Fracturing Sparse-Voxel-Octree objects using dynamical Voronoi patterns

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 introduce a new Voronoi-based method to fracture objects represented by sparse voxel octrees (SVOs). Our approach is inspired by the pattern-based methods, however, in contrast to them, it doesn't require pattern precomputation. Moreover, thanks to the octree structure, the surfaces of the fractured pieces of geometry are created efficiently and robustly. Every fracture pattern is unique and centered at the impact location. A novel islands detection technique is also provided, which is tunable to a desired level-of-detail accuracy. The fractured pieces, which are determined as a consequence of the object's destruction, are represented by individual SVOs, and treated and simulated as rigid bodies. For this purpose, we also propose a new collision detection technique, which extends the previous image-based methods to voxels. As a result, deep penetrations of colliding objects, resolved on various levels of physics that can be specified individually for each pair of the objects, are handled in parallel with no extra cost. In order to demonstrate our technique, a number of scenarios are presented, including a partial fracturing of objects with fine details.

Keywords

SVO, Voronoi decomposition, pattern fracturing, rigid body physics

1 INTRODUCTION

For many years voxels have been successfully used in lots of applications in computer graphics. From special effects up to medical imaging, we benefit from volumetric information delivered by voxels. Transparent, layered and with vague surfaces models are the main target for this object representation. There were many limitations that came along with voxels, such as large memory consumption and lack of hardware support, and some of them are still present nowadays. However, many new techniques have been developed, which made voxels more competitive than ever. With constant increase in computational power of modern graphics processing units, we may be facing situation, when methods based on voxels will gradually supersede the ones based on triangles.

In this context, probably one of the best promising concepts is Sparse Voxel Octree (SVO) [Cra11] — a hierarchical structure that lately has made voxels popular as a representation for solid objects in computer graphics.

Less memory consumption, various levels of detail, and necessary access only to small subset of full voxel data are some benefits of SVO that have led to creation of realistic, high-resolution voxel models and large voxel environments. In this paper, we made an attempt to take advantage of this representation and shed a new light on the problem of objects fracturing.

Special effects of destruction are widely present in today's computer games and movie industry. Depending on a purpose, they can be either highly realistic but requiring lots of computational power, or fast but giving only an approximate illusion of observed phenomena. The latter approach is aimed for real-time performance, where accuracy is not as important, that this is usually the case in games. Many methods are used in order to do that in a fast but somewhat "fake" way, such as pre-fracturing of objects and replacing models when collision happens. Such an approach, however, requires preparations of lots of game assets. Recently, pattern-based fracture methods have been proposed for the mesh representation. They deal with this problem by decomposing the destruction process into two parts: the generation of a fracture pattern and then its multiple application on number of objects. While the first part can be done by an artist, the latter requires specialized techniques in order to robustly cut objects, mainly due to their geometrical complexity. Similarly, after the ob-

ject's destruction, the movement of fractured pieces is simulated and it also suffers from the same reason: the more detailed object, the harder to calculate collision.

We strongly believe, that representing objects with sparse voxel octrees is a remedy to all these problems. With spatial information about an object, built from basic geometric figures like boxes, many algorithms can be simplified. What is more, due to the SVO hierarchical structure, many calculations can be avoided and performed on different levels of details. Our main contribution include:

- a method to dynamic fracture with Voronoi decomposition, which can be applied locally;
- a method for detecting separated volumes in a SVO;
- an algorithm for collision detection, that adopts the image-based approach [FBAF08] to voxels.

2 RELATED WORK

Fracture Simulation. Fracture modeling for computer graphics greatly enhances physics simulation widely present in modern computer games and special effects in movies. Probably it wouldn't be so common today if it weren't for work by Terzopoulos et al. [TF88] and Norton et al. [NTB*91] which pioneered this area of study in computer graphics. O'Brien and Hodgins followed them and focused in their papers on simulating brittle [OH99] and ductile fracture [OBH02]. Their approach was based on the finite element method (FEM) used to compute internal stresses and fracture propagation directions. Unfortunately, cutting a mesh and its actualization, which is one of the most challenging issues in such approaches, are still present nowadays [WRK*10]. There are also methods that formulate the problem of fracture simulation as a quasi-static stress analysis [ZBG15]. They have the potential to be cheaper than a fully dynamic deformation simulation, but tends to produce deadened motions. An example of another interesting work can be [CYFW14], in which low resolution objects are enriched during fracture with extra details based on material strength field.

Although, from the "physical" viewpoint, the results of the mentioned methods can be perceived as acceptable, our main interest lies in solutions aimed for real-time systems. Since the pattern-base approach, at least conceptually, plays well with voxel representations and, moreover, is efficient and robust, we decided to utilize it in our SVO fracture method. A number of methods have been proposed to generate fracture patterns, such as the ones based on Voronoi diagrams [SSF09][BCC*11][MCK13] or engaging simulations [IO09]. In our approach, in contrast to the previous ones, we don't need any precalculated fracture pattern, as it is generated on-the-fly (like in [SO14]), when collision happens.

Collision Detection and Response. Collision detection itself covers broad area of investigation in computer graphics. Due to colliding objects geometrical complexity it is common to take advantage of hierarchical bounding structures, including bounding boxes [GLM96] and bounding spheres [Hub95]. Fortunately, in the case of SVO, objects are inherently represented hierarchically, so there is no need for any additional space searching optimizing structure.

More recently, some work on collision detection has been focused on exploiting GPUs, taking advantage of their inherent computational parallelism. Most of the presented techniques return pairs of colliding primitives delivering necessary information for objects separation. In particular, Layer Depth Images (LDIs) proved to be very useful for this goal [HTG04]. In this method, at the first stage, a broad phase collision check is performed, resulting in bounding boxes representing the intersection volume of boxes' pairs that enclose colliding objects. Then, the volumes are rendered to LDIs using GPU. Finally, iterating over LDIs along a chosen viewing axis, one can calculate the collision volume by inspecting the pairs of consecutive texels. In the original LDI method as presented by Heidelberger et al. only collision detection but not its response was taken into account. That was addressed later by Faure et al. [FBAF08] by delivering the derivatives at the vertices of the meshes of colliding objects in order to generate forces for minimizing the volume of collision. In addition to the penalty-based method used in [FBAF08], Allard et al. [AFC*10] proposed also the constraint-based one including Coulomb friction.

In the context of collision detection, our method can be viewed as an extension of that by Faure et al., and it can also be incorporated into the method by Allard et al. Our main contribution lies in the efficient determination of the collision area. The main advantage of our method is that we can generate collision information in parallel for all collided objects at accuracy specified individually for a given pair of objects. While Faure dealt with objects represented with meshes, we utilize SVOs. One should notice that this new problem formulation fits very well to the LDI-based solution by Faure.

Sparse Voxel Octrees. The pursuit of efficient exploitation of voxel structures as representations of solid objects in computer graphics lasts for years. Nowadays, with the aid of modern GPUs along with the new GPU-specialized programming techniques, the benefits of voxel model of solids become not only evident but even more and more spectacular. Probably the best example of this new life of voxels in computer graphics is Crassin's research on SVO [Cra11] — one of the notable results is a global illumination method based on SVO and cone tracing [CNS*11]. Moreover, there are also some work that focuses mainly on efficient raytrac-

ing over SVO [LK10], which shows that ray casting using SVO can be faster than when objects are represented with meshes. Although the majority of papers focus on visualization of static voxel models, there were also some approaches to animation of voxel models. Crassin proposed to animate objects by constant voxelization to SVO, which unfortunately requires object's input mesh and scales poorly. Nevertheless, the SVO animation can be realized in other ways. An example is the method developed by Bautembach [Bau11], in which SVO is animated only during visualization, without any influence on the structure of an input model. His approach was then extended by Willcocks [Wil13], who presented a method for volumetric deformation and animation of large number of objects in the scene. What is more, octrees can be built very fast using recent techniques presented in [ZGHG11][GPM11][Kar12]. If there is not enough memory in GPU to build SVO by means of the methods, one can use the out-of-core approaches described by [BLD14][PK15], which utilize only a fraction of memory required to store the model.

The above and inevitably incomplete set of citations show that the computer graphics community nowadays experiences a renaissance of voxels, which manifests in constant development of new voxel techniques and applications. Nevertheless, to the best of our knowledge, this paper is the first report on the application of SVO in the tasks such as physics of rigid body collisions and fracture simulation.

3 SVO STRUCTURE

Our SVO structure is very similar to the one used by Crassin et al. [Cra11]. First of all, there is an octree for nodes' descriptors, represented as two 32 bit integers. Each node descriptor has the information about whether the node: is terminal (1 bit), represents empty space (1 bit), is internal (1 bit). The remaining 29 bits of the first integer are used as a pointer to the descriptor of the first of node's eight children. What is more, with each descriptor there is associated another pointer, as a second integer, to the densely packed voxel's data: color and normal vector.

3.1 Voxel Types

In our SVO structure, there are three kinds of nodes: boundary, internal and empty. *Boundary nodes* are the nodes that represent the boundary of the object and they are used in visualization. If the SVO is derived from a mesh, then the nodes are created during voxelization process and contains information about color and normal vector. They can be expanded further up to the finest SVO level. One of their child nodes can be an empty node. The *empty node* indicates that its volume doesn't intersect the object's surface, that is the node lies either inside or outside of the object — it is always a terminal node. There are also *internal nodes* that, from the algorithmic point of view, can be considered as a special case of empty nodes, because an empty node is flagged as internal when it passes the test (Sec. 3.2) whether it is located inside of the object. As such, internal nodes represent the interior of the object. These nodes are very important as they deliver the information that is necessary for objects fracturing and physics calculations.

3.2 Internal Nodes Test

After the SVO creation, the test for internal nodes is conducted for each empty node on every SVO level, because empty non-internal nodes are skipped during the fracturing process and collision detection. The test is based on ray casting. A ray is shot from the empty node center and has the same direction as the normal vector of the parent node. Next, the ray traverse the SVO down to its leaves. The two outcomes of this operation are possible. Either the ray doesn't hit anything or it encounters one of the leaves. In the first case, the internal node test has failed. In the second case, an additional test must be performed to determine which side of the leaf node was hit. The test rely on the comparison of the ray direction and the node normal vector. If the angle between them is smaller than 90 degrees, then the internal node test passed and the node is assumed to belong to the interior of the object (in other case, the empty node lies outside the object, so the test failed).

4 FRACTURING SVO OBJECT

4.1 Fracture Algorithm

In this section we outline our algorithm for fracturing SVO objects.The underlying assumption is that a SVO object is cut into pieces with a fracture pattern represented by a space-partitioning structure that divides the 3D Euclidean space into convex regions (Sec. 4.2). As such, the fracture pattern consists of (planar) faces determining the slicing areas which are then used to partition the SVO object.

The main part of the algorithm is to determine subsets of the SVO voxels that represent the surfaces of fractured pieces at the accuracy of the SVO highest level. The voxels that constitute the subsets are the boundary voxels (Sec. 3.1) from the SVO leaves as well as internal voxels that are intersected by a pattern face at the SVO highest level — we call the latter voxels the *HL internal voxels*. One should note, however, that internal voxels do not have children in the SVO, so the HL internal voxels that are not children of boundary voxels are not physically present in the original SVO. Therefore the HL internal voxels have to be dynamically created during the fracturing process.

In order to determine the HL internal voxels 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 (Sec. 4.3). If an internal voxel is intersected by a pattern face and the voxel is not the HL internal voxel, then its eight child internal voxels are dynamically created at the next SVO level. With regard to the voxel-face intersection tests the two following facts should be pointed out. First, the necessary condition for a child voxel to be intersected with a pattern face is the presence of the intersection of the face with the voxel's parent. Secondly, the parent of an internal voxel may be an internal voxel (in this case the child voxel was dynamically created) or a boundary voxel. Therefore at each SVO level the voxel-face intersection tests are performed for both interior and boundary voxels whose parent voxels have been intersected at the previous SVO level (Fig. 1).

Having the subsets of voxels, we regard the voxels as the boundary voxels of the fractured pieces (Sec. 4.4) represented by the subsets, and for each subset we build a SVO (Sec. 4.6) on the basis of the subset's component voxels treated as the SVO leaves.

One should note, however, that the SVO partition procedure does not guarantee that the resulting subsets of voxels will be connected sets. It may happen that a subset can be partitioned into two groups of voxels such that for all voxels in one group there is no adjoint voxel in the other group. In such a case the subset represents two (or more) disjoint fracture pieces which should be treated as individual objects by a physics engine. Such autonomous groups of voxels within a single subset we call *islands*. In order to detect them, the additional test (Sec. 4.5) has to be performed before the SVO creation.

4.2 Voronoi Fracture Pattern

Patterns used to fracture objects can be obtained in many ways. Usually, it is a precomputed decomposition of space, based on Voronoi diagrams. During the fracture process performed at run-time, the pattern is aligned with the target object at an impact location and properly rotated. Such an approach was recently presented by Sue et al. in [SSF09]. In their method, the fracture pattern is applied to a mesh object using the level set-based approach, which requires high resolution grids to get thin features and, consequently, it is both computationally expensive and memory consumptive. On the other hand, the fracture pattern may be applied directly to a mesh. However, a naive application of this idea would be cumbersome to implement robustly. To deal with it, Müller et al. [MCK13] implement a fracture pattern as a set of meshes, and a mesh to fracture is initially split to convex pieces with the use of the Volumetric Approximate Convex Decomposition (VACD). As a result, they are capable to locally destruct mesh objects in real-time.

In contrast to the previous methods, in our approach we represent a fracture pattern by a finite set of 3D points

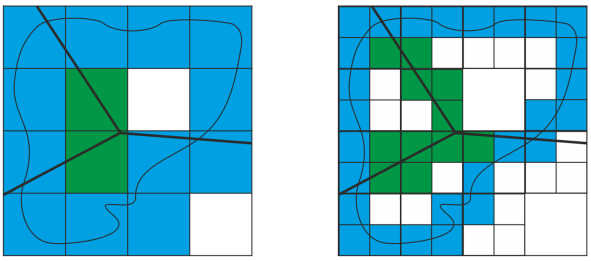


Figure 1: Voxels located on both the object's boundary and the Voronoi faces are expanded into child nodes on the next SVO level. The voxels marked with blue color lie on the boundary, while the green ones are internal, generated on-the-fly.

which are treated as seeds of a Voronoi diagram used to fracture a SVO object and generated dynamically at run-time. To deliver the impression that the fracture concentrates around the impact location, the Voronoi seeds are generated at random on a set of spheres of growing radii and centered at the point of impact. By manipulating the length of the radii we can obtain a variety of patterns and different sizes of fractured pieces. Of course, there is also a possibility to locate the seeds in a more physical way, which would make the outcome of the fracture process even more realistic.

4.3 Voxel-Pattern Faces Intersection Test

One of the main operations of our algorithm for fracturing SVO objects is the voxel-pattern faces intersection test which is performed at each level of the SVO as described in Sec. 4.1. The intersection test does not require the determination of the faces of the Voronoi diagram since it is realized directly on the basis of the set of the Voronoi seeds.

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

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

Since there is the one-to-one correspondence between the Voronoi seeds and the Voronoi cells, given a point $x \in \mathbb{R}^3$, the function γ determines the set of the indices of the Voronoi cells that include x. (The resulting set of indices is not a singleton, if x lies on a face, an edge, or a vertex of the Voronoi diagram).

Now, we can move to the voxel domain and define a function that, 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 located in:

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

Since the voxel V is intersected by a face of the Voronoi pattern if V has a (nonempty) intersection with at least two Voronoi cells, it is easy to see that the intersection

exists if the set of indices given by $\mho(V)$ is not a singleton. Of course, in a practical implementation of the intersection test we can stop computing the set $\mho(V)$ if its subset obtained in an intermediate step contains at least two indices.

4.4 Fracture Boundary Set

As a result of the voxel-pattern faces intersection tests performed while traversing the SVO, we obtain an unordered set of the HL internal voxels (see Sec. 4.1). The set together with the set of the SVO leaf boundary voxels constitute the set $\mathscr{B}$ of voxels which are the boundary voxels representing the surfaces of the fractured pieces at the accuracy of the SVO finest level. In order to assign these boundary voxels to the adequate pieces we should decompose the set $\mathscr{B}$ into the *disjoint* family of the connected subsets of voxels that represent, first of all, the pieces induced by the Voronoi cells, and then islands within a piece, if the piece proves to be a disconnected set.

Nevertheless, the voxels in $\mathscr{B}$ that have a nonempty intersection with a pattern face belongs to two or more Voronoi cells of the pattern and, as such, they should be "divided" between the cells. This leads us to the *fracture boundary set* (FBS) which is constructed from $\mathscr{B}$ by computing for each $V \in \mathscr{B}$ the set of indices $\mho(V)$, creating copies of V in a number equal to the cardinality of $\mho(V)$, and assigning to each copy the consecutive index from $\mho(V)$. Since the indices from $\mho(V)$ determine the Voronoi cells that V belongs to (Sec. 4.3), the copies of a voxel in FBS are treated as disjoint sets from the point of view of the voxel space partitioned with the fracture pattern. The FBS is the set on which we carry out the operations of grouping voxels into connected sets of fractured pieces — the algorithm described in the next section.

Apart from an index of the Voronoi cell, each voxel in FBS has to store attributes required for visualization, i.e., color, position, and normal vector. The attributes are determined on the basis of the attributes of original voxels from the SVO. The voxel's position is computed during the SVO traversal on the basis of the location of the voxel within the SVO. The remaining two attributes depend on whether the original voxel is a boundary voxel or a HL internal voxel. The copies of a boundary voxel inherit a color and normal vector from their original. In the case of the copies of a HL internal voxel the color may be obtained from a volumetric texture or it can be generated procedurally. In turn, their normal vectors are determined on the basis of the normals of the Voronoi faces that intersect the HL internal voxel. If the HL internal voxel does not lie on an edge or a vertex of the intersecting Voronoi face, the normal for the copy of the voxel in FBS is just equal to the face normal pointing outside the Voronoi cell associated to the copy; otherwise the normal is computed by averaging the normals of all faces that intersect the HL internal voxel.

4.5 FBS connected-component detection

Since FBS is just an unordered set of boundary voxels, we have to group the voxels into the connected subsets of the individual fractured pieces, i.e., the pieces induced by the Voronoi cells and eventually the islands within a piece, if the piece is a disconnected set (Sec. 4.1). However, before the actual voxel grouping takes place, in the initial step, FBS is sorted by the voxel positions represented by Morton codes [Kar12] (Fig. 2). Thanks to that, the voxels can be addressed by one-dimensional index, which is essential for the island detection algorithm as well as later, for constructing the SVOs (Sec. 4.6).

Now, we obtain the FBS voxels grouped with regard to the fracture pattern cells, just by carrying out the stable sorting by the $\mho(V)$ indices, which were assigned to the voxels during the FBS construction (as described in the previous section). As a result, the groups are placed one after another in linear memory, and the voxels within each group are ordered by Morton code. We call the groups of voxels *Voronoi cell pieces* (VCPs). In the next step each of the VCP groups has to be checked for connectivity, that is the island detection test is performed.

4.5.1 Island detection algorithm

We assume that a voxel is connected with another voxel from its nearest neighborhood if the voxels share a face, an edge or a vertex, so we assume 26-connectivity. Our solution to the island detection problem is based on the well-known conception of *connected-component labeling* (CCL — see e.g. [SB10]). The CCL underling idea is to label the elements of an input set with consecutive numbers, and then, to maximize iteratively the labels in the range of the elements' adjacent neighbors. This way, for each connected subset, the maximum label in the subset propagates between the subset's elements. At the end of the procedure the connected subsets are distinguished from one another by a maximum label that is associated to each element of a connected subset.

Although there are a number of efficient CCL algorithms, both sequential and parallel ones, to our best knowledge, all of them assume and operate on the input data organized in a regular structure such as a 2D uniform grid of pixels or a 3D uniform grid of voxels. Therefore the necessary neighborhood information for each basic structure element is easily accessible. Unfortunately, this is not the case for SVO and, as a consequence, for FBS, since the voxel neighborhood information does not follow from the ordering of elements inherent for these structures. Of course, one could augment the structures in the additional information by storing the pointers to neighbors of each voxel,

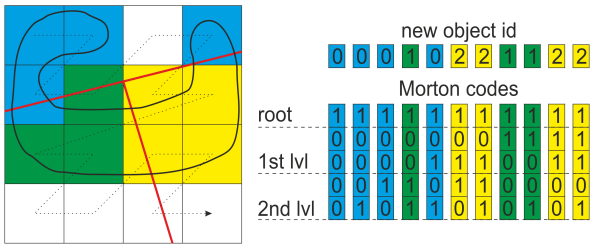


Figure 2: Fractured object represented by a quadtree. Node positions are described with Morton codes and new objects' ids are assigned.

but this would require a lot of additional memory space and extra processing during the SVO constructions. Instead, we decided to base our island detection algorithm on the direct application of the CCL underling conception realized in the form of a parallel implementation on GPU in CUDA.

In the initial step of the algorithm the voxels in FBS are labeled with consecutive numbers. Then, for each voxel within each VCP group one thread is launched. Every thread loops over the VCP voxels existing in the 26-neighborhood of the relevant central voxel, taking the maximum of the label values of the central voxel and the neighboring voxels and relabeling with the maximum the voxels of the lower label value. Since, at the same time, other "neighboring" threads may try to relabel the voxels, the relabeling is done using an atomic operation (`atomicMax` in CUDA). The central voxel's neighbors are determined on the basis of their Morton codes by means of the binary search on voxels from the same VCP group, and their memory addresses are cached in thread local memory. Each thread continues until there is no change in the labels of the voxels processed by the thread.

Then, for each VCP group in which relabeling took place, the threads are launched again, and the algorithm continues this way until there is no VCP group to relabel.

Finally, in the FBS voxels, the $\mho(V)$ indices are substituted with the labels, and FBS itself is stably sorted by the labels to obtain the required partition on the groups of the connected voxel subsets, which preserves the order of the Morton codes within each group.

4.5.2 Optimizations

The original 26-neighborhood each thread is supposed to operate on can be reduced by half. Let $\mathbb{V}$ be the voxel space induced by the cubic lattice that coincides with VCP; thus VCP $\subset \mathbb{V}$. Let $N_k(V)$ be a k-element subset of the 26-neighborhood $N_{26}(V) \subset \mathbb{V}$ of a voxel $V \in \mathbb{V}$. Assuming that the threads operate on the subsets $N_k(V) \cap$ VCP, $V \in$ VCP, it is easy to see that the sufficient condition for the correct propagation of labels by the threads within a connected component is that for all voxels $V, V' \in \mathbb{V}$ such that $V' \in N_{26}(V) \setminus N_k(V)$, the voxel V belongs to $N_k(V')$. If $N_k(V)$ satisfies this condition, then this guarantees that if $label(V) < label(V')$ for some voxel $V' \in N_{26}(V) \cap$ VCP, then there is a thread to propagate $label(V')$ to V. A little thought shows that for $k = 13$, $N_{13}(V)$ can be composed with 9 voxels of coordinates $(x+a, y+b, z-1)$ and 4 voxels of coordinates $(x+a, y+1, z)$, $(x-1, y, z)$, where $a, b \in \{-1, 0, 1\}$, and (x, y, z) are coordinates of the voxel V. Moreover, one can easily check that $k = 13$ is the minimum number of voxels for $N_k(V)$ to obey the above condition.

There are also other possible optimizations that we use in the island detection algorithm. First, there are dependency chains between voxels in the same neighborhood, which results in equivalence trees [SB10], that can be flatten in each iteration. Secondly, there is no need to re-launch threads which have propagated the VCP maximum labels to their neighborhoods. Since we label the FBS voxels, we know the maximum value of the labels within each VCP group, so we exclude such "maximum" threads from further operations. Thirdly, each voxel V stores the information about its *existing* neighbors (i.e., the voxels from $N_{13}(V) \cap$ VCP) in order to reduce the costly binary search being done by a thread only to these neighbors. The information is obtained during the first execution of the thread and coded in one integer by setting appropriate bits. Lastly, some speed/precision trade can be made, as the island detection can be performed on other SVO level than the finest one. In order to do so, one have to construct an approximation of FBS composed of the ancestor voxels of the FBS voxels at a given SVO level. The appropriate ancestors can be easily determined on the basis of the Morton codes of the FBS voxels. Then the island detection test can be carried out on that approximated FBS set, and the result propagated down to the voxels from the original FBS.

4.6 SVO Building

Having FBS partitioned on the connected voxel subsets distinguished by labels (Sec. 4.5.1), for each subset a SVO has to be built. As mentioned in Sec. 2, there are a number of fast methods for building a SVO, but we are interested on those that utilize Morton code. A SVO can be build in depth-first order [Kar12] by creating all levels in one kernel call. In turn, the breadth-first order is obtained iteratively [ZGHG11], and this approach is most suitable for our purposes. Moreover, the original algorithm can be easily extended to build many SVOs simultaneously (voxels from different SVOs are distinguished by the piece labels). Our implementation remains almost the same as in [ZGHG11], with two exceptions. First, the parallel prefix scan primitive is replaced with its segmented version [HB10]. Secondly, the sorting primitive operation has to be changed into

the stable sorting primitive and performed two times: first by Morton codes and second by the piece label.

4.7 Local Fracture

Apart from fracturing the whole object, there is also a possibility to perform this operation locally. In order to do that, one can attribute to every Voronoi seed an extra object identifier. Then, on the last level of applying the fracture pattern, voxels are assigned to piece objects on the basis of these identifiers instead of the seed indices (see Sec. 4.4). The assignment of the same identifier to many seeds allows large parts of the fractured object to remain intact and to properly create surfaces for pieces. However, the voxels intersected by Voronoi faces must still be detected, and on the last SVO level the intersection test must be altered. The modification concerns internal voxels intersected by a Voronoi face that is induced by seeds with the same identifier — these voxels should not be considered in the output result. Otherwise, some extra surfaces would be generated inside a part of the SVO that is supposed to stay intact.

5 SVO COLLISION AND RESPONSE

In this section we address the problem of the rigid body simulation of objects represented by SVOs. Lots of techniques have been proposed over the years to simulate physics behavior of objects in virtual environments. The methods have been mainly designed for meshes, but often tended to use other forms of representation and structures to model physical shapes and properties. For example, one may utilize particles located on the object's surface to detect collision and generate repulsion forces. However, the surface-based methods are characterized by short range of reactions and may not easily recover from deep penetrations. Another example are the distance-based methods. They rely on a spatial map that encodes the signed distance from a given point in 3D space to the object surface. Although they better handle deep intersections, the calculation of the distance map can be computationally expensive and memory consumptive.

The approach we chose to adapt to voxel domain is based on the volume minimization method [FBAF08]. In order to resolve collision, the technique models the intersection of colliding objects, and then computes the size of the intersection volume and the repulsion forces—the latter derived from the volume gradients. No precomputation is required and the efficiency of the surface-based methods is combined with the robustness of the distance-based ones.

5.1 Collision

In [FBAF08] to model the intersection of colliding objects the Layer Depth Images (LDIs) were used along with GPU to benefit from parallelism. Our approach is slightly different. First, the voxelization of objects into LDI is no longer necessary, since our objects are build with voxels. Secondly, in place of images of different resolutions, we take advantage of the SVO hierarchy and utilize it in the collision test.

5.1.1 Intersection of SVOs

In order to detect collision, we start with a broad-phase check in which the intersections of SVOs' roots of the objects are tested. Then, the potentially colliding objects are paired and their SVOs traversed simultaneously. When traversing down, pairs of voxels (a voxel from one SVO and a voxel from the other SVO) on a given level of the SVOs are examined for an intersection (performed between two AABBs in the local space of one of the SVOs), until the desired level of detail is reached. The process is very efficient, since the intersection test is continued only for the pairs of those child voxels whose parents intersect each other. Moreover, the pairs of internal voxels are omitted, since our goal is to follow the boundary voxels so as to enclose the intersection of the chosen levels, and calculate the size of the volume of the set and gradients of this volume. In turn, we use this information in order to generate the collision response forces (see 5.2).

5.1.2 Collision Points

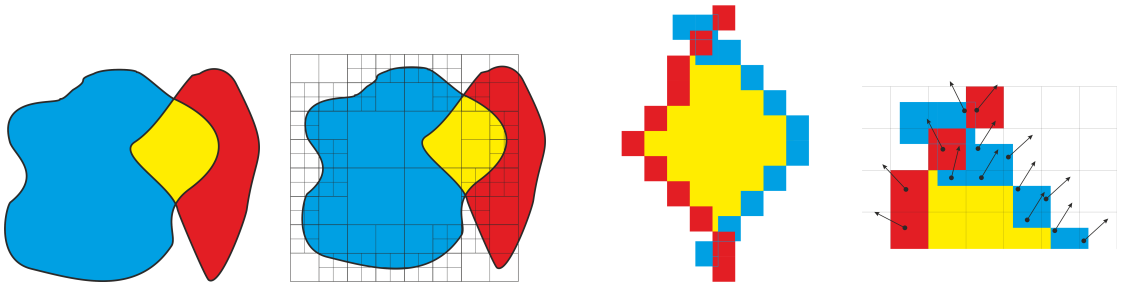
The next step is the approximation of the surface of the SVOs' intersection. For this purpose we generate a set of *collision points* when the desired SVO level is reached during the traversal of the trees. The collision points correspond to the intersections between the pairs of voxels from the colliding SVO objects. For a boundary-internal voxel intersection, the position of the collision point is determined as the center of the intersection volume between the pair of the voxels. In turn, for a boundary-boundary voxel intersection two collision points are generated; each is positioned on the face of one of the voxels from the pair by translating the center of the intersection volume along the normal vector of the voxel.

Moreover, with each collision point the normal vector of the surface of the SVOs' intersection is associated, which is just the normal vector of the corresponding boundary voxel.

Finally, we perform discretization of the collision points by averaging their positions and normal vectors within the cells of the cubic lattice coincident with the voxels on the chosen SVO level. The operation is straightforward as the collision points data is kept in the local space of one of the SVOs.

5.1.3 Intersection Volume and Gradients

Having the collision points, we can calculate the size of the intersection volume of the colliding objects. In order to do that, we integrate the intersection volume by



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

Figure 3: (a) Two colliding objects. (b) The SVO structure drawn on the top of the blue object. (c) The intersection area modeled with voxels. (d) The dots represent averaged collision points and volume gradients.

iterating over the collision points located in the columns of the cubic lattice used for discretization. We examine pairs of subsequent collision points. Our goal is to find the pairs of the collision points such that their normal vectors' coordinates in the axis we iterate on have opposite signs, and the same requirement is fulfilled for the normal vector of the first collision point and the search direction. If this condition is satisfied, the pair of collision points represent the local entry and exit of the intersection volume surface, so the difference between the positions of such collision points (in the axis we iterate on), when multiplied by the voxel face area, is a partial volume of the intersection. Therefore, to obtain the entire volume of the intersection we sum these partial volumes. Moreover, we compute the volume gradients as the collision points' normal vectors multiplied by the voxel face area [FBAF08].

5.2 Response Forces

To resolve collision, the colliding objects have to be separated. For this purpose we utilize the penalty-based method as in [FBAF08]. It results in the generation of two forces for each collision point. First, there is a repulsion force, which can be formulated as:

$$f_{\text{rep}} = -kV_{\text{int}}g_i \tag{3}$$

where k is an arbitrary positive constant, V_{int} – the size of the intersection volume, and g_i – the volume gradient at the i-th collision point. The second force is the friction, expressed by the equation:

$$f_{\text{frc}} = \mu v_t a \tag{4}$$

where μ is an arbitrary positive constant, v_t – the relative tangential velocity, and a – the voxel face area.

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.

6.1 Varying Physics Level

Our collision detection method, due to the hierarchical structure of SVO, supports varying physics levels at no extra cost. As a result, one can easily balance between precision of performed calculations and performance. However, the object's volume is changing with physics level. The higher SVO level, the more accurate object's description. In order to obtain the volume of an object, its boundary voxels must be sorted along one axis and the pairs of the entry and exit voxels must be found. As a consequence, small objects at lower SVO levels may not have volume at all and collision forces would not have affect on them.

Moreover, the chosen physics level doesn't have to be the same for all collided objects' pairs. It can be adapted to the current situation or, for example, based on objects distance to the camera. When something happens far away from the observer, it doesn't have to be presented precisely, as the observer couldn't see all the details anyway. In addition, sorting collision points and forces calculations can be performed in parallel for pairs on different physics level.

6.2 Fracture Performance

Table 1 presents timings of consecutive fracture stages on chosen test scenes. Studying the results, one can observe that increasing SVO level with the same number of Voronoi seeds, multiplies the fracture time by a factor of 3–4 with reference to the fracture time on the previous SVO level. This is directly connected with the number of voxels on the current SVO level, which changes in a similar manner. Moreover, the impact location and the fracture pattern itself also has great influence on fracture time. The more fragmented objects, the more surfaces to create, and the more voxels to process.

6.3 Physics Performance

The physics part of our solution was tested using a number of various scenarios. First, the simplest scene is presented in fig. 4. The Bunny was fractured and new

Scene	SVO level	Voronoi nodes	FBS extraction	Voxels sorting	Islands detection	SVOs building	Internal nodes detection	Total
Bunny (Fig. 4)	8	90	2.5 ms	1.1 ms	2.7 ms	4.1 ms	2.5 ms	12.9 ms
	9	90	6.4 ms	2.5 ms	9.2 ms	5.9 ms	6.7 ms	30.7 ms
	10	90	21.0 ms	10.4 ms	40.1 ms	13.8 ms	25.5 ms	110.8 ms
Columns (Fig. 5)	10	70	17.5 ms	9.5 ms	38.8 ms	11.8 ms	22.2 ms	99.8 ms
Buddha (Fig. 6)	10	40	5.1 ms	3.4 ms	12.9 ms	6.3 ms	17.6 ms	45.3 ms
Dragons (Fig. 7)	9	15	2.1 ms	1.7 ms	6.8 ms	4.5 ms	7.0 ms	22.1 ms
		35	2.5 ms	2.1 ms	7.3 ms	4.9 ms	7.7 ms	24.5 ms
		50	2.7 ms	2.2 ms	8.5 ms	5.4 ms	8.8 ms	27.6 ms

Table 1: Fracture timings on different scenes. For the Columns and Buddha timings are an average of all fractures. In the Dragons scene, every dragon has an independent result, as different fracture parameters where used.

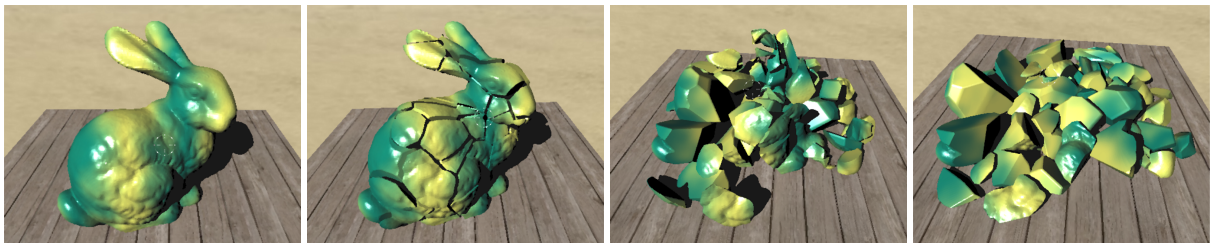


Figure 4: Fracturing the Stanford Bunny with the 8th physics level. Physics time was 4–9 ms per time step.

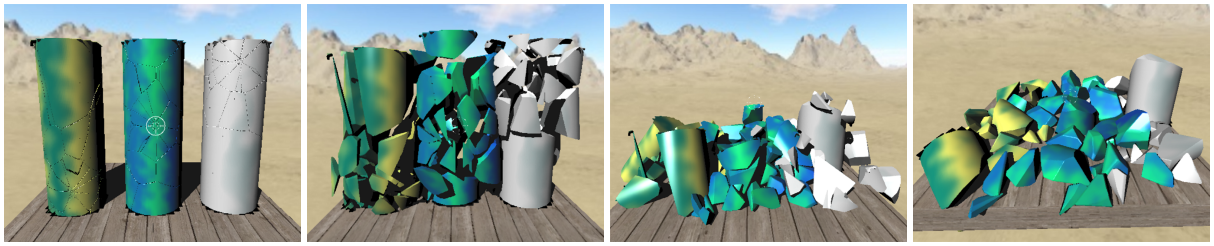


Figure 5: Fracturing three columns with physics calculated on the 7th SVO level. Physics time was 4–8 ms per time step.

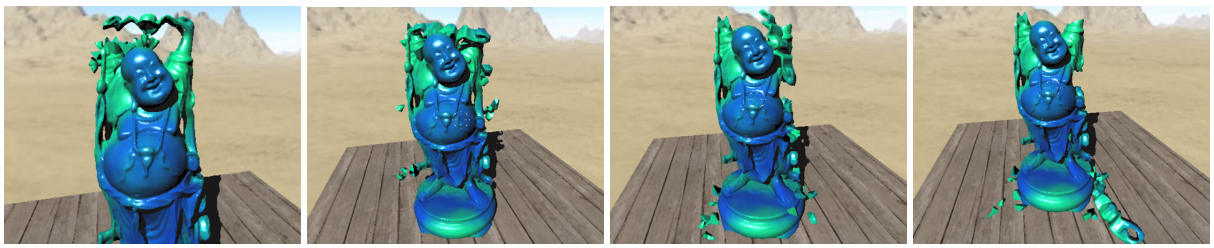


Figure 6: Local fracture of the Happy Buddha figure with the 9th physics level. Physics time was 5–7 ms per time step.

Figure 7: Three dragons dropped and fractured on collision points' locations. The objects had a different number of Voronoi seeds used during fracturing: 15, 35, 50. Physics was set to the 8th level and calculation time varied from 3 to 11 ms per time step.

pieces were scattered over the box's surface. The operation was rather energetic and rich with objects from a wide range of volumes. The physics time was 4–9 ms obtained on the 8th physics level. Even though, the lower physics level (7th) was used in *Columns* scene (fig. 5), the average time was similar to the previous one, and varied from 4 to 8 ms. It was the result of a larger number of collision points detected during the collision test. The opposite situation happened in *Happy Buddha* scene (fig. 6), where the 9th physics level produced better performance (5–7 ms) for local fracture. The last scene (fig. 7) presents three dragons dropped and fractured on collision points locations. The objects were cut into rather big pieces. However, due to their complexity, the average physics time varied from 3 to 11 ms. The time step for all simulations was set to 0.01 s, as we use the penalty-based response method with the implicit Euler method as an integrator.

7 CONCLUSION

We have presented a novel method for fracturing objects represented with sparse voxel octrees. Our method applies a fracture pattern to the object in the impact location, and cut the object into pieces, which are then also represented with SVOs. No precomputation of the pattern is required, as it is generated on-the-fly. After fracturing, new rigid objects are simulated. Thanks to our collision detection and response method, which is an extension of Faure's image-based approach [FBAF08], we can efficiently calculate physics in parallel on different levels of details with no extra cost.

8 REFERENCES

[AFC*10] Allard J., Faure F., Courtecuisse H., Falipou F., Duriez C., Kry P. G.: Volume contact constraints at arbitrary resolution. ACM Trans. Graph. 29, 4 (July 2010), pp. 82:1-82:10.

[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), 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.

[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).

[GLM96] Gottschalk S., Lin M. C., Manocha D.: OBBTree: A hierarchical structure for rapid interference detection, 1996.

[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.

[HB10] Hoberock J., Bell N.: Thrust: A parallel template library, 2010. Version 1.7.0.

[HTG04] Heidelberger B., Teschner M., Gross M. H.: Detection of collisions and self-collisions using image-space techniques. In WSCG (2004), pp. 145-152.

[Hub95] Hubbard P. M.: Collision detection for interactive graphics applications. IEEE Transactions on Visualization and Computer Graphics 1 (1995), 218-230.

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

[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.

[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.

[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.

[SB10] Stava O., Benes B.: Connected component labeling in CUDA. GPU computing gems emerald edition (2010), pp. 569-581.

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

[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 (to appear) (2015).

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