JCE EST 1998 Journal of Computer Engineering ISSN 10003428

Original Research Article
Vol 12, Issue 7, 2023 | ISSN: 1000-3428

GPU-Based Framework for Training 3D Convolutional Neural Networks with Voxelized Polygonal Models

Julian A. Moreno-Martínez, Sofia I. Hernández-González, Pablo R. Morales-Rodríguez, Laura M. Gómez-López, and Ricardo A. Díaz-Pérez

Julian A. Moreno-Martínez and Sofia I. Hernández-González, Universidad de Salamanca, Facultad de Informática, 37007, Spain;

ABSTRACT In this paper we present an efficient GPU-based framework to dynamically perform the voxelization of polygonal models for training 3D convolutional neural networks. It is designed to manage the dataset augmentation by using efficient geometric transformations and random vertex displacements in GPU. With the proposed system, every voxelization is carried out on-the-fly for directly feeding the network. The computing performance with this approach is much better than with the standard method, that carries out every voxelization of each model separately and has much higher data processing overhead. The core voxelization algorithm works by using the standard rendering pipeline of the GPU. It generates binary voxels for both the interior and the surface of the models. Moreover, it is simple, concise and very compatible with commodity hardware, as it neither uses complex data structures nor needs vendor-specific hardware or additional dependencies. This framework dramatically reduces the input/output operations, and completely eliminates the storage footprint of the voxelization dataset, managing it as an implicit dataset.

I. INTRODUCTION

training a neural network. Instead, current trends are based

D object recognition is a key component of many on larger sets of synthetic objects, mainly Computer-Aided systems, such as vision, surveillance, augmented reality, Design (CAD) models, which are normally represented by

robotics and self-driving vehicles, among others. With the increasing popularization of 3D acquisition technologies, those systems will greatly benefit from better 3D object classification methods. Among them, the most successful are based on machine learning, and especially deep learning. Within this research field, one of the most promising methods is the 3D Convolutional Neural Network (3D-CNN), which relies on a volumetric representation of the data. Several methods based on 3D-CNNs have been recently proposed for the purposes of object classification [1]–[9].

Generally speaking, machine learning methods heavily rely on the availability of a large amount of data, so a scarce dataset consisting of real-world objects (typically captured with 3D scanning technologies) is not the best option for

a boundary representation (B-rep) [1], [9]–[12]. With a 3DCNN, one of the most appropriated object representations is the voxel model, obtained by voxelization or 3D rasterization [1]–[4], [9].

The voxelization of a 3D model consists of converting its continuous geometric representation into a set of voxels that approximates it. *Voxel* stands for *volume element*, in the same way that *pixel* stands for *picture element*. Voxel representations of 3D models have many applications in solid modeling, volume graphics and physical simulation. They have been extensively used for rendering objects which are difficult to represent with traditional surface representations, like clouds, fire, smoke or terrain models [13]. A voxelization suffers from the limitations of any discrete representation

© 2023 The Authors. Open Access under CC BY 4.0.
How to cite: Julian A. Moreno-Martínez, Sofia I. Hernández-González, Pablo R. Morales-Rodríguez, Laura M. Gómez-López, and Ricardo A. Díaz-Pérez (2023). GPU-Based Framework for Training 3D Convolutional Neural Networks with Voxelized Polygonal Models. Journal of Computer Engineering, 12(7), 78–99. DOI: https://doi.org/10.5281/zenodo.19349528

of a continuous signal. Therefore, the resulting data will be limited by the sampling frequency, which is directly related to the voxel space resolution. Moreover, each sampling (a voxel in our case) of the original signal (the geometry of the 3D model) will be quantized to a given level of precision. Typical voxelizations for machine learning are binary, having a single presence value that can be represented by a single bit (Fig. 1).

In this paper we present an efficient framework for dynamically performing the voxelization of B-rep models based on polygonal meshes. The proposed method is designed to be used for training 3D CNNs, and takes into account several aspects such as the dataset augmentation, geometrical transformations and the storage footprint of the voxelizations. The GPU-based algorithm used for voxelizing 3D models is concise, very compatible with commodity hardware, and it neither uses complex data structures nor has additional dependencies such as CUDA. Our proposal reduces the loading and saving operations to the bare minimum, and completely eliminates the storage footprint. Consequently, this framework manages the voxelized 3D models dataset as a truly implicit dataset.

II. RELATED WORK

Deep learning is one of the most active fields in machine learning. Among the available techniques, CNNs have been successfully used on images for many tasks, such as image classification, semantic segmentation or object detection. Later, in the context of Deep Learning for Graphics [14], CNNs have been used for identifying and classifying 3D data, from single points to structured data considered as objects. Since deep networks need a large amount of training data to achieve a good performance, real-world objects are not the best option, because their capture with 3D scanning technology, processing and labeling, is a cumbersome task. Instead, there are many approaches that depend on 3D CAD models for the training [1]–[4], [9]. By using CAD models, the features of the samples from the training dataset are totally under control, especially the topology. This also allows to have other useful attributes, such as surface normals, colors, labels for segmentation, etc. [1], [9]–[12].

There are several network architectures for learning from CAD models, which intrinsically involve the use of a specific data structure. The main representations used in deep learning for graphics are images, volume elements, meshes and point clouds [8], [14]. All of them can be used to describe 3D objects, although meshes are preferred for most tasks in computer graphics, due to their completeness and precision. In deep learning there are many 3D object segmentation and classification methods based on CNNs [2], [4], [8]. Most of them use volumetric information for representing the objects [1], [2], [4]–[6], [15]–[18], or a set of images resulting from projections of the objects from several virtual camera positions [7], [8]. However, volumetric representations provide more suitable spatial information to identify and classify 3D objects. Among current research results we can highlight the work from Wu *et*

© 2023 The Authors. Open Access under CC BY 4.0.
How to cite: Julian A. Moreno-Martínez, Sofia I. Hernández-González, Pablo R. Morales-Rodríguez, Laura M. Gómez-López, and Ricardo A. Díaz-Pérez (2023). GPU-Based Framework for Training 3D Convolutional Neural Networks with Voxelized Polygonal Models. Journal of Computer Engineering, 12(7), 78–99. DOI: https://doi.org/10.5281/zenodo.19349528

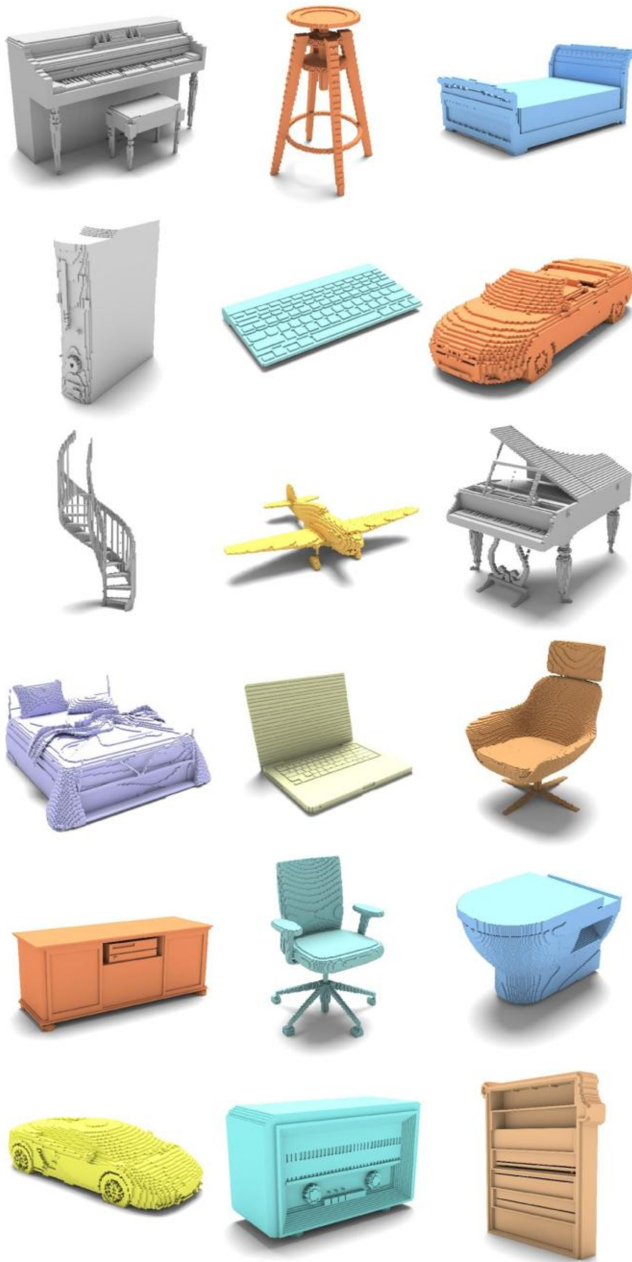

FIGURE 1. Some examples of voxelized CAD models.

al. [1], which introduces a dataset to evaluate 3D shape classifiers (ModelNet), and Maturana *et al.* [2] which introduces Voxnet, an architecture for recognizing objects from point clouds in real time that integrates a voxelization with a supervised 3D-CNN. The authors tested their system with LiDAR, RGBD and CAD data. Other works are also based on this latter approach [18]. In addition to this, some works mix several object representations in order to improve the learning, including voxelizations [19].

Working with 3D-CNNs worsen the so called *curse of dimensionality*. Because of this, voxelizations are normally generated at low resolutions. However, sparse 3D-CNNs can reduce this issue [20]–[22], allowing greater volumetric resolutions in the architecture of the network. Other works also use hierarchical representations like octrees [23] to further lessen the effect of this issue. Related to this, Ghadai *et al.* [24] propose a multi-resolution CNN for object recognition based on a multi-level voxel grid. In addition to this, with upcoming hardware and more available memory, it is expected that in the near future more detailed voxelizations could be used with CNNs, so an efficient voxelization pipeline as the one proposed here would be very useful.

A. RELATED WORK ON VOXELIZATION

The voxelization of a 3D model has many applications and it is a well researched topic. There are many works that propose solutions for converting a 3D model B-rep representation to voxels. Most of these works can be grouped according to four main features: (1) the use of densities in contrast to binary values for each voxel, (2) the voxelization of the surface and/or the interior, (3) the geometrical method for calculating each voxel, or (4) the computational method used for the processing. Following, some previous works on voxelization are discussed according to these criteria.

1) Voxel values

In general, most voxelization methods are designated as binary rasterizarion algorithms [25]–[33], because the result is a simple classification of whether the voxels lie inside or outside the 3D model. The main drawback

is that they suffer from aliasing problems. To alleviate this, other algorithms focus on the quality and accuracy of the result by using some form of filtering [34] or distance field techniques [35]. In contrast to binary approaches, these algorithms assign a density value to each voxel that reflects its degree of occupancy by the 3D model. However, most 3D-CNN architectures work with binary voxels [1], [2], [4], therefore this matter remains open for further experimentation.

2) Surface voxelization and full voxelization

A voxelization can produce presence values for each resulting voxel from the surface of a 3D model [25]–[30], [34] or also from the interior [26], [31]–[33]. The first voxelization algorithms were mere extensions of 2D rasterization algorithms to a 3D space. These methods produce a 3D rasterization of graphical primitives, mainly polygons. In this case, the voxelization of a 3D model results in a volumetric representation of the surface of the model, but not of its interior. Other methods can also calculate the interior. Fig. 2 shows the difference between voxelizing only the surface of a model and performing a full voxelization.

Both approaches can be implemented in GPU. Some proposals perform a GPU-based surface voxelization of polygonal models [25], [28], [29]. Those methods are very fast, but only rasterize the surface of the models. In this category, the best performance is achieved by Schwarz *et al.* [26], and later by Pantaleoni [27]. Both of them use

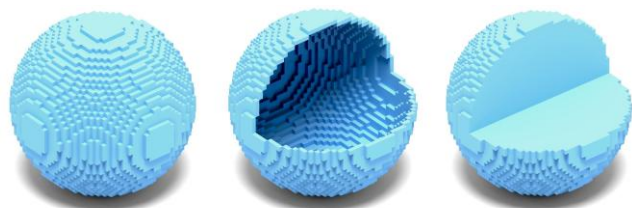

FIGURE 2. Voxelization of an sphere (left), a cut of the voxelization of its surface (middle) and a cut of the full voxelization (right).

a massively parallel approach for computing the surface voxelization, and can also perform the voxelization of the interior. The main drawback is their complexity and specific hardware requirements. The work from Zhang *et al.* [30] is another method for voxelization that produces a high-quality surface voxelization, with better results than previous works [27], especially at higher resolutions.

Apart from those mentioned above, there are other voxelization methods that take into account the interior of the 3D models (Fig. 2). This is usually not addressed because full voxelizations are computationally more expensive, or because it is unnecessary for certain applications. However, as there are 3D-CNNs that require the interior of the objects, in this work we are more interested in this type of algorithm. The best voxelization methods that rasterize the interior of the solid are implemented in GPU. One of the most cited is the work presented by Fang *et al.* [31], which performs a slice-based rasterization using the standard rendering pipeline. Other classic methods use several views or a multi-pass rendering approach in order to obtain better results, but with a performance penalty [36]. The work from Eisemann *et al.* [33] presents a single-pass technique for the voxelization of the interior of the solids, with a filtering algorithm that builds a density estimation for calculating surface normals. The main drawback is that the implementation is not so straightforward. As mentioned before, the method from Schwarz *et al.* [26] can rasterize the surface of a 3D model, but it can also perform a second step for obtaining the interior, with the possibility of constructing an octree. The method presented by Young *et al.* [37] performs a multi-level voxelization using ray-triangle tests for the finer levels.

3) Methods for calculating voxels

Another important feature of a voxelization algorithm is the method used for calculating

each voxel. Some proposals are based on a direct rasterization of the graphical primitives [34], [35]. The work presented by Zhang *et al.* [30] is an example of a modern scan-line based method. It produces a high-quality surface voxelization, with better results than the fastest alternative [27], especially at higher resolutions. These 3D scan-conversion methods are extensions of their 2D counterparts, and are usually very efficient. However, they suffer from aliasing, only keep a part of the surface of the objects, and cannot rasterize their interior. Many other algorithms are based on a triangle-voxel 3D intersection test for checking whether a given voxel intersects a given triangle [26], [27]. These computationally intensive solutions have the drawback of the cost of the triangle-voxel intersection test. However, this is compensated by using GPU-based massively parallel implementations that present a remarkable performance.

There are some approaches that rely on a GPU slicingbased voxelization [31]–[33]. In general, this approach works by moving a slicing plane, in some cases parallel to the projection plane. This plane is displaced with a constant step size that depends on the resolution of the voxelization. For each position of the slicing plane, the object to be voxelized is rendered, and the intersection between the geometry and the plane is calculated. These methods also apply masks for using an integrated presence function that allows the definition of different materials for the interior. With the introduction of the geometry shader in modern GPUs, other methods took advantage of this option, like the proposal of Chang *et al.* [28], which is is fast, but only rasterizes the surface of the models. In general, slicing-based algorithms suffer a performance penalty linked to the resolution, because they require multiple rendering passes, and therefore the object to be voxelized must be rendered multiple times. Moreover, this approach can also produce some aliasing that can be resolved by a conservative rasterization [25].

4) Computing approaches for voxelization

The technology used to implement a voxelization method is of paramount importance. It can be CPU-based or GPUbased, and the latter can be further classified into methods that use the standard rendering pipeline [28], [29], [31]–[33], and those that use computational methods, mostly based on APIs like CUDA or OpenCL [26], [27], [30], [37]. The use of the rendering pipeline is compatible with almost every system, which is always a desirable feature. With the GPU programmable stages (shaders), a particular voxelizer can be designed with a custom rendering strategy [32], with the possibility of using more advanced rendering resources like layered depth images [38]. The geometry shader is often the preferred GPU programmable resource, as it allows the modification or creation of new geometry during the rendering [28], [29]. Our work uses this approach.

On the other hand, modern computational methods run in a massive parallel fashion for achieving their highest performance [26], [27], [30], [37]. The algorithm presented by Schwarz *et al.* [26] is considered the state of the art of the voxelization methods. Later, this method was enhanced by the work of Pantaleoni [27], which improved the parallel approach. The algorithm from Young *et al.* [37] performs a voxelization with different levels of detail, using raytriangle intersection tests computed in GPU. The main drawback with these last proposals is that they are somewhat difficult to implement efficiently. They also require additional dependencies like CUDA (with specific hardware), which prevents the deployment on a variety of systems.

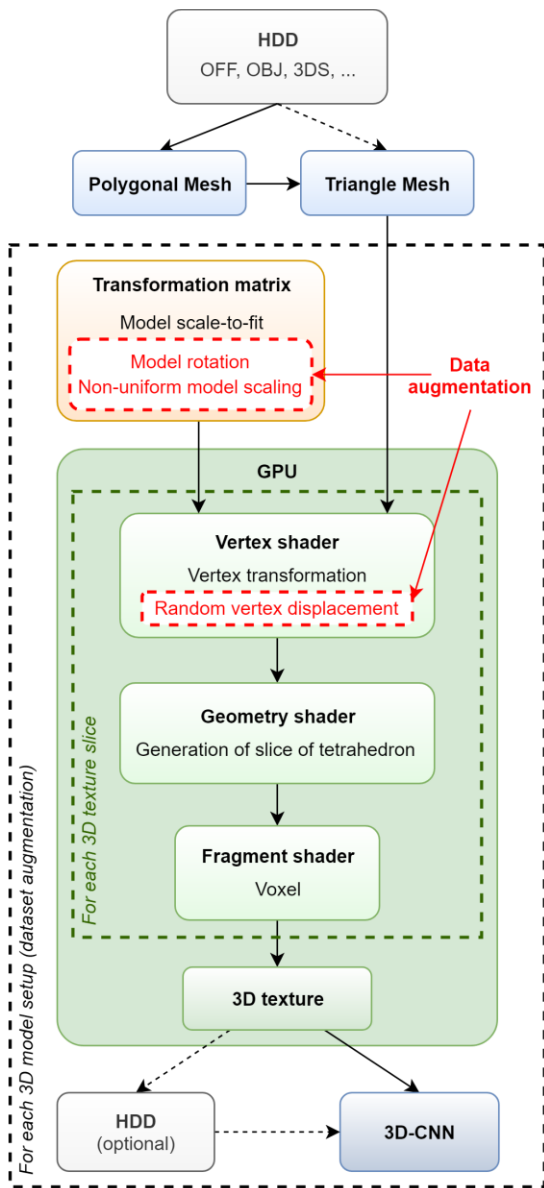

FIGURE 3. The proposed voxelization framework for training a CNN.

III. VOXELIZATION FRAMEWORK FOR CNNS

In this section we present the proposed voxelization framework along with its design principles. Fig. 3 shows the overall architecture of the framework for training a voxelbased CNN. The main goal is to feed the CNN with voxelizations of 3D models as fast as possible, including all the data augmentations achieved using a series of geometric transformations. The best scenario allows us to get each voxelization from the GPU and directly feed the CNN. For achieving this, both components must be integrated into the same program or be directly available as services, which heavily depends on the platform used. It would be also possible to use temporary files for a batch or storing the full dataset of voxelizations in secondary memory. However, this would partially defeat the purpose of this work.

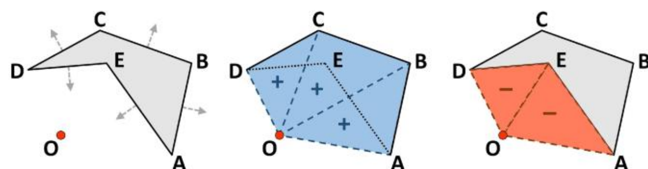

FIGURE 4. A 2D simplicial covering of a n-sided polygon. A triangle is generated for every segment, and has a sign for the covering that depends on the orientation of the segment with respect to *O*. In this example the simplicial covering of the polygon is composed of 3 positive triangles and 2 negative triangles.

For the voxelization framework to be efficient, the core 3D rasterization method must be as fast as possible. The fastest voxelization methods in the previous section only work with the surface of the solids [27]–[30], which is inadequate for working with some CNNs that require the interior of the objects for learning properly. Among the state of the art of the methods that work with the interior of the solids [26], [32], [33], our voxelization algorithm [32] is one of the most concise and easy to implement, while having a reasonably performance. It can directly handle any kind of polygon, not only triangles, including non-convex, holed or non-manifold. This is a very desirable feature for working with CAD models, not needing a polygon tesselator for decomposing n-sided polygons into non-overlapping triangles. The only drawback is that it needs topologically closed solids as input in order to give correct results.

On the other hand, in this work the voxelization algorithm has been updated for allowing dataset augmentation strategies based on efficient geometric transformations and deformations, which are applied in every voxelization batch. Moreover, it is a very compatible GPUbased approach, because it does not need vendor-specific hardware nor additional dependencies, such as CUDA [26], [27].

3D models can be loaded into memory from several available file formats. Since the voxelization method is binary, only topological information is taken into account. Materials, textures, animations and other data are ignored. The first thing to consider is the B-rep representation. We have adapted and optimized an existing voxelization algorithm [32], of which more details are presented in section III-A. Unlike most voxelization algorithms, it can directly handle n-sided polygonal faces without the need for tessellating them into non-overlapping triangles. Each n-sided polygon is converted to a triangle set using a 2D simplicial covering (Fig. 4). Logically, this step is omitted if the polygon is already a triangle. After that, for each triangle a tetrahedron is created using the centroid of the model as the fourth point. This tetrahedra set forms a 3D simplicial covering of the model, and is used for the rasterization of both the surface and the interior of the model [32]. Actually, the union of the 2D simplicial coverings of all the faces of the model can be treated as a normal triangle mesh. Therefore, the data structure to be handled by the GPU is completely standard: a list of vertices shared by a set of triangles, in

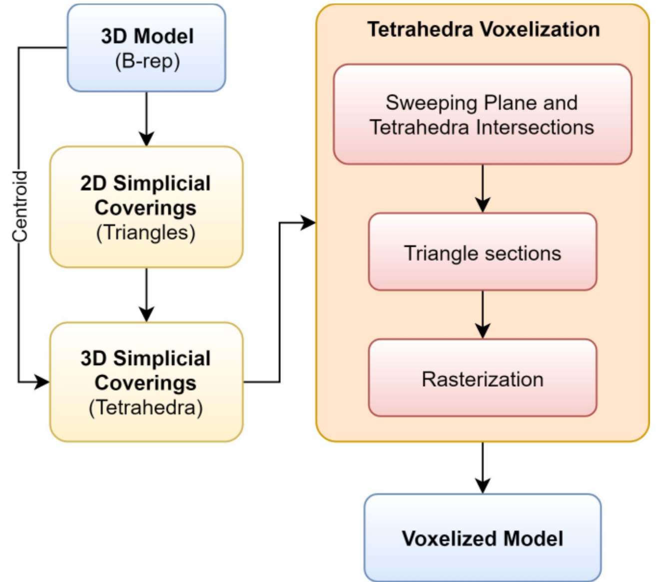

FIGURE 5. 3D model voxelization workflow. This is the core of the framework.

which each triangle is specified by three vertex indices. The construction of a tetrahedron from each triangle is directly carried out in GPU during the voxelization. Therefore, the first step consists of uploading vertices and triangles to GPU memory. From then on, all the voxelizations of the same model can be directly carried out only by setting several input parameters of the GPU shaders. In section III-B we present more details about the GPU-based implementation.

A. 3D MODEL VOXELIZATION ALGORITHM

In this section we present the fundamentals of the 3D model voxelization algorithm used, which is based on a previous work [32]. The theoretical foundation is the pointin-tetrahedron inclusion test. This method works with 3D solids defined with a polygonal B-rep, or polyhedron, which is a standard representation in CAD and solid modeling.

Let G be a polyhedron and O an arbitrary origin point. G is defined by the polygonal faces $f_1,f_2,\ldots,f_n$. Then let $S = T_1,T_2,\ldots,T_n$ be a covering of G with 3D simplices (tetrahedra).

For the simplest case, that is, a polyhedron defined only by triangles, T_i is defined by O and the triangular face f_i. For the general case, which is a polyhedron defined by n-sided polygons, each face is represented by a 2D simplicial covering which results in a signed n-triangle set [39] (see Fig. 4), and therefore a set of tetrahedra is defined for that triangle set. Then, an arbitrary point P is inside polyhedron G if the following condition is met:

$$\sum_i sign(T_i) \cdot inclusion(P,T_i) > 0 \quad (1)$$

where $inclusion(P,T_i) = 1$ when $P \in T_i$ and 0 otherwise; $sign(T_i) = +1$ when the vertices of the triangular faces of the tetrahedron T_i follow a counterclockwise ordering,

TABLE 1. Edges required for point interpolation depending on the sweeping plane position.

P_0	$\overline{AD}$	$\overline{AD}$	$\overline{AD}$
P_1	$\overline{AB}$	$\overline{BD}$	$\overline{BD}$
P_2	$\overline{AC}$	$\overline{AC}$	$\overline{CD}$
P_3	–	$\overline{BC}$	–

−1 when they follow a clockwise ordering, and 0 when the tetrahedron is degenerated.

This way, a point P inside G is covered by an odd number of tetrahedra from S. By using the formulation of equation 1, the voxelization algorithm for a given polyhedra is as follows (Fig. 5):

1) Clear the presence buffer (see below) with a value of 0.
2) Compute the centroid of the polyhedron. Set this point as origin O.
3) Take every face of the polyhedron and construct its 2D simplicial covering, which results in a triangle set [39]. For the simplest case, that is, the polygonal face is a triangle, this step is unnecessary since the 2D simplicial covering of a triangle is that same triangle.
4) Take each triangle ΔABC from the previous step, and construct the tetrahedron $OABC$. This result in a tetrahedra set.
5) Voxelize (scan-convert) the tetrahedra set in the presence buffer using an XOR logical operation.
6) The final state of the 3D presence buffer represents the voxelization of the polyhedron.

The presence buffer is a 3D array with the same dimension as the voxel space. Each voxel will have a presence value, which can be represented with a single bit; 1 means that the voxel belongs to the polyhedron, whereas 0 means that the voxel is not occupied by the polyhedron. The scanconversion of a tetrahedron in the buffer is done by flipping all the presence values covered by it. Once every tetrahedron has been scan-converted, a value of 1 is only present at those voxels that approximate the interior of the polyhedron. The centroid of the polyhedron is the best choice for the origin O. This way, the average size of the tetrahedra is smaller, which implies a lower total amount of voxels to be processed. This algorithm is very simple and can handle any kind of polyhedron, including non-convex, self-intersecting or holed.

This algorithm produces a full voxelization, therefore it includes both the interior and the surface of the solid. If only the surface is needed, it can be extracted from the full voxelization by using a filtering operation (Fig. 2 shows an example). This way, every voxel that is adjacent to an empty voxel is marked as part of the surface. The adjacency is controlled by the filter. Each voxel has 26 neighboring voxels that can be queried: 6 at the faces, 8 at the corners, and 12 at the edges. The most common connectivity function is the 6-connectivity, which takes the six neighbors at the

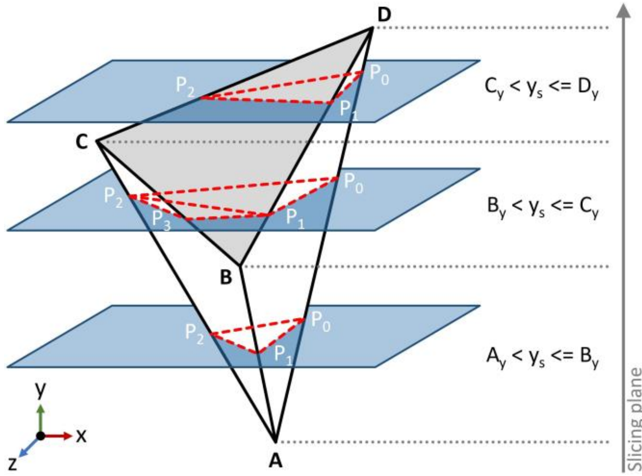

FIGURE 6. Slicing planes for rasterizing a tetrahedron.

faces, although different results can be obtained with other functions, such as 18-connectivity and 26-connectivity.

1) Tetrahedra rasterization

As pointed out before, the core step in the voxelization algorithm is the scan-conversion of a tetrahedron in the presence buffer, which is in turn a voxelization itself. For this purpose, each tetrahedron is processed by a scan-line algorithm. Let *ABCD* be a tetrahedron as depicted in Fig. 6. Let us assume that we use a right-handed coordinate system. The algorithm works as follows.

1) A slicing direction must be chosen. We will assume the slicing plane will move along the *Y* axis. Then, sort the vertices of the tetrahedron by their y coordinate. Let A be the vertex with the lowest y coordinate, B the next, and so on with C and D (see Fig. 6). The slicing plane starts at $y_s = A_y$ and finishes at $y_s = D_y$.
2) Compute the intersections of the edges of the tetrahedron with the current slicing plane. We denote these points P_0, P_1, P_2, P_3, as shown in Fig. 6. These intersections can be computed by a simple linear interpolation or by applying a faster incremental approach. Point P_3 is only needed in the interval $B_y < y_s \leq C_y$ (Table 1).
3) Voxelize the slice y_s of the tetrahedron. This is achieved by rasterizing the triangle $\Delta P_0P_1P_2$. In the interval $B_y < y_s \leq C_y$, a second triangle $\Delta P_1P_3P_2$ must also be rasterized. During this operation, the presence values of all the voxels (x,y_s,z) covered by the triangles must be flipped with an XOR logical operation.
4) Increment y_s and repeat steps 2 and 3 until $y_s = D_y$.

B. GPU-BASED VOXELIZATION

As presented just before section III-A, prior to performing the voxelization of a 3D model, all required data must be available in GPU memory. Our implementation uses a vertex buffer and an index buffer for the triangles. It must be noted that these buffers are not modified at any time after they have been loaded into GPU memory. For every step of each possible voxelization of the current 3D model, including data augmentation techniques, the stored triangles are taken as the source of all the GPU processing, which generates the needed intermediate data on the fly. This allows the voxelization framework to perform a full batch of voxelizations of a 3D

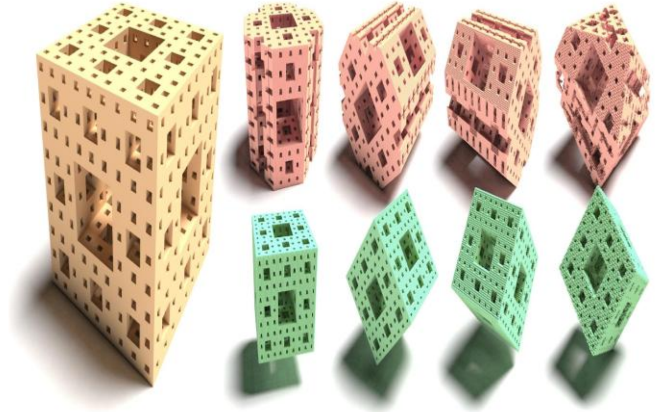

FIGURE 7. Voxelizations of a level-3 Menger Sponge fr
safe scale-to-fit. Bottom row: with safe scale-to-fit.

model without the need for continuously accessing files or transferring data from CPU to GPU. Section VI presents the source code of the shaders.

Apart from the buffers of vertices and triangles, in order to perform a single voxelization of a 3D model several parameters have to be specified to the shaders. First, the model has to be centered and scaled to fit the voxel space. This work assumes that this space is bounded by a cube, and has a given resolution (N^3). As several geometric transformations can be applied to the model, it is necessary to limit the scaleto-fit operation in order to get correct results. In this work we have used the maximum diagonal of the bounding box for calculating the scaling factor, although there are other choices, such as the diameter of the bounding sphere, the convex hull of the model, etc. This way, the object can be transformed without being clipped against the borders of the voxel space. The only drawback of this safe scale-to-fit operation is that the object will appear a bit smaller when voxelized (Fig. 7). All the transformations that can be applied are mainly used as a data augmentation technique for training the network. This matter will be further explained in section
III-C).

1) Vertex processing

All the geometric transformations mentioned before are applied in the vertex shader, in which all the vertices are processed and their new attributes are passed to the next stage of the GPU pipeline. Only the position of each vertex is relevant for our framework. Relating to this, a random vertex displacement can be performed at this point. This is one of the techniques for the dataset augmentation, and it allows us to get slight variations of the morphology of a 3D model (more details in section III-C). The main drawback with this step is that there is no truly support for random number generation in the GPU. Moreover, as every pseudorandom number generator depends on a previous state, this data cannot be shared by all GPU threads in an efficient way without seriously harming the parallelism. A simple solution to this problem consists of generating a large enough user controlled random sequence to be used along all the voxelizations, and uploading it to the GPU as a read-only circular buffer. This operation is very fast and is performed only once for each 3D model. Moreover, the performance impact on the vertex shader is minimal, because the random number generation in the shader is actually reduced to a memory access and a simple calculus. In order to control the starting of the series of random numbers within this buffer, an additional input parameter for the offset is used.

2) Tetrahedra generation and slicing

As mentioned before, for each triangle a tetrahedron must be constructed using the centroid of the model as the fourth vertex. The vertices shared by the triangles have been modified by the vertex shader, and now they are passed to the geometry shader. The centroid of the model is specified as an input parameter as it is fixed for every tetrahedron. The slicing plane is determined by the current slice number, which is also specified as an input parameter. Source code is shown in section VI. The geometry shader uses the algorithm from section III-A1. First, it computes the tetrahedron by taking the vertices of the input triangle and the centroid. It sorts the vertices of this tetrahedron by their y coordinate. Then, it calculates the intersections of the edges of the tetrahedron with the current slicing plane by applying linear interpolation (Table 1). Finally, it generates one or two triangles that result from these intersections, depending on the case (Fig. 6). Those triangles will be processed by the GPU built-in standard rasterizer and the corresponding fragments will be generated in

the next stage of the rendering pipeline. Of course, if the tetrahedron does not intersect the slicing plane, no triangles are generated for rasterizing.

It is necessary to remark the following. As we stated before, for every slice of the voxelization, a tetrahedra set is dynamically generated from the input set of triangles, which corresponds to the entire surface of the 3D model, including the simplicial coverings for the n-sided polygons. At first look, this seems to be quite inefficient, because the best option would be to previously calculate the tetrahedra set for a whole model voxelization, and then use it for rasterizing each slice. We have implemented and tested three alternatives for this: (a) calculate the tetrahedra in CPU and upload them as a buffer to the GPU; (b) generate the tetrahedra in a first GPU pass using transform feedback; (c) calculate the tetrahedra in a first GPU pass using a compute shader. Surprisingly enough, none of these options improved the performance of the algorithm. In fact, they were about 15-20% slower with our hardware configuration. Both alternative GPU versions, based on transform feedback and compute shader, had worse results probably because the generation of the tetrahedra only comprises a few algebra calculations, and the rest of the code is basically conformed by several random memory accesses and conditional instructions. This clearly does not take advantage of massive parallel processing of the GPU. Therefore, the best option is to calculate a tetrahedron when necessary, including the sorting of their vertices.

3) Voxels generation and storage

As the last stage of the GPU processing, the fragment shader stores the result of the rasterization of every slice of every tetrahedron. Each pixel stored in the current rendering buffer is actually a voxel write operation. At this point, an aspect function can be applied in order to get a material value for each voxel. However, since we are only interested in a binary voxelization, this step is not included in our implementation. This algorithm is implemented using a simple data structure for storing the result. The presence buffer can be represented by a 3D array of bits, but for most cases it is more convenient to store each voxel in a byte or even an integer for performance reasons. Most time is spent in the rasterization of 2D triangles corresponding to the intersections of the tetrahedra with the slicing plane. In this aspect, the GPUbased implementation is logically far more efficient than any CPU-based version, and is one of the main reasons why this framework is based on GPU processing.

The result of the voxelization can be stored in GPU in several ways. The best option is to use a 3D texture. While it can be used as a standard buffer for retrieving the data into CPU memory space, it can also be directly used for rendering purposes. Moreover, when using a 3D texture for storing the voxelization, two options are available: (a) render all the geometry in a single pass, generating triangles that will be targeted to a specific slice of the texture, (b) render the corresponding part of the geometry to each texture slice separately.

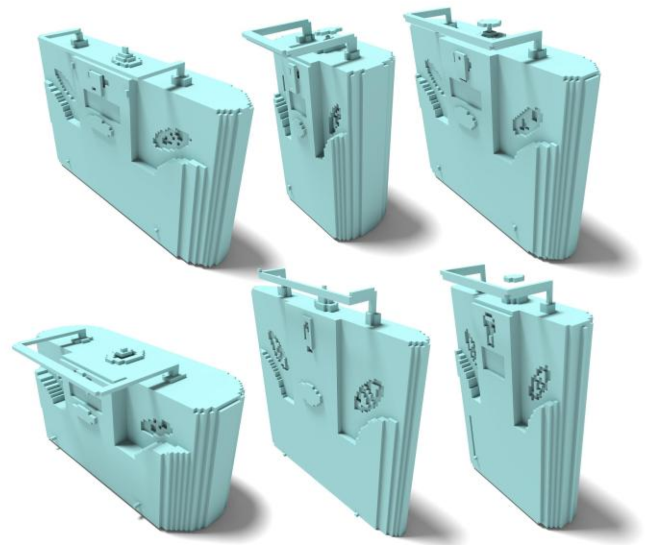

FIGURE 8. Some examples of non-uniform scaling transformations applied to a CAD model of a radio prior to rasterization.

Modern GPUs can render to multiple buffers simultaneously, including several 2D slices from a 3D texture. The rendering is still done in 2D, but in the shaders we can select the target buffer (texture slice) for each rasterized triangle. At first glance, this is the ideal solution, mostly because each tetrahedron is processed only once, and multiple tetrahedron slices are generated and rasterized into the proper texture slice for a single geometry shader execution. However, there are some drawbacks. The number of triangles that can be generated from a single shader execution is limited by hardware, forcing a cumbersome multi-pass approach. In any case, we have tested this solution with the result of a noticeable performance loss compared to the second option mentioned before. With this option the rasterization is carried out by a loop that moves the slicing plane. For each 2D slice of the resulting 3D buffer, all tetrahedra are processed in parallel in a rendering pass. Each tetrahedron is processed in the geometry shader, which tests whether the tetrahedron intersects the slicing plane and, if so, calculates the corresponding tetrahedron sections that must be rasterized, which are formed by a triangle or a pair of triangles (Fig. 6). This option is much faster that rasterizing all the slices of all the tetrahedra in a single pass. It is possible that, if multi-target rendering to 3D texture is improved in upcoming hardware, this situation could revert in the future.

C. DATASET AUGMENTATION STRATEGIES

Although there are more available CAD models than segmented real-world objects, they are clearly not sufficient for a proper training of a neural network. In order to increase the performance of the CNN it is

FIGURE 9. A series of random vertex displacement distribution limited to the 5% of the voxel space width.

convenient, even necessary, to augment the training dataset. As noted before, several actions can be carried out. The most important one is to perform multiple rotations on the models in order to improve the learning of rotational invariance (Fig. 7), and to break any possible symmetries along any axis [17]. The rotational invariance depends on the purpose of the network. For example, to identify urban furniture in an outdoor scene, rotational invariance along the gravity axis could be enough, making the training much faster, but at the cost of raising the probability of not properly recognizing fallen or damaged structures. In order to properly identify and classify a realworld object, rotational invariance is essential for achieving a good performance of the network. For achieving a whole model rotation, a rotation matrix is a part of the calculation of the model transformation matrix that is passed as an input parameter to the vertex shader.

Another convenient form of augmentation is based on the concept of object similarity. A

given type of object (a class) can have multiple variations in dimensions and proportions. As the object is always maximized to fit the voxel space, the original size loses relevance. On the other hand, different proportions can mislead the network. To solve this, several controlled non-uniform scaling transformations can be applied to each model (Fig. 8). By performing this, the network can learn invariance to that type of transformation. As before, this operation is implemented with a transformation matrix that is part of the model transformation matrix (Fig. 3). This step is straightforward, and have no impact on the GPU performance.

Related to the above, more variations of an object can be obtained by slightly displacing the spatial coordinates of the vertices of the models, which allows us to get variations of the morphology (Fig. 9). The displacement carried out is random and has a user-controlled distribution. Section III-B1 presented some details about the management of the random numbers generation in GPU. Since we work with polygonal meshes (polyhedra), this operation is straightforward, and can be performed on-the-fly in the vertex shader without changing the original mesh data stored in GPU. This way, the system allows the network to learn invariance to small deformations within the same object class.

It must be noted that combining all the possible data augmentation options, the total number of voxelizations to be performed for each model greatly increases. This must be taken into account because this data augmentation would improve the overall performance of the network, but at the cost of increasing the training time. In any case, the dataset augmentation strategy must be adapted to meet several criteria imposed by the nature of the data, because not every option is desirable for every situation. For example, variations in proportions could be considered as different versions of the same object or not, depending on whether we use only one class for that type of object or we also use subclasses.

D. CONNECTION WITH THE CNN

The last step of the framework is the connection with the neural network. As it was mentioned earlier, in order to have an efficient training process, the feeding of the network must be as fast as possible. To be more precise, only a minor part of the hardware resources must be used for the voxelization of 3D models, including all the data augmentations. Most processing power must be left for the neural network processing. By no means the network training should be interrupted, waiting for the voxelizer to finish the next sample. Therefore, this connection between the voxelizer and the neural network is of upmost importance. The standard approach performs every voxelization of every model separately by using different programs. On the other hand, the best implementation possible is the direct connection, which is accomplished by having the neural network interface and the voxelizer in the same program. In order to perform several training iterations with an implicit dataset of voxelizations, it must be always the same for each iteration. This is typically motivated by a change of the CNN parameters, such as the learning rate, the dropout, the minibatch size, etc., but not by a change to the dataset. This means that random-based augmentation techniques must always give the same output. This is achieved by processing the same

TABLE 2. Results of the benchmarks carried out with a commodity computer. Two resolutions are presented, 64^3 and 256^3. The dataset augmentation was a combination of 10 non-uniform scalings, 5 random vertex displacements and 100 rotations. 5000 voxelizations were generated for each CAD model, making a total of 5 million samples for the training of the CNN.

	Standar	One-Model	Direct	Standar	One-Model	Direct
Program Models	783s 462h	18	039	783s 462h	18	039 s
GPU		85			02	
GPU-CPU		25			28	
Total voxelizations		23			3	
Voxelization		2	Non		21	Non
Voxelization loading		4	Non		3	Non
Total	473 2h	66	32	545 3h	92	39
Storage footprint (RLE)	18,02GB	47,677 MB	None	795,57GB	1,91GB	None
Storage footprint (RAW)	152,58GB	156,25 MB	None	9,53TB	9,76GB	None

3D models in the same order, and controlling the seeds for all the random number generators used. This is the way the dataset of voxelizations can be truly considered as implicit, because it is always calculated on demand without the need of any data storage.

As an alternative, the voxelizer and the CNN can be in different programs. An example would be a Python script that uses the voxelizer as an external program written in C++. In this case, each voxelization must be stored on secondary memory and then read from the Python script in order to feed the neural network. Logically, this approach is much slower, especially when using a data augmentation configuration that causes a great number of voxelizations to be generated for each 3D model to be learnt. However, we can still take advantage of the framework. For achieving this, it is necessary to produce all the voxelizations needed for a given 3D model in a single step. In this work, we name this process the *one-model batch* approach. Consequently, the voxelizer will generate all the voxelization variants using a given data augmentation setup, and a mini-dataset of files will be written into secondary memory. In this case, it is mandatory that all the data of the current 3D model are kept in GPU until the last voxelization of that model is finished.

A direct connection with the neural network is not only convenient for performance reasons. Its storage footprint is zero, which supposes a total saving of secondary memory. In the case of using the one-model batch approach, the footprint would be minimal, because only the temporary files for the current model will be kept in secondary memory. On the contrary, the standard approach imply a huge storage footprint,

which possibly will force the user to configure the system for generating, saving and then loading the voxelizations as needed by the CNN.

IV. EXPERIMENTS AND RESULTS

We have implemented the entire framework in C++, including the three approaches described before: the standard approach, the one-model batch, and the direct connection with the CNN. We have implemented a dummy 3D-CNN used through its C++ interface for testing the performance of the framework. The hardware used was a commodity computer based on an Intel© Core™ I7-8700 CPU with 16 GB of main memory, and an Nvidia GeForce GTX 1060 with 6GB of memory. For testing the performance of the storage input and outputs operations we have used *M.2 SSDs*. All the tests have been performed with a single-thread implementation of the framework, in order to show the benefits of our approach with the simplest configuration. However, it can handle multiple CPU threads and also several GPUs. This ultimately depends on the needs of the CNN training. The main goal is to avoid the training from being interrupted, waiting for the voxelizer to finish the next sample.

We tested our framework with 3D CAD models from standard datasets, mainly from Princeton ModelNet10 and ModelNet40 [1]. The largest dataset used is ModelNet40, which consists of 12311 CAD models, split into 9843 models for training and 2468 models for testing. However, since the main goal of our tests is to analyze the performance of the voxelization framework, no division was made into training, test and validation sets for the neural network. We chose a total of 1000 models with different topological complexities. For analyzing the impact of performing a high number of voxelizations per model, we have used all the augmentation techniques described in section III-C. First, for each model 10 versions have been generated by using a random non-uniform scaling. This produces variations in term of proportions along the three axes. These variations in width, length and height have a maximum factor of ±50%. Combined to this, other 5 variations have been created using random vertex displacement for helping the network to learn from small deformations. In these tests we have used a uniform distribution in order to obtain a high number of noticeable variations, with a maximum of the 5% of the width of the voxel space. However, for the actual training of the CNN, more subtle and convenient results can be obtained using a normal distribution with a standard deviation of 5-10% of the width of the voxel space. Finally, 100 rotations around the three axes have been done in order to improve the learning of rotational invariance. This is to ensure that every object is learnt regardless the original vertical alignment of the CAD model. In total, 5000 voxelizations have been generated for each CAD model, making a total of 5 million samples for the CNN. Unfortunately, a considerable number of CAD models tested did not meet the topology conditions for the voxelization algorithm used to work correctly. Therefore, an straightforward pre-processing step was carried out for the models after loading. The main problem with the original models was the duplicity of the triangles (with opposite normal vectors), that was resolved by a merging operation.

We performed our experiments with several resolutions, albeit we have only included 64^3 and 256^3 in Table 2, for the sake of clarity. Although voxelizations for CNNs are nowadays generated at low resolutions, with upcoming hardware improvements more detail could be used with more complex network architectures. Therefore, the resolutions used in our experiments are feasible nowadays. Table 2 shows the results for the voxelization of 1000 models with a geometrical complexity ranging from about 10000 to about 265000 triangles. There are

several aspects that should be discussed. At first glance, there is a clear difference between the three approaches tested, especially regarding to loading operations. With the standard approach the system must be restarted for every voxelization. The program setup overhead is negligible, but the loading of the 3D models have a severe accumulated performance penalty. On the contrary, both direct connection and one-model batch approaches load each model only once.

The performance of the GPU-based voxelization algorithm depends on both resolution and geometrical complexity of the models. Its performance cost is the same for all the approaches. As can be seen, the higher the resolution, the bigger the impact of the GPU-CPU memory transfers. This also applies to the time taken for saving and loading the voxelized model files. For the direct connection approach, these operations do not apply.

For the footprint storage testing, all the voxelization files have been generated. In the case of the standard approach, we have used temporary files for simulating a full storage. For achieving realistic results, a FIFO queue has been used for managing a set of files that occupies a fixed and large enough portion of the storage space. This is to avoid the optimizations of the cache policies of the system, and to force the hard drive to write data in different physical locations.

Two file formats have been used for storing the voxelized models: a 8-bit per voxel using run length encoded compression (RLE), and a 1-bit per voxel uncompressed RAW binary format. The total times shown in Table 2 include the input and output operations only in RLE format. Table 3 shows the storage footprint of the 9843 models of the training dataset from ModelNet40, augmented up to 5000 samples per model. The one-model batch approach only needs to store the voxelizations of the model that is being processed. When using RLE compression, the peak of storage occupation is reached with the model whose voxelization has the highest entropy.

TABLE 3. Storage footprint for the training models from ModelNet40 dataset with the different approaches presented. The dataset augmentation setup is the same as in the main tests, which produces 5000 voxelizations for each one of the 9843 models, making a total of 49,21 million samples. The storage footprint of the direct connection is not included here as it is always zero.

Resolutio	Standard		One-Model Batch	
	RAW	RLE	RAW	RLE
	187.7G	04G	53M	31M
	46T	177.5G	156.2M	18M
	73T	20T	22G	516.0M
	87T	22T	76G	61G

As can be seen in Tables 2 and 3, results clearly reflect that the main weakness of the standard approach is the need for using secondary storage. It has a severe impact both on the storage footprint and the performance of the overall process. On the other hand, direct connection is logically the best approach, but if the voxelizer system and the CNN can not have that level of cohesion in the system used, there is always the option of using the one-model batch approach with a moderate impact on the performance and an almost negligible storage footprint.

V. CONCLUSIONS

In this paper we have presented an efficient framework for dynamically performing 3D model voxelizations for training CNNs. The GPU-based algorithm used for voxelizing 3D models is concise, very compatible with commodity hardware, and it neither uses complex data structures nor has additional

dependencies such as CUDA. The method is designed to manage the dataset augmentation by using efficient geometric transformations and random vertex displacements directly in GPU. Every voxelization is carried out on-the-fly for directly feeding the network. Results show that the overall performance with this approach is much better than with the standard method, which carries out every voxelization of every model separately, having much higher setup and data processing overhead. This framework dramatically reduces the input/output operations to a minimum. Moreover, the storage footprint of the generated voxelization dataset is zero, since every voxelization is carried out on-thefly for directly feeding the network. This way the framework manages the voxelized 3D model dataset as a truly implicit dataset.

VI. APPENDIX - SOURCE CODE

```
// GLSL Shaders source code

// Vertex shader --------------------------------
#version 430

uniform mat4 matrix; uniform float
randomSequenceFactor; uniform int
randomSequenceSize; uniform int
randomCurrentOffset;

layout(std430, binding=0) buffer
  InRandomSequence { vec3 random[]; };
layout(location = 1) in vec3 vertex;

void main() {
  vec4 displacement =
  randomSequenceFactor *
    vec4(random[(gl_VertexID+randomCurr
    entOffset)% randomSequenceSize], 0.0);
  gl_Position = matrix * vec4(vertex,1.0) +
    displacement;
}

// Geometry shader -----------------------------
-
#version 430

#define INTERP(A,B,s) \\
  (mix(A, B, (s - A[1]) / (B[1] - A[1])))
#define ORDERY(A,B) if (A.y > B.y) \\
  {vec4 tmp = A; A = B; B = tmp;}

layout(triangles) in; layout(triangle_strip,
max_vertices = 6) out;

uniform float slice; uniform vec3
centroidPoint;

void main() {
  vec4 tVertexA = gl_in[0].gl_Position; vec4
  tVertexB = gl_in[1].gl_Position; vec4
  tVertexC = gl_in[2].gl_Position; vec4
  tVertexD = vec4(centroidPoint,1);

  ORDERY ( tVertexA, tVertexB );
  ORDERY ( tVertexC, tVertexD );
  ORDERY ( tVertexA, tVertexC );
  ORDERY ( tVertexB, tVertexD ); ORDERY (
  tVertexB, tVertexC );

  if (tVertexA.y < slice && slice <= tVertexD.y)
  {
    gl_Position = vec4(INTERP(tVertexA,
    tVertexD, slice).xz, 0.0, 1.0);
    EmitVertex();

    vec4 v1 = (slice <= tVertexB.y) ?
      vec4(INTERP(tVertexA, tVertexB,
        slice).xz, 0.0, 1.0) :
      vec4(INTERP(tVertexB, tVertexD,
      slice).xz,
        0.0, 1.0); gl_Position = v1;
    EmitVertex();

    vec4 v2 = (slice <= tVertexC.y) ?
      vec4(INTERP(tVertexA, tVertexC,
        slice).xz, 0.0, 1.0) :
      vec4(INTERP(tVertexC, tVertexD,
      slice).xz,
        0.0, 1.0); gl_Position = v2;
    EmitVertex();

    EndPrimitive();

    // Extra triangle between vertices B and C
if (tVertexB.y < slice && slice <= tVertexC.y) {
      gl_Position = vec4(INTERP(tVertexB,
      tVertexC,
```

```
            slice).xz, 0.0, 1.0); EmitVertex();
          gl_Position = v2; EmitVertex();
          gl_Position = v1;
          EmitVertex();
          EndPrimitive();
        }
    }
}
// Fragment shader --------------------------
#version 430 out float result;
void main() { result = 1; }
```

REFERENCES

[1] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1912–1920, 2015.

[2] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 922–928, IEEE, 2015.

[3] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, "Multi-view convolutional neural networks for 3d shape recognition," in Proceedings of the IEEE international conference on computer vision, pp. 945–953, 2015.

[4] C. R. Qi, H. Su, M. Nießner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view cnns for object classification on 3d data," in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 5648–5656, 2016.

[5] A. Kar, S. Tulsiani, J. Carreira, and J. Malik, "Category-specific object reconstruction from a single image," in Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1966–1974, 2015.

[6] C. B. Choy, D. Xu, J. Gwak, K. Chen, and S. Savarese, "3d-r2n2: A unified approach for single and multi-view 3d object reconstruction," in European conference on computer vision, pp. 628–644, Springer, 2016.

[7] E. Johns, S. Leutenegger, and A. J. Davison, "Pairwise decomposition of image sequences for active multi-view recognition," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3813– 3822, 2016.

[8] D. Shin, C. C. Fowlkes, and D. Hoiem, "Pixels, voxels, and views: A study of shape representations for single view 3d object shape prediction," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3061–3069, 2018.

[9] S. Koch, A. Matveev, Z. Jiang, F. Williams, A. Artemov, E. Burnaev, M. Alexa, D. Zorin, and D. Panozzo, "Abc: A big cad model dataset for geometric deep learning," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 9601–9611, 2019.

[10] P. Shilane, P. Min, M. Kazhdan, and T. Funkhouser, "The princeton shape benchmark," in Proceedings Shape Modeling Applications, 2004., pp. 167–178, IEEE, 2004.

[11] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, et al., "Shapenet: An informationrich 3d model repository," arXiv preprint arXiv:1512.03012, 2015.

[12] Q. Zhou and A. Jacobson, "Thingi10k: A dataset of 10,000 3d-printing models," arXiv preprint arXiv:1605.04797, 2016.

[13] T. Akenine-Moller, E. Haines, and N. Hoffman, Real-time rendering. AK Peters/CRC Press, 2018.

[14] N. J. Mitra, I. Kokkinos, P. Guerrero, N. Thuerey, V. Kim, and L. Guibas, "Creativeai: Deep learning for graphics," in

SIGGRAPH 2019 Courses, Siggraph 2019, 2019.

[15] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Generative and discriminative voxel modeling with convolutional neural networks," arXiv preprint arXiv:1608.04236, 2016.

[16] X. Xu and S. Todorovic, "Beam search for learning a deep convolutional neural network of 3d shapes," in 2016 23rd International Conference on Pattern Recognition (ICPR), pp. 3506–3511, IEEE, 2016.

[17] N. Sedaghat, M. Zolfaghari, E. Amiri, and T. Brox, "Orientation-boosted voxel nets for 3d object recognition," arXiv preprint arXiv:1604.03351, 2016.

[18] C. Ma, Y. Guo, Y. Lei, and W. An, "Binary volumetric convolutional neural networks for 3-d object recognition," IEEE Transactions on Instrumentation and Measurement, vol. 68, no. 1, pp. 38–48, 2018.

[19] L. Minto, P. Zanuttigh, and G. Pagnutti, "Deep learning for 3d shape classification based on volumetric density and surface approximation clues.," in VISIGRAPP (5: VISAPP), pp. 317–324, 2018.

[20] B. Graham, "Sparse 3d convolutional neural networks," in Proceedings of the British Machine Vision Conference (BMVC) (M. W. J. Xianghua Xie and G. K. L. Tam, eds.), pp. 150.1–150.9, BMVA Press, September 2015.

[21] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey, "Faster cnns with direct sparse convolutions and guided pruning," arXiv preprint arXiv:1608.01409, 2016.

[22] X. Chen, "Escoin: Efficient sparse convolutional neural network inference on gpus," 2018.

[23] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: learning dense volumetric segmentation from sparse annotation," in International conference on medical image computing and computer-assisted intervention, pp. 424–432, Springer, 2016.

[24] S. Ghadai, X. Yeow Lee, A. Balu, S. Sarkar, and A. Krishnamurthy, "Multi-level 3d cnn for learning multi-scale spatial features," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 0–0, 2019.

[25] L. Zhang, W. Chen, D. S. Ebert, and Q. Peng, "Conservative voxelization," The Visual Computer, vol. 23, no. 9-11, pp. 783–792, 2007.

[26] M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on gpus," ACM transactions on graphics (TOG), vol. 29, no. 6, p. 179, 2010.

[27] J. Pantaleoni, "Voxelpipe: a programmable pipeline for 3d voxelization," in Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, pp. 99–106, ACM, 2011.

[28] H.-H. Chang, Y.-C. Lai, C.-Y. Yao, K.-L. Hua, Y. Niu, and F. Liu, "Geometry-shader-based real-time voxelization and applications," The Visual Computer, vol. 30, no. 3, pp. 327–340, 2014.

[29] Z. Zhang and S. Morishima, "Application friendly voxelization on gpu by geometry splitting," in International Symposium on Smart Graphics, pp. 112–120, Springer, 2014.

[30] Y. Zhang, S. Garcia, W. Xu, T. Shao, and Y. Yang, "Efficient voxelization using projected optimal scanline," Graphical Models, vol. 100, pp. 61–70, 2018.

[31] S. Fang and H. Chen, "Hardware accelerated voxelization," Computers &

Graphics, vol. 24, no. 3, pp. 433–442, 2000.

[32] C. J. Ogáyar, A. Rueda, R. J. Segura, and F. R. Feito, “Fast and simple hardware accelerated voxelizations using simplicial coverings,” The Visual Computer, vol. 23, no. 8, pp. 535–543, 2007.

[33] E. Eisemann and X. Décoret, “Single-pass gpu solid voxelization for realtime applications,” in Proceedings of graphics interface 2008, pp. 73–80, Canadian Information Processing Society, 2008.

[34] M. Sramek and A. E. Kaufman, “Alias-free voxelization of geometric objects,” IEEE transactions on visualization and computer graphics, vol. 5, no. 3, pp. 251–267, 1999.

[35] M. W. Jones, “The production of volume data