Procedural Fracture of Shell Objects

Jakub Domaradzki
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
J.Domaradzki@stud.elka.pw.edu.pl

Tomasz Martyn
Institute of Computer Science
Warsaw University of Technology
ul. Nowowiejska 15/19
00-665 Warsaw, Poland
martyn@ii.pw.edu.pl

ABSTRACT

We propose a novel algorithm to fracture brittle objects that are characterized by an empty interior and thick surface (which we denote as *shell objects*), such as: vases, pots, pitchers, antique ceramic, etc. Our method augments the previous ones based on fracture patterns and utilizes sparse voxel octrees (SVOs) as a highly efficient and detailed object representation. In our method, the fracture pattern relies on Voronoi diagrams and is calculated on-the-fly. The outcomes of applying the fracture pattern differ from the ones obtained with the previous methods in that it solves the problem of planar faces of the newly generated pieces of geometry, allowing them to have concave shapes. Without any precomputation, we are able to achieve various and interesting fractures that are unique to each destructed object. Finally, our approach is intuitive, adaptable and fast, which makes it a good candidate for applications in computer game industry.

Keywords

sparse voxel octree, Voronoi decomposition, pattern fracturing, procedural surface generation

1 INTRODUCTION

Nowadays various effects of object fracturing are widely present in computer games. Wandering through the game worlds we can usually interact with lots of environment items and in order to increase the realism of "being-in-the-world", we are more often allowed to destruct them. Nevertheless, the players' requirements and expectations pertaining the nowadays gameplays in general and in particular freedom in the player's interactions with the virtual world are continuously rising. Needless to say that the objects populating the game level are usually required not only to possess a capability of being fractured, splintered, crushed, or destroyed in some other way, but also the process of breaking them into pieces should be *unique* for each instance of a destructible geometrical asset and for the same instance—with every restart of the game.

Therefore the approaches based on a predefined destruction of geometrical assets and successfully utilized in games of the previous generations are now slowly being replaced with more and more sophisticated techniques, in which some of the calculations are performed on-the-fly during the gameplay itself. Due to the constant increase of computer power, especially graphics cards, we are now very often able to perform all calculations underlying the object destruction in real-time. On the other hand, the amount of details of the geometry that constitutes the scenes in modern games is also still increased, which makes the task of performing such a unique destruction of objects directly during gameplay even more challenging.

Thanks to the development of new voxel-based techniques in a few recent years, it is now possible to utilize voxels as a representation of solid objects in real-time systems. Memory usage, one of the main problems associated with voxels, has been drastically reduced mainly due to taking advantage of the sparse voxel octree structure (SVO) [Cra11]. This way a genuine, highly detailed geometry with many levels of detail can be easily accessed.

In this paper, we show how to enrich with details and make unique the process of destruction when applied to shell objects, which are featured by an empty interior and a thick surface like vases, pots, pitchers, antique ceramic, etc. Our approach augments the previous ones based on fracture patterns and benefits from the SVO representation. The outcomes of our method stand out from those obtained with the related solutions in that the pieces the destroyed object is falling apart to can be concave in shape and, moreover, the geometry of the pieces' surface is not limited to planar facets. And what is most important, all that we achieve in time sufficient for computer game applications.

2 RELATED WORK

There is a wide selection of literature on fracturing brittle objects. Over the years, many methods have been developed. The early works in the animation of fracture used connected point-masses and the fracturing process was performed by the removal of some of the links. This method was introduced by Terzopoulos et al. [TF88] and Norton et al.[NTB*91], who pioneered this area of study in computer graphics. Afterwards, the approach was improved by O'Brien and Hodgins, who followed them and focused on simulating brittle [OH99] and ductile fracturing [OBH02]. They used a fully dynamic finite element method (FEM), in order to compute the internal stress in the object being destructed. However, this approach requires very small time steps and costly cutting mesh operations [WRK*10]. To avoid this issues, the fracture simulation tends to be formulated as a quasi-static stress analysis [GMD13][ZBG15].

On the other hand, the real-time systems, especially computer games, utilize a more practical approach which is based on a predefined partition of the destructible object into pieces, which replace the original object while it is destroyed. While this method is fast and gives the designer a full control over the shape and location of fractures, its results are often a far cry from realism. In order to improve the visual quality of this approach and, at the same time, to stay within the limits imposed by real-time applications, the geometry-based methods were introduced [SSF09][BCC*11][MCK13]. Although the methods also utilize a predefined fracture pattern, the pattern is separated from the actual geometry and is applied, oriented and scaled on demand at the impact location. That way the observer gets the impression that the fracturing is unique for each destruction. The common technique to generate the fracture patterns consists in constructing a 3D space decomposition based on the Voronoi diagram or by means of a simulation [IO09]. The further development in this direction resulted in generating the fracture pattern on-the-fly while a destructive impulse occurs [SO14][DM16].

Although the current pattern-based approaches usually produce good results, there is still room for improvement. From our standpoint, there are two essential problems, which we regard as challenges and would like to address, namely: the non-planar surfaces of the fractured pieces and their concave shape. One should note that both problems have already been referenced in [SO14], however, the solution presented there requires a pre-computation step and the calculations are performed on a rather fine tetrahedral mesh. The method we present in this paper follows Domaradzki's work et al. [DM16] and takes a different approach to tackle the mentioned issues. Looking for a predecessor of the general conception our method is founded on, one can point out a paper by Chen [CYFW14], in which visual details were added to coarse simulation results in a post-process step.

What is more, we strongly believe that the solution to the addressed problems can be found, quite naturally, in the voxel-based representation. Currently, the popular data structure for voxels to represent boundaries of solid objects is the sparse voxel octree (SVO) [Cra11]). Thanks to the computational power offered by today's GPUs, along with new GPU-specialized programming techniques newly developed algorithms designed for sparse voxel octrees are ready for real-time applications.

Although the SVO representation is not yet well established in the commercial real-time graphics software, such as professional game engine being still dominated by triangle meshes, it has already proven its usefulness in a number of fields. To begin with, Cyril Crassin was able to visualize global illumination in real time using voxel cone tracing [CNS*11]. Following this work, Laine developed an efficient SVO ray-tracing algorithm [LK10], proving that way that ray-casting the SVO can be done faster than when using triangle meshes. Furthermore, the search for even more compact scene representations led to the development of sparse voxel directed acyclic graphs (SVDAG [KSA13]), which have been then improved to Symmetry-Aware SVDAGs [VMG16], that reduce memory footprint even further. There are also other ways to represent the SVO-based geometry effectively, to mention only a method of unlimited object instancing presented by Jabłoński et al. [JM17], that can be combined with a continuous and symmetrical LOD transition [JM16].

Apart from visualization itself, there has also been a development in other graphics areas including object animation [Bau11], deformation [Wil13], and fracturing in real-time [DM16]. Last, but not least, octrees can be built very fast using some recent techniques presented in [ZGHG11][GPM11][Kar12] and even by means of the out-of-core approaches described in [BLD14][PK15], which utilize only a fraction of memory required to store a model.

3 SVO FRACTURING

In this section, we outline our algorithm for fracturing SVO objects. We especially target shell objects that are empty inside and are formed by thick surfaces, such as vases, pitchers, pots, antique ceramics, etc. Our goal is to cut them into pieces with a slightly and locally disturbed fracture pattern that is calculated on-the-fly. The algorithm consists of two separate parts that combined together produce a final result. The first part, we call *Basic Fracture Algorithm* (or BFA—Sec. 4), is based

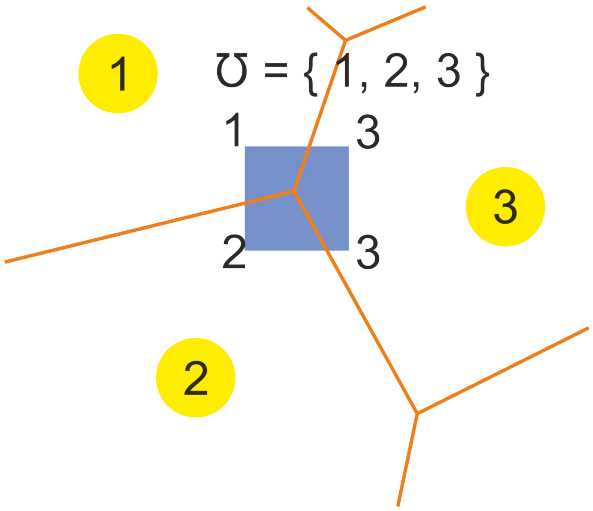


Figure 1: A voxel-pattern intersection test with three Voronoi seeds.

on the previous work of Domaradzki et al. [DM16]. Its underlying idea is to divide the SVO object into convex pieces by means of a fracture pattern consisting of planar faces, which determine the slicing areas used to partition the object. The outcome is then processed in the second part, we call *Enhanced Fracture Algorithm* (or EFA—Sec. 5), in which the cuts are additionally deformed by means of a local procedural surface creation. As a consequence, we enhance the previous Domaradzki's algorithm in that the final cuts are less regular and more intricate geometrically and, thus, they render a more natural fracture.

4 BASIC FRACTURE ALGORITHM

The goal of BFA is to determine the subsets of SVO voxels that represent the surfaces of the particular fractured pieces at the accuracy of the SVO highest level. To this end, the SVO is traversed from the root to the leaves, and at each SVO level, the intersections of voxels with the pattern faces are tested. Next, if a face-voxel intersection is detected, the voxel is either expanded to its eight children or, in the case the voxel belongs to the object's interior (i.e. it is not an element of the surface), the children are dynamically created. The resulting subsets are composed of the voxels intersected by the fracture pattern faces at the highest SVO level and, of course, the original leaf voxels of the SVO object surface. A voxel is assigned to a given subset on the basis of the closest Voronoi seed to the voxel's location (Sec. 4.1). Then, for each subset, we build a SVO founded on the subset's voxels treated as the SVO leaves. A more detailed description of the algorithm can be found in [DM16].

4.1 Fracture Pattern

The fracture pattern used in this algorithm is represented by a finite set of 3D points, which represent the seeds of a Voronoi diagram. In order to achieve a more realistic fracture, we want it to concentrate around the impact location. This can be obtained by aligning the center of an existing fracture pattern with the impact location as presented in [SSF09] and [MCK13]. However, following work in [DM16], we also create the fracture pattern on-the-fly. In this goal, we generate the Voronoi seeds at random on a set of spheres of growing radii, which are centered at the point of impact. Having this set of seeds it is not required to determine the faces of the Voronoi diagram for the voxel-pattern intersection test, which can be performed directly based on the set as follows:

Let $S = \{1,\ldots,n\}$ be the set of the indices of the Voronoi seeds $\{s_i\}_{i\in S}$. Define a function $\gamma : \mathbb{R}^3 \to 2^S$ such that

$$\gamma(x) = \{k \in S : \|s_k - x\| = \min_{i\in S} \|s_i - x\|\}. \qquad (1)$$

Given a point $x \in \mathbb{R}^3$, the function γ returns the set of the indices of the Voronoi cells that include x. (Note that the resulting set is not a singleton, if x is located on a face, an edge, or a vertex of the Voronoi diagram).

The voxel-pattern intersection test can be done by means of the following function:

$$\Ʊ(V) = \bigcup_{v\in V} \gamma(v). \qquad (2)$$

which, given a voxel specified by the set $V = \{v_i\}_{i=1,\ldots,8}$ of its vertices, maps the voxel into the set of the indices of the Voronoi cells the vertices are situated in.

It is easy to see that the voxel is intersected by a face of the fracture pattern if and only if the set of indices given by $\Ʊ(V)$ is not a singleton (fig. 1).

5 ENHANCED FRACTURE ALGORITHM

In the second step, we improve the result obtained by BFA to give the fracture a more realistic appearance. In this purpose, we must face the two main deficiencies of the previous algorithm: the lack of concave fractured pieces and their totally planar surfaces.

5.1 Distance Metric Approach

To begin with, there is a solution that might be used to produce outcomes satisfying our needs. However, it is expensive in memory and computation. The approach is based on influencing the distance metric that is used in the calculations in the voxel-pattern intersection test (sec. 4.1). The distance metric can be altered in two ways.

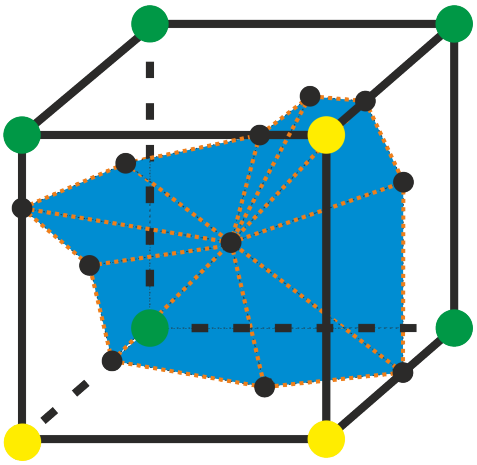


Figure 2: An example of a crack surface represented by a set of triangles, created within a voxel on the SVO transition level.

First, one can deform the fracture pattern as in [M05]. Following that, we would have to use a 3D noise texture and sample it along the ray from a voxel corner to a Voronoi seed, accumulating the result and treat it as a distance. Such an approach would take a lot of memory to store the texture and load operations to sample it with the 3D interpolation to receive a smooth outcome.

Another method, presented in [SO14], first applies a deformation to the object and then the fracturing process operates on the deformed object. Finally, the reverse deformation is applied to the result, producing deformed pieces of fractured geometry. Although this method is faster than the previous one, it encounters difficulties with the hierarchical representation of the destructed object. What is more, it would have to be performed on-the-fly, as building the new deformed SVO object comes with the loss of information (as presented in [Wil13]).

5.2 Procedural Crack Surface Creation

Taking into consideration challenges presented in the previous section, we propose another solution. Following BFA (sec. 4), we traverse the SVO from the root to the leaves. On each SVO level we get a set of voxels containing the information whether a voxel is intersected by a face of the fracture pattern and, if so, which Voronoi seed is the nearest to the voxel's vertices. As a result, if we stop BFA on any intermediate SVO level, then the outcome can be viewed as a partition of the SVO object with an approximation of the Voronoi diagram. Depending on the level we stop, the size of voxels differs and so the volumes of the pieces the voxels make up. The volumes may be treated as subspaces, within which *crack surfaces* can be procedurally generated. The crack surfaces will be used in the final step of the algorithm to enrich the original planar and convex fracture geometry provided by BFA. The SVO level at which we stop the algorithm and construct the crack surfaces we call the *transition level*.

There are two main problems that we need to address. First, how to generate the geometry of these crack surfaces with the voxels delivered by BFA. Secondly, we aim at a GPU-based parallel implementation that constructs the crack surfaces concurrently in the voxels. Therefore the algorithm is to operate locally within a voxel, and this implies the problem of connectivity of the surfaces between neighboring voxels.

We tackle the second problem by founding the crack surface construction on an information shared by vertices of neighboring voxels. As the aftermath of BFA, each vertex possesses the information about its nearest Voronoi seed, which we now additionally enrich with a procedurally generated value, for example, obtained from a 3D noise function.

The surface we want to create should separate the voxel's vertices that have assigned different Voronoi seeds and thereby belong to different fracture pieces of the objects. To this end, for each voxel's edge that connects vertices having different Voronoi seeds, the *edge separation point* is created. The position of this point is determined by adding to one of the edge's endpoints an offset computed with the following equation:

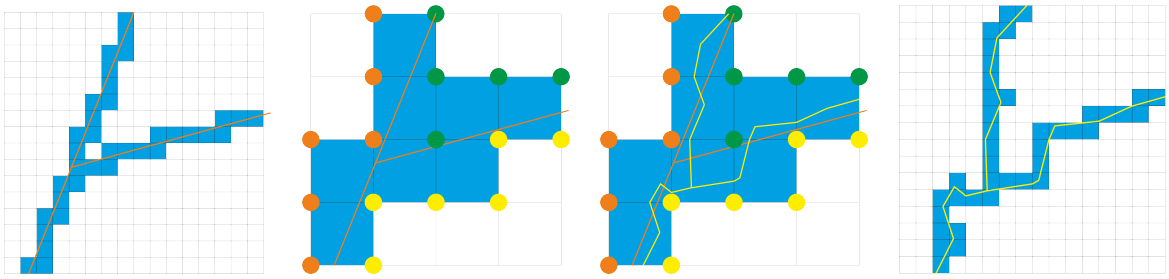
$$eo_n = f(a,b) * v_s, \tag{3}$$

where: eo_n is the edge offset on $n \in \{X,Y,Z\}$ axis; $a,b \in [0,1]$—values assigned to the voxel's vertices; f is a user function returning a value in range (0, 1) based on the values a and b; and v_s is the size of the voxel edge. The user function can be implemented, for example, as a balance point based on the given weights or a deviation from the intersection point with the Voronoi pattern steered by the given weights.

Subsequently, for each voxel's face, the *face separation point* is calculated as the average of the *edge separation points* belonging to this face, and then the point is displaced by a slight offset within the face's area. Next, the *voxel separation point* is determined from *face separation points* in an analogous way. Finally, for each tuple of three points: *edge separation point*, *face separation point* and *voxel separation point* a triangle is created.

The set of the triangles forms a portion of the crack surface that will then be approximated by descendant voxels of the processed transition voxel (fig. 2).

The presented construction of continuous crack surfaces within voxels is only exemplary and one can develop different methods for this purpose. For example, one can base the construction of the surface on mathematical functions.



(a) (b) (c) (d)

Figure 3: (a) A result of BFA on the SVO highest level. (b) A result of BFA on the SVO transition level. (c) A new crack surface constructed by EFA relative to the BFA result. (d) The final voxelized crack obtained with EFA.

5.3 SVO Traversal

Putting all together, we can describe the overall method for fracturing a SVO object as follows:

We begin with BFA, which traverses the SVO tree from the root to the leaves: voxels on each SVO level are being tested for intersections with faces of the fracture pattern (sec. 4), and the ones that pass the test, are expanded to their children (which are generated on-the-fly in the case of the intersected voxels located inside of the object). This way we proceed over a chosen number of the SVO levels to the desired transition level. At this level, we assign the nearest Voronoi seeds to the voxels' vertices, and EFA begins.

From now on, the final surface of fractured pieces is being created using the crack surfaces. We generate the crack surfaces within the voxels at the transition level in the way described in the section 5.2. Then, for every voxel that gave birth to a crack surface, each child of the voxel is tested against an intersection with the surface by means of the method described in [AA05]. That way we proceed to the SVO leaves, where each voxel is assigned to an appropriate group of voxels which defines a fractured piece, using the same rules as in BFA (fig. 3). The final voxels properties comes from a volumetric texture for color and normal vector taken from the triangles the voxel intersects.

In order to fully define a fractured piece, apart from the voxels intersected by the crack surfaces, we also need to identify the appropriate voxels that come from the original surface of the destructed object. In order to assign these voxels to the proper Voronoi seed (and thus the group specifying a fractured piece), while executing EFA we utilize the following test:

While checking a voxel for intersections with the triangles of a crack surface, we also check on which side of each triangle the voxel is located and accumulate this information. By the construction of the crack surface (Sec. 5.2), each triangle has a vertex located on an edge of a transition level voxel, such that the edge ends with the voxel's vertices assigned to two different Voronoi seeds. On that basis, we assign the appropriate Voronoi seed to each side of the triangle. Using this information, we can assign a surface voxel (not intersected by a crack surface) to the appropriate Voronoi seed on the basis of the voxel's location relative to the sides of the relevant crack surface's triangles—we choose the seed that dominates in the sides of the triangles, as shown in fig. 4.

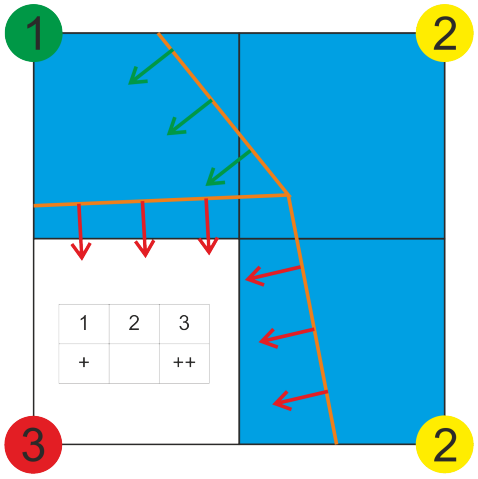


Figure 4: The assignment test for a voxel (marked with white) that doesn't intersect a crack surface.

The method presented above features a lossless fracturing process. It means that the surfaces of cracks match perfectly each other. In the real world, however, there are also materials bearing different properties, which result in less stable cracks. During fracturing process, apart from a separation of an object into a number of pieces, a part of the object's volume is converted into separate tiny dust particles of an individual size much smaller than the volume of the SVO leaf voxel. The formation of these dust particles during fracturing process results in irregular empty volumes between crack surfaces (fig. 5). We can easily incorporate this effect into our method by changing the crack surface creation algorithm presented in Sec. 5.2 to a more suitable one. For this purpose, we utilize the surface creation technique used in the Marching Cubes algorithm [LC87]. Specifically, using the information of the assignment of the transition voxels' vertices to Voronoi seeds, we match the voxel to one of the cube configurations used

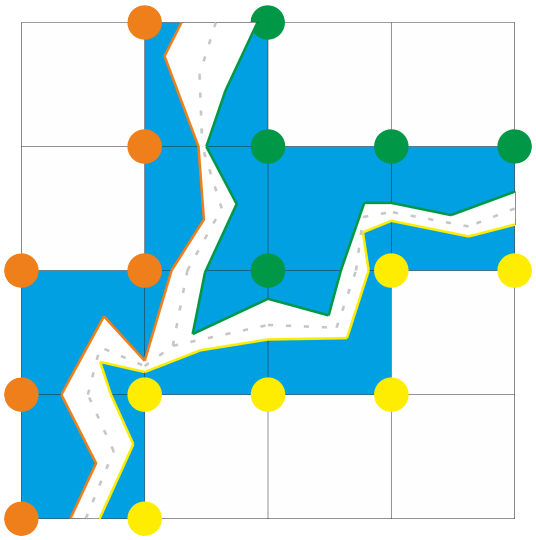


Figure 5: An empty volume between cracks' surfaces.

in the Marching Cubes algorithm. Moreover, we need to split the *edge separation points* and shift them in order to separate new surfaces.

6 RESULTS

In this section, we discuss the performance and quality of the presented solution. All depicted timings were obtained on Intel Core i7 960 CPU with Nvidia GeForce GTX Titan Black GPU. All algorithms were implemented using CUDA framework to fully exploit the parallelism delivered by GPU.

The presented results show that our new fracturing technique greatly improves on the visual quality of fractured objects relative to the relevant methods that utilize fracture patterns. A direct comparison with work in [DM16] is shown in fig. 6. One can notice that thanks to the distortions in the planar fracture pattern faces, the result looks more natural. What is more, the outcomes of our technique are always unique as they are created on-the-fly while a destruction occurs.

Furthermore, our method gives the user a simple way to influence the fracturing process outcome by controlling the level at which the transition between BFA and EFA takes place. Studying the pictures in fig. 7 reveals the relationship between this parameter value and the final result. The closer the transition level to the root, the more the basic fracture pattern (build from the Voronoi seeds) is distorted. The change in the transition level, when regarded from the point of view of the procedural creation of crack surfaces, also has an influence on the smoothness of the cracks.

We also presented a way to incorporate in the fracturing process a decline in the original object's volume within the area of cracks—fig. 9 exemplifies the feature. The variety of the cracks' possible shapes and widths we can obtain with our method greatly enhances the visual quality of results and allows one to produce countless of unique outcomes. In addition, the cracks are not only more detailed but also more noticeable.

We also succeeded in the integration of the results of our fracturing method with a physics simulation technique presented in the paper [DM16]. The performance of the approach presented there depends strongly on the volume of the objects subjected to simulation. Even though the fractured pieces generated with our method are usually relatively thin, the results of the physics simulation are acceptable (fig. 8).

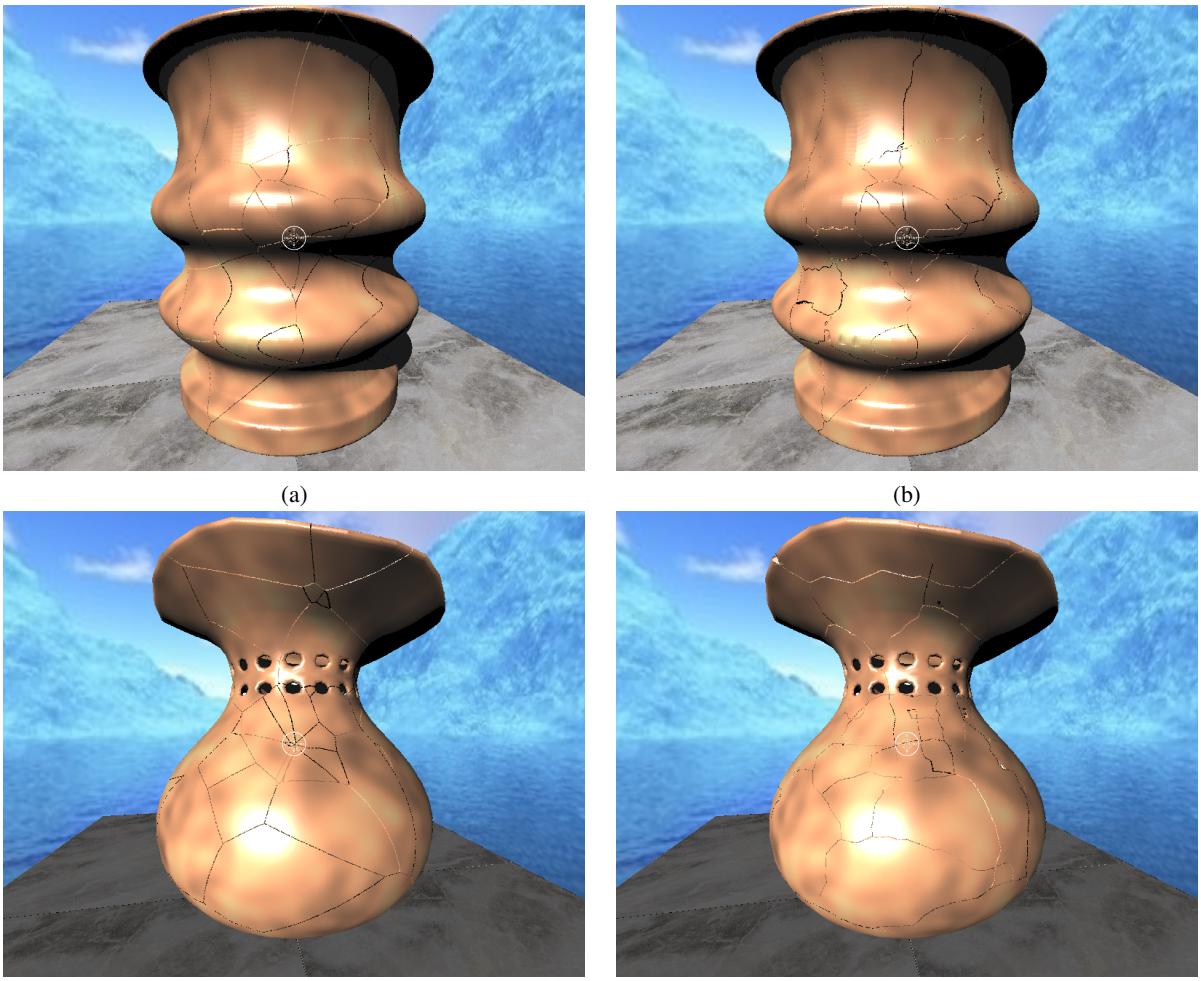
Last but not least, the time performance of our method, as measured per voxel, is slower than in the one featuring the method in [DM16]. First, it is mainly due to that our technique produces more voxels in the final outcome and, secondly, the test for voxel intersection with a procedurally generated crack surface is more computationally demanding. It should be also noted that the performance of all the algorithms used in the fracturing process (fracture boundary set extraction, islands detection, internal nodes detection, etc.—details in [DM16])) depends directly on the total number of voxels, which also influence the timing of the whole process. However, in our tests, fracturing the SVOs with the number of levels up to 9 (included), resulted in the timings not exceeding 50 ms.

7 CONCLUSION AND FUTURE WORK

We have presented a novel method for fracturing shell objects represented with sparse voxel octrees. Our method allows for creating detailed surfaces of fractured pieces. To this end, it applies a fracture pattern to the object at the impact location and cuts the object with the pattern into pieces, and then enhances this partial result by creating the final surfaces of the pieces procedurally. As a consequence, new detailed pieces of geometry are created and represented as individual SVOs, which can then be subjected to a rigid body simulation using the approach presented in [DM16].

Although in this paper, we target only the objects that are empty inside but possesses thick surfaces, we strongly believe that our method could also be applied to objects with noticeable inner volumes. In that case, it would require either the change of the method used to procedurally create new surfaces or the application of more suitable weights for the current method for voxels' vertices on the transition level, so that the method would be aware of the voxels' boundaries in a local neighborhood. It is due to the fact that the current method generates cracks without any global structure, which is desirable for objects' surfaces but not necessarily for objects' interior.

Finally, the presented fracturing method operates on voxels, however, with some changes, it could also be



(a) (b)

(c) (d)

Figure 6: A comparison of results obtained by our fracturing technique (fig. b and d) and the method presented in [DM16] (fig. a and c).

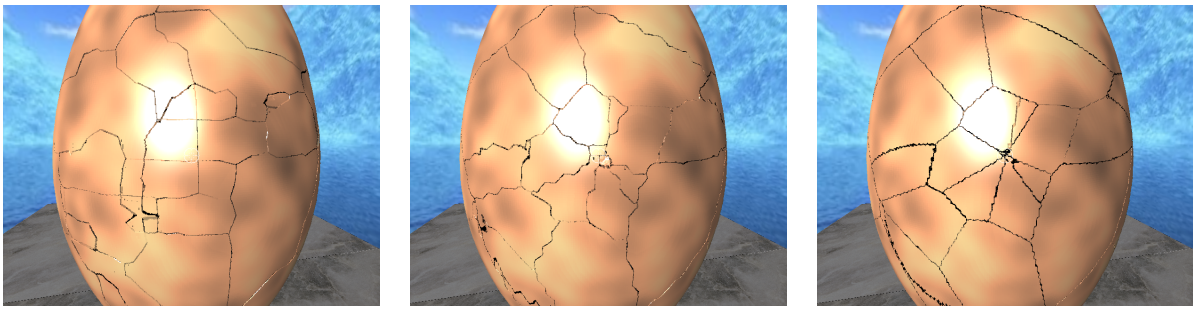


Figure 7: Different transition levels (from the left accordingly: 4, 6, 8) and the same fracture pattern.

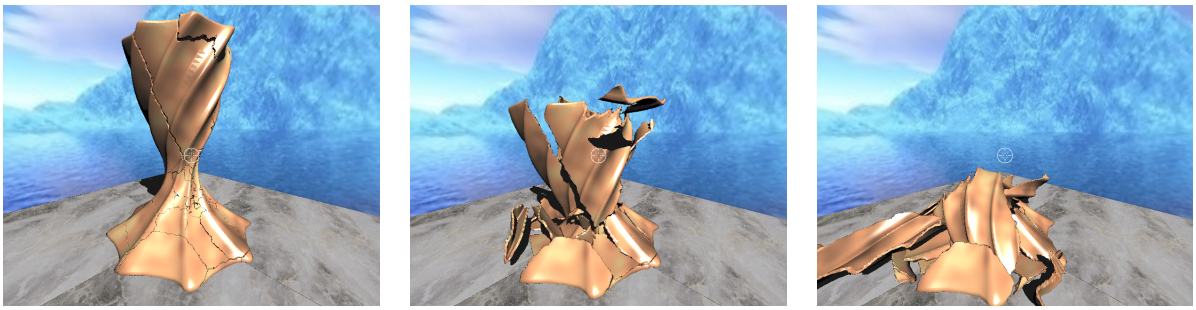


Figure 8: Fracturing and physics simulation of a vase.

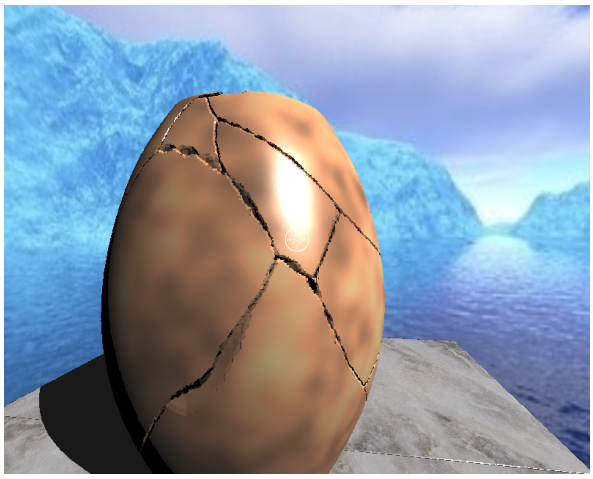


Figure 9: Cracks with irregular space between them.

adapted for objects represented with meshes. For this purpose, the first part of the algorithm could be performed on a tetrahedral mesh and within its cells, new surfaces would be created and cut against the destructed object.

8 REFERENCES

[AA05] Akenine-Möller, T., Aila, T.: Conservative and tiled rasterization using a modified triangle set-up. In Journal of Graphics Tools 10 (2005), 3, pp. 1-8.

[Bau11] Bautembach D.: Animated sparse voxel octrees. Bachelor's Thesis (feb 2011).

[BCC*11] Baker M., Carlson M., Coumans E., Criswell B., Harada T., Knight P., Zafar N. B.: Destruction and dynamic artist tools for film and game production. In ACM SIGGRAPH 2011 course notes (2011).

[BLD14] Baert J., Lagae A., Dutra' P.: Out-of-core construction of sparse voxel octrees. Computer Graphics Forum 33, 6 (2014), pp. 220-227.

[CNS*11] Crassin C., Neyret F., Sainz M., Green S., Eeisemann E.: Interactive indirect illumination using voxel cone tracing. Computer Graphics Forum (Proceedings of Pacific Graphics 2011) 30, 7 (sep 2011).

[Cra11] Crassin C.:. PhD thesis, Grenoble University, 2011.

[CYFW14] Chen Z., Yao M., Feng R., Wang H.: Physics-inspired adaptive fracture refinement. ACM Trans. Graph. 33, 4 (July 2014), 113:1-113:7.

[DM16] Domaradzki J., Martyn T.: Fracturing Sparse-Voxel-Octree objects using dynamical Voronoi patterns. In Computer Graphics, Visualization and Computer Vision WSCG 2016. Full Papers Proceedings, Computer Science Research Notes, vol. 2601, 2016, pp. 37-46.

[FBAF08] Faure F., Barbier S., Allard J., Falipou F.: Image-based Collision Detection and Response between Arbitrary Volume Objects. In Eurographics/SIGGRAPH Symposium on Computer Animation (2008).

[GPM11] Garanzha K., Pantaleoni J., Mcallister D.: Simpler and faster HLBVH with work queues. In Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11, ACM, pp. 59-64.

[GMD13] Glondu, L., Marchal, M., Dumont, G.: Realtime simulation of brittle fracture using modal analysis. IEEE TVCG 19, 2013, pp. 201-209.

[IO09] Iben H. N.,O'Brien J. F.: Generating surface crack patterns. Graph. Models 71, 6 (Nov. 2009), 198-208.

[JM16] Jabłoński S., Martyn T.: Real-Time Rendering of Continuous Levels of Detail for Sparse Voxel Octrees. In Computer Graphics, Visualization and Computer Vision WSCG 2016. Short Papers Proceedings, Computer Science Research Notes, vol. 2602, 2016, pp. 79-88.

[JM17] Jabłoński S., Martyn T.: Unlimited Object Instancing in real-time. In Computer Graphics, Visualization and Computer Vision WSCG 2017. Short Papers Proceedings, Computer Science Research Notes, vol. 2702, 2017.

[KSA13] Kämpe, V., Sintorn, E., Assarsson, U.: High resolution sparse voxel DAGs. In ACM Trans. Graph. 32, 4, 2013, pp. 101:1-101:13.

[Kar12] Karras T.: Maximizing parallelism in the construction of BVHs, Octrees, and k-d Trees. In High Performance Graphics (2012), Eurographics Association, pp. 33-37.

[LC87] Lorensen W., Cline H.: Marching Cubes: A high resolution 3D surface construction algorithm. In Proceedings of the 14th annual conference on Computer graphics and interactive techniques (1987), vol. 21, pp. 163-169.

[LK10] Laine S., Karras T.: Efficient sparse voxel octrees. In Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '10, ACM, pp. 55-63.

[M05] Mould, D.: Image-guided fracture. In Proceedings of Graphics Interface 2005. Canadian Human-Computer Communications Society, 2005. p. 219-226.

[MCK13] 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.

[NTB*91] Norton A., Turk G., Bacon B., Gerth J., Sweeney P.: Animation of fracture by physical modeling. The Visual Computer 7, 4 (1991), 210-219.

[OBH02] O'Brien J. F., Bargteil A. W., Hodgins J. K.: Graphical modeling and animation of ductile fracture. In Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '02, ACM, pp. 291-294.

[OH99] O'Brien J. F., Hodgins J. K.: Graphical modeling and animation of brittle fracture. In Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '99, pp. 137-146.

[PK15] Pätzold M., Kolb A.: Grid-free Out-of-core Voxelization to Sparse Voxel Octrees on GPU. Proceedings of the 7th Conference on High-Performance Graphics, HPG '15, ACM, pp. 95-103.

[SO14] Schvartzman S. C., Otaduy M. A.: Fracture Animation Based on High-dimensional Voronoi Diagrams. Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '14, ACM, pp. 15-22.

[SSF09] 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, ACM, pp. 155-164.

[TF88] Terzopoulos D., Fleischer K.: Modeling inelastic deformation: Viscolelasticity, plasticity, fracture. SIGGRAPH Comput. Graph. 22, 4 (June 1988), pp. 269-278.

[VMG16] Vllanueva, A. J.; Marton, F.; Gobbetti, E.: SSVDAGs: Symmetry-aware sparse voxel DAGs. In Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. ACM, 2016. pp. 7-14.

[Wil13] Willcocks C. G.: Sparse volumetric deformation. PhD Thesis (apr 2013).

[WRK*10] Wicke M., Ritchie D., Klingner B. M., Burke S., Shewchuk J. R., O'Brien J. F.: Dynamic local remeshing for elastoplastic simulation. ACM Transactions on Graphics 29, 4 (July 2010), 49:1-11. Proceedings of ACM SIGGRAPH 2010, Los Angles, CA.

[ZBG15] Zhu Y., Bridson R., Greif C.: Simulating rigid body fracture with surface meshes. ACM Transactions on Graphics (2015).

[ZGHG11] Zhou K., Gong M., Huang X., Guo B.: Data-parallel octrees for surface reconstruction. IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS (2011).