

M. Sagardia / T. Hulin / C. Preusche / G. Hirzinger

Improvements of the Voxmap-PointShell Algorithm — Fast Generation of Haptic Data-Structures

ABSTRACT

The Voxmap-PointShellTM (VPS) Algorithm is a haptic rendering algorithm able to compute collision responses with 1 kHz update-rates with arbitrarily complex scenarios. This work introduces fast algorithms to generate the two haptic data-structures used by the VPS algorithm: *voxmaps* —voxelized volume structures for static objects— and *pointshells* —point-clouds describing moving objects—. For generating *voxmaps*, collision tests based on the *Separating Axis Theorem* are carried out. On the other hand, *pointshells* are generated utilizing optimization strategies to place points uniformly on the surface of the objects. The improvements in performance and quality obtained compared to the previous generating algorithms from DLR are included, all of them carried out on virtual car models provided by Volkswagen and artificial models.

1. INTRODUCTION

Virtual Reality (VR) simulations with haptic or touch feedback present appealing features for the industry. They make possible assembly checks of complex vehicle models in early stages of the product development process. It is also possible to use VR with training purposes for the future maintenance workers.

Features of VR with Haptic Feedback

In a VR set-up with force feedback the user moves a haptic device in the real world altering the position of objects in the virtual environment. Whenever a collision occurs, the collision force is felt by the user through the haptic device. Both collision detection and force computation are performed by the haptic rendering algorithm.

Whereas visual feedback requires update rates of around 30 Hz, touch or haptic feedback must be generated at 1 kHz rate in order to simulate collisions realistically. Due to this challenging requirement it was in the past 20 years when the computers and the emerging methods acquired enough features to be used in haptic rendering.

Haptic Rendering Algorithms

Most of the collision detection algorithms work with polygonal objects, and a big part of them rely on convex objects. The strategies that most usually appear are based on the *Separating Axis Theorem* (SAT) [5,1], when the complexity of the polygonal objects is low.

As far as the number of features —vertices, edges, faces— increases, other algorithms are used, such as the ones based on Minkowski differences [3], or algorithms that incrementally track collision areas, using the information of previous collision steps [8].

In 1995, Zilles et al. [15] introduced a new approach that uses a point —the Haptic Interface Point (HIP)— which symbolizes the end-effector of the haptic device. Every time the HIP penetrates an object, a Surface Constrained Point (SCP) is generated so that it remains as close as possible to the HIP, but yet on the surface. The distance between the HIP and the SCP is used to compute a spring-like repulsion force, which pushes the HIP out of the penetrated object.

This idea has been further extended to algorithms that go beyond the single-point paradigm. Amongst them lie “God-Object” methods that replace the HIP for complex objects [11], or the Voxmap-PointShell™ (VPS) Algorithm. The latter, introduced by McNeely et al. in 1999 [9], was reimplemented and improved at DLR [7,12]. It represents an appealing solution to generate haptic feedback within 1 kHz update-rates even with almost arbitrarily complex scenarios¹.

The scope of this work is focused on generating *voxmaps* and *pointshells*, which are the haptic data-structures used by the VPS algorithm. To this end, the work has been

¹ Although the complexity does not affect the time required for computing the collision response, there are parameters that influence it, e. g.: the number of points of the *pointshell* data-structure sets the computation time; or several shapes such as thin surface-like regions produce weaker collision forces than bulky solid objects.

divided into the following sections: Section 2 explains the VPS algorithm, while Sections 3 and 4 introduce new methods to generate the *voxmap* and the *pointshell*, respectively. Finally, Section 5 gathers generation results for models provided by Volkswagen, comparing former algorithms used at DLR; and Section 6 states future steps after summarizing the content of the paper.

2. THE VOXMAP-POINTSHELL™ (VPS) ALGORITHM

The VPS algorithm uses two data structures to generate the collision response: *voxmaps* and *pointshells*. *Voxmaps* are volume-based structures composed of *voxels*—three-dimensional analogs of pixels— that represent static models; a voxel is named to be a surface-voxel whenever it represents a solid part of the model. On the other hand, *pointshells* are point-clouds that describe the surface of dynamic (moving) models, having each point a normal pointing inwards the object. Figure 1 gathers schematic examples of these data-structures.

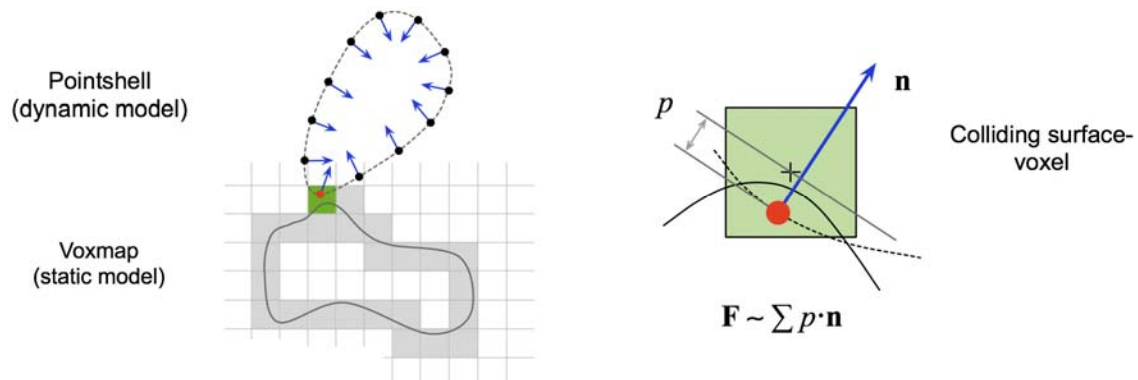


Figure 1: Schematical explanation of the Voxmap-PointShell™ (VPS) Algorithm.

During the haptic simulation, collision detection and force computation are performed every 1 ms in the original VPS algorithm [9], traversing all the *pointshell*-points that are in the scene. Every time a point is inside a surface-voxel, a collision is detected. The penetration is calculated measuring the distance from the point to the normal plane that goes through the center of the voxel. The penetration and the normal yield a single collision force, and all the collision forces summed together yield the total repulsion force. More detailed explanations concerning the collision detection and force computation are given in [9,10,13].

Improvements of the Original VPS

Further features have been added to the algorithm. In [12] a dynamic shaping filter was implemented to soften the force discontinuities. On the other hand, in [10] outwards-growing layers were used in the *voxmap* generating a *distance-field*, and the volume data-structure inherited geometrical properties of the polygonal model, as the vertices and the edges were considered to be the most important contact elements. Moreover, the constant-time performance set by the number of points was abandoned for the benefits of *spatial* and *temporal coherence*, also known as incremental behavior. Barbič et al. applied a hierarchy to the *pointshell* [2] in order to decrease the dependency on the total number of points by targeting only the points on the likely colliding regions.

Data-Structure Generation at DLR

As it can be seen in Figure 1, some of the most important elements in a *voxmap* are the surface-voxels, which are detected whenever a voxel in the three-dimensional grid which is fixed to the model contains or collides with a piece of the static object. Furthermore, inner and outer-voxels can be distinguished, according to whether they lie inside or outside the objects, respectively, and layers are generated around the voxelized surface both outwards and inwards. *Pointshells*, as mentioned, are formed with points spread on the surface of the moving models; each point has a vector which is normal to the surface pointing inwards the object. Hence, note that it is essential to detect outer and inner regions in order to compute the normal vectors that point inwards.

The old version for generating haptic data-structures at DLR was based on the open-source library SOLID [14]. The generation algorithm obtained first the *voxmap* of the polygonal model performing collision detection between the voxels of the three-dimensional grid described in the bounding box of the model: if a voxel was colliding with an object, it was marked as a surface-voxel. The next step consisted in projecting the surface-voxel centers on the polygonal model, obtaining an accurate *pointshell* [12].

This work presents algorithms based on the same idea, but instead of SOLID library, own algorithms are implemented in order to speed up the process and increase the quality of the outcoming data-structures.

3. VOXMAP GENERATION

The voxelization algorithm presented in this work is the following one: as first step the polygonal model is placed in the *voxmap*, where at the beginning each cube is an empty voxel. The algorithm goes through all the objects of the polygonal model, and per object, through all the polygons, which are split into triangles. For each triangle, the candidate voxels within its bounding box are traversed and checked for collision against the triangle. In case they collide, they are marked as a surface-voxel —this is done assigning the value 0 to the voxel—. Therefore, two main procedures appear in the algorithm: (i) fast navigation through the bounding box of the triangle detecting candidate surface-voxels and (ii) collision detection between triangle and candidate surface-voxel.

3.1. Navigation in the Bounding Box of the Triangle

The aim of the navigation is to traverse in the bounding box of the triangle all the likely colliding voxels as fast as possible. Figure 2 helps to understand how this is solved.

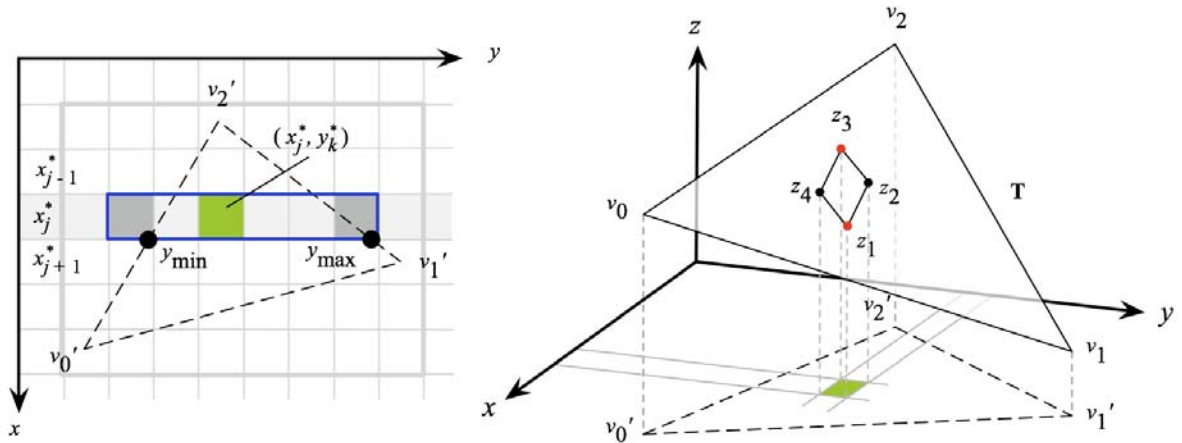


Figure 2: Navigation in the bounding box of the triangle detecting likely surface-voxels.

For a given triangle \mathbf{T} with vertices $v_i \in \mathcal{R}^3, \forall i \in \{0,1,2\}$, first, the discrete² axis-aligned bounding box —the one formed by the voxels— is detected. Second, taken the projection of the triangle on the xy -plane, the algorithm starts navigating along the x -axis stepping into successive discrete x^* values. For a given x_j^* , that defines a range of

continuous x (shadowed), the maximum and minimum y values on the triangle are detected (y_{\max}, y_{\min}). The pixels related to these extreme y values bound (in blue) the range of voxels in y -direction that are likely surface-voxels for this given x_j^* value. Third, for a pair (x_j^*, y_k^*) that represents a pixel in the bounded segment of the x_j^* row, the algorithm evaluates the four corners on the plane of the triangle, obtaining $z_i, \forall i \in \{1,2,3,4\}$. In a similar way as before, $\max_i(z_i)$ and $\min_i(z_i)$ are the real extremes to search for surface in the column fixed by the discrete (x_j^*, y_k^*) couple. Translating them into voxel discrete coordinates, the start and end navigation-points in z -direction are obtained.

Therefore, given a discrete x^* , the y^* navigation values are delimited. For each discrete (x^*, y^*) couple —which is contained in the pixelized shadow of the triangle— the algorithm goes up in z -direction and checks for collision every voxel contained between the delimited discrete z^* coordinates.

This first approach can be understood as constructing a tight bounding box around the triangle, contained in the former axis-aligned bounding box; the walls of this new box are aligned with the plane of the triangle and its edges. Obviously, the number of collision tests to be carried out with this approach decreases drastically compared to a more naïve approach that would consist in checking for collision all the voxels in the bounding box.

Nonetheless, it is possible to decrease more the number of collision checks working with a modified second approach. Instead of checking for collision all the voxels between the z^* extreme values in the column of a pixel (x^*, y^*) , it is possible to narrow the span in z^* by checking for collision the voxels upwards from maximal z^* and downwards from minimal z^* . Once the first surface-voxels are detected in both directions, the voxels between them are directly marked to be surface-voxels, without performing more collision checks.

² Discrete entities are marked with *.

However, this second approach turns out to be faster only with high resolutions, as it is going to be shown in Section 5.

3.2. Collision Detection between Triangle and Voxel

The algorithm performs collision detection between the candidate surface-voxel and the triangle using the *Separating Axis Theorem* (SAT), in a similar way as explained in [1]. The SAT states that two convex objects collide with each other if and only if there is an axis where their projections overlap. If no such axis exists, both objects are disjoint. Hence, the strategy consists in simplifying the problem into one dimension.

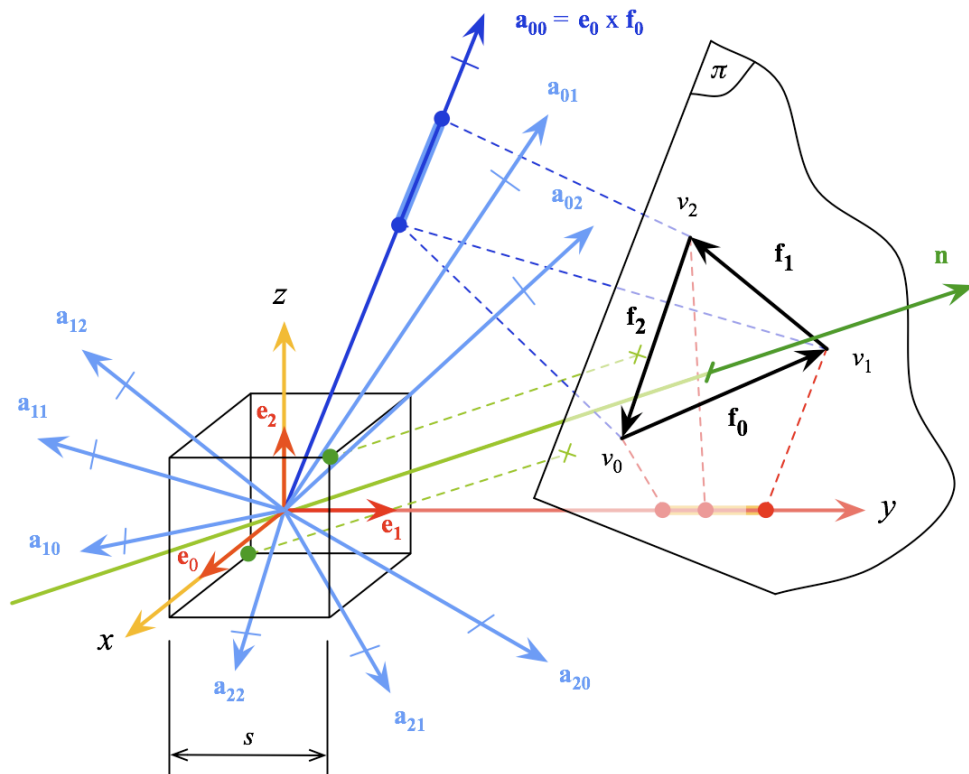


Figure 3: Collision detection between triangle and voxel using the *Separating Axis Theorem* (SAT). Red: three coordinate axes $\{e_0, e_1, e_2\}$; Green: the normal axis n ; Blue: nine cross-axes $\{a_{ij}\}$.

In the three dimensional world, the axes to be checked for collision are the normal vectors of the faces and the cross products between each pair of edges. As shown in Figure 3, the triangle is described by its three vertices $v_i \in \mathbb{R}^3, \forall i \in \{0,1,2\}$, being its edges $f_0 = v_1 - v_0$, $f_1 = v_2 - v_1$ and $f_2 = v_0 - v_2$. The voxel is determined with its center—which is considered to be the origin—and its edge-size s .

There are 13 axes to be checked for overlap, which can be classified into three groups:

1. The three COORDINATE AXES $\mathbf{e}_0 = \{1,0,0\}$, $\mathbf{e}_1 = \{0,1,0\}$ and $\mathbf{e}_2 = \{0,0,1\}$, representing the normal vectors to the faces of the voxel (in red in Figure 3). The three vertices of the triangle are projected on each axis to check whether the projection of the triangle on those axes overlaps the projection of the voxel, which is a region of width s centered in the origin.
2. The NORMAL OF THE TRIANGLE \mathbf{n} (in green). The vertices of the voxel (green points) that define the vector which is aligned with \mathbf{n} are checked for their relative position against the plane of the triangle π .
3. The nine axes that result from performing the CROSS PRODUCT between the edges of the voxel and the triangle: $\mathbf{a}_{ij} = \mathbf{e}_i \times \mathbf{f}_j, \forall i, j \in \{0,1,2\}$. A detailed explanation of the procedure is given at [1,13].

As soon as the algorithm detects that there is overlap between a projected triangle-voxel pair in one of the axes, it can be stated that the triangle and the voxel are colliding, thus, the algorithm ends and the voxel is marked as surface-voxel. A triangle-voxel pair is not colliding if and only if all the tests in the 13 axes return “no overlap”.

Layering

After having traversed all the objects detecting each surface-voxel, the inner and outer parts of the model are recognized on the *voxmap* using the *Flood-fill* algorithm extended to the third dimension³. Afterwards, layers are added to the voxelized surface—the number of layers is a parameter chosen in the function-call—: the first outer layer is formed by voxels with value -1, whereas the first inner layer is created with voxels of value 1. The absolute voxel-value increases linearly according to the number of layer away from the surface.

4. POINTSHELL GENERATION

The *pointshell* of a virtual model is obtained from its previously generated *voxmap*, projecting on the polygonal objects the surface-voxel centers. Similarly to how it was

³ The algorithm works with a LIFO stack that speeds up the filling process. It starts at a corner-voxel of the virtual model.

done in the previous section, the algorithm traverses each object and triangle, and for each triangle the surface-voxels are regarded. Having a surface-voxel and its triangle, two algorithms are compared in order to determine the best approach.

The first method projects the center of the voxel on the plane of the triangle obtaining P' ; subsequently, P' is projected on the boundaries of the triangle, taking into account in which Voronoi region of the triangle the former P' is located. This last step yields P'' , which is the targeted point.

The second method consists in solving an optimization problem. This approach is the finally chosen one, given that its implementation requires on average 37 % of the time spent by the first algorithm, using arbitrary triangles and points. In the following lines the optimization approach is explained.

A triangle \mathbf{T} defined by its vertices $v_i \in \mathfrak{R}^3, \forall i \in \{0,1,2\}$, can be described using the scalar variables s, t and the vectors $\mathbf{B} = v_0$, $\mathbf{E}_0 = v_1 - v_0$ and $\mathbf{E}_1 = v_2 - v_0$ [4], as follows:

$$\begin{aligned} \mathbf{T}(s,t) &= \mathbf{B} + s\mathbf{E}_0 + t\mathbf{E}_1 \\ (s,t) \in D &= \{(s,t) : s \in [0,1], t \in [0,1], s+t \leq 1\}. \end{aligned}$$

Equation 1

Given a point $P \in \mathfrak{R}^3$, the closest point on the triangle to P can be obtained by minimizing the square distance function between P and $\mathbf{T}(s,t)$: $Q(s,t) = \|\mathbf{T}(s,t) - P\|^2$.

The statement of the optimization problem yields

$$\begin{aligned} \min Q(s,t) &= as^2 + 2bst + ct^2 + 2ds + 2et + f \\ \text{subjected to} &\begin{cases} g_1(s,t) = s \geq 0 \\ g_2(s,t) = t \geq 0 \\ g_3(s,t) = 1 - s - t \geq 0, \end{cases} \end{aligned}$$

Equation 2

where $a \dots f$ are easily obtainable scalar parameters⁴. The Karush-Kuhn-Tucker conditions applied to Equation 2 yield the following nonlinear system with five equations

⁴ $a = \mathbf{E}_0 \cdot \mathbf{E}_0, b = \mathbf{E}_0 \cdot \mathbf{E}_1, c = \mathbf{E}_1 \cdot \mathbf{E}_1, d = \mathbf{E}_0 \cdot (\mathbf{B} - P), e = \mathbf{E}_1 \cdot (\mathbf{B} - P), f = (\mathbf{B} - P) \cdot (\mathbf{B} - P)$.

and five unknowns —variables s, t and the Lagrange multipliers $\lambda_j, \forall j \in \{1, 2, 3\}$ —:

$$\left. \begin{array}{l} \nabla Q(s, t) - \sum_j \lambda_j \nabla g_j(s, t) = 0 \\ \lambda_j g_j = 0 \end{array} \right\} \forall j \in \{1, 2, 3\}.$$

Equation 3

This system in Equation 3 generates a set of seven solutions. One of them denotes a point that is inside the triangle, whereas the other six represent points which are located on the boundary of the triangle, being each one linked to one of the six Voronoi regions of the triangle. The discrimination between the possible seven solutions is done regarding whether the values of the variables fulfill their requirements: $(s, t) \in D, \lambda_j \geq 0$.

The *pointshell* generation is performed in two phases. In the first one, all the projected points are computed using the optimization method, but only the projected points that lie inside the triangle are stored into the structure of the *pointshell*, marking the surface-voxels which they belong to as “projected”. The rest of the boundary-points are stored separately, indexing also the voxel from which they were obtained.

In the second phase all the boundary-points are traversed looking whether their origin-voxel was already projected. In case it was not, the boundary-point is added to the *pointshell*, marking the origin-voxel as “projected”.

In the performed experiments only 0.06 - 1.3 % of the boundary-points were recovered in the second phase, but considering them is essential to avoid possible gaps in the point-cloud.

After generating each *pointshell*-point, the next step is computing the normal associated to each node, because the normal vectors generated by CAD programs point sometimes in the wrong direction. The normal-vectors are obtained analyzing the neighborhood of the *pointshell*-point in the *voxmap* — a region formed by 5 x 5 x 5 voxels—. Setting the origin in the voxel where the point is located, all the vectors that go from this origin to each neighbor voxel-center are summed premultiplied by their voxel-value —recall that the *voxmap* has inwards and outwards layers, with positive and negative voxel-values,

respectively—. The normal of the *pointshell* is the normalized value of this vector.

5. RESULTS

This section contains the generation results performed on a computer with an Intel® D processor at 3.4 GHz, with Linux openSuse 10.2 operating system and 2 GB of memory. The models that were used in the tests are shown on Figure 4.

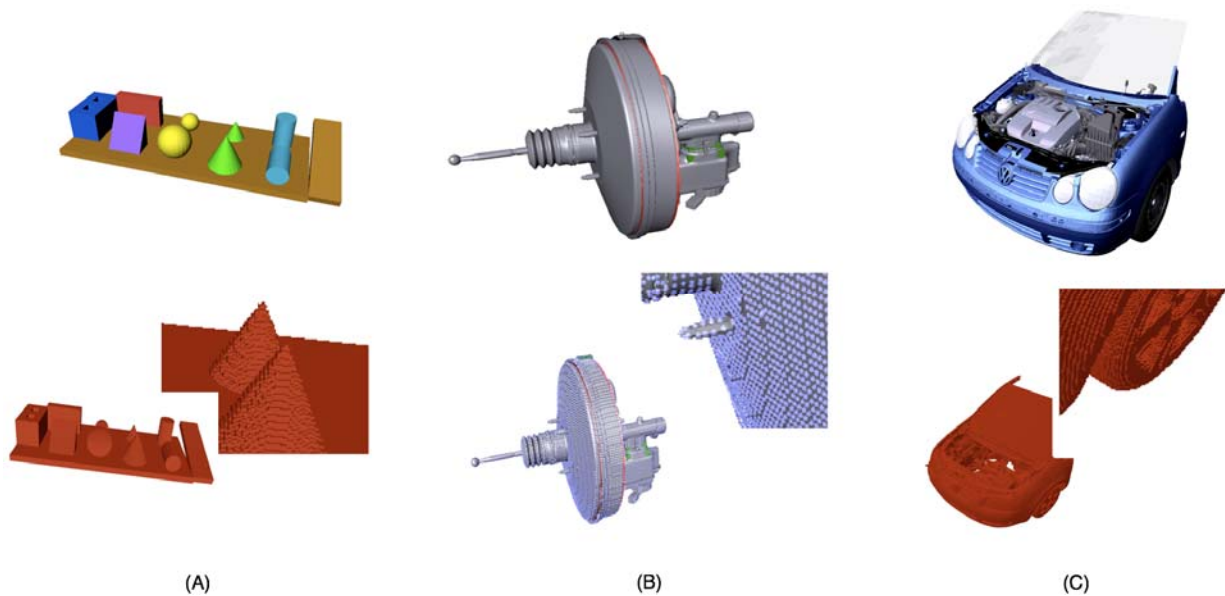


Figure 4: Models used for the evaluation. *Top: polygonal models; bottom: haptic data-structures (voxmap, pointshell) with different resolutions (voxel-size, s).* (A) Artificial model: 7,088 triangles, $s=2.0$; (B) Brake: 24,251 triangles, $s=4.0$; (C) VW Polo: 3,024,231 triangles, $s=5.0$.

Quality

Figure 5 shows the quality improvements obtained with the new algorithms. Concerning the *voxmap*, the old voxelizer that used SOLID library generated too few or too many voxels, according to several tuning parameters.

The lack of surface-voxels yielded holes that made impossible the filling task mentioned in Section 3. Due to this, the layering could not be performed, and the normal vectors of the *pointshell* could not be computed. The excess of voxels degenerated in cases like the one shown in the picture, were surface-voxels appeared where there is free space. All these problems disappear with the new voxelizing algorithm, obtaining accurate discrete representations.

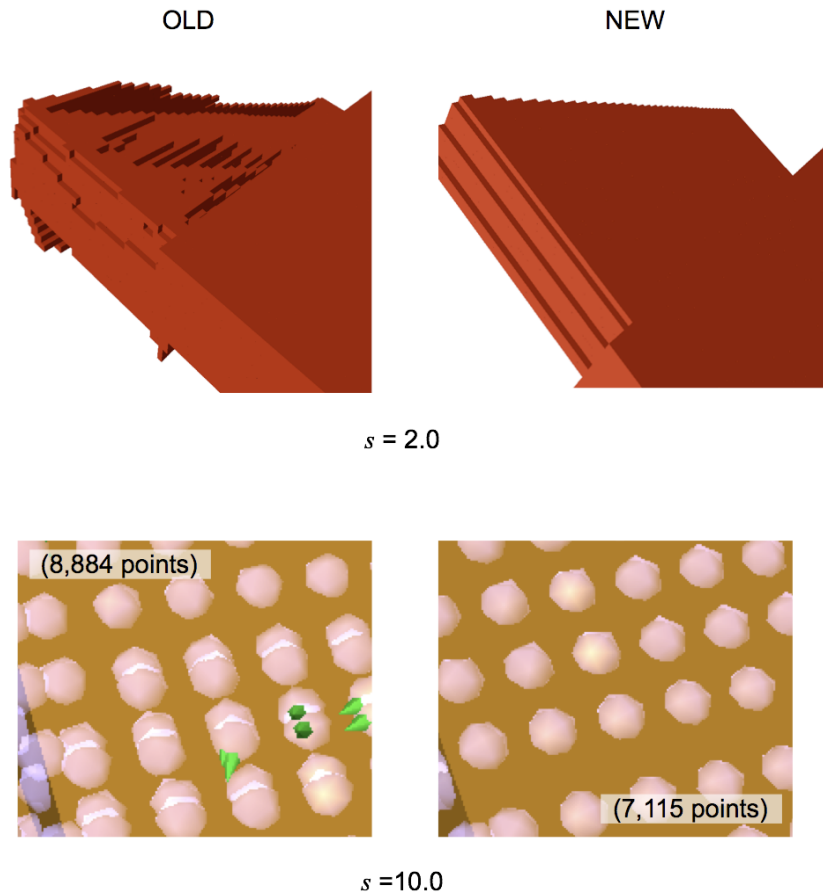


Figure 5: Quality enhancements achieved on the *voxmap* and the *pointshell* using the new algorithms.

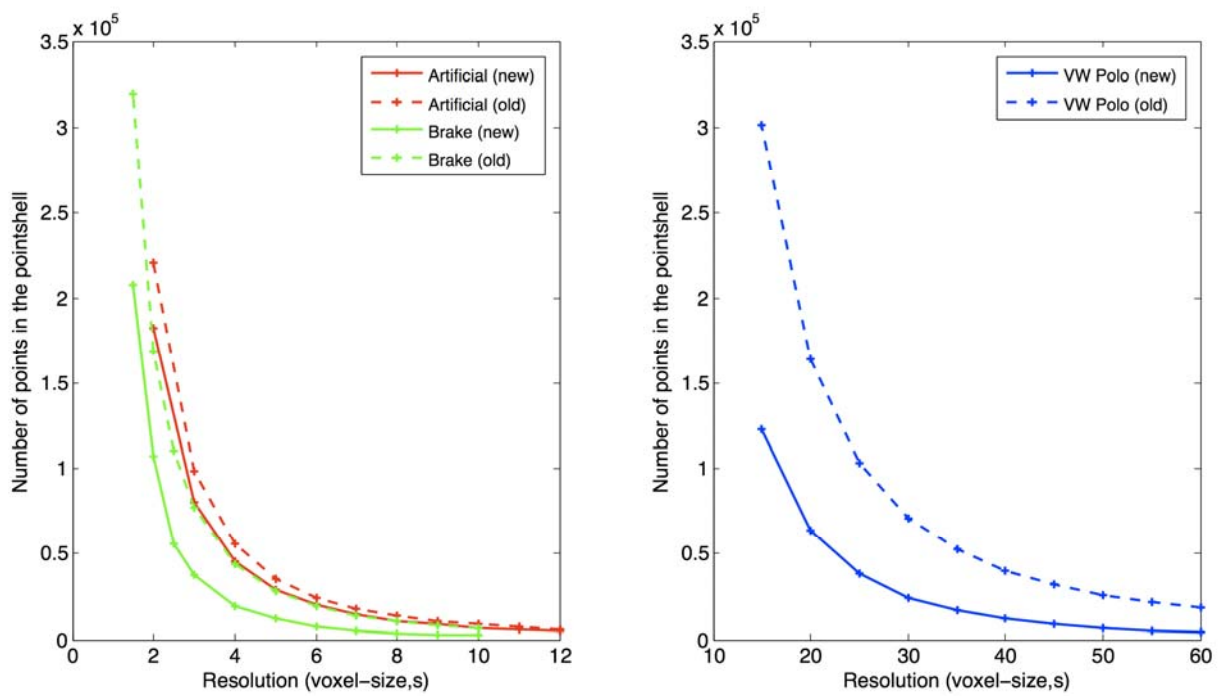


Figure 6: Number of *pointshell*-points obtained with the old and the new algorithms.

In the case of the *pointshell*, the new algorithm avoids generating superfluous points that appeared with the old versions. In fact, the aim of the *pointshell* is to describe a model for a given resolution (voxel-edge size, s) uniformly and with the fewest possible points; in this way, the new algorithm is able to describe a model with higher resolutions for a given amount of points. The chart on Figure 6 shows how the dashed curves that represent the number of points generated with the old algorithms against the resolution lie always above the solid ones, which are related to the new algorithms.

Performance

Figure 7 shows performance increase ratios. These ratios were calculated dividing the generation times of the old algorithms by the generation times of the new ones. It can be seen that the *voxmap* generation is from 2 to 52 times faster than in the old process, depending on the model and the resolution of the haptic data-structure. This gain in performance increases with the resolution. Table 1 contains some simulation results, in order to show the absolute time which is required. There, additional time records are shown in the case of the *voxmap* generation, such as the loading time —changing in memory the polygonal models— and the initialization time —arranging the empty *voxmap* grid after computing the bounding box of the model—.

Notice that in the case of the voxelization of the VW Polo (case (e)), the new algorithm is able to obtain a *voxmap* of resolution $s=2.0$ in less than eight minutes, whereas the old version required almost three hours. The explained second approach turns out to be faster only from certain high resolutions on —approximately, artificial model: 1.98×10^8 voxels; brake: 6.1×10^7 voxels; VW Polo: 4.66×10^7 voxels—. Further tests with different models can lead to a condition that decides to use the first or the second approach according to parameters like the number of triangles and the resolution —number of voxels in the *voxmap*—.

In the case of the *pointshell*, the speed increase factor ranges from 1.8 to 19, depending on the models and the resolutions that are used.

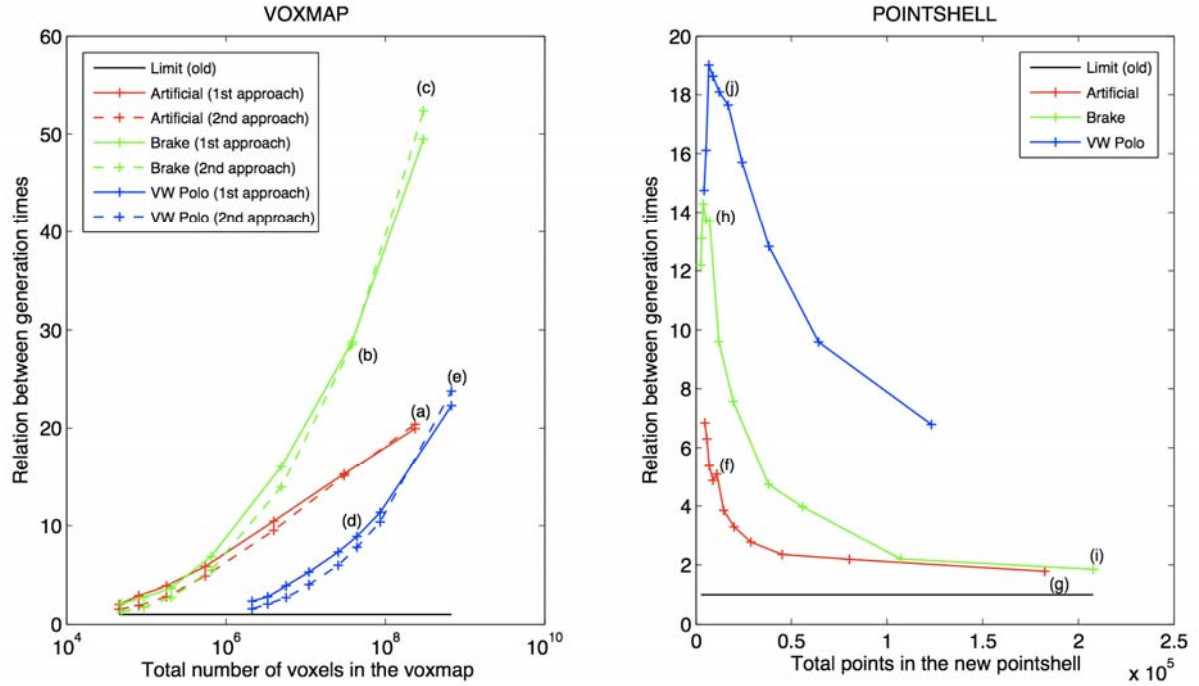


Figure 7: Increase in performance comparing the time required by the new and the old algorithms to generate the haptic data-structures (Table 1: t_{old}/t_{new}). Left: voxmap generation; right: pointshell generation. Examples from Table 1 are shown with letters.

VOXMAP								
Model	s	Voxels	Load	Initialize	Generation		Filling	$\Delta = t_{old}/t_{new}$
					t_{old}	t_{new}		
–	–	#	sec.	sec.	sec.	sec.	sec.	–
Artificial (a)	0.5	237,717,488	0.1190	1.1856	338.24	16.562	2.3423	20.42
Brake (b)	1.0	38,524,026	0.3954	0.1919	104.1	3.614	0.4994	28.8
Brake (c)	0.5	302,685,600		1.5112	821.5	15.684	3.7512	52.4
VW Polo (d)	5.0	44,311,554	18.972	0.2290	614.7	69.179	0.6992	8.9
VW Polo (e)	2.0	680,652,000		3.6188	9860.0	442.29	9.9486	22.3
POINTSHELL								
Model	s	Points (new)	t_{old}	t_{new}	$\Delta = t_{old}/t_{new}$			
–	–	#	sec.	sec.	–			
Artificial (f)	8.0	10,954	3.215	0.6319	5.09			
Artificial (g)	2.0	182,464	244.75	137.412	1.78			
Brake (h)	6.0	7,287	6.107	0.4465	13.68			
Brake (i)	1.5	207,490	559.9	302.974	1.85			
VW Polo (j)	40.0	12,161	203.2	11.223	18.11			

Table 1: Examples containing time values obtained with old and new algorithms. Each case is marked with a letter; the chart of Figure 7 displays all the performance increases of the marked cases.

Remark 1: Given that the required time depends on the computer, this work focuses on relative values that are not affected by the computer which is used. The absolute values in Table 1 give a practical impression of the speed.

Remark 2: Cases (f), (h) and (j) from Table 1 correspond to *pointshells* with around 8,000 points. This amount is the approximate number of points for which collision detection and force computation can be performed within 1 ms running the VPS algorithm with the used computer.

Remark 3: An example that is comparable case (e) is given in [10]. There, a landing gear model with 2.76×10^6 triangles is used to generate a *voxmap* of 4.59×10^8 voxels. The required time using a two processor 2.8 GHz Xeon PC with 2 GB of RAM running Windows XP is of 1,353 seconds, compared to the 442.29 seconds required by voxelization algorithm explained here —running on the computer specified at the beginning of this section.

6. CONCLUSION AND FUTURE WORK

Very fast algorithms for generating *voxmaps* and *pointshells*, the haptic data-structures of the Voxmap-PointShell™ (VPS) Algorithm, were presented in this work.

The voxelization algorithm navigates in the bounding box of each triangle of the original polygonal model detecting the probable surface-voxels, and performs collision tests to these candidate voxels based on the *Separating Axis Theorem*. In comparison to the old algorithms implemented at DLR, the obtained discrete representations show an improved quality —no holes or excessive surface-voxels— and faster generation speed —with increase factors between 2 to 52, depending on the model and the resolution—.

The *pointshell* generator projects the surface-voxel centers on the original polygonal model based on a nonlinear optimization method that finds the closest points on the surface to the voxel-centers. The new *pointshells* are described without the redundant points that appeared in the old versions, being possible to work with higher resolutions.

Besides of that, the generation speed is between 1.8 and 19 times faster than the old one, depending on the resolution and the model.

The quality improvements are significantly important, given that the original VPS algorithm strongly relies on its data-structures to obtain realistic collision responses. The generation speed, although not as critical as the performance of the VPS algorithm itself, is an important feature to build virtual environments almost instantaneously or at least within few minutes, compared to the hours required before for some resolutions.

Regarding data-structure generation, possible future steps involve comparing the current algorithms to others. A candidate voxelization algorithm has already been chosen [6], and possible comparison indicators are listed in [13]. In the case of the *pointshell* generation, the main improvements are focused on increasing the performance of the algorithm, given that the actual methods handle big amounts of boundary points, of which only 0.06 – 1.3 % are taken into the final *pointshell*, investing a considerable amount of time in discarding superfluous points.

Besides of that, implementing hierarchies for *voxmaps* and *pointshells* is very interesting; in the case of the *voxmap* it produces storage improvements [9], whereas hierarchical *pointshells* increase the performance of the VPS itself. In [2] an appealing sphere-point tree is presented, which is able to localize interference areas. Hence, it checks for collision only the points within these areas.

Finally, it is worth to mention the idea of recognizing the shape, both in the *voxmaps* and the *pointshells*, in order to exploit this information to increase the performance of the VPS when using thin or stick-like objects.

Acknowledgements

The authors of this work would like to express their gratitude to Volkswagen for supporting this research topic and for providing the models used to perform the tests.

References:

- [1] Akenine-Möller, T.: *Fast 3D Triangle-Box Overlap Testing*, Journal of graphic tools, vol. 6, nr. 1, pp. 29-33, 2001.
- [2] Barbič, J.; James, D.L.: *Time-critical distributed contact for 6-DoF haptic rendering of adaptively sampled reduced deformable models*, Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation, SESSION: Real-time simulation, pp. 171-180, 2007.

- [3] Cameron, S.: *Enhancing GJK: Computing Minimum and Penetration Distances between Convex Polyhedra*, Int. Conf. Robotics & Automation, 1997.
- [4] Eberly, D.: *Distance between Point and Triangle in 3D*, 1999.
- [5] Gottschalk, S.; Lin, M. C.; Manocha, D.: *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*, Computer Graphics Journal, vol. 30, Annual Conference Series, pp. 171-180, 1996.
- [6] Huang, J.; Yagel R.; Filippov V.; Kurzion Y.: *An Accurate Method for Voxelizeing Polygon Meshes*, IEEE Symposium on Volume Visualization, pp. 119-126, 1998.
- [7] Hulin, T.; Preusche C.; Hirzinger, G.: *Haptic Rendering for Virtual Assembly Verification (Poster)*, WorldHaptics Conference 2005, Pisa - Italy, 2005.
- [8] Lin, M. C.; Canny, J. F.: *A Fast Algorithm for Incremental Distance Calculation*, IEEE International Conference on Robotics and Automation, pp. 1008-1014, 1991.
- [9] McNeely, W.A.; Puterbaugh, K.D.; Troy, J.J.: *Six Degree-of-Freedom Haptic Rendering Using Voxelman Sampling*, Proc. of SIGGRAPH, 1999.
- [10] McNeely, W.A.; Puterbaugh, K.D.; Troy, J.J.: *Voxel-Based 6-DOF Haptic Rendering Improvements*, Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation, pp. 171-180, 2006.
- [11] Ortega, M.; Redon, S.; Coquillart, S.: *A Six Degree-of-Freedom God-Object Method for Haptic Display of Rigid Bodies*, IEEE Transactions on Visualization and Computer Graphics, vol. 13, iss. 3, pp. 458-469, 2007.
- [12] Renz, M.; Preusche, C.; Pötke, M.; Kriegel, H.-P.; Hirzinger, G.: *Stable Haptic Interaction with Virtual Environments Using an Adapted Voxmap-Pointshell Algorithm*, Proc. Of Eurohaptics, 2001.
- [13] Sagardia, M.: *Enhancements of the Voxmap-PointShell Algorithm*, Master's Thesis (DLR-TECNUN), 2008.
- [14] van den Bergen, G.: *Proximity Queries and Penetration Depth Computation on 3D Game Objects*, Proc. Game Developers Conf., 2001.
- [15] Zilles, C.; Salisbury, J.: *A constraint based god-object method for haptic display*, Proceedings of the IEE/RSJ International Conference on Intelligent Robots and Systems, Human Robot Interaction, and Cooperative Robots, 1995.

Authors:

Mikel Sagardia
Thomas Hulin
Carsten Preusche
Prof. Dr.-Ing. Gerd Hirzinger
DLR (German Aerospace Center)
Member of the Helmholtz Association
Institute of Robotics and Mechatronics
Oberpfaffenhofen, D-82234 Wessling
Muenchner Str. 20
Phone: +49 8153 28-1221
Fax: +49 8153 28-1134
E-mail: Mikel.Sagardia@dlr.de