simulating Visual Geometry

Matthias Müller Nuttapong Chentanez Miles Macklin

NVIDIA Research

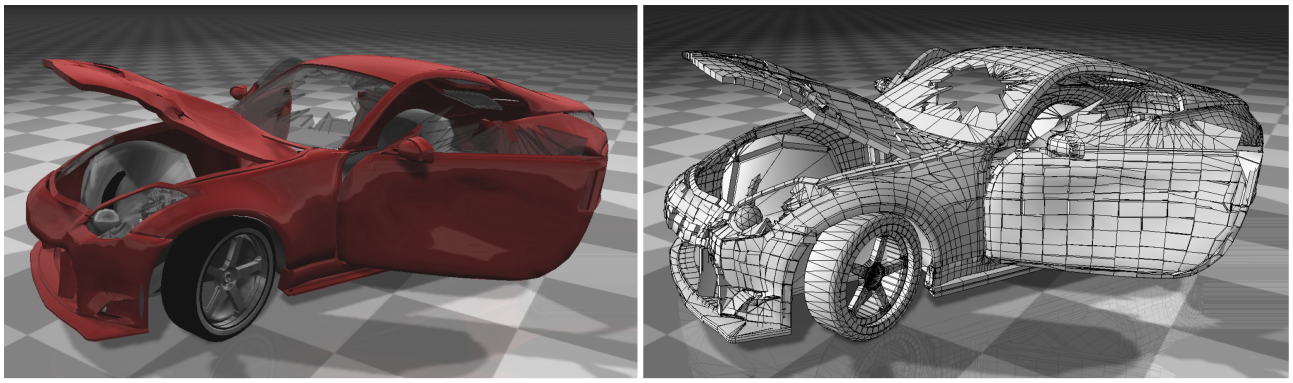

Figure 1: *Left: A car model brought to life with our method. It can be driven, deformed and shattered and interacts with the environment as expected. Right: Our simulation model based on convex polyhedra as primitives which are derived from the faces of the input mesh.*

Abstract

In computer graphics, simulated objects typically have two or three different representations, a visual mesh, a simulation mesh and a collection of convex shapes for collision handling. Using multiple representations requires skilled authoring and complicates object handing at run time. It can also produce visual artifacts such as a mismatch of collision behavior and visual appearance. The reason for using multiple representation has been performance restrictions in real time environments. However, for virtual worlds, we believe that the ultimate goal must be *WYSIWYS* – what you see is what you simulate, what you can manipulate, what you can touch.

In this paper we present a new method that uses the same representation for simulation and collision handling and an almost identical visualization mesh. This representation is very close and directly derived from a visual input mesh which does not have to be prepared for simulation but can be non-manifold, non-conforming and self-intersecting.

Keywords: deformation, fracture, tearing, convex primitives, oriented particles, rigid body simulation

Concepts: •**Computing methodologies** → **Physical simulation;**

1 Introduction

Objects in films and games are typically created in a digital content creation (DCC) tool as triangle or quad meshes that represent the object's surface. These meshes are fine tuned for rendering and rigged kinematic animation but not prepared for physically-based animation. One of the main difficulties is that visual meshes are typically not conforming meaning that elements overlap. Often objects such as a cup and its handle are not modeled as one connected mesh but as separate meshes that intersect each other.

A popular way to handle these problems is to not simulate the visual mesh directly but to embed it in a simulation mesh or attach it to a more general structure such as a mass-spring network or a skeleton. There are several advantages of this method. The simulation mesh can be tuned independenty of the tessellation of the visual mesh. Often, the resolution of the simulation mesh is chosen to be lower so small visual details do not slow down the simulation. Also, there are no restrictions on the structure of the input mesh, it can even be a triangle soup.

On the other hand, creating an appropriate simulation mesh can be a non-trivial task. Also working with two representations complicates the simulation code. Tearing and fracturing the visual mesh along with the physical mesh becomes a complex problem. In addition separate representations for simulation and visualization can yield visual artifacts.

For real time applications, simulating detailed and complex visual geometry has been too slow because of the memory and compute power restrictions of game consoles. With the recent introduction of cloud gaming, this situation might change because the simulation can be done on a powerful remote machine in this case. Therefore, we were interested in finding a method to potentially simulate every visible detail. We call this *WYSIWYS*, i.e. What You See Is What You Simulate. Ultimately, an ideal virtual world should be *WYSIWYS*, especially in virtual reality environments which have become very popular in recent years.

To make a step in that direction, we propose a method that uses a simulation mesh which is very close to the visual mesh. We use convex shapes as primitives. These are either created by extruding polygonal faces of the visual input mesh along the negative normal direction or by identifying convex shapes surrounded by visual faces.

MiG '16,, October 10-12, 2016, Burlingame, CA, USA
ISBN: 978-1-4503-4592-7/16/10
DOI: http://dx.doi.org/10.1145/2994258.2994260

The primitives are connected via a general graph. In this graph, two primitives are connected if they touch or overlap in the rest state. We simulate this structure efficiently treating each primitive as an oriented particle as in [MC11]. Representing these particles with convex polyhedra rather than ellipsoids as in the original approach lets us model flat surfaces and sharp corners and edges. Precise shapes are essential especially after fracture events when the pieces are in close proximity as Figure 7 shows. Since our primitives are convex polyhedra, we can use the approach of Müller et al. [MCK13] for subdivision and fracturing. In a hierarchical manner, larger parts such as the car doors or the hood in Figure 1 are grouped into bodies and connected via joints. Since rigid body engines such as PhysX represent objects as a collection of convex shapes, they can be used directly to simulate the object on this higher level. Our representation can be created quickly and maintained in a straight forward way through deformations and fracture events.

Our main contributions are

- The representation of objects with convex polyhedra derived directly from the visual mesh and a connectivity graph defined by their proximity in the rest state.
- Using the oriented particle approach to simulate these convex polyhedra instead of ellipsoids.
- The idea of deforming the primitives using the affine transformation provided by shape matching. This idea is general and can be applied to other frameworks as well.
- The definition of a graphical mesh that follows the primitives closely, stays watertight throughout deformations and is straight forward to tear and fracture.
- Extending the method of [MCK13] for brittle fracture to ductile fracture and the simulation of detailed deformations in coarse parts of the mesh.

2 Related Work

Our method is built on a few core ideas, namely using a general, non-conforming unstructured mesh, combining the rigid body formulation with a deformable model, fracture and tearing algorithms and a unified solver based on position based dynamics (PBD) [BMM15]. There is a large body of work in all those fields so we will mention only the most relevant papers for our method.

2.1 Simulation Models

In solid simulations, a variety of models have been used to represent objects. Early solutions are mass-spring networks [SLF08] and regular grids in connection with finite differences [TF88]. Regular grids have also been used in connection with finite elements (FEM) [MTG04]. The most popular representation for deformable volumetric objects are tetrahedral meshes mostly simulated with the finite element method such as [OH99], [ITF04]. Another solution for solid simulations is to use particles without connectivity. FEM can be formulated for this representation via the moving least squares technique (MLS) as in [MKN*04]. The material point method has become pupular in recent years. It is a hybrid method that alternates between particles and a grid [SSC*13], [SSJ*14], and [RGJ*15]. The use of Eulerian grids has been proposed for solid simulation as well [PLF14].

We use the oriented particles method proposed by Müller et al. [MC11]. The underlying model is a set of particles with orientation which are connected in an unstructured, arbitrary way. The approach is a generalization of the shape matching method [MHT05], [RJ07]. The oriented particle idea is somewhat related to the concept of elastons [MKB*10] in the sense that it can handle one-, two- and three dimensional objects in a unified way. Another related method is the approach of Faure et al. [FGBP11] which uses control points with orientation to define a continuous deformation field. Recently Choi et al. [Cho] have extended the oriented particles to support tearing and fracturing.

For the simulation we use a unified solver based on PBD as Macklin et al. [MMCK14] with the extension of handling convex polyhedra as primitives in addition to sphere shaped particles and a position based rigid body formulation similar to the one proposed by Bender et al. [DCB14]. The PDB formulation was extended by Bouaziz et al. [BML*14] by including an inertia term to make the method compatible with implicit Euler integration and reduce damping. The latter can also be achieved by using a second order velocity update [BMM15].

2.2 Embedding of a Visual Mesh

In games and film, objects are defined by visual surface meshes which are typically not tuned for simulation. Therefore, the most popular approach has been to embed the visual mesh in a simulation mesh. This fundamental idea has been introduced by Sedeberg et al. [SP86] with free form deformation. Here objects are deformed by deforming a surrounding cage. Müller et al. have embedded a separate visual mesh in a tetrahedral mesh [MG04] and a regular grid [MTG04] and have shown how to split it when the simulation mesh fractures. Instead of interpolating vertex positions of a surrounding simulation mesh, the transformations of bones of a skeleton are blended in linear blend skinning [MTLT]. In [MC11], Müller et al. use a similar idea to attach a visual mesh to oriented particles by blending the transformations defined in nearby particles by their position and orientation. Without a surrounding volumetric mesh, splitting the visual mesh becomes a difficult problem. Jones et al. [JML*16] do not work with an arbitrary surface mesh but create a conforming tetrahedral mesh for visualization which they split along the tetrahedral boundaries. This approach can yield artifacts when the structure of the tetrahedral mesh becomes visible. In contrast, applying the fracture method of Müller et al. [MCK13] makes sure that the mesh structure remains hidden and only the tear lines in the fracture pattern become visible after tearing and fracturing.

2.3 Fracturing and Tearing

As the pioneers in physically based animation, Terzopoulos et al. [TF88] also introduced fracture simulation to computer graphics. They broke links in a regular grid finite difference model if the stresses exceeded a given threshold. Later O'Brien et al. used a tetrahedral mesh in connection with FEM for the simulation of brittle [OH99] and ductile [OBH02] fracture. They determined the fracture directions based on the direction of the stresses inside the material. A large body of work based on similar ideas followed. For computer games however, FEM based stress analysis is typically computationally too expensive. The traditional and most popular approach here is to use pre-fractured models and simply replace the whole by the parts at run time. With this approach, the actual impact location cannot be taken into account. To bridge the gap between these too extremes, Su et al. [SSF09] proposed to use pre defined fracture patterns that are applied at the impact location at run time using signed distance fields. Müller et al. [MCK13] made this approach pixel accurate and applicable to games by representing objects and fracture patterns with convex decompositions which is the approach we use in our method.

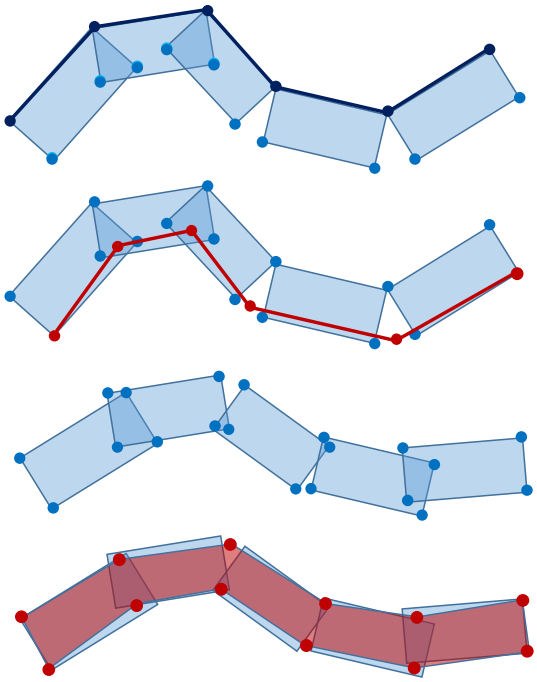

Figure 2: *From visual to simulation mesh (2d cut). Top: the input triangles (dark blue) are extruded to form volumetric convex primitives (light blue). Second row: the vertices of the primitives (light blue) are grouped and their position averages (red) for visualization. This yields a consistent inner surface in the rest state. Third row: during simulation the primitives change their poses potentially yielding gaps. Bottom: the gaps are closed when the averaged positions are used for visualization.*

3 The Method

We will now describe the steps from a visual mesh to a physical simulation in more details.

3.1 Physical Mesh Creation

Our input is a visual mesh with triangle and quad faces potentially intersecting each other. The physical properties that cannot be derived from the geometry automatically are defined by the user for each sub-mesh in analogy to the graphical material parameters. Examples of physical parameters are, thickness, material type such as soft, plastic or brittle and stiffness.

With this additional information, the input mesh is automatically turned into a simulation mesh as shown in Figures 2 and 3.

Each face is turned into a convex polyhedron by extruding it along the negative normal by the user defined thickness. These polyhedra become the primitives of our simulation mesh. The top row of Figure 2 shows this in a 2d cut, where the faces are shown as dark blue segments and the primitives as light blue boxes. If desired, we also create single primitives from visual geometry that encloses a convex shape as shown in Figure 3. Only the input faces that are not used in this way are extruded. This allows the definition of more general convex primitives and filled shapes. We then create a connection between each pair of primitive that touch or intersects. This results in a general mesh structure. Independent input meshes that are connected only visually by overlapping parts get physically connected.

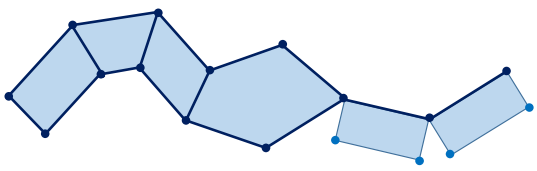

Figure 3: *Left part: if enclosed convex shapes are detected in the visual mesh, these are transformed into single primitives. The visual faces on the right are extruded because they do not belong to an enclosed volume.*

Figure 4: *Joint definition. Joints are defined by boxes. Their pose determines the location and extent of the joint while their names defines the joint type and the participating bodies.*

To build objects like the car in Figure 1 we allow the user to connect sub-meshes via joints. The joints are defined via additional boxes where the mesh name defines the joint type and the orientation the local axes as shown in Figure 4. In addition, each primitive of a soft body is automatically attached to overlapping rigid bodies. In this way, soft tires get automatically attached to the wheel.

3.2 The Simulation Method

For the simulation, we use a unified solver based on convex polyhedra as primitives. Since the primitives have an extent and therefore an orientation and since their connectivity is a general graph, the oriented particle approach [MC11] is ideal for the simulation of the objects. We define one shape matching group per primitive and all its 1-ring neighbors. These are used for both, soft body simulations of entire objects and plasticity simulations inside rigid objects. Due to the stability of the method, fine structures are handled robustly. Since all primitives are convex polyhedra we can use standard collision detection based on sweep and prune for the broad phase and the separating axis theorem for the narrow phase. Due to the large number of primitives, collision detection becomes the bottle neck, especially in situations where only a few rigid objects have to be simulated as in the car scene shown in Figure 1 but we still want detailed collision handling. We therefore use a three stage collision detection approach. In a broad phase, we find intersections of the bounds of entire objects. For pairs of intersecting objects we identify all their primitives that intersect the cut of the two object bounds. After identifying intersecting bounds of these primitive pairs in a second sweep and prune run, we perform the narrow phase convex - convex test using the separating axis theorem.

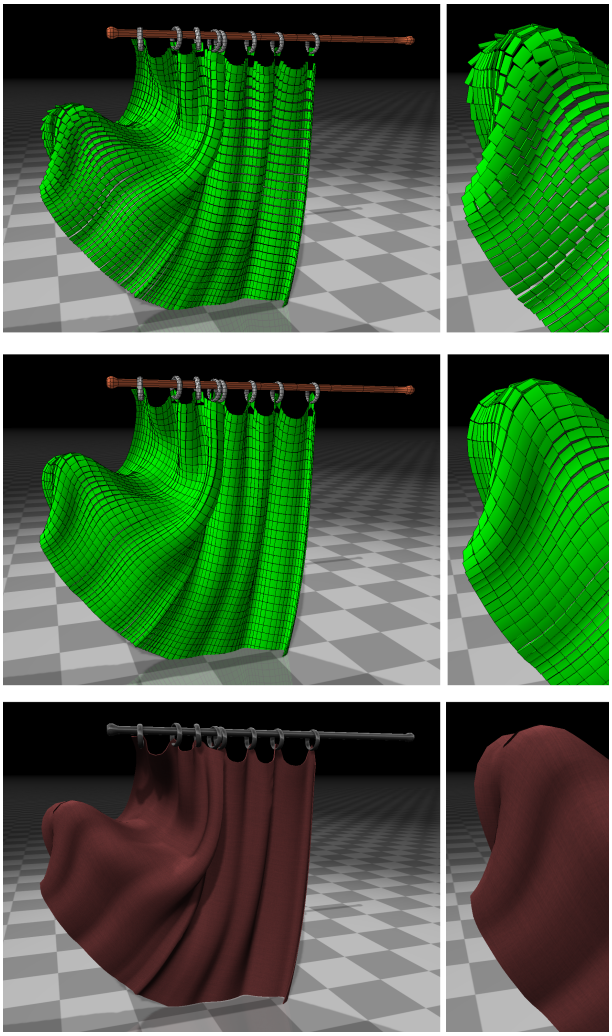

Figure 5: *Simulating a curtain. Top: using standard oriented particles. Middle: the local affine transformation of shape matching is applied to the primitives which reduces gaps significantly. Bottom: The visual mesh.*

3.3 Deforming Primitives

The standard oriented particle approach only modifies the location and orientation of primitives. This potentially yields gaps and collision artifacts as shown in the third row of Figure 2 and the top row of Figure 5. However, changing the shapes of primitives arbitrarily can destroy their property of being convex, the property that is essential for collision handling and fracturing. Fortunately we found a simple effective way to reduce these gaps substantially while keeping the primitives convex as shown in the second row of Figure 5. For this we deform the primitives using the local affine deformation matrices computed in the oriented particle approach. Being linear, this deformation preserves convexity of the primitives.

The local deformation matrix is computed from the positions of a particle and all its neighbors [MC11] as

$$\mathbf{A} = \sum_i \left(\mathbf{A}_i + m_i \mathbf{x}_i \bar{\mathbf{x}}_i^T \right) - M \mathbf{c} \bar{\mathbf{c}}^T, \tag{1}$$

where $\mathbf{A}_i = \frac{1}{5} m_i r_i^2 \mathbf{R}_i$ and $\mathbf{R}_i, m_i$, r_i, $\bar{\mathbf{x}}_i$ and $\mathbf{x}_i$ are the orientation matrix, mass, radius, rest and current position of primitive i, M the total mass and $\bar{\mathbf{c}}$ and $\mathbf{c}$ the rest and current center of mass of the neighborhood.

The matrix $\mathbf{A}$ defined in this way is sufficient to extract an optimal rotation but does not correspond to the true best fit affine transformation as noted in [MHT05]. To get our desired deformation matrix $\mathbf{D}$, we generalize the normalization of [MHT05] to the orientated particle approach and get

$$\mathbf{D} = \mathbf{A} \bar{\mathbf{A}}^{-1}, \tag{2}$$

where

$$\bar{\mathbf{A}} = \sum_i \left(\bar{\mathbf{A}}_i + m_i \bar{\mathbf{x}}_i \bar{\mathbf{x}}_i^T \right) - M \bar{\mathbf{c}} \bar{\mathbf{c}}^T \tag{3}$$

and $\bar{\mathbf{A}}_i = \frac{1}{5} m_i r_i^2 \mathbf{I}$. This normalization does not have to be performed at every solver iteration but only once per time step before collision handling.

3.4 The Visualization Mesh

Although the affine deformation of the primitives greatly reduces collision artifacts, it still yields small visual gaps as shown in Figure 5.

Therefore, we propose a new type of visualization mesh that stays close to the simulated primitives and for which fracturing and tearing can be implemented in a straight forward way. This is in contrast to attaching a general mesh to an independent simulation mesh where complex boolean operations have to be performed on the fly. Our basic idea is that vertices of different primitives are joined for visualization. For this each vertex of each primitive stores an identifier *id* that is global to the containing body. Vertices with the same global *id* are joined visually in two passes. First, all the positions of the vertices of the primitives are added to an array with an entry for each global *id*. These sums are then averaged. In a second pass, each vertex reads back the visual position from this array. Even without deformations, the inner surface of the simulation mesh contains gaps as shown at the top of Figure 2. The second row of Figure 2 shows that the visualization mesh has a consistent inner surface. The bottom row shows the visual mesh (red) when the simulation mesh (blue) is deformed.

Since our primitives are generated from faces of a polygonal mesh, we can use the vertex number of this mesh as the global id inside each primitive. For the vertices that are generated through extruding we generate the same connectivity structure. This way, the mesh is closed on the inside as well even though we use the face normals and not the vertex normals for the extrusion.

3.5 Plastic Deformation

In case of rigid parts, we treat the entire object as one rigid body between deforming impacts. Only when they are plastically deformed, the oriented particle approach is used to rearrange the primitives locally in a quasi-static simulation. This yields a substantial speed up. The car in Figure 1 has 17k primitives but only 20 rigid parts and runs at over 100 fps. Since we still use the detailed mesh for collision handling in this case, the fidelity of the simulation remains unchanged. To decide whether an object needs to be plastically deformed, we iterate through the contacts provided by the collision engine and check whether the relative normal velocity is above a threshold defined in the material. We define the deformation offset to be the relative normal velocity multiplied by the time step. This vector is transformed into the local frames or the objects to displace the participating primitive. After all contacts are processed and the corresponding primitives moved, we simulate all the primitives in a quasi static way holding all primitives that intersect

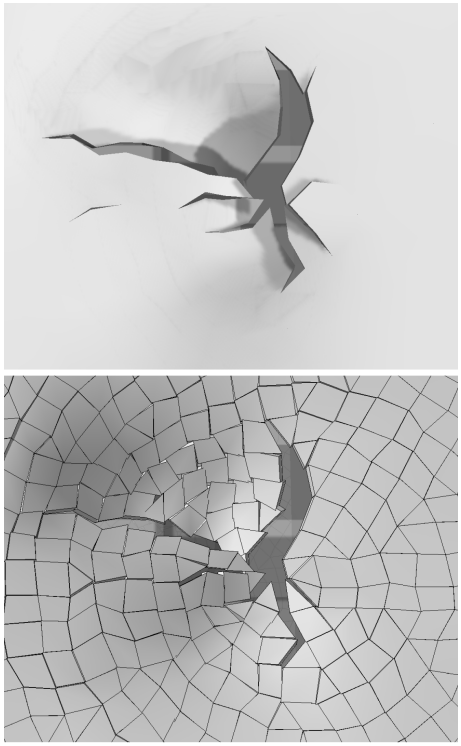

Figure 6: *Tearing. To tear a mesh, a fracture pattern is applied to subdivide the surface. Only a subset of the links between the primitives are marked as tearable, yielding a pre-defined tear pattern.*

the shape of joints fixed. This is why joints have to be defined with a spatial extent. It is because they act as boundary conditions for the static solver. After the modification of the local positions of primitives, the center of mass and inertia tensor of the containing object have to be re-computed.

3.6 Subdivision, Fracture and Tearing

A further advantage of using convex polyhedra as primitives is that we can use the approach of Müller et al. [MCK13] for fracturing the mesh. The main idea is to use a *fracture pattern* which is a connected set of convex polyhedra like the objects to be fractured. First, the fracture pattern is aligned with the impact location and then the cuts of all the primitives of the objects against all fracture cells are computed. All resulting pieces within a fracture cells are combined to form new independent objects. This process simulates brittle fracture and is used for the shattering of the car windows shown in Figure 7.

We propose a modification of this approach that lets us simulate ductile fracture and adaptive level of detail. The idea is to not separate the resulting objects immediately but keep all the pieces in the same object. The effect of applying a pattern in this way is that a mesh is subdivided close to an impact location allowing the formation of detailed deformation patterns in regions where the original mesh is coarse. The wings of the door shown in Figure 10 only contain a few large faces. After applying a fracture pattern, small scale deformations are generated. With deformation patterns, fine tesselation is created only where needed at run time.

Two further extensions allow the simulation of ductile fracture and tearing as shown in Figure 6. First, we test whether the distance between a connected pair of primitives exceeds a threshold. This

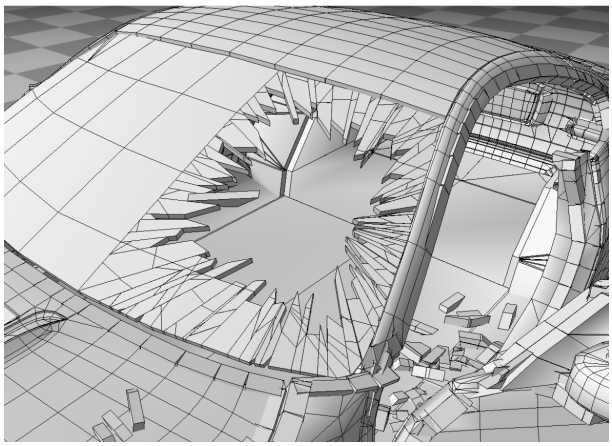

Figure 7: *A glass fracture pattern is applied to the car's window. The small pieces remain at the correct positions on the detailed surface and inside the car.*

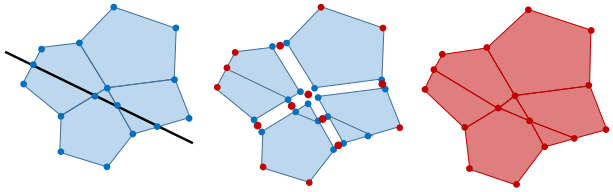

Figure 8: *Fracture of the visual mesh. The fracture operation is performed in the undeformed state. The vertices to be grouped are determined from scratch after a fracture event. Left: all vertices (light blue) that share the same location are grouped. Middle: the averaged positions in the deformed state. Right. The fractured mesh in the deformed state.*

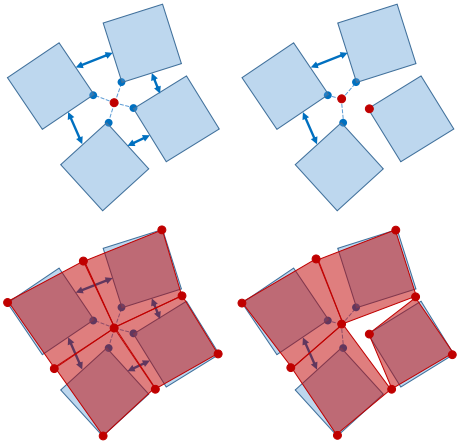

Figure 9: *Tearing of the visual mesh. Top: a set of primitives only shares a common visual vertex if they are mutually connected. If a torn link separates this set, each set gets its own visual vertex. Bottom: this yields the correct crack in the visual mesh.*

threshold is defined as a strain limit defined in the material times the rest length. If this is the case, we mark the connection as broken. Second, it is important to only make a subset of the links breakable. Only links introduced by applying a fracture pattern and not the ones created from the initial mesh should be broken, otherwise the initial tessellation becomes visible in the fracture lines. In addition, by only making a subset of the links of the fracture pattern breakable, an artist can pre-define tear lines such as the ones shown in Figure 6.

Fracturing and tearing require updates in the visual mesh. These are substantially less complex than updating an independent visual mesh because we have a one to one correspondence between visual faces and simulation primitives. All we have to do is to update or re-compute the global ids of the vertices. Figure 8 shows how fracture is handled.

It would be a difficult task to derive new global ids from the old ones after a fracture event in which many new primitives are generated. Fortunately we found a simple solution for this problem. We always apply the fracture patterns in the undeformed configuration. In this configuration the newly introduced as well as the previous vertices that should be grouped, i.e. get the same global id can be identified as the ones that have the same positions. This can be tested efficiently by sorting the vertices along the x, y, or z axis. In Figure 8, the cut in the undeformed configuration is shown on the left. The middle image shows how co-located vertices get the same global ids and the same averaged positions which results in the deformed configuration shown on the right.

To handle tearing, we maintain the following condition: Primitives that share a common global id need to be linked. To ensure this we need to know the number $val(id)$ of vertices that share each global id. This is computed for averaging during the evaluation of the visual positions. Given a primitive and one of its global ids, we perform the tearing as follows. We flood the graph starting from the primitive and visiting only neighbors that contain the given global id. If the number of visited primitives is smaller than $val(id)$, we create a new global id' and place id with id' in all visited primitives. Then we subtract $val(id')$ from $val(id)$. This local procedure is executed for both primitives adjacent to a broken link and all their referenced global ids. We show the result of this process in Figure 9.

Finally the joints have to be updated as well after a tearing or fracture event. Here we leverage the fact that joints are defined with spatial extents again. If an object gets separated into two or more parts it is deleted and replaced by a set of new objects. We then check all new objects against all the joints adjacent to the original object. If a new object intersects the volume of the joint, a new joint is created with the same attributes but referencing the new object. After all new objects and joints are created, the old body and all the joints referencing it are deleted.

4 Results

In all our examples we used a single core of an Intel Core i7 CPU at 3.1 GHz. At the moment we have a non-optimized serial implementation and expect to get substantial speedups with a GPU parallelization.

To simulate the car shown in Figure 1 we removed some interior geometry and left most of the outer input mesh as is. There are 21 objects, 28 joints and a total of 16k primitives. Creating the authored model in Blender shown in Figure 4 took less than an hour. The car simulates at more than 60 fps because the simulation consists of handling the high level objects only. When a deformation happens, the object's internal static solver is called and takes in the order of 1-10 ms per hit. In the accompanying video we show how the bullets and shattered pieces interact with the detailed geometry of the car.

Figure 5 shows a simulated curtain which demonstrates the effectiveness of applying the local affine transformation given by shape matching to the primitives.

The door scene shown in Figure 10 has 4 objects, 4 joints and 700 primitives at the start. Each first hit per object adds about 1k new primitives when the deformation pattern is applied. Between the impacts this scene runs at more than 100 fps.

The fourth experiment shown at the bottom of Figure 11 shows how soft body attachments persist throughout tearing. We did not modify the model of the monster truck for simulation but only added 9 joints. There are 45k primitives and the scene runs at 30 fps where most of the time is spent for solving the soft tires composed of 2.5k primitives each.

In the final scene shown in the top two shots of Figure 11 we simulated the monster truck driving over a shaky bridge. In addition to the monster truck there are 600 primitives and 1.5k attachments coming from the bridge. This complex scenario runs stably with our unified position based solver at 20 fps. The ropes are attached to the static blocks and simulated as soft bodies. The wooden steps are simulated as rigid bodies and are attached to the ropes. They interact with the soft tires via collisions. The tires are attached to the wheels which are in turn connected to the rest of the car via joints.

5 Conclusion and Future Work

We have presented a method to directly simulate the visual geometry provided by artists with the goal of generating a WYSIWYS virtual environment in which a user can potentially manipulate all details in a scene. For this we turn visual mesh faces into convex polyhedral primitives and define a mesh structure by connecting all touching or overlapping primitives. We simulate this general mesh using the oriented particle approach. On a higher level submeshes are turned into simulation objects and connected via joints. To substantially accelerate the simulation of plastic objects, we treat them as rigid objects between collisions. Using convex polyhedra as primitives allows us to use standard collision algorithms, the representation of sharp features and precise tearing and fracture operations.

At the moment all visual details are turned into simulation primitives. It might be reasonable to turn very small features into normal maps or similar visual representations and only keep elements larger than a threshold in the physical mesh.

Figure 10: *A metal door is deformed and torn. Initially, there only 8 faces per wing. At run-time, the application of the deformation pattern increases the tessellation density to represent the small scale features.*

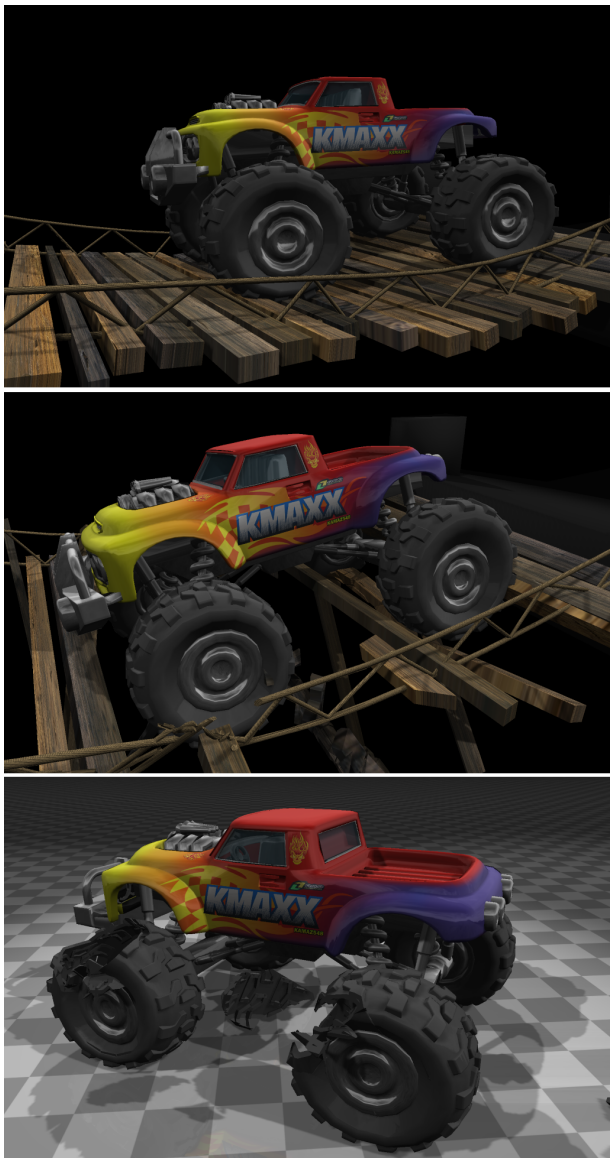

Figure 11: *Top: A complex interaction of various material types, joints and attachments. Bottom: The bursting tires show how attachments of soft bodies to rigid object persist when fractured.*

References

BOUAZIZ S., MARTIN S., LIU T., KAVAN L., PAULY M.: Projective dynamics: Fusing constraint projections for fast simulation. *ACM Trans. Graph. 33*, 4 (July 2014), 154:1–154:11. URL: http://doi.acm.org/10.1145/2601097.2601116.

BENDER J., MÜLLER M., MACKLIN M.: Position-based simulation methods in computer graphics. *EUROGRAPHICS Tutorial Notes, Zürich, May 4-8* (2015).

CHOI M. G.: Real-time simulation of ductile fracture with oriented particles. *Computer Animation and Virtual Worlds 25*, 3-4. URL: http://dx.doi.org/10.1002/cav.1601.

DEUL C., CHARRIER P., BENDER J.: Position-based rigid body dynamics. In *Computer Animation and Social Agents (CASA)* (2014).

FAURE F., GILLES B., BOUSQUET G., PAI D. K.: Sparse meshless models of complex deformable solids. In *ACM SIGGRAPH 2011 Papers* (2011), SIGGRAPH '11, pp. 73:1–73:10.

IRVING G., TERAN J., FEDKIW R.: Invertible finite elements for robust simulation of large deformation. In *Proceedings of the ACM SIGGRAPH Symposium on Computer Animation* (2004), pp. 131–140.

JONES B., MARTIN A., LEVINE J. A., SHINAR T., BARGTEIL A. W.: Ductile fracture for clustered shape matching.

MÜLLER M., CHENTANEZ N.: Solid simulation with oriented particles. *ACM Trans. Graph. 30*, 4 (July 2011), 92:1–92:10. URL: http://doi.acm.org/10.1145/2010324.1964987.

MÜLLER M., CHENTANEZ N., KIM T.-Y.: Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph. 32*, 4 (July 2013), 115:1–115:10. URL: http://doi.acm.org/10.1145/2461912.2461934.

MÜLLER M., GROSS M. H.: Interactive virtual materials. In *Graphics Interface 2004* (London, Ontario, Canada, 2004), pp. 239–246.

MÜLLER M., HEIDELBERGER B., TESCHNER M.: Meshless deformations based on shape matching. In *Proc. SIGGRAPH 2005* (2005), pp. 471–478.

MARTIN S., KAUFMANN P., BOTSCH M., GRINSPUN E., GROSS M.: Unified simulation of elastic rods, shells, and solids. *ACM Trans. Graph. 29*, 4 (July 2010), 39:1–39:10. URL: http://doi.acm.org/10.1145/1778765.1778776.

MÜLLER M., KEISER R., NEALEN A., PAULY M., GROSS M., ALEXA M.: Point Based Animation of Elastic, Plastic and Melting Objects. In *Symposium on Computer Animation* (2004), Boulic R., Pai D. K., (Eds.), The Eurographics Association. doi:10.2312/SCA/SCA04/141-151.

MACKLIN M., MÜLLER M., CHENTANEZ N., KIM T.-Y.: Unified particle physics for real-time applications. *ACM Trans. Graph. 33*, 4 (July 2014), 153:1–153:12. URL: http://doi.acm.org/10.1145/2601097.2601152.

MÜLLER M., TESCHNER M., GROSS M.: Physically-based simulation of objects represented by surface meshes. In *in Proceedings of Computer Graphics International (CGI)* (2004), pp. 26–33.

MAGNENAT-THALMANN N., LAPERRIERE R., THALMANN D.: Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics Interface 88 (Toronto, Ont., Canada, Canada)*.

O'BRIEN J. F., BARGTEIL A. W., HODGINS J. K.: Graphical modeling and animation of ductile fracture. In *Computer Graphics (SIGGRAPH 2002 Proceedings)* (San Antonio, Texas, July 2002), pp. 291–294.

O'BRIEN J. F., HODGINS J. K.: Graphical modeling and animation of brittle fracture. In *Computer Graphics (SIGGRAPH '99 Proceedings)* (New York, Aug. 1999), ACM Press, pp. 137–146. doi:http://doi.acm.org/10.1145/311535.311550.

PAI D. K., LEVIN D. I. W., FAN Y.: Eulerian solids for soft tissue and more. In *ACM SIGGRAPH 2014 Courses* (2014), SIGGRAPH '14, pp. 22:1–22:151. URL: http://doi.acm.org/10.1145/2614028.2615413.

RAM D., GAST T., JIANG C., SCHROEDER C., STOMAKHIN A., TERAN J., KAVEHPOUR P.: A material point method for viscoelastic fluids, foams and sponges. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2015), SCA '15, pp. 157–163. URL: http://doi.acm.org/10.1145/2786784.2786798.

RIVERS A. R., JAMES D. L.: Fastlsm: Fast lattice shape matching for robust real-time deformation. In *ACM Transactions on Graphics (Proc. SIGGRAPH 2007)* (2007), vol. 26(3), pp. 82:1–82:6.

SELLE A., LENTINE M., FEDKIW R.: A mass spring model for hair simulation. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 64:1–64:11. URL: http://doi.acm.org/10.1145/1360612.1360663.

SEDERBERG T. W., PARRY S. R.: Free-form deformation of solid geometric models. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986), SIGGRAPH '86, pp. 151–160. URL: http://doi.acm.org/10.1145/15922.15903.

STOMAKHIN A., SCHROEDER C., CHAI L., TERAN J., SELLE A.: A material point method for snow simulation. *ACM Trans. Graph. 32*, 4 (July 2013), 102:1–102:10.

SU J., SCHROEDER C., FEDKIW R.: Energy stability and fracture for frame rate rigid body simulations. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2009), SCA '09, pp. 155–164. URL: http://doi.acm.org/10.1145/1599470.1599491.

STOMAKHIN A., SCHROEDER C., JIANG C., CHAI L., TERAN J., SELLE A.: Augmented mpm for phase-change and varied materials. *ACM Trans. Graph. 33*, 4 (July 2014), 138:1–138:11. URL: http://doi.acm.org/10.1145/2601097.2601176.

TERZOPOULOS D., FLEISCHER K.: Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. In *the Proceedings of ACM SIGGRAPH 88* (1988), pp. 269–278.