Computers & Graphics 125 (2024) 104104

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical Section

Generating implicit object fragment datasets for machine learning ☆

Alfonso López [a,*], Antonio J. Rueda [a], Rafael J. Segura [a], Carlos J. Ogayar [a], Pablo Navarro [b], José M. Fuertes [a]

[a] Department of Computer Science, Campus Las Lagunillas s/n, Jaén, 23071, Spain
[b] Instituto Patagónico de Ciencias Sociales y Humanas, Centro Nacional Patagónico, CONICET, Puerto Madryn, Argentina

ARTICLE INFO

Keywords:
Voxel
Fragmentation
Fracture dataset
Voronoi
GPU programming

ABSTRACT

One of the primary challenges inherent in utilizing deep learning models is the scarcity and accessibility hurdles associated with acquiring datasets of sufficient size to facilitate effective training of these networks. This is particularly significant in object detection, shape completion, and fracture assembly. Instead of scanning a large number of real-world fragments, it is possible to generate massive datasets with synthetic pieces. However, realistic fragmentation is computationally intensive in the preparation (e.g., pre-factured models) and generation. Otherwise, simpler algorithms such as Voronoi diagrams provide faster processing speeds at the expense of compromising realism. In this context, it is required to balance computational efficiency and realism. This paper introduces a GPU-based framework for the massive generation of voxelized fragments derived from high-resolution 3D models, specifically prepared for their utilization as training sets for machine learning models. This rapid pipeline enables controlling how many pieces are produced, their dispersion and the appearance of subtle effects such as erosion. We have tested our pipeline with an archaeological dataset, producing more than 1M fragmented pieces from 1,052 Iberian vessels (Github). Although this work primarily intends to provide pieces as implicit data represented by voxels, triangle meshes and point clouds can also be inferred from the initial implicit representation. To underscore the unparalleled benefits of CPU and GPU acceleration in generating vast datasets, we compared against a realistic fragment generator that highlights the potential of our approach, both in terms of applicability and processing time. We also demonstrate the synergies between our pipeline and realistic simulators, which frequently cannot select the number and size of resulting pieces. To this end, a deep learning model was trained over realistic fragments and our dataset, showing similar results.

1. Introduction

Numerous applications rely on incomplete views of objects, often caused by occlusion or fragmentation. When the complete shape holds crucial significance, fracture assembly and completion emerge as valuable techniques for inferring the full structure from one or a few fragments. These applications span diverse domains, including heritage preservation, archiving, geometry processing, computer vision, and robotics. Akin to numerous other applications, recent works have solved these tasks with artificial intelligence models; however, their training is a significant bottleneck and source of challenges. For instance, acquiring extensive datasets of scanned fragments and their corresponding ground truth is arduous in terms of material resources and time. Instead, generating large datasets featuring fragmented rigid bodies holds promise for addressing this gap more effectively.

On the other hand, rigid body fragmentation and deformation have longstanding challenges in computer graphics and related fields such as fabrication and mechanics. While mechanics often focuses on accurate numerical models that reflect reality by integrating continuum dynamics, calculus, and differential geometry [1], the needs of realistic fragmentation differ significantly from those of real-world fracture assembly and completion. Moreover, physically based approaches, although capable of producing realistic results, are computationally expensive. Consequently, real-time applications like video games frequently compromise realism. They typically employ precomputed fracture patterns that can be dynamically adapted to simulate real-time impacts, yielding visually appealing results. However, pre-computations are time-consuming, especially for generating large fragment datasets. Additionally, realistic fragmentation processes may produce many fragments of

☆ This article was recommended for publication by Marco Attene.

* Corresponding author.
E-mail addresses: allopezr@ujaen.es (A. López), ajrueda@ujaen.es (A.J. Rueda), rsegura@ujaen.es (R.J. Segura), cogayar@ujaen.es (C.J. Ogayar), pnavarro@cenpat-conicet.gob.ar (P. Navarro), jmf@ujaen.es (J.M. Fuertes).

https://doi.org/10.1016/j.cag.2024.104104
Received 9 May 2024; Received in revised form 30 September 2024; Accepted 4 October 2024
Available online 15 October 2024

insufficient size for meaningful Machine Learning (ML) training. This further prolongs dataset generation until many adequately sized pieces are available.

This work introduces a simple pipeline for rapidly fracturing rigid bodies to generate huge fracture datasets. These pieces are provided in a voxel implicit format, though other derived representations can also be inferred: point clouds and triangle meshes. The Discrete Voronoi Chain (DVC) [2] is extended to leverage realism and response time while simulating other convenient effects. In addition, it is implemented in the Graphics Processing Unit (GPU) and can generate nearly five fragments per second. According to this, the main contributions of this research are the following:

1. We propose a GPU-based fragment generator that outputs implicit pieces in the form of voxels and thus can be directly fed into ML pipelines.
2. Our method enables configuring the number of fragments and their distribution, thus complementing physically-based fragmentation for generating much larger datasets.
3. Implementation of subtle voxel-level effects, such as erosion and impacts, that enhance the appearance of our fragments, despite not being entirely realistic.
4. We provide a detailed analysis of computation time and resource usage for generating a large dataset, comparing our approach to state-of-the-art work. Additionally, we integrate our fragments into an assembly pipeline to demonstrate their applicability.
5. We made available a dataset composed of 1M fragments from 1,052 archaeological artefacts [3], published in Github.

2. Related work

This section is guided by the state-of-the-art on brittle fractures, highlighting current trends and limitations. Then, the relevancy of generating fracture datasets is evaluated by exploring fracture assembly and completion works. Publicly available datasets are also revised for comparison.

2.1. Fragmentation

Decades of comprehensive research have delved into the fracture simulation of brittle objects. Unlike elastic and ductile materials, brittle materials exhibit a failure or fracture threshold close to their elastic limits. Consequently, these materials lack deformability. The revision in this section focuses on these specific materials.

The inception of fracture modelling finds its roots in continuum mechanics, employing parameters such as strain, stress, and plastic properties. In the realm of simulating the fracturing process, mass–spring methods [4] transform polygonal meshes into tetrahedral volumes. This category encompasses a diverse array of methods, ranging from finite element to boundary element techniques (FEM [5] and BEM [6], respectively), as well as mesh-less approaches. These methodologies are typically categorized based on their re-meshing requirements, degrees of freedom (DOF), and the equations they aim to solve. The growing significance of the Material Point Method (MPM) in physically-based simulations has also extended to the simulation of brittle fractures [7].

Traditional FEM approaches necessitate re-meshing during crack propagation due to the initial limitations in DOF imposed by the number of vertices. Although alternatives like the eXtended Finite Element Method (XFEM) [8] address this issue, FEM-based methods are often susceptible to instability and challenges in managing bifurcations. In contrast, the BEM employs a surface-mesh discretization, formulating governing equations based on boundary integral forms rather than volumetric ones. This method can be efficiently streamlined to reduce response time by swiftly estimating stress using level sets, albeit requiring pre-computation of stiffness matrices. Unlike its predecessors, MPM exhibits superior adaptability to topological changes inherent in deformable materials, making it a longstanding choice for fracture simulations, even in brittle materials [7]. Moreover, MPM is esteemed for its enhanced numerical stability.

Completely renouncing realism, Velić et al. [2] introduced Voronoi diagrams applied to polygonal meshes, aligning with geometry-based approaches. This algorithm incorporates Voronoi regions either as boundaries during region growth or as initial points for expansion. However, this stochastic method tends to yield regular and convex fragments that differ from realistic outcomes [9]. Similarly, Müller et al. [10] projected Voronoi diagrams into convex hulls of meshes for real-time destruction. They were able to recursively fragments to simulate impact points. However, they handle better meshes of polygons rather than only triangles, since the first are easier to cut. This approach is similar to the Cell Fracture plugin in Blender [9], although the latter does not project the Voronoi diagram over the convex hull, but on the bounding box. To mitigate this lack of realism, Oh et al. [11] introduced noise to the fragmented geometry. While this may sacrifice realism, it proves beneficial for efficiently simulating diverse scenarios using Voronoi cells as cutting planes [12]. Another avenue involves the application of Boolean operations to achieve fragmentation, parametrized by primitive shapes and their quantities [13]. Introducing more irregular patterns is feasible by employing primitives with noisy geometry [14] or incorporating level sets [15]. The latter approach is particularly effective over volumetric representations, enhancing crack details when voxel size is significantly reduced. Although categorized as non-physical, this category of methods is faster in generating extensive datasets.

In addressing the time-consuming nature of real-time fractures, an alternative strategy involves pre-computing fracture patterns that can dynamically adapt to real-time collisions. The activation of fractures during a specific impact often relies on criteria such as Euclidean distance or learning from examples, as demonstrated by Schvartzman and Otaduy [16]. Introducing a novel and advanced method, Sellán et al. [17] extends a linear subspace into which impacts can be projected. Employing an optimizer, this method identifies points prone to separating first into fragments. The process involves simplifying polygon meshes into a set of tetrahedrons, with pre-computed fracture modes. While it entails a substantial preparation phase, this approach yields rapid real-time fragmentation. The variability in the generated fragments is contingent on the number of pre-computed modes.

2.2. Shape assembly and completion

Shape assembly and completion problems address the incomplete knowledge of 3D shapes that may come from occlusion and insufficient resolution during its acquisition by scanning, or directly from parts that have been lost. Current state-of-the-art works approach these tasks using GAN, Encoder-Decoder architectures [18–20] as well as auto-decoders [13] and transformers [21]. For point cloud datasets, optimization is also possible by ensuring the final shape fits in the depictions of multiple views [22].

Polygonal meshes are not frequently used as input in ML models; instead, several alternative representations have been used, each having advantages and disadvantages. The most straightforward is the voxel model [19,23]. Over these, convolutional networks can be directly used, though dimensionality is a huge problem which limits the voxel space resolution. Point clouds are another common representation in the literature [20,21]. Otherwise, signed distance functions (SDFs) [24], signed directional distance functions (SDDFs) [25] and unsigned distance functions [26] are inferred from triangle meshes and point clouds.

Regarding the availability of quality datasets for training and testing the different solutions, the vast majority of previous studies use partial views of ShapeNet [27], ModelNet [19], BuildingNet [27], cultural heritage models [28], scanned objects [29], 3D vehicles [30] and SemanticKITTI [19,31]. Note that datasets such as SemanticKITTI also

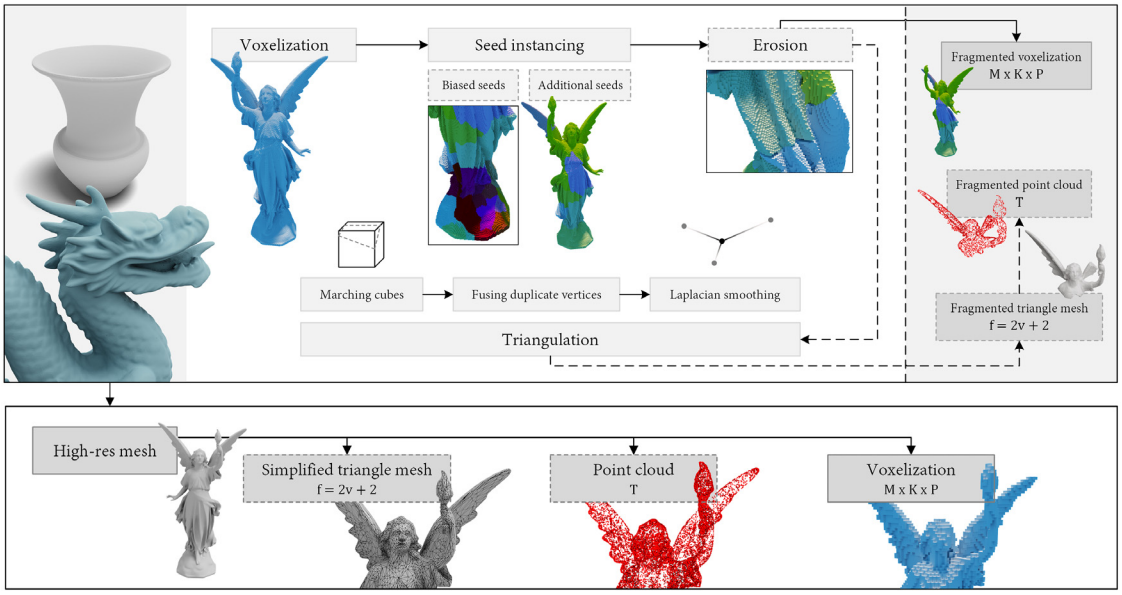

Fig. 1. Overview of the proposed dataset generation. High-res triangle meshes are voxelized, fragmented, eroded and optionally converted to point cloud and triangle mesh. The starting high-res mesh can be stored in the same formats. The number of points is illustrated as T, triangles as f, vertices as v and the voxelization size as $\mathcal{M}, \mathcal{K}, \mathcal{P}$.

provide pose data and are highly suitable for multi-view completion. Besides this, completion can be further guided by text feed [27,32]. Other works have constructed their datasets. For example, Lamb et al. [13] build their fractures over ShapeNet and Greek pottery from Koutsoudis et al. [33]. To this end, input models were modified by applying Boolean operations with regular shapes such as icospheres and cubes. These regular primitives may have some noise to provide more irregular cuts, and even be combined to create more realistic fractures [14]. More recently, Sellán et al. [34] built a dataset based on their fracturing method [17].

3. Methodology

This section explains the details of the fragmentation pipeline depicted in Fig. 1. The inputs of our pipeline are rigid bodies given by triangle meshes of any number of vertices and triangles. No preprocessing is required besides merging repeated vertices. The primary output is a voxel-based fragmentation, while others such as point clouds and triangle meshes can be optionally calculated from voxels.

3.1. Voxelization

We simulate mesh fracturing in voxel space since it provides a simpler data structure where fragments can be calculated by trivially accessing surrounding voxels. In addition, this data can be directly used to feed models [23] and enable simulating subtle effects, such as erosion. Other common representations in ML, such as distance functions, are derived from voxels and polygonal meshes. In comparison, tetrahedralization also requires a high resolution and provides hard boundaries unless these are subdivided and smoothed. The voxelization is performed as proposed by Ogayar-Anguita et al. [35] since it is a fast, simple and reliable method that also voxelizes the interior of the solid, unlike other studies that only voxelize the surface [36]. This work has its theoretical foundation in the point-in-tetrahedron inclusion test, and it is resolved using plane-sweeping in the geometry shader. Further details can be found in Ogayar-Anguita et al. [35], though a brief overview of this method is shown in Fig. 2.

The dimensionality of the voxel space, $V^{\mathcal{M}\times\mathcal{K}\times\mathcal{P}}$, is obtained by applying a fixed multiplier, η (voxel/metric unit), to the axis-aligned bounding box (AABB) of the mesh, rather than being a square, to lower the CPU/GPU memory footprint. However, it can be later padded to $\mathcal{O}^3$ on disk storage, with $\mathcal{O} \leftarrow \max \mathcal{M}, \mathcal{K}, \mathcal{P}$. The matrix is filled with `free`, a yet-to-determine value, or `empty` voxels that will never be occupied.

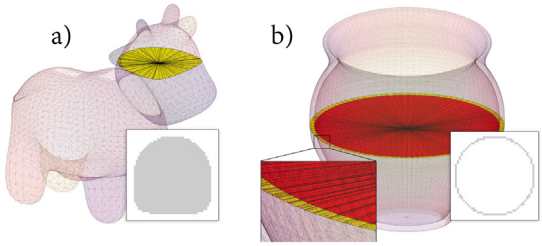

Fig. 2. Voxelization results obtained using the method of Ogayar-Anguita et al. [35] over (a) a solid mesh and (b) a mesh whose surface has some thickness. Red-coloured slices indicate that there is an even number of overlapping triangles, whereas green colouring indicates that there is an odd number. Hence, red-coloured areas do not have active voxels.

3.2. Fragmentation

The implemented fragmentation is based on the Discrete Voronoi Chain (DVC) [2]. It generates a list of Voronoi region centres, from now on referred to as seeds, that expand to their neighbourhood until every voxel belongs to a region. Note that, seeds are internally represented as 3D indices within the voxelization. Flooding in the limits between regions may lead to an incorrect Voronoi diagram, therefore distance checks to the region centres are required. However, disabling the distance checks also enables disrupting the Voronoi diagram to create concavities.

The first matter of discussion is how to sparse seeds. Trivially, they should be evenly distributed over `free` voxels. Despite Monte-Carlo samplers behaving more uniformly, we found the Mersenne Twister pseudo-random generator fair enough to sparse seeds. As a result, a set $\mathcal{S}$ composed of seeds is extracted, with $s_i \in \mathcal{S}$ being a vector such as $\{s_{i_x}, s_{i_y}, s_{i_z}\}$.

In the straightforward DVC approach, each `free` voxel from $\mathcal{V}$ is assigned the fragment ID i of the nearest seed s_i. Various distance functions can be implemented, from Euclidean or Chebyshev to Manhattan, ultimately leading to different fracture patterns. However, the main drawback of this method is that voxels are flooded without checking their connectivity to seeds. For instance, vessel handles are characterized by an inter-leaved, empty space, which is omitted during this procedure. A better solution is to assign each voxel by propagating

Fig. 3. Voronoi regions expanded after ten iterations using Moore (left) and Von Neumann neighbourhoods.

the fragment ID of each seed using a growing-region approach. Albeit slightly slower in the GPU, this procedure intrinsically tackles the previous connectivity drawback.

The growing procedure is implemented with a stack that is iteratively updated. In the GPU, two stacks with a length of $\mathcal{M} \times \mathcal{K} \times \mathcal{P}$ are required; while one is updated during an iteration, the other is checked and emptied. For each voxel of a stack, its `free` neighbours (yet to flood but inside the mesh) are set to the ID of the current voxel. Then, they are appended to the stack of the following iteration. Neighbourhoods may have different shapes; for instance, the two most common are Von Neumann (six faces, as in a cube) and Moore, which also integrates the corners (twenty-six voxels). They can be easily integrated into the GPU as buffers containing a variable number of 3D vectors. The aftermath of different flooding neighbourhoods is depicted in Fig. 3.

3.2.1. Additional seeds

Although real-world fragments are predominantly convex, they also include more intricate cuts leading to concave shapes. This feature can be smoothly incorporated into our fragmentation pipeline by generating additional seeds. These are intended to push the fragment limits from several points, thus creating more complex shapes. Note that, new seeds are not assigned different IDs, but the value of the closest initial seed using a distance function such as Chebyshev. This new seed set is represented as $M = \{m_1, m_2, \ldots, m_k\}$ with m_i belonging to any region from S.

A drawback of this approach is that it can create isolated regions if two seeds are too far and the growing regions cannot merge before the region-growing procedure stalls. Therefore, these isolated regions must be identified and removed (i.e., their voxels must be reset to `free`) to allow a new assignment by region growing from neighbouring fragments. Notice that isolated fragments could be attached to a surrounding fragment; however, the implemented approach leads to more visually appealing pieces.

A simple solution to detect isolated fragments is to use disjoint sets (see Fig. 4). In this approach, a set of voxels is considered isolated if its ID matches the ID of another set with which it is disconnected. Our GPU implementation of this test is mainly supported by the voxel data type: `uint16_t`. This amount of bits can be split into two parts to save (1) the seed ID in the most significant 16 bits (identifier of the random point that flooded a voxel) and (2) the fragment ID in the least significant bits. That is, additional seeds have different IDs but share their fragment ID. Then, the flooding procedure is repeated by exchanging the seed ID with surrounding voxels from the same fragment ID, hence storing the minimum seed ID. As a result, the same voxel could be processed more than once if a lower seed ID is encountered. In practice, the voxels to be processed are handled using one buffer for reading and another for writing. With this approach, disjoint sets are groups of voxels whose seed ID is not the minimum possible ID for their fragment, i.e., zero. Once detected, their voxels are freed and the fragmentation process is restarted.

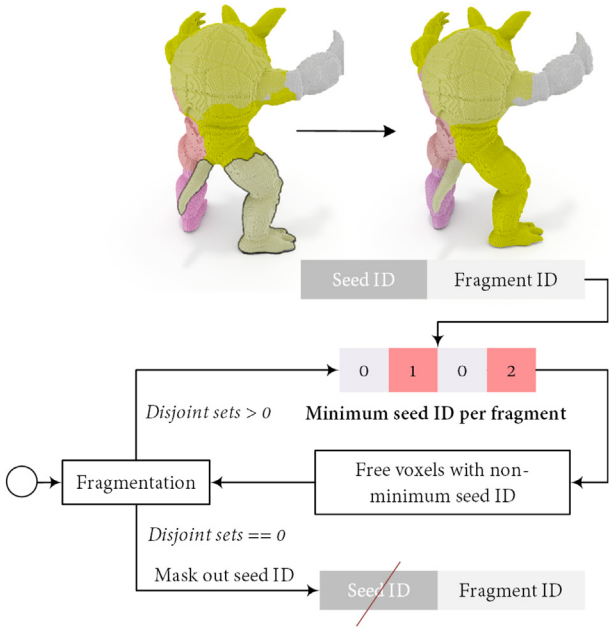

Fig. 4. Fragmentation procedure, during which isolated regions are checked. On the top side, the Armadillo model is displayed with isolated regions (left) and fixed (right).

3.2.2. Biased seeds

Another benefit of randomly sampling the voxelization is that it can be trivially biased towards specific parts to obtain more fragments. This can be viewed as a specific case of additional seeds. Rather than placing seeds uniformly, we can bias them towards one or more specific locations. Moreover, any alternative random distribution generating values in $[0, 1]$ can be used, leading to different volume histograms, as illustrated in Fig. 5. Biased seeds emulate impacts focused on a surface point (high bias) or a dispersed fragmentation that could rather correspond to squashing (spread over a larger area). Other tools such as Cell Fracture [9] emulate this effect by recursively subdividing previous fragments. In our work, biased seeds are implemented through Gaussian random distributions configured with a bias factor, S. The higher the bias, the less the seeds are scattered.

Fig. 5 illustrate the differences between additional seeds, which are uniformly sparsed to create more intricate shapes, and biased seeds, which concentrate new seeds in a few surrounding voxels to recreate collisions. This feature holds particular relevance to simulating fracture patterns guided by strained points. For instance, it can simulate interactive impacts (Fig. 6) by scattering seeds over an impact point. Simulating incomplete findings lacking small fragments could be even advantageous, as occurs in archaeology. Nevertheless, biased pieces tend to be smaller, thus more likely to be discarded according to their volume. Hence, this feature is more relevant for interactive fragmentation than for generating datasets.

3.2.3. Erosion

Previous voxelized fragments have a perfect fit, i.e., joining all of them produces the starting voxelization. However, the breakage of artefacts can result in tiny shards detaching, often going unnoticed or being discarded during excavations. In addition to small fragments, larger shards may also remain unrecovered, as shown in Fig. 8. Furthermore, we can also find fragments with smoothed edges due to erosion or abrasion caused by contact with soil, especially in wet environments [37]. The loss of larger shards can be simulated using the previous bias seeds, while the erosion of details and the loss of tiny shards can be replicated with erosion techniques. These factors contribute to misfitting shards, making it difficult to fully reconstruct the original artefact.

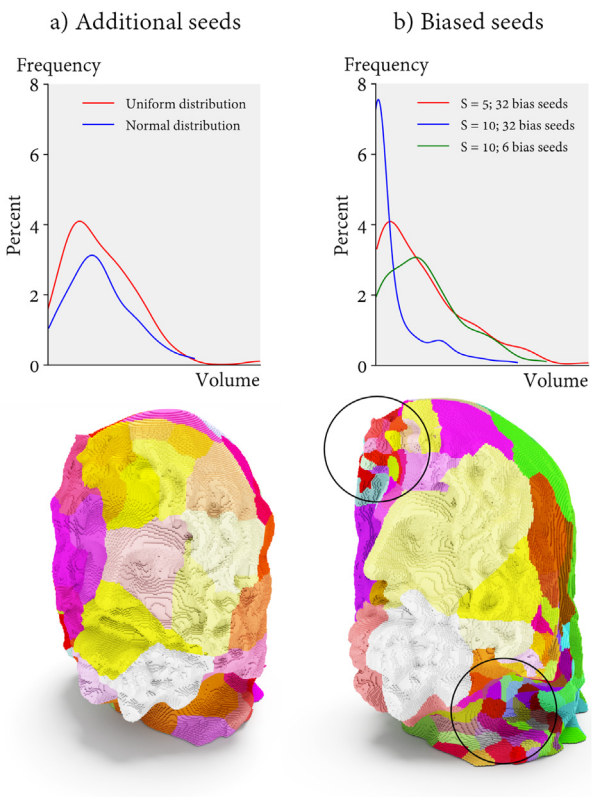

Fig. 5. On the left side, ten regions grow with no bias using two different random distributions, resulting in larger pieces. In the right image, eight regions grow with two bias points (circled) emulated with different numbers of seeds and spreading (S).

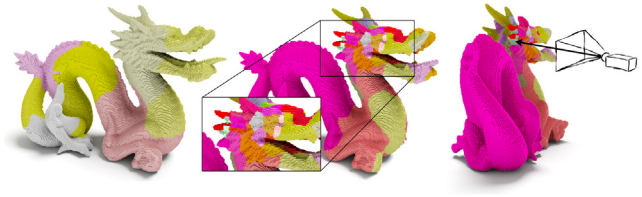

Fig. 6. Initial fragmentation and on the right side, the fragmentation produced by an impact interactively selected over the dragon's head.

In this regard, voxelization helps to integrate erosion through 3D erosion kernels. Different convolutions (C^3), from squares, circles or crosses, can be utilized for sampling the neighbourhood of a voxel. Following this procedure, voxels are eroded (discarded and set to `free`) whether the summed convolution is not equal to the number of voxels activated in C^3. In practice, this is hard to control and parametrize as voxels not in the fragment boundaries may also be eroded. To tackle this, boundary voxels (in contact with a `free` voxel) are detected from each fragment to create a list of erasable voxels. We define a boundary voxel as one with direct connectivity with at least one `free` voxel. We emphasize this as this principle will also be used later.

The erosion is further parametrized by establishing the number of iterations during which the voxelization is eroded. In addition, it is possible to relax or harden the erosion by requiring less activations from a given convolution. We also introduced random checks to avoid smoothed boundaries; instead, voxels are convoluted with a probability in $[0, 1]$. If the probability is one, every voxel is convoluted. It is configured with a threshold, where a lower value results in gentler erosion. Fig. 8 shows the erosion results in triangle meshes, compared to another set of non-eroded shards, whereas Fig. 7 displays the erosion kernels.

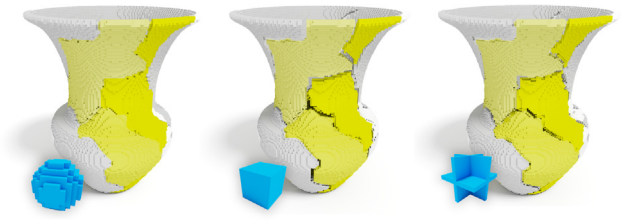

Fig. 7. Three erosion kernels and their effects. Voxels were convoluted and freed if less than 50% of the expected voxels were active.

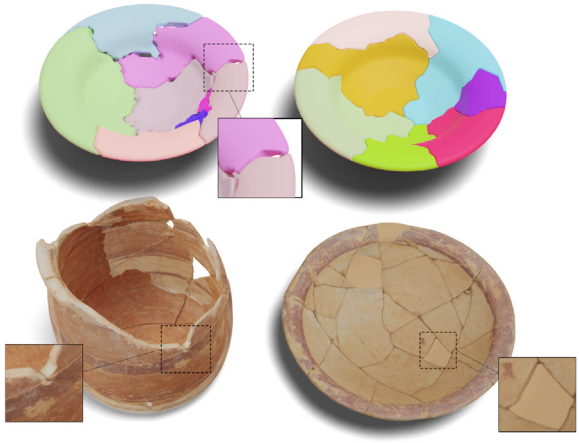

Fig. 8. Comparison of eroded (left) and fitting fragments (right) on the top side. Fitting pieces are obtained by weighting the Laplacian operator to zero in the boundary voxels. Below, two real-world vessels are compared: (1) an incomplete vessel with notable erosion cues, and (2) an incomplete vessel artificially reconstructed by replicating missing pieces.

3.3. Mesh generation

The primary aim of this work is to simulate fragmentation within voxel space. However, additional representations can be derived from this, including triangle meshes, point clouds, and SDFs. We generated triangle meshes by implementing the marching cubes algorithm in the GPU. The main bottleneck is to transfer large triangle meshes back from VRAM (Virtual Random Access Memory), with $[5, 12]$ triangles being returned per voxel. This drawback is partially suffocated by fusing duplicated vertices in the GPU. First, points are encoded as 3D Morton codes of 30 bits (10 bits per coordinate) which can be efficiently sorted with the Radix sort algorithm ($\mathcal{O}(nk)$ complexity, with n being the number of points and $k = 30$ bits). Once sorted, contiguous values which satisfy distance$(p_1, p_2) < \epsilon$ are merged. In the GPU, this test is implemented by finding the indices of the first points that are identical to the following points. This also involves reallocating points via prefix-scan and modifying the triangle indices.

Another drawback arises from the staggered appearance of resulting meshes. This is particularly evident in the curved surfaces of ceramic vessels. To alleviate this, we perform several iterations of the Laplacian operator, as depicted in Fig. 10. However, as a side effect, this also removes the sharp details at the edges of the fragments. We handled this by weighting the smoothing operator in faces within boundary voxels with a lower factor, even zero. Note that border voxels were previously identified for erosion and this information is still present in this stage. Finally, meshes can be optionally decimated with quadric error metrics [38] to reduce the storage footprint, as illustrated in Fig. 9.

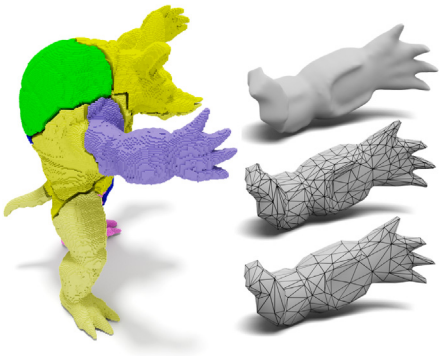

Fig. 9. Marching cubes over a fragment of the armadillo. The mesh was iteratively simplified with quadric decimation.

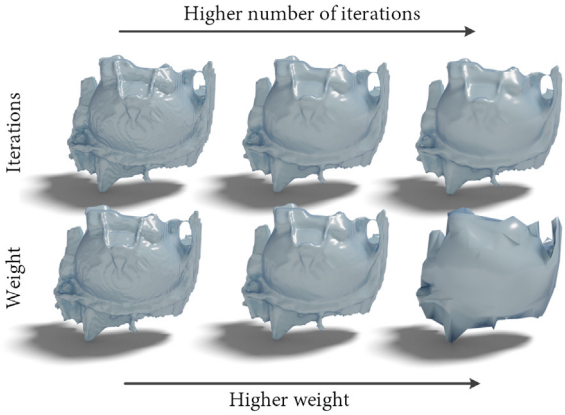

Fig. 10. Triangle meshes obtained by using different Laplacian smoothing factors. On the top side, the number of iterations varied from $0.01 \cdot \max(\mathcal{M}, \mathcal{K}, \mathcal{P})$ to 0.08. On the bottom side, the weight factor goes from 0.1 to 0.6.

Fig. 11. Sampling of the Armadillo mesh using the Mersenne Twister pseudo-random sampler (left) and Halton sampling (right). On the left point cloud, the right ear of the model is missing.

3.4. Point cloud generation

Point clouds are another successful representation in DL that can be inferred from triangle meshes. They are composed of k points sampled from triangles according to their area and the overall summed-area. We considered that at least one point was to be extracted from every triangle. Then, the generated points are shuffled and subsampled if the number of points is higher than k. Unlike seed instancing, where uniformity is not required, the random distribution in this step holds particular significance, as it could potentially omit relevant parts with low k. Fig. 11 compares the sampling with the Mersenne Twister pseudo-random generator and a Halton sampler.

4. Experimentation and results

The experiments are focused on measuring the memory footprint and efficiency of the parallel fragment generation in the form of voxels, meshes and point clouds. Firstly, it is compared against Breaking Good [17], a realistic fragmentation method which was recently published. It is based on pre-fractured modes that do not provide direct control over the number of fragments; instead, fragments are determined by the projected impact. We have also integrated the Cell Fracturer plugin from Blender [9], which utilizes Voronoi diagrams as in [10,15]. Measurements were performed on a PC with Intel®Xeon®Silver 4210R (2.4GHz), 176 GB RAM, GeForce RTX 3070 GPU with 8 GB VRAM, and Windows 10 OS. The proposed pipeline is implemented in C++23 using OpenMP (Open Multi-Processing) for parallel processing. We use OpenGL 4.6 for rendering and GPGPU (general-purpose computing on GPU). Note that OpenGL storage buffers are limited to 2 GB regardless of the VRAM capacity. Our work employs voxelizations with low resolution since these are appropriate for convolutional operations, and besides this, the footprint is diminished by quantization (with the data type being `uint16_t`). Still, the VRAM is rapidly outgrown by buffers allocated for solving marching cubes. Hence, our experimentation will be clamped to 128^3 voxels.

Overall, we have considered two ways the fragments can be fed into the network: (1) storing the whole dataset in an out-of-core fashion, and (2) generating batches consumed by the model in the GPU. However, the second approach has not been annotated in the following experiments since it has no storage footprint.

4.1. Dataset preparation

The generation of fragment datasets has three results: voxels, triangle meshes and point clouds. Additionally, the resulting dataset should have a similar number of samples for each number of pieces. To this end, we fragmented every item in $[2, 10]$ pieces during a number of iterations interpolated from $[15, 5]$. We tested our pipeline over a dataset of 1,052 artefacts comprising different kinds of Iberian ceramic vessels [3]. These were generated as surfaces of revolution from profiles reconstructed by archaeologists. Other parts, such as their handles, were later included under their supervision. These models oscillate between 500k and 3M vertices, and between 200k and 1.6M triangles. The dimensionality of voxelizations was downscaled according to the object's bounding box. In this manner, the triangles obtained from marching cubes have a similar scale to the starting mesh. Point clouds were generated by sampling 1000 points, similar to previous fracture assembly works [34], and triangle meshes were stored with no quadric decimation when possible. The pieces were smoothed using the Laplacian operator: the boundary voxels were smoothed with a weight of 0.2, whereas the rest were smoothed with a weight of 0.9. It was prolonged during $0.048 \cdot k$ iterations, with k being the maximum dimension of the voxelization. In this manner, the number of iterations grows as voxelizations get larger.

The instructions to download our fragment dataset are published online. The details of the dataset are shown in Table 3 for two different voxel resolutions: 64 (for comparisons) and 128 (released version). Two variants of our dataset have been released: (1) 1M pieces of 1,052 artefacts in every possible binary file format, and (2) 200K pieces from only 200 artefacts, stored as plain `.obj` and `.ply` (~25 GB). The latter is released to offer an overview of our dataset.

4.2. Computational resources

Besides the storage footprint addressed in Section 4.4, we investigated the computational resources used during the dataset generation, in contrast to Breaking Good. They released their code in Python and we adapted it to behave similarly to our fragmentation (grid of size 128, simplification to 5,000 tetrahedra, 20 fracture modes and stops

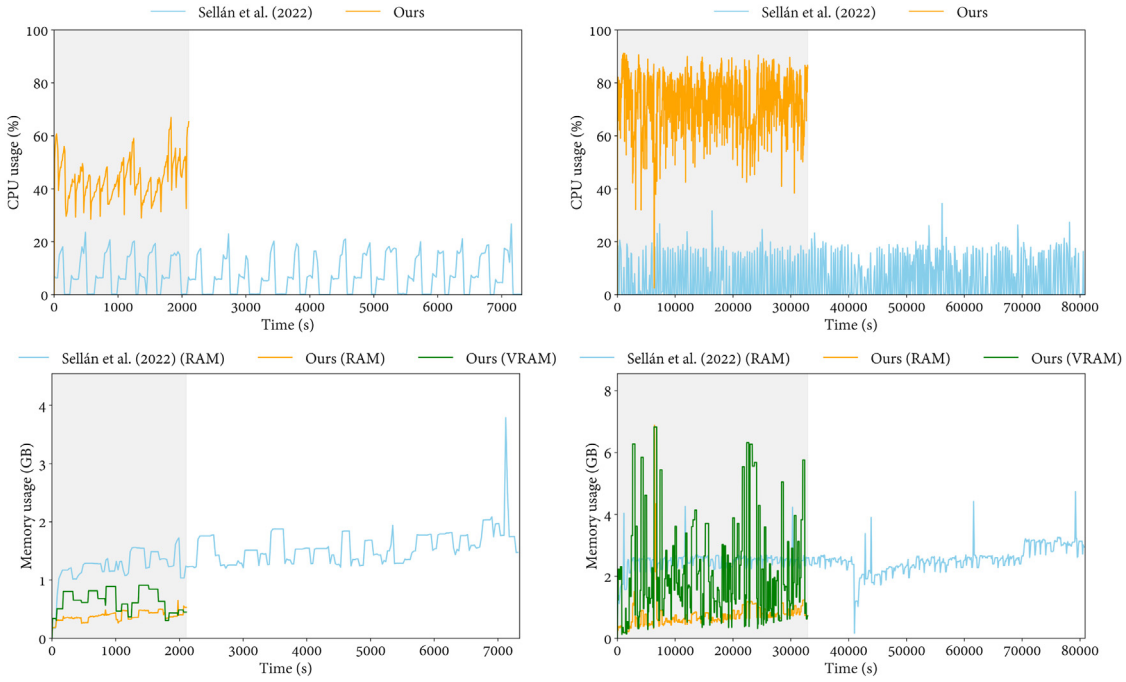

Fig. 12. Usage of CPU, RAM and VRAM resources while generating a dataset with Breaking Good fragmentation and our method. On the left side, the results are obtained by fragmenting 20 random vessels, while the results on the right side were extracted from 200 artefacts of the Thingi10K dataset.

for every model when 1000 fragments are generated). Their results are written as compressed binary files (`.npy`). On the other hand, the Cell Fracture plugin was not included in this experiment since it stalled with some of the meshes in Thingi10K. However, it managed to fragment the twenty vessels in 1.91 h (slightly better than the performance of Sellán et al. [17]). The performance on this small dataset also showed minimal CPU usage and no utilization of the GPU.

Fig. 12 illustrates the overall response time and resource utilization over two mesh collections: a small subset of 20 vessels from our dataset and 200 artefacts from the Thingi10K dataset. Our approach not only exhibits parallelism in the GPU but also in the CPU, optimizing resource utilization across both components. Additionally, we minimize virtual memory usage and optimize performance by reusing GPU buffers instead of allocating new ones for each step. The data type of GPU and CPU buffers was also adapted to have a reduced footprint while keeping the required capabilities (e.g., using voxels of 16 bits that still enabled instancing up to 2^8 seeds). It is important to note that our approach may result in a slightly higher RAM footprint due to the mapping of VRAM to RAM for data transfers. This mapping conceptually resembles an open stream connecting the GPU and CPU. Finally, we must highlight that the pipeline is notably optimized for dataset generation, as buffers are allocated only once per mesh, and the same applies to stages like voxelization. Note that the larger dataset also had a higher variability of voxelization dimensions, leading to higher usage of computational resources. As the meshes get more detailed, such as in Thingi10K (right side), the compared work seems to struggle even when computing impacts.

Fig. 13 groups the measured response time by events, showing the fracturing and impact projection stages are far more time-consuming than our core events. Yet, reads notably impact the time measured from Sellán et al. [17]. For this reason, load and storage stages are dashed and omitted in the summed time below. The second summation, however, includes these stages. Furthermore, we omitted the data type conversion stage since it transforms voxels into triangle meshes and point clouds, which are not required for implicit datasets.

4.3. Convexity

Although realism is difficult to quantify, an overabundance of convex pieces is a common indicator of unrealistic shapes, as seen in Voronoi diagrams. However, measuring convexity in 3D is not straightforward. For instance, shards from Voronoi diagrams are not perfectly convex due to hollowed objects and complex shapes, but their convexity is expected to be closer to one. To assess fragment convexity, we measured the distance of mesh vertices from the convex hull. Rays were cast from each vertex along the surface orientation, and the collision distances were averaged across all vertices.

This test was conducted on 10,000 fragments from the three approaches being compared, using Thingi10K artefacts. As shown in Fig. 14, fragments generated by Cell Fracture were the most convex, as expected, followed by Breaking Bad and our method. We also evaluated several configurations of our dataset, including varying numbers of additional seeds. As the number of seeds decreases, the fragments become more convex. Additionally, the total number of fragments significantly impacts convexity. It tends to increase as the fragment count rises, as fragments become more similar to their convex hull, with less space for complex shapes and hollowness. However, there is a noticeable gap between Breaking Bad and our dataset when the fragment count is lower. We hypothesize this is due to Sellán et al. [17] showing a higher level of detail on fracture boundaries, which are relatively close to the convex hull. As the number of vertices per fragment decreases (i.e., with more fragments), the difference in this metric diminishes. It is also worth noting that, even with fewer seeds, our results remain far from those of Voronoi diagrams. This is because we modified DVC to allow multiple fragments to push boundaries without considering proximity to seed locations. As a result, the outcome is not a Voronoi diagram even without additional seeds.

4.4. Compression

Large datasets can scale up to a few TB if not properly handled. Hence, different file formats were checked to minimize the storage footprint. Notice that a few have a lower footprint at the expense of requiring some preprocessing when loading. Experimentation has been

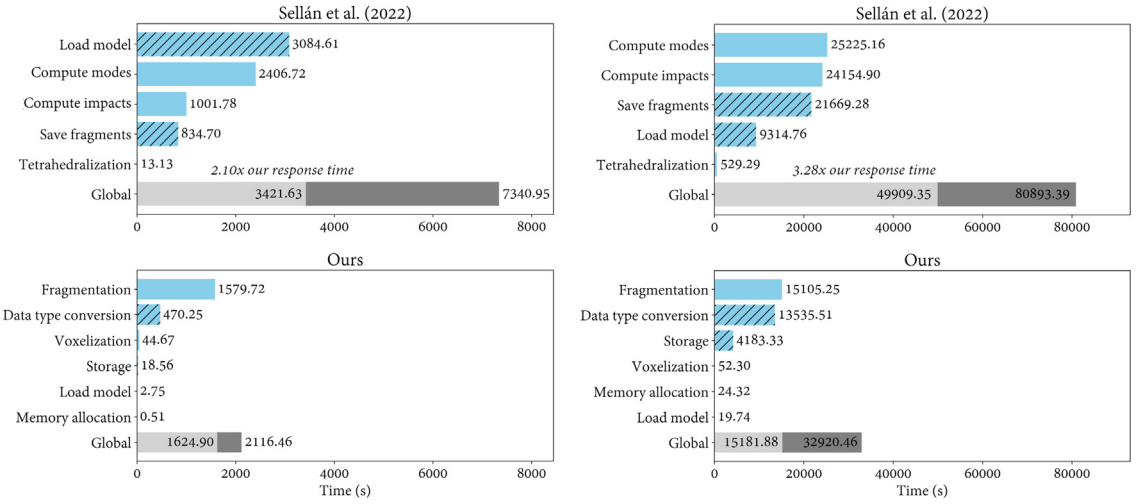

Fig. 13. Response time reported in every pipeline step. Steps are vertically ordered according to the response time. Global time depicts summed time without accumulating dashed steps (light grey), and including them (dark grey).

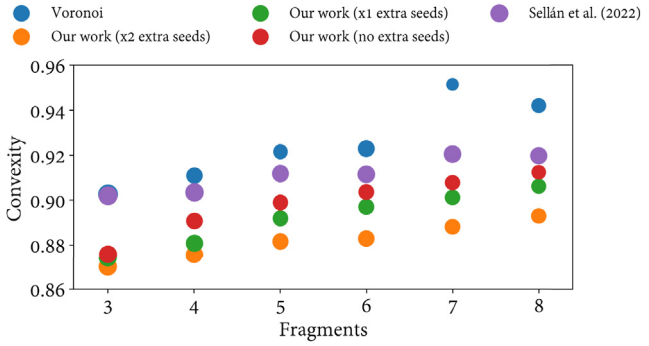

Fig. 14. Average convexity of the fragments produced by the three compared approaches, based on the number of splits. The number of seeds was set according to the number of required fragments (×1, ×2). The size of the circles represents the standard deviation.

narrowed to [64, 128] voxels since formats such as `.vox` are notably heavier and therefore are not recommended for dataset generation. The following formats have been contemplated:

- Point clouds: PLY (binary), XYZ (human-readable) and compressed binary (compression with the Point Cloud Library (PCL) package, 1 mm^3 resolution).
- Voxel: raw binary (uncompressed), Run-Length Encoding (RLE) binary [39], quad-stacks binary [40] and MagicaVoxel.
- Mesh: STL, OBJ and binary mesh (using the internal format of our program).

Table 1 shows the footprint if files are in their original format (in parentheses) or zipped. Formats such as the quad-stack binary are lighter when uncompressed, presumably because they present a significant compaction of the raw data. However, other simpler algorithms such as RLE have a lower footprint when compressed as zipping contemplates repeated data not addressed by the algorithm. According to this, the dataset was stored using RLE (voxel), binary PCL compression (point cloud) and binary mesh formats.

4.5. Evaluation of applicability

One of the main concerns during the data type transformation is whether significant changes in the geometry could worsen the performance of trainable models. Factors such as the voxelization LOD have their weight in the loss of precision, as do the Laplacian smoothing factors. Yet, adequately trained models should be able to generalize despite fragments having small changes in their geometry, thus attending to global features rather than local features. The voxel models from this pipeline were used in a previous shape completion work [23], whereas point clouds and polygonal meshes were not yet checked. We used multiple datasets, including (1) the artefact dataset of Breaking Bad (200 artefacts, 74,698 pieces), (2) a subset of our vessel dataset (200 artefacts, 70,000 pieces) and (3) a mix of both (200 artefacts, 71,632 pieces). The latter was generated by removing half of the artefacts from (1) and including half of (2) while maintaining the test set (25% of the available meshes). All these datasets were tested using the DGL (Dynamic Graph Learning) [41] fracture assembly model. Fragments are processed before training by sampling them, zeroing their location and randomly rotating them. On the other hand, the performance of DGL was measured by annotating (1) the translation and rotation Root Mean Square Error (RMSE) and Mean Absolute Error (MAE), (2) the Chamfer distance between the starting point clouds and (3) part accuracy, i.e., the percentage of fragments whose Chamfer distance is below $\varsigma = 0.01$.

We used the Breaking Bad dataset as a baseline for comparison, as realistic fragments are hypothesized to be optimal for training machine learning models. While we believe that larger datasets would benefit the training process, this experiment was designed to assess whether our generated fragments negatively impact assembly metrics. Therefore, we did not test larger datasets but instead worked with one comparable in size to the Breaking Bad dataset.

Table 1
Storage footprint of 200 vessel models broken into [2, 10] fragments, using 15 iterations for 2 fragments and 5 for 10 fragments. The footprint after and before zipping files is reported out and in parentheses, respectively.

	64	128
Binary raw grid	0.09 GB (8.59 GB)	0.57 GB (68.75 GB)
MagicaVoxel	5.09 GB (17.19 GB)	44.88 GB (137.52 GB)
RLE	0.08 GB (1.99 GB)	0.32 GB (8.10 GB)
QuadStack	0.21 GB (0.92 GB)	0.65 GB (2.94 GB)
PLY	4.34 GB (8.22 GB)	4.38 GB (8.22 GB)
XYZ	3.93 GB (9.13 GB)	3.96 GB (9.12 GB)
Comp. binary	0.97 GB (0.98 GB)	1.00 GB (1.04 GB)
OBJ	4.98 GB (13.89 GB)	5.80 GB (16.42 GB)
STL	17.37 GB (79.82 GB)	16.40 GB (94.61 GB)
Binary	4.23 GB (16.92 GB)	4.85 GB (19.84 GB)

Table 2
Metrics obtained by training the DGL fracture assembly model over the artefact dataset and our fragmented vessels.

	Breaking Bad	Ours	Breaking Bad & Ours
T. MAE ($\times 10^{-2}$)	12.26	15.82	13.31
R. MAE (°)	75.36	68.39	73.04
T. RMSE ($\times 10^{-2}$)	14.71	18.88	15.84
R. RMSE (°)	86.58	77.56	84.15
Chamfer d. ($\times 10^{-3}$)	25.78	11.38	22.60
Part accuracy (%)	4.05	4.19	5.32

Table 3
Details of the released fragment dataset (up to 128 voxels in any dimension), and another version with a lower number of voxels (64). # is used to abbreviate 'number of'.

	Up to 64^3	Up to 128^3
#Voxel files	187,257 (0.77 GB)	187,257 (2.86 GB)
#Triangle mesh files	1,040,428 (115.55 GB)	1,040,428 (432.35 GB)
#Point cloud files	1,040,428 (4.32 GB)	1,040,428 (4.32 GB)
#Models	1,052	
Response time	0.90 days	2.42 days
#Fragments/s	13.33	4.96
Average #vertices	4,490	17,933
Average #faces	9,084	36,703

As reported in Table 2, both models are far from producing perfectly fitted fragments, though our dataset obtained similar results to those from the Breaking Bad fragments. Even when both datasets are fused, the majority of metrics are similar to the baseline *Breaking bad* training. Therefore, incorporating our fragments does not seem to disrupt the training of the model. However, the results show that there is still room for improvement in models that work with partial views of objects. It is important to note that DGL was originally released as a model trained on semantic parts from PartNet, rather than on fractured objects.

4.6. Visual comparison

Fig. 15 illustrates fractures produced by the methods under comparison, including Cell Fracture. Notably, the code released by Sellán et al. [17] lacks control over the number of resulting fragments, resulting in a lower count. This does not necessarily translate to larger fragments, as evidenced by the plate artefact broken into a few thin pieces. Conversely, our methodology tends to generate fragments that, while geometrically intricate, are not completely realistic. Thus, there is a potential synergy between both approaches for training ML models. Additionally, Sellán et al. [17] introduces rough boundaries by decimating the edges of the fragments. In contrast, our technique leads to fragments with smoother edges, due to utilizing marching cubes, smoothing operators and erosion, as illustrated in the bottom images of Fig. 15. Finally, Cell Fracture produces notably convex pieces. Although the Blender implementation is not as efficient, this method could be potentially more rapid than previously compared approaches. However, some subtle effects are harder to integrate, such as erosion. Therefore, our work is halfway between a simple Voronoi diagram and realistic fragmentation.

5. Conclusions and future work

In this paper, we have presented an efficient pipeline for fragmenting 3D meshes and generating huge voxel datasets. It is implemented in the GPU operating volumetric models to rapidly fragment models based on the Discrete Voronoi Chain. We improved this unrealistic method to introduce effects such as erosion, concavities and impact simulations. Over this method, we generated a large dataset of more than 1M fragments (~450 GB) from 1,052 archaeological artefacts. Since it is implemented in the GPU, the resulting pieces can even be used on the fly to feed ML models without intermediate storage. We tested our pipeline by (1) comparing resource usage and response time with previous work, (2) testing the convexity of our fragments against those of a simple Voronoi diagram, (3) evaluating metrics of popular ML models for fracture assembly and (4) comparing visual results. Our method was proven more efficient, and the released dataset was successfully used for fracture assembly tasks. Despite successfully integrating our data into DL tasks, we emphasize the synergy between our method and realistic approaches, such as Sellán et al. [17], for generating larger fragment datasets.

While this work primarily focuses on voxel-based fragments, other common data representations have been derived from them. However, these representations exhibit greater geometric inaccuracies due to converting from voxel to mesh. Therefore, we present these optional representations to trivially integrate them into other assembly and completion solutions based on triangle meshes and point clouds.

As a future work, the efficiency of the pipeline could be substantially enhanced by exploring alternatives to marching cubes, which currently represent the primary bottleneck in terms of both storage footprint and response time. Moreover, thorough assessments regarding the impact of various factors on the trained models should be conducted, including but not limited to the number of fragments, degree of convexity/concavity and voxel resolution.

CRediT authorship contribution statement

Alfonso López: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Antonio J. Rueda:** Writing – review & editing, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Rafael J. Segura:** Writing – review & editing, Visualization, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Carlos J. Ogayar:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Formal analysis, Conceptualization. **Pablo Navarro:** Writing – review & editing, Validation. **José M. Fuertes:** Writing – review & editing, Validation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This result has been partially supported by the Spanish Ministry of Science, Innovation and Universities via a doctoral grant to the first author (FPU19/00100).

Appendix

Dataset details

Fig. 17 illustrates the distribution of fragments regarding the number of triangles and vertices. In this manner, we can observe that fragments obtained by breaking artefacts into ten fragments have volumes above 10%, which is a frequently used threshold to filter out fragments that are otherwise difficult to classify and hardly relevant during training. For instance, the compared fragmentation work [34] firstly tested the deep learning models over every piece, whereas a later released version was decimated by removing those pieces whose normalized volume felt under 2.5%. Finally, Fig. 16 provides an overview of our triangle mesh dataset.

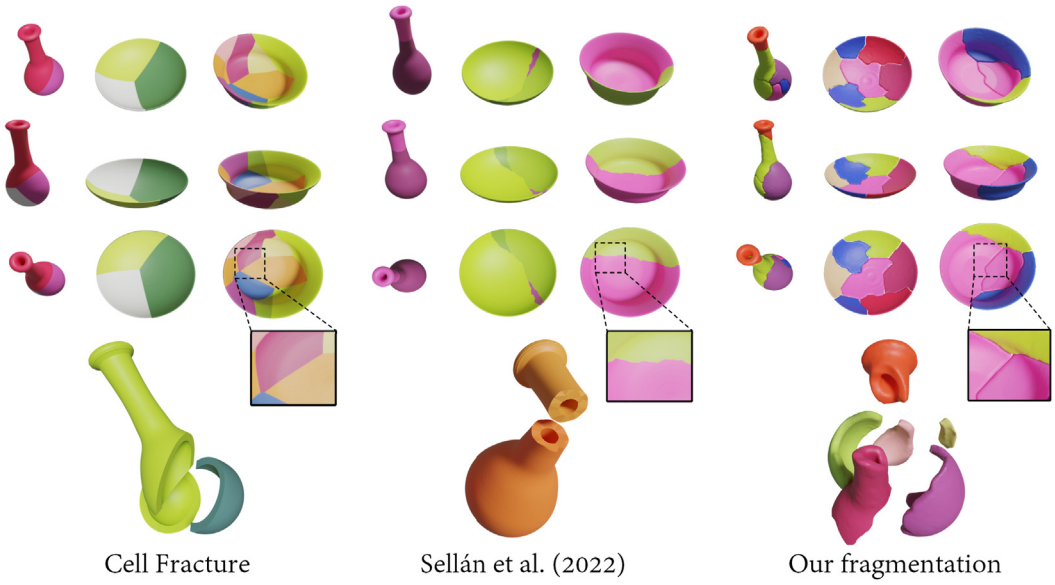

Fig. 15. At the top: front, back and top view of fragments generated by the Cell Fracture plugin, Sellán et al. [17] and our method. On the bottom side, the fragments of a single vessel are displayed.

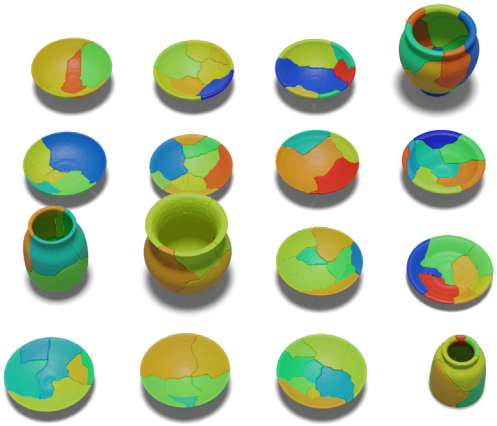

Fig. 16. Overview of sixteen fragmented vessels.

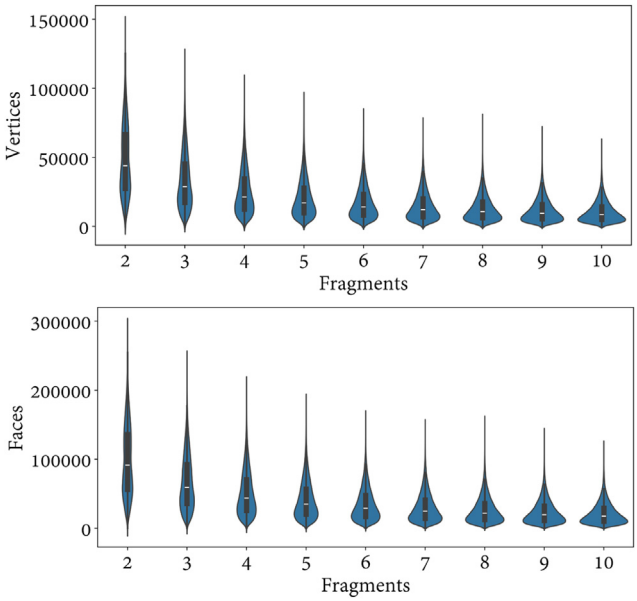

Fig. 17. Distribution of the number of faces and vertices in our fragment dataset.

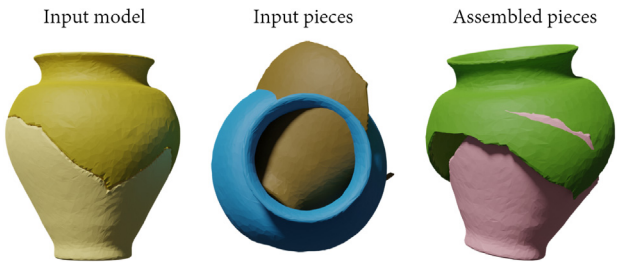

Fig. 18. Pieces assembled with the DGL network.

Fracture assembly: training details

Fig. 17 shows that triangle meshes are large, especially when artefacts are broken into less fragments. Therefore, training deep learning models over these kinds of meshes is time-consuming. A widespread approach to save RAM and VRAM is to sample meshes on the fly to extract another data representation (e.g., point clouds). However, this is a particularly intensive task for large triangle meshes. For this reason, we implemented the quadric decimation that narrows the number of faces and vertices. Sellán et al. [17] provide a low-resolution dataset of 74,698 pieces over which we trained in barely ~1 day using an NVIDIA A600 GPU. In comparison, our dataset comprises 200 artefacts and 70,000 pieces with up to 10,000 faces. The training over both datasets was prolonged for approximately one day using 200 epochs, and the top-1 checkpoint was used to assemble the pieces in the test dataset, as shown in Fig. 18. It shows that the current state-of-the-art is far from the ideal assembly, and therefore, we hope our dataset contributes to this challenging problem.

Data availability

The data is already shared in the manuscript.

References

[1] Muguercia L, Bosch C, Patow G. Fracture modeling in computer graphics. Comput Graph 2014;45:86–100. http://dx.doi.org/10.1016/j.cag.2014.08.006, URL https://www.sciencedirect.com/science/article/pii/S0097849314000806.

[2] Velić M, May D, Moresi L. A fast robust algorithm for computing discrete voronoi diagrams. J Math Model Algorithms 2009;8(3):343–55. http://dx.doi.org/10.1007/s10852-008-9097-6.

[3] Lucena M, Fuertes JM, Martínez-Carrillo AL, Ruiz A, Carrascosa F. Classification of archaeological pottery profiles using modal analysis. Multimedia Tools Appl 2017;76(20):21565–77. http://dx.doi.org/10.1007/s11042-016-4076-9.

[4] O'Brien JF, Bargteil AW, Hodgins JK. Graphical modeling and animation of ductile fracture. ACM Trans Graph 2002;21(3):291–4. http://dx.doi.org/10.1145/566654.566579, URL https://dl.acm.org/doi/10.1145/566654.566579.

[5] Koschier D, Lipponer S, Bender J. Adaptive tetrahedral meshes for brittle fracture simulation. The Eurographics Association; 2014, http://dx.doi.org/10.2312/sca.20141123.057-066, URL https://diglib.eg.org:443/xmlui/handle/10.2312/sca.20141123.057-066. Accepted: 2014-12-16T07:33:42Z ISSN: 1727-5288.

[6] Hahn D, Wojtan C. Fast approximations for boundary element based brittle fracture simulation. ACM Trans Graph 2016;35(4):104:1–104:11. http://dx.doi.org/10.1145/2897824.2925902, URL https://dl.acm.org/doi/10.1145/2897824.2925902.

[7] Fan L, Chitalu FM, Komura T. Simulating brittle fracture with material points. ACM Trans Graph 2022;41(5):177:1–20. http://dx.doi.org/10.1145/3522573, URL https://dl.acm.org/doi/10.1145/3522573.

[8] Chitalu FM, Miao Q, Subr K, Komura T. Displacement-Correlated XFEM for simulating brittle fracture. Comput Graph Forum 2020;39(2):569–83. http://dx.doi.org/10.1111/cgf.13953, URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13953. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13953.

[9] Blender. Cell fracture: Blender 4.1 manual. 2024, URL https://docs.blender.org/manual/en/latest/addons/object/cell_fracture.html.

[10] Müller M, Chentanez N, Kim T-Y. Real time dynamic fracture with volumetric approximate convex decompositions. ACM Trans Graph 2013;32(4):115:1–115:10. http://dx.doi.org/10.1145/2461912.2461934, URL https://dl.acm.org/doi/10.1145/2461912.2461934.

[11] Oh S, Shin S, Jun H. Practical simulation of hierarchical brittle fracture. Comput. Animat. Virtual Worlds 2012;23. http://dx.doi.org/10.1002/cav.1443.

[12] Zafar NB, Stephens D, Larsson Ma, Sakaguchi R, Clive M, Sampath R, Museth K, Blakey D, Gazdik B, Thomas R. Destroying LA for "2012". In: ACM SIGGRAPH 2010 talks. SIGGRAPH '10, New York, NY, USA: Association for Computing Machinery; 2010, p. 1. http://dx.doi.org/10.1145/1837026.1837059, URL https://dl.acm.org/doi/10.1145/1837026.1837059.

[13] Lamb N, Banerjee S, Banerjee NK. MendNet: Restoration of fractured shapes using learned occupancy functions. Comput Graph Forum 2022;41(5):65–78. http://dx.doi.org/10.1111/cgf.14603, URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14603. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14603.

[14] Gregor R, Bauer D, Sipiran I, Perakis P, Schreck T. Automatic 3D object fracturing for evaluation of partial retrieval and object restoration tasks - benchmark and application to 3D cultural heritage data. In: Pratikakis I, Theoharis T, Spagnuolo M, Gool LV, Veltkamp RC, Godil A, editors. 3DOR. Eurographics Association; 2015, p. 7–14.

[15] Museth K. OPENVDB. In: ACM SIGGRAPH 2021 courses. SIGGRAPH '21, New York, NY, USA: Association for Computing Machinery; 2021, p. 1–197. http://dx.doi.org/10.1145/3450508.3464577, URL https://dl.acm.org/doi/10.1145/3450508.3464577.

[16] Schvartzman SC, Otaduy MA. Fracture animation based on high-dimensional Voronoi diagrams. In: Proceedings of the 18th meeting of the ACM SIGGRAPH symposium on interactive 3D graphics and games. i3D '14, New York, NY, USA: Association for Computing Machinery; 2014, p. 15–22. http://dx.doi.org/10.1145/2556700.2556713, URL https://dl.acm.org/doi/10.1145/2556700.2556713.

[17] Sellán S, Chen Y-C, Wu Z, Garg A, Jacobson A. Breaking Bad: A dataset for geometric fracture and reassembly. 2022, http://dx.doi.org/10.48550/arXiv.2210.11463, URL http://arxiv.org/abs/2210.11463. arXiv:2210.11463 [cs].

[18] Gu J, Ma W-C, Manivasagam S, Zeng W, Wang Z, Xiong Y, Su H, Urtasun R. Weakly-supervised 3D shape completion in the wild. In: Vedaldi A, Bischof H, Brox T, Frahm J-M, editors. Computer vision – ECCV 2020. Lecture notes in computer science, Cham: Springer International Publishing; 2020, p. 283–99. http://dx.doi.org/10.1007/978-3-030-58558-7_17.

[19] Stutz D, Geiger A. Learning 3D shape completion under weak supervision. Int J Comput Vis 2020;128(5):1162–81. http://dx.doi.org/10.1007/s11263-018-1126-y.

[20] Deng Z, Jiang J, Chen Z, Zhang W, Yao Q, Song C, Sun Y, Yang Z, Yan S, Huang Q, Bajaj C. TAssembly: Data-driven fractured object assembly using a linear template model. Comput Graph 2023;113:102–12. http://dx.doi.org/10.1016/j.cag.2023.05.003, URL https://www.sciencedirect.com/science/article/pii/S0097849323000560.

[21] Yu X, Rao Y, Wang Z, Liu Z, Lu J, Zhou J. PoinTr: Diverse point cloud completion with geometry-aware transformers. In: ICCV. 2021, p. 12498–507.

[22] Kerbl B, Kopanas G, Leimkühler T, Drettakis G. 3D Gaussian splatting for real-time radiance field rendering. ACM Trans Graph (SIGGRAPH Conf Proc) 2023;42(4). URL http://www-sop.inria.fr/reves/Basilic/2023/KKLD23.

[23] Navarro P, Cintas C, Lucena M, Fuertes JM, Rueda A, Segura R, Ogayar-Anguita C, González-José R, Delrieux C. IberianVoxel: Automatic completion of iberian ceramics for cultural heritage studies. In: Thirty-second international joint conference on artificial intelligence. Vol. 6, 2023, p. 5833–41. http://dx.doi.org/10.24963/ijcai.2023/647, URL https://www.ijcai.org/proceedings/2023/647. ISSN: 1045-0823.

[24] Zheng Z, Yu T, Dai Q, Liu Y. Deep implicit templates for 3D shape representation. 2021, http://dx.doi.org/10.48550/arXiv.2011.14565, URL http://arxiv.org/abs/2011.14565. arXiv:2011.14565 [cs].

[25] Zobeidi E, Atanasov N. A deep signed directional distance function for object shape representation. 2021, http://dx.doi.org/10.48550/arXiv.2107.11024, URL http://arxiv.org/abs/2107.11024. arXiv:2107.11024 [cs].

[26] Tang J, Lei J, Xu D, Ma F, Jia K, Zhang L. SA-ConvONet: Sign-agnostic optimization of convolutional occupancy networks. In: 2021 IEEE/CVF International Conference on Computer Vision. ICCV, 2021, p. 6484–93. http://dx.doi.org/10.1109/ICCV48922.2021.00644, URL https://ieeexplore.ieee.org/document/9710632/. Conference Name: 2021 IEEE/CVF International Conference on Computer Vision (ICCV) ISBN: 9781665428125 Place: Montreal, QC, Canada Publisher: IEEE.

[27] Cheng Y-C, Lee H-Y, Tulyakov S, Schwing AG, Gui L-Y. SDFusion: Multimodal 3D shape completion, reconstruction, and generation. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2023, p. 4456–65, URL https://openaccess.thecvf.com/content/CVPR2023/html/Cheng_SDFusion_Multimodal_3D_Shape_Completion_Reconstruction_and_Generation_CVPR_2023_paper.html.

[28] Payne A, Limp F. Virtual hampson museum project.

[29] Choi S, Zhou Q-Y, Miller S, Koltun V. A large dataset of object scans. 2016, arXiv:1602.02481.

[30] Manivasagam S, Wang S, Wong K, Zeng W, Sazanovich M, Tan S, Yang B, Ma W-C, Urtasun R. LiDARsim: Realistic LiDAR simulation by leveraging the real world. In: 2020 IEEE/CVF conference on computer vision and pattern recognition. CVPR, IEEE Computer Society; 2020, p. 11164–73. http://dx.doi.org/10.1109/CVPR42600.2020.01118, URL https://www.computer.org/csdl/proceedings-article/cvpr/2020/716800l1164/1m3o170Go2k.

[31] Behley J, Garbade M, Milioto A, Quenzel J, Behnke S, Stachniss C, Gall J. SemanticKITTI: A dataset for semantic scene understanding of lidar sequences. In: 2019 IEEE/CVF international conference on computer vision (ICCV). 2019, p. 9296–306. http://dx.doi.org/10.1109/ICCV.2019.00939, URL https://ieeexplore.ieee.org/document/9010727. ISSN: 2380-7504.

[32] Chen K, Choy CB, Savva M, Chang AX, Funkhouser T, Savarese S. Text2Shape: Generating shapes from natural language by learning joint embeddings. 2018, arXiv preprint arXiv:1803.08495.

[33] Koutsoudis A, Pavlidis G, Arnaoutoglou F, Tsiafakis D, Chamzas C. Qp: A tool for generating 3D models of ancient Greek pottery. J. Cult. Herit. 2009;10(2):281–95. http://dx.doi.org/10.1016/j.culher.2008.07.012, URL https://www.sciencedirect.com/science/article/pii/S1296207409000326.

[34] Sellán S, Luong J, Mattos Da Silva L, Ramakrishnan A, Yang Y, Jacobson A. Breaking good: Fracture modes for realtime destruction. ACM Trans Graph 2023;42(1):10:1–10:12. http://dx.doi.org/10.1145/3549540, URL https://dl.acm.org/doi/10.1145/3549540.

[35] Ogayar-Anguita CJ, Rueda-Ruiz AJ, Segura-Sánchez RJ, Díaz-Medina M, García-Fernández AL. A GPU-based framework for generating implicit datasets of voxelized polygonal models for the training of 3D convolutional neural networks. IEEE Access 2020;8:12675–87. http://dx.doi.org/10.1109/ACCESS.2020.2965624, URL https://ieeexplore.ieee.org/document/8955843. Conference Name: IEEE Access.

[36] Zhang Y, Garcia S, Xu W, Shao T, Yang Y. Efficient voxelization using projected optimal scanline. Graph Models 2018;100:61–70. http://dx.doi.org/10.1016/j.gmod.2017.06.004, URL https://www.sciencedirect.com/science/article/pii/S152407031730053X.

[37] Skibo JM, Schiffer MB. The effects of water on processes of ceramic abrasion. J Archaeol Sci 1987;14(1):83–96. http://dx.doi.org/10.1016/S0305-4403(87)80008-0, URL https://www.sciencedirect.com/science/article/pii/S0305440387800080.

[38] Garland M, Heckbert PS. Surface simplification using quadric error metrics. In: Proceedings of the 24th annual conference on computer graphics and interactive techniques. SIGGRAPH '97, USA: ACM Press/Addison-Wesley Publishing Co.; 1997, p. 209–16. http://dx.doi.org/10.1145/258734.258849, URL https://dl.acm.org/doi/10.1145/258734.258849.

[39] Tsukiyama T, Kondo Y, Kakuse K, Saba S, Ozaki S, Itoh K. Method and system for data compression and restoration. 1986, URL https://patents.google.com/patent/US4586027A/en.

[40] Graciano A, Rueda AJ, Pospíšil A, Bittner J, Benes B. QuadStack: An efficient representation and direct rendering of layered datasets. IEEE Trans Vis Comput Graphics 2021;27(9):3733–44. http://dx.doi.org/10.1109/TVCG.2020.2981565, URL https://ieeexplore.ieee.org/document/9040672. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[41] Huang J, Zhan G, Fan Q, Mo K, Shao L, Chen B, Guibas L, Dong H. Generative 3D part assembly via dynamic graph learning. In: The IEEE conference on neural information processing systems. neurIPS, 2020, p. 1–12.