COMPUTER-AIDED DESIGN & APPLICATIONS
2018, VOL. 15, NO. 6, 852–862
https://doi.org/10.1080/16864360.2018.1486961

Taylor & Francis
Taylor & Francis Group

Check for updates

Fast and cross-vendor OpenCL-based implementation for voxelization of triangular mesh models

Mohammadreza Faieghi [a], O. Remus Tutunea-Fatan [b] and Roy Eagleson [c]

[a]Biomedical Engineering, Western University, Canada; [b]Mechanical and Materials Engineering, Western University, Canada; [c]Electrical and Computer Engineering, Western University, Canada

ABSTRACT

As the open standard for parallel programming of heterogeneous systems, OpenCL has been used in this study in the context of a particular and intensive computing task, namely the voxelization of tessellated objects. For this purpose, OpenCL platform has been utilized to develop a parallelized voxelization algorithm that relies on a fast and efficient triangular mesh facet/cube overlapping test. The extensive numerical tests conducted with heterogeneous hardware configurations on geometric objects of varying complexities, mesh/domain sizes, and voxel resolutions suggest that up to 99.6% or 260 times decrease in the computation time can be obtained when GPU- or CPU-based parallelized techniques are used instead of the conventional single-thread CPU approach. Future developments will attempt the integration of the current implementation into a virtual orthopaedic surgery platform.

KEYWORDS

OpenCL; Voxelization; GPGPU; GPU Computing

1. Introduction

The recent advancements in the field of computer graphics have led to a number of changes in the way the geometry of objects is modeled and rendered. Among the technologies that have been used for these purposes, voxel-based modeling continues to receive an increased attention. According to a more or less generic definition, voxels are small cubes used to represent volumetric units. As such, a voxel-based representation consists of a set of cubes that belong to a – typically uniform - 3D grid placed over the domain of the object. Since any point on the grid is easily identifiable, the structure of the voxelized geometry can be conveniently described in terms that are comparable to – if not simpler than - the polygonal mesh counterparts. This advantage of voxelized representations has been typically exploited in the context of computer-aided design and manufacturing such is the case of material removal simulation [2],[10],[15],[19],[23], skeleton extraction from mesh [18], composite model representations of functionally gradient materials [21], 3D printing [20], generation of porous surfaces [3], mesh repair [11], thickness analysis [14], etc. Since in many cases, the geometry is solely available in a triangular mesh format, voxelization becomes one of the mandatory preprocessing steps to enable subsequent geometry manipulation/computation operations.

As implied by Fig. 1, the core of the voxelization is represented by the computation of the intersection between the triangular facets of the mesh and the voxel grid used to discretize the bounding box of the 3D object. To calculate the required intersections, conventional approaches relied on iterative single-thread routines that loop over all voxels of the grid by repeatedly testing for intersections with each of the mesh facets. In many cases, robust and reliable overlapping test techniques – such as the separating axis theorem [1] – were used to identify and process the required voxel-facet intersections. However, the efficiency of these techniques is somewhat limited when it comes to fine meshes and/or voxels.

To accelerate voxelization, some of the prior studies have involved the graphics pipeline for parallelization of the voxelization in rendering passes [4–7],[9]. Nevertheless, the point sampling method used by the conventional rasterizers of the pipeline is known to cause inaccurate voxelizations for thin structures. To address this, [22] proposed a "conservative voxelization" technique that in turn introduced redundant voxels, an issue that was later rectified in [8] by means of hardware-based tessellation and point-based rendering. Even though the latter technique is faster by two orders of magnitude, the fixed-function tesselator of the pipeline was found to introduce inaccurate results such that while the involvement of the graphics pipeline might be sufficient for real-time

CONTACT O. Remus Tutunea-Fatan rtutunea@eng.uwo.ca, Mohammadreza Faieghi mfaieghi@uwo.ca, Roy Eagleson eagleson@uwo.ca

© 2018 CAD Solutions, LLC, http://www.cadanda.com

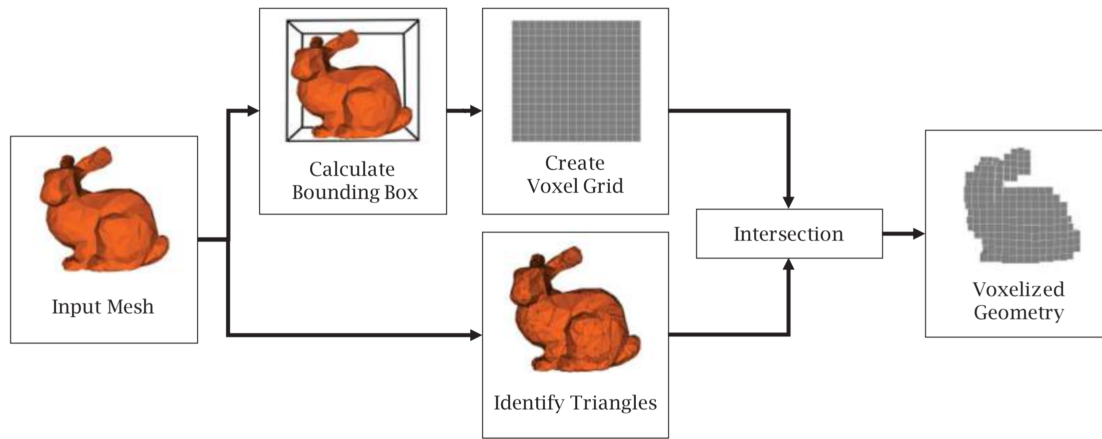


Figure 1. Phases of the voxelization process.

voxelization, its limited flexibility might be responsible for inaccurate results.

In the recent years, General Purpose Graphics Processing Units (GPGPUs) gradually became a versatile tool for high-performance computations. Both CUDA [12] and OpenCL [13] programming platforms were designed to facilitate the access to the many GPU cores that can be used to parallelize a multitude of computing tasks for an efficient solving of the complex problems. As an illustration of the CUDA-capabilities, Schwarz and Seidel [17] have shown that the GPU-based parallelization of the triangle-cube overlapping test can reduce the computing time reported in [22] by one order of magnitude, while preserving the accuracy of the voxelization.

Contrasting with the vendor-specific nature of CUDA, OpenCL-based developments are compatible with a wide-range of graphics hardware from all major vendors. However, unlike CUDA, OpenCL tends to be more verbose in a sense that more low-level code is required to establish the parallel computing infrastructure. Because of this, OpenCL-based parallelization continues to remain a challenging task and this might explain its relatively limited deployment. With this in mind, the primary goal of the current work was to assess the feasibility of OpenCL platform to an extensive computational problem such as voxelization. While not apparent in the limited context of this study, the problem of fast and accurate voxelization to be performed on cross-vendor hardware represents one of the first milestones to be completed as part of an extensive virtual surgery simulator that is presently under development.

2. OpenCL framework

The OpenCL standard defines a set of data types, data structures, and functions that augment C and C++ to enable parallel processing by means of either CPUs or GPUs. The major components of a typical OpenCL program are depicted in Fig. 2. While all these components will be briefly explained further, for more in-depth details on the OpenCL programming framework, the reader is invited to consult a more extensive resource, such as [16].

Kernels are functions to perform parallel computing tasks and they are programmed in OpenCL C, a language established on C99 specifications [13], but further enriched to accommodate parallel programming. Kernels are executed by OpenCL *devices* that can be either multi-core CPUs or GPUs. It is important to note that most computers have several OpenCL devices in their architecture. The OpenCL *context* allows to choose from these devices and manage them for a specific computational task. Control of the kernels and devices in a context are initiated by a segment of regular C/C++ code, termed *host application*.

According to OpenCL terminology, control instructions are called *command queues*, while the CPU running the host application is called OpenCL *host*. In a typical computing task, the input data for kernels is prepared on the host side and is stored within the *host memory* (*i.e.*, general computer RAM). Following this, OpenCL *buffers* transfer the prepared data to the *device memory* and if the device is a GPU, the device memory becomes the memory of graphics card. Once the kernel is executed, the output can be either fetched by the host memory or stored in the device memory for subsequent computations.

The generic OpenCL device model is presented in Fig. 3. According to this, the processing cores of a CPU or GPU are called *compute units*. After the kernel is invoked, the computing task is divided into several subtasks, called *work-groups,* each of them being executed by a compute unit. Furthermore, several processors are available inside of every compute unit, each of them being tasked

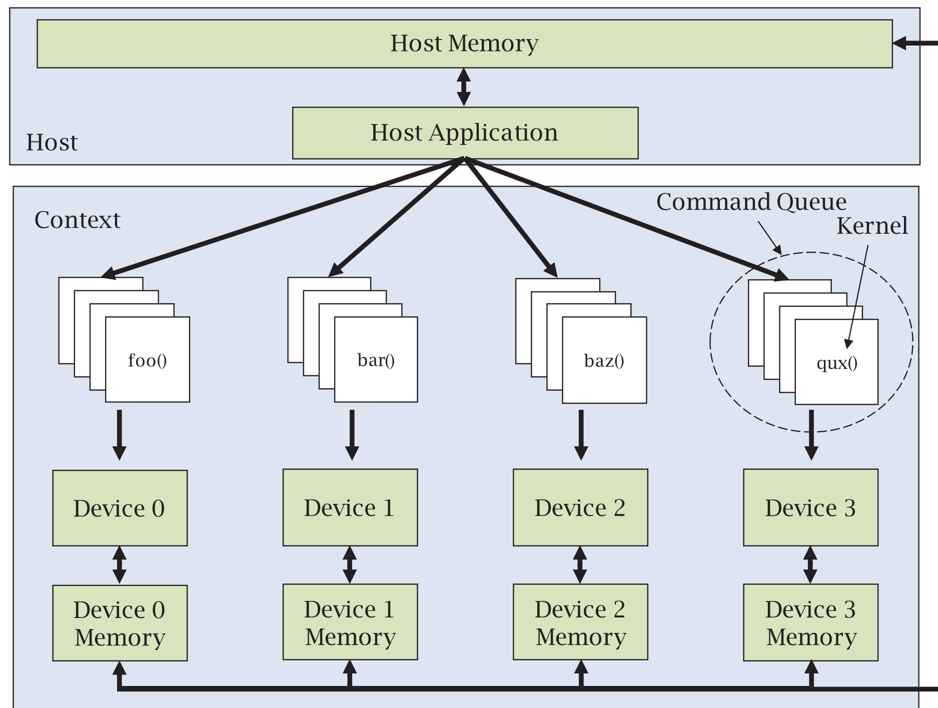


Figure 2. Generic structure of the OpenCL program.

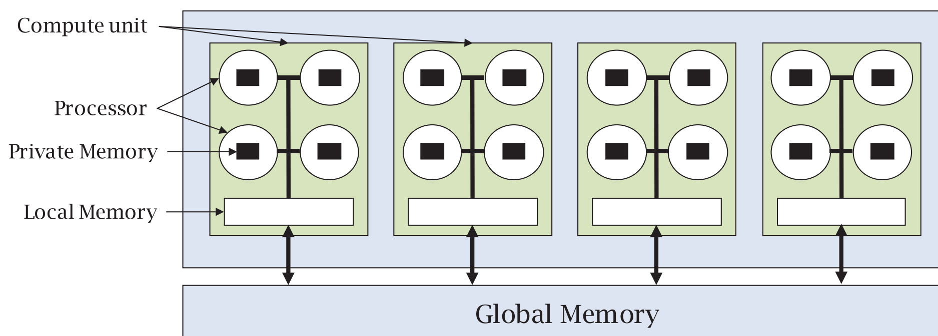


Figure 3. Generic structure of the OpenCL device.

to complete a certain *work-item*, *i.e.*, a certain portion of the work-group.

To accommodate this complex type of hierarchical computing pattern, different layers of memory are embedded into the device. Every processor from the compute unit has a set of dedicated registers called *private memory*, while all processors belonging to the same compute unit share a segment of memory called *local memory*. All compute units can also access the *global memory* (*i.e.*, device memory). A brief comparative analysis of all different types of memory available suggests that private memory remains the fastest, but also the most limited one. By contrast, global memory is the largest, but the also the slowest option. Because of this, efficient OpenCL programming needs to limit the number of times of access to global memory to a maximum of two: one for kernel input reading and another one for output writing.

It is also important to optimally parallelize a compute task by partitioning it into several work-groups. The number of work-items in a work-group is called *local size* and the total number of work-items is denoted by the term *global size*. While the best practice recommends the maximization of the local size, local memory tends to be limited such that the global memory has to be used in most cases and its extensive access will inevitably compromise the performance of the algorithm. As such, data partitioning often comes down to a challenging tradeoff between the maximum use of the local size and the minimal access to the global memory.

3. Implementation

3.1. Input data preparation

The vast majority of common triangular mesh formats store mesh data in two different arrays: a float array used for vertex coordinates (*i.e.*, mesh geometry) and an integer array used to describe how the vertices are connected to form the triangles in the mesh (*i.e.*, mesh topology). This particular type of data storage saves memory space but results in a cluttered memory access during kernel execution that in turns slows down the entire process. To address the issue, a data preparation step has been devised to construct a new float array in which vertex coordinates were sorted per triangle. To be more specific, if $\mathcal{M}$ is a mesh with n triangles, then a float array of length $9n$ will hold the vertex coordinates such that the elements $9i$ to $9i + 8$ will correspond to the vertices of the i-th triangle. Because of this, the entire vertex information for each of the mesh triangles will be orderly available in one location and this will facilitate the coalesced memory access (Fig. 4).

3.2. Voxel data

3.2.1. Voxel grid construction

The voxel grid $\mathcal{G}$ is a set of cubes that reside within the Axis Aligned Bounding Box (AABB) of the mesh. To construct $\mathcal{G}$, the AABB is first identified by means of minimum and maximum of mesh vertex coordinates and then divided into identical cubes whose size is user-defined/controlled. This manner of grid construction guarantees that all resulting voxels will be aligned with the axes of the coordinate system and thereby the calculation of the normal vectors for all voxels becomes trivial. In addition, $\mathcal{G}$can be uniquely represented by means of three independent parameters:

(i) corner/origin of the grid $\mathbf{p}_0 \in \mathbb{R}^3$,
(ii) grid dimension *i.e.*, the number of voxels in each direction $\mathbf{d} \in \mathbb{N}^+$ and
(iii) voxel diagonal $\Delta\mathbf{p} \in \mathbb{R}^3$.

To access a voxel $\mathcal{V} \in \mathcal{G}(\mathbf{p}_0, \mathbf{d}, \Delta\mathbf{p})$, the three integers (x, y, z) ranging from $(0, 0, 0)$ to $(d_x - 1, d_y - 1, d_z - 1)$ were used for indexing. As a result, the spatial information of every voxel can be readily calculated. For example, the minimum corner of the voxel $\mathcal{V}(x, y, z)$ is yielded by:

$$\mathbf{o} = \mathbf{p}_0 + < \Delta\mathbf{p}, (x, y, z) > \tag{3.1}$$

where $< >$ denotes the dot product operation.

3.2.2. Voxel information storage

Following the construction of the voxel grid, an array of length $d_x d_y d_z$ was dynamically allocated in order to store material-specific information to be subsequently associated with each voxel. The spatial information of every voxel can be determined by their location in this array. As such, the corresponding index of voxel $\mathcal{V}(x, y, z) \in \mathcal{G}(\mathbf{p}_0, \mathbf{d}, \Delta\mathbf{p})$ in the array is:

$$i = x + yd_x + zd_x d_y \tag{3.2}$$

Conversely, the indices of the corresponding voxel in the 3D grid can be calculated by solving the following equations in the order

$$z = \frac{i}{d_x d_y}, \quad y = \frac{i - zd_x d_y}{d_x}, \quad x = i - zd_x d_y - yd_x \tag{3.3}$$

Therefore, the array represents a mapping between the 3D voxel grid and a contiguous 1D stack of memory which ensures coalesced access to the voxel data.

3.3. Voxelization kernel

3.3.1. Mesh facet/Voxel overlap test

The method utilized here was adopted from [21] and essentially represents an enhanced version of the separating axis theorem [11] that requires a lower number of operations and is thereby faster. The verification of the intersection between triangles and voxels is a four-step process centered on the verification of facet plane/voxel intersection/overlapping. For this purpose, let $\mathcal{T}$ with vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ be a triangular mesh facet and $\mathcal{V}$ be a

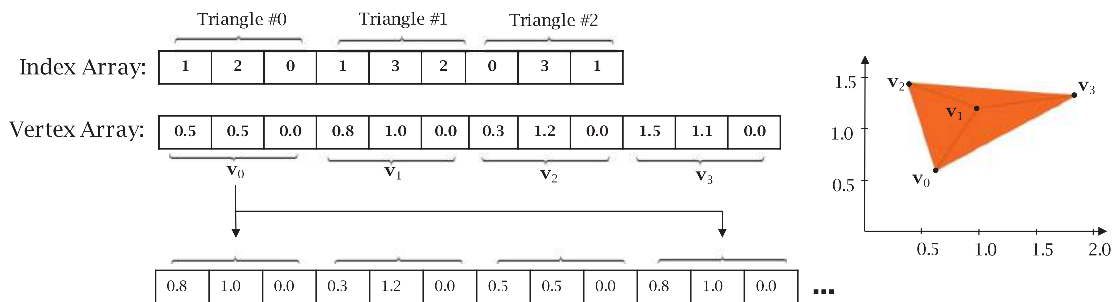


Figure 4. Example of input data preparation.

voxel characterized by the extreme corners $\mathbf{p}$ and $\mathbf{p}+\Delta\mathbf{p}$, respectively. Under these conditions, the facet/voxel overlap test comes down to the calculation of $\mathcal{T}$'s normal and critical point:

$$\mathbf{c} = \left(\begin{Bmatrix} \Delta\mathbf{p}_x, & \mathbf{n}_x > 0 \\ 0, & \mathbf{n}_x \leq 0 \end{Bmatrix}, \begin{Bmatrix} \Delta\mathbf{p}_y, & \mathbf{n}_y > 0 \\ 0, & \mathbf{n}_y \leq 0 \end{Bmatrix}, \begin{Bmatrix} \Delta\mathbf{p}_z, & \mathbf{n}_z > 0 \\ 0, & \mathbf{n}_z \leq 0 \end{Bmatrix} \right) \tag{3.4}$$

followed by the verification of validity for:

$$(<\mathbf{n}, \mathbf{p}> + a_1)(<\mathbf{n}, \mathbf{p}> + a_2) \leq 0 \tag{3.5}$$

where $a_1 =< \mathbf{n}, \mathbf{c} - \mathbf{v}_0 >$ and $a_2 =< \mathbf{n}, \Delta\mathbf{p} - \mathbf{c} - \mathbf{v}_0 >$.

The rest of the intersection test comes down to the evaluation of the projections of $\mathcal{T}$ and $\mathcal{V}$ onto the principal planes of the coordinate system. For instance, the following expressions are to be calculated with respect to XY plane:

$$\mathbf{n}_{e_i}^{\mathbf{xy}} = (-\mathbf{e}_{i,y}, \mathbf{e}_{i,x})^T \cdot \begin{Bmatrix} 1, & \mathbf{n}_z \geq 0 \\ -1, & \mathbf{n}_z < 0 \end{Bmatrix}, \tag{3.6}$$

$$a_{e_i}^{\mathbf{xy}} = - < \mathbf{n}_{e_i}^{\mathbf{xy}}, \mathbf{v}_{i,xy} > + \max\{0, \Delta\mathbf{p}_x \mathbf{n}_{e_i,x}^{\mathbf{xy}}\} + \max\{0, \Delta\mathbf{p}_y \mathbf{n}_{e_i,y}^{\mathbf{xy}}\} \tag{3.7}$$

for all three edges $\mathbf{e}_i = \mathbf{v}_{i+1 \bmod 3} - \mathbf{v}_i$. If the three tests in Eq. (3.8) are all true, then the projections of $\mathcal{T}$ and $\mathcal{V}$ on XY are intersecting.

$$< \mathbf{n}_{e_i}^{xy}, \mathbf{p}_{xy} > + a_{e_i}^{xy} \geq 0, i \in \{0, 1, 2\} \tag{3.8}$$

One of the important advantages of this routine is that if only one of the statements in Eqns. (3.5) or (3.8) are false, then it can be immediately concluded that $\mathcal{T}$ and $\mathcal{V}$ are separated. This approach saves the unnecessary computations and ultimately leads to a faster voxelization algorithm.

3.3.2. Algorithm parallelization

In the context of the present study, the primary target of data parallelization technique were the nested loop structures that could be sped up by either launching a work-item: i) per voxel of the grid, or ii) per mesh facet. In the first approach, the number of work-items required to accomplish voxelization is essentially equal to the total number of grid voxels. Every work-item starts up with voxel selection followed by a loop over mesh facets to verify their relative positioning with respect to the analyzed voxel. As soon as an intersection is detected, the testing loop breaks off and the voxel is assigned a boundary status. At the end of the loop, voxels found in a non-intersecting condition with mesh facets have to be removed from the voxel-based representation of the object. However, the limited size of the local memory requires the storage of the extensive mesh data in global memory and in turn this diminishes the performance of the technique.

Table 1. Comparison between triangle- and voxel-based parallelization schemes for Stanford Bunny model.

	Number of Intersection Tests		
Voxel Size (mm)	Approach 1 (VB)	Approach 2 (FB)	Order of Magnitude Ratio (VB/FB)
3	1,767,431	93	4.2789
2	4,615,507	224	4.3140
1	44,406,021	1,107	4.6033
0.5	368,832,950	5,874	4.7979

In the second approach, the vertices of each triangular facet are assigned to an individual work-item. Since the voxels outside of the AABB of a triangle will not be overlapping with the triangle itself, the voxelization problem reduces in this case to the verification of the relative positioning between the facet and its AABB within the voxel grid. The data in Tab. 1 demonstrates that the switch from Approach 1 (voxel-based = VB) to Approach 2 (facet-based = FB) enables reductions of more than four orders of magnitude for the number of intersection tests required.

Furthermore, each work-item in Approach 2 requires for processing only the coordinates of the vertices for a single triangular facet instead of those of the whole mesh. Since this small amount of information can be handled by the local memory, the number of I/O operations with global memory will be reduced to two: one read of the raw data followed by one write of the results. As such, facet-based parallelization is deemed more efficient than the conventional voxel-based method and Fig. 5 depicts the pseudocode of the kernel based on this method.

3.4. Overall structure of the OpenCL program

An overview of the OpenCL code used to implement parallel voxelization is shown in Fig. 6. In brief, the program starts with the identification of the OpenCL platform that is available through the hardware followed by the selection of a device to perform the required computations. Next, the kernel code is compiled and brought to an executable form to be run on the selected device.

As discussed above, the number of work-items required to complete voxelization is equal to the number of mesh triangles n. Since OpenCL requires that global size is a multiple of the local size, the program has to determine first the maximum local size that is available in the device such that it can set the global size to

```
    INPUT: n, mesh_vertices, p0, d, Δp
    OUTPUT: voxel_array

1   Set i to work-item number
2   IF i < n
3   | Extract v0, v1, v2 of triangle[i] from mesh_vertices
4   | Copy v0, v1, v2, p0, d, Δp to local memory
5   | Calculate the AABB of the triangle[i]
6   | Find the voxels within AABB
7   | FOR j = 1 to number of voxels in AABB
8   | | IF triangle[i]'s plane overlaps voxel[j] THEN
9   | | | IF triangle[i] and voxel[j] projections on xy-plane overlaps THEN
10  | | | | IF triangle[i] and voxel[j] projections on xz-plane overlaps THEN
11  | | | | | IF triangle[i] and voxel[j] projections on yz-plane overlaps THEN
12  | | | | | | Calculate voxel[j]'s location in voxel_array
13  | | | | | | Assign material information to the voxel_array location
14  | | | | | ELSE
15  | | | | | | Continue to the next iteration
16  | | | | | ENDIF
17  | | | | ELSE
18  | | | | | Continue to the next iteration
19  | | | | ENDIF
20  | | | ELSE
21  | | | | Continue to the next iteration
22  | | | ENDIF
23  | | ELSE
24  | | | Continue to the next iteration
25  | | ENDIF
26  | ENDFOR
27  ENDIF
```

Figure 5. Pseudocode of the voxelization kernel.

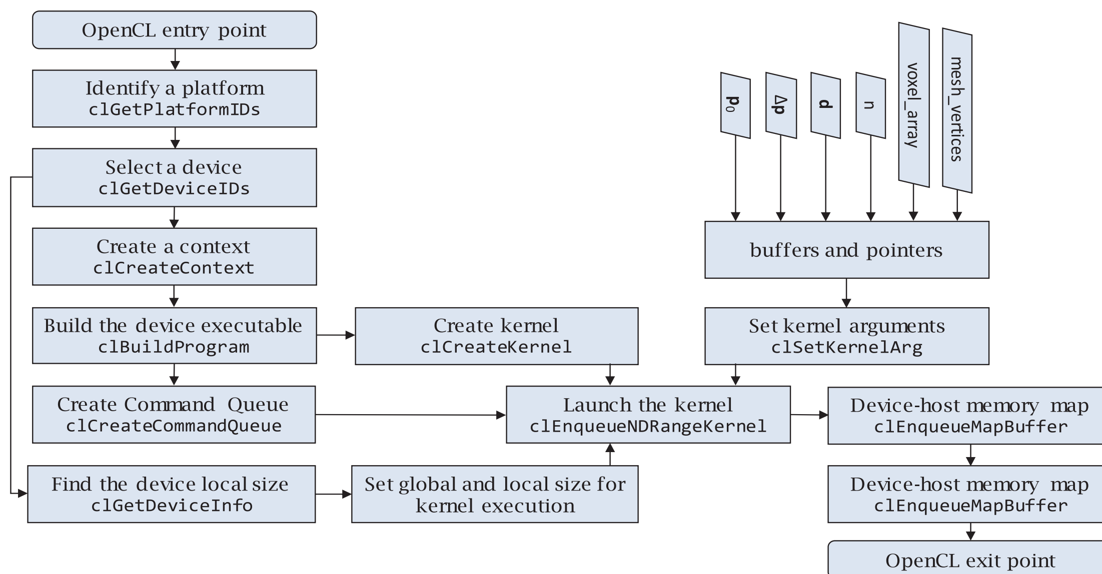


Figure 6. Core structure of an OpenCL algorithm to implement parallelization in the context of a voxelization task.

the appropriate value that is greater than or equal to n. Evidently, OpenCL buffers and pointers will pass further all the variables and arrays required by the kernel. Once the entire data is passed to the device memory and kernel work-items are configured, the command queue will launch the kernel. Following the start of the kernel, a write-only buffer operation is used to pass the voxel data to the host. OpenCL allows two primary modes of device to host memory transfer: i) clEnqueueReadBuffer = reading from the buffer and ii) clEnqueueMapBuffer = mapping device to host memory. The direct comparison of the two options – illustrated in Fig. 7 – implies that memory mapping consistently outperforms the read buffer approach.

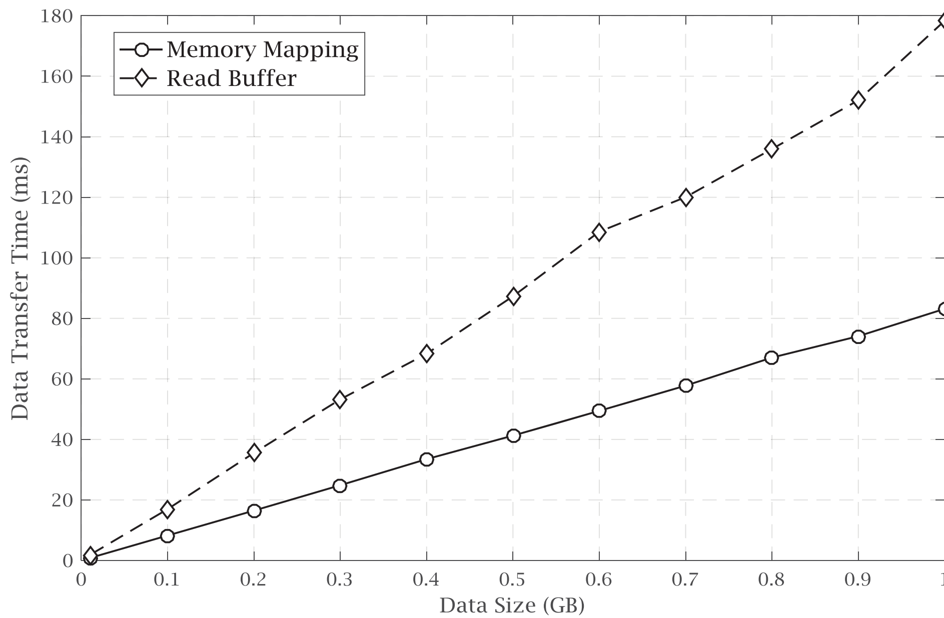


Figure 7. Comparison between techniques used to transfer data between video and system RAM.

Because of this clear superiority, memory mapping was the only method employed for voxel data transfer following the kernel execution. Both techniques were tested when transferring variable amounts of data from the GDDR5 memory of a GeForce 970 GTX graphics card to a DDR4 RAM and as Fig. 7 demonstrates, the advantage of using memory mapping becomes more prominent as the amount of data to be transferred increases.

4. Results

This section presents several test results that were obtained during the evaluation of the performance for the developed voxelization toolkit. The analyzed configurations included high- and mid-range desktop video cards (NVIDIA GeForce 970 GTX, AMD Radeon R7 240), a high-end video card for laptops (Nvidia GeForce 960 GTXM) as well as an integrated desktop video card (Intel HD Graphics 530). Since OpenCL allows CPU, not only GPU parallelization, additional tests were also conducted on desktop Intel 6700 K and AMD FX 770 K processors, as well as the mobile Intel 6700 HQ. Table 2 summarizes the specifications of all the aforementioned hardware components.

Here, the number of compute units indicates the number of work-groups that can be concurrently executed in a device, the maximum local size represents the limit of the work-groups, while the local memory size constitutes the amount of dedicated memory that is available for each of the work-groups within a certain processing unit. As it can be inferred from the entire discussion above, these three parameters play a critical role on the parallelization capabilities of a certain hardware configuration that in turn are dependent on an appropriate partitioning of the problem into parallel work-items. The last two columns of the table, namely maximum clock frequency and global memory, denote the speed of the GPU/CPU processor along with its associated video memory size (for GPU)/system RAM (for CPU), respectively. As expected, the size of the global memory limits the maximum resolution of the voxel grid since – at some point during the execution of the code – the entire voxel data has to be stored in it.

Table 2. Summary of hardware components used in voxelization tests.

Hardware Component Name	Processor Type	Number of Compute Units	Maximum Local Size (KB)	Local Memory Size (KB)	Maximum Clock Frequency (MHz)	Global Memory Size (GB)
NVIDIA GeForce 970 GTX	GPU	13	1,024	48	1,177	4
NVIDIA GeForce 960 GTXM	GPU	5	1,024	48	1,176	4
AMD Radeon R7 240	GPU	6	256	32	780	2
Intel HD Graphics 530	GPU	24	256	64	1,050	1
Intel Core i7 6700K	CPU	8	8,192	32	4,000	16
Intel Core i7 6700HQ	CPU	8	8,192	32	2,600	12
AMD FX 770K	CPU	4	1,024	32	3,493	8

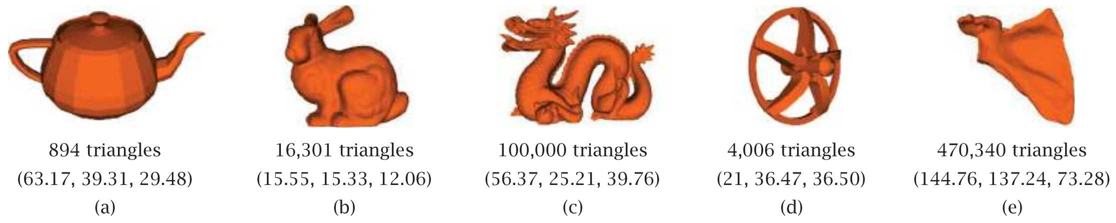


Figure 8. Benchmarked models (including mesh and domain sizes): (a) Utah teapot, (b) Stanford bunny, (c) Stanford dragon, (d) surgical tool/reamer, and (e) scapula.

Table 3. Effect of geometric model, voxel resolution, and hardware configuration on GPU-based parallelization.

			Voxelization Time (ms)						
			GeForce 970 GTX		GeForce 960 GTXM		Intel HD Graphics 530	Radeon R7 240	
Model	Voxel Size (mm)	Voxel Grid Dimension (X, Y, Z)	OpenCL	CUDA	OpenCL	CUDA	**OpenCL**		Single-Thread
Utah Teapot	0.5	127, 79, 59	2.12	1.39	2.07	1.28	2.4	6.49	21
	0.25	253, 158, 118	13.57	8.41	13.1	7.78	14.66	25.9	108
	0.1	632, 394, 295	166.52	102.61	164.68	93.53	151.95	315.31	1081
Stanford Bunny	0.5	32, 31, 25	0.05	0.11	0.09	0.12	0.11	0.21	13
	0.25	63, 62, 49	0.08	0.14	0.16	0.15	0.19	0.22	18
	0.1	156, 154, 121	0.32	0.30	0.66	0.44	0.86	1.81	65
Stanford Dragon	0.5	113, 51, 80	0.31	0.30	0.72	0.53	0.76	1.77	85
	0.25	226, 101, 160	1.09	0.76	2.75	1.70	2.42	4.04	159
	0.1	564, 253, 398	10.17	5.57	25.55	12.96	19.44	34.52	757
Surgical Tool	0.5	42, 73, 73	0.77	0.50	0.75	0.48	0.82	2.46	25
	0.25	84, 146, 146	5.32	3.45	5.13	2.88	5.56	8.51	61
	0.1	211, 365, 366	71.89	41.54	68.84	36.54	69.42	109.23	412
Scapula	0.5	290, 275, 147	2.31	1.34	5.44	2.72	3.8	5.47	420
	0.25	580, 549, 294	11	4.87	25.79	10.06	13.3	24.14	868
	0.1	1488, 1373, 733	112.66	43.68	283.97	88.80	82.10	256.34	4913

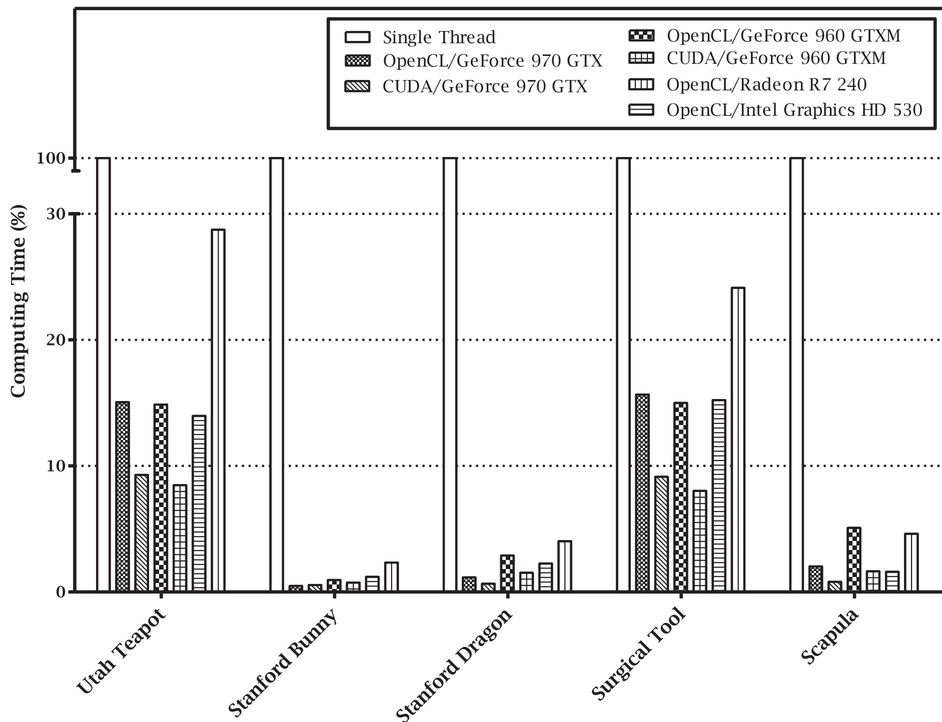
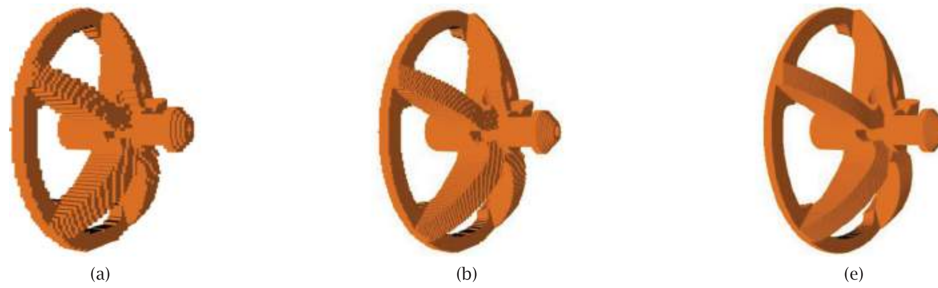


Figure 9. Normalized results of the GPU-based parallelization.



(a) (b) (e)

Figure 10. Sample voxelization results at different voxel sizes: (a) 0.5 mm, (b) 0.25 mm, and (c) 0.1 mm.

Table 4. Effect of geometric model, voxel resolution, and hardware configuration on CPU-based parallelization.

			Voxelization Time (ms)			
Model	Voxel Size (mm)	Number of Voxels in the Grid (X, Y, Z)	Intel 6700K	Intel 6700HQ	AMD FX 77K	Single-Thread
Utah Teapot	0.5	127, 79, 59	3.82	4.85	6.54	21
	0.25	253, 158, 118	20.84	27.5	36.5	108
	0.1	632, 394, 295	244.84	290.92	403.63	1081
Stanford Bunny	0.5	32, 31, 25	0.66	0.71	0.91	13
	0.25	63, 62, 49	1.24	1.91	1.9	18
	0.1	156, 154, 121	5.39	7.66	6.75	65
Stanford Dragon	0.5	113, 51, 80	2.12	3.94	6.17	85
	0.25	226, 101, 160	6.8	7.91	18.53	159
	0.1	564, 253, 398	48.64	55.81	106.87	757
Surgical Tool	0.5	42, 73, 73	2.09	2.72	2.39	25
	0.25	84, 146, 146	9.18	11.79	7.37	61
	0.1	211, 365, 366	94.31	117.2	75.65	412
Scapula	0.5	290, 275, 147	7.59	9.13	32.5	420
	0.25	580, 549, 294	26.94	35.33	82.56	868
	0.1	1488, 1373, 733	244.71	292.21	610.57	4913

The numerical experiments whose results will be detailed further have been conducted on tessellated representative geometries whose principal characteristics are presented in Fig. 8. In addition to the size of the mesh, the data also includes the size of the bounding box.

As it can be noticed, while the Utah teapot features a coarse mesh, both Stanford bunny and dragon are characterized by fine meshes. By returning to the general surgical orthopaedic context of the present study, the last two objects represent the cutting end of a surgical tool used in glenoid reaming procedures along with the geometry of a scapula obtained through the reverse engineering of a patient-specific CT model.

The main metric used during the numerical trials was the voxelization time, *i.e.* the time required to generate voxels at a preset size/resolution for each of the five faceted geometries. To eliminate subjectivity, voxelization time was measured by means of the built-in profiling tool available in OpenCL. The comparison baseline used for all voxelization tests was generated by a single-thread implementation running on an i7-6700 K processor with 16GB DDR4 RAM and the results obtained were summarized in Tab. 3 as well as graphically represented in a normalized form in Fig. 9.

As it can be noticed from the table, although the Utah teapot geometry has the smallest mesh size, its larger domain has determined voxelization times surpassing those associated with Stanford bunny model. On the other hand, the topological complexity of the surgical tool object has led to voxelization times that were sometimes over six times larger than those obtained for Stanford dragon, a geometric object characterized by considerably larger domain and mesh sizes. While the exact root cause of the increase might be difficult to identify, it can be probably assumed that the aforementioned topological complexity of the model has a direct effect on the partitioning of the work-items that – in turn – will significantly prolong the voxelization time.

Moreover, while no definite conclusion can be drawn with respect to GPU performances – other than perhaps the mid-range Radeon R7 240 card seems to be somewhat overall weaker than the rest, the exception being the scapula model for which GeForce 960 GTXM has managed to underperform it – the data shown in Fig. 9 implies that GPU-based parallelization can speed up voxelization anywhere between 73.5% or 3.8 times (surgical tool on Radeon R7 240) and 99.6% or 260 times (Stanford bunny on a GeForce 970 GTX) when compared to single-thread CPU. With respect to OpenCL-CUDA comparisons, while the latter seems to perform better for most NVIDIA-based hardware configurations, its major drawback remains related to its portability to other hardware.

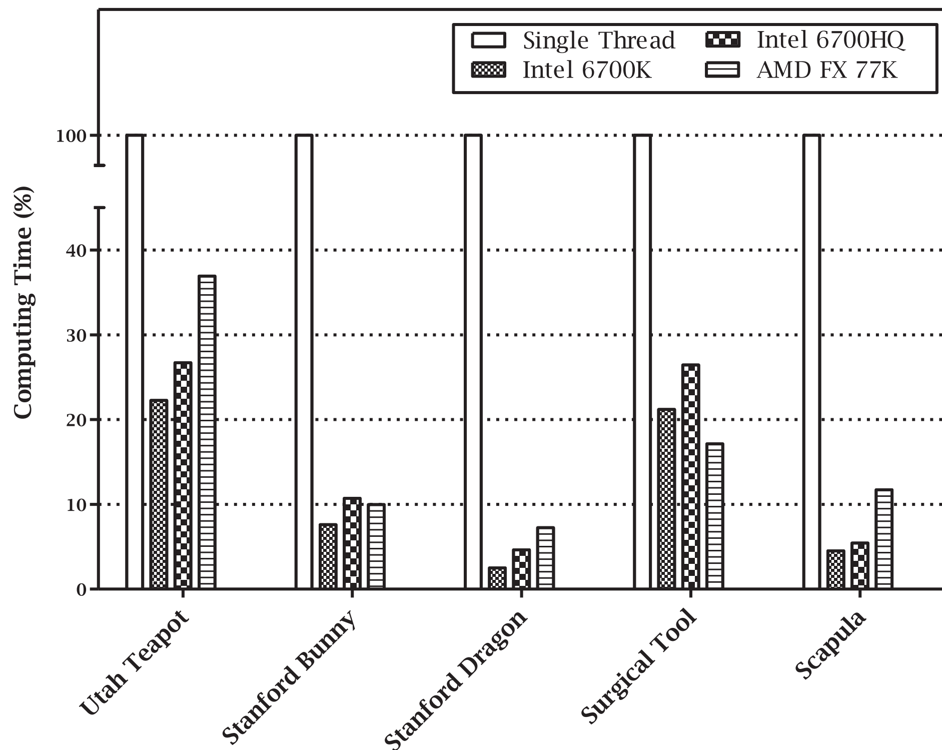


Figure 11. Normalized results of the CPU-based parallelization.

A sample of final voxelized geometry obtained for different voxel resolutions is presented in Fig. 10.

Since CPU-based parallelization is also an option, the next set of tests/numerical experiments have quantified the performances of the three CPUs included in Tab. 2. The results obtained are summarized in Tab. 4 and graphically represented in a normalized form in Fig. 11.

A brief comparative analysis of the results shown in Tab. 3 and 4 suggests that the parallelized execution of the voxelization kernel takes longer on CPUs than on GPUs. However, same as in the GPU-based case, CPU-based parallelization led to voxelization times that were between 62.7% or 2.7 times (Utah teapot on AMD FX 77 K) and 97.5% or 40.1 times shorter than those obtained on the single-thread CPU (Stanford Dragon on Intel 6700 K). Interestingly, the OpenCL voxelization algorithm ran faster on a slower/older CPU (AMD FX 77 K) than it did in the single-thread implementation executed on more powerful CPU (Intel i7 6700 K).

5. Conclusion

The OpenCL algorithm developed in the context of this study represents one of the first attempts made to demonstrate the portability as well as the efficiency of this programming platform in the context of a voxelization task. The actual/final voxelization time is largely dependent on the size/complexity of the tessellated object used as an input, the target voxel resolution as well as the configuration of the computing hardware involved. The results obtained demonstrate that GPU-based parallelization can bring significant reductions of the voxelization – or in general computing – time such that whenever the efficiency of an algorithm is of concern, parallelization of the code should be considered. Furthermore, while CPU-based parallelization might not be as effective as its GPU-based counterpart, significant computation time decreases are still possible even on low-end/less powerful processors. For better or worse, the results of the present study do not identify any of the hardware devices as being an absolute "winner". This essentially means that the decision to select one over the other should always be relative to the computational task at hand that in turn would require a more extensive and in-depth testing of all hardware configurations/options available.

Future extensions of this work will attempt to integrate the current OpenCL voxelization algorithm into a fast material removal technique that it is hoped that will be capable to function in a real-time manner within the virtual orthopaedic surgery platform that is currently under development. Finally, while CUDA remains an efficient and robust parallelization platform, it is hoped that the current results will inspire future OpenCL-based applications whose versatility/independence of vendor-specific hardware offers certain developmental advantages.

Acknowledgements

The authors would like to acknowledge the financial support provided in part by Natural Sciences and Engineering Research Council (NSERC) of Canada and Canadian Institutes of Health Research (CIHR) that was received under the framework of the Collaborative Health Research Projects (CHRP) program.

ORCID

Mohammadreza Faeighi http://orcid.org/0000-0003-0075-2969

O. Remus Tutunea-Fatan http://orcid.org/0000-0002-1016-5103

Roy Eagleson http://orcid.org/0000-0001-9264-8135

References

[1] Akenine-Moellser, T.: Fast 3D triangle-box overlap testing, Journal of Graphics Tools, 6(1), 2002, 29–33. http://doi.org/10.1080/10867651.2001.10487535
[2] Arbabtafti, M.; Moghaddam, M.; Nahvi, A.; Mahvash, M.; Richardson, B.; Shirinzadeh, B.: Physics-Based Haptic Simulation of Bone Machining, IEEE Transactions on Haptics, 4(1), 2011, 39–50. http://doi.org/10.1109/ToH.2010.5
[3] Dinis, J. C.; Moraes, T. F.; Amorim, P. H. J.; Moreno, M. R.; Nunes, A. A.; Silva, J. V. L.: POMES: An Open-Source Software Tool to Generate Porous/Roughness on Surfaces, Second CIRP Conference on Biomanufacturing, 49, 2016, 178–182. http://doi.org/10.1016/j.procir.2015.07.085
[4] Dong, Z.; Chen, W.; Bao, H. J.; Zhang, H. X.; Peng, Q. S.: Real-time voxelization for complex polygonal models, in Proceedings of the 12th Pacific Conference on Computer Graphics and Applications, 2004, 43–50.
[5] Eisemann, E.; Decoret, X.: Fast scene voxelization and applications, in Proceedings of the 2006 symposium on Interactive 3D graphics and games, 2006, ACM: Redwood City, California. p. 71–78.
[6] Eisemann, E.; Decoret, X.: Single-pass GPU solid voxelization for real-time applications, in Proceedings of the Graphics Interface 2008, Canadian Information Processing Society: Windsor, Ontario, Canada. p. 73–80.
[7] Fang, S. F.; Chen, H. S.: Hardware accelerated voxelization, Computers & Graphics, 24(3), 2000, 433–442. http://doi.org/10.1016/S0097\T1\textendash8493(00)00038-8
[8] Fei, Y.; Wang, B.; Chen, J., Point-tessellated voxelization, in Proceedings of Graphics Interface 2012, Canadian Information Processing Society: Toronto, Ontario, Canada. p. 9–18.
[9] Li, W.; Fan, Z.; Wei, X.; Kaufman, A.: Flow simulation with complex boundaries, in "GPU Gems 2", 2005, Addison Wesley Professional, 747–764.
[10] Lin, Y. P.; Wang, X. D.; Wu, F. L.; Chen, X. J.; Wang, C. T.; Shen, G. F.: Development and validation of a surgical training simulator with haptic feedback for learning bone-sawing skill, Journal of Biomedical Informatics, 48, 2014, 122–129. http://doi.org/10.1016/j.jbi.2013.12.010
[11] Nooruddin, F. S.; Turk, G.: Simplification and repair of polygonal models using volumetric techniques, IEEE Transactions on Visualization and Computer Graphics, 9(2), 2003, 191–205. http://doi.org/10.1109/Tvcg.2003.1196006
[12] NVIDIA. CUDA Toolkit Documentation v8.0. [cited 2017 March 7]; Available from: http://docs.nvidia.com/cuda/#axzz4Qpkp88jf
[13] OpenCL. OpenCL C Specifications. [cited 2017 March 7]; Available from: https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf
[14] Patil, S.; Ravi, B.: Voxel-based representation, display and thickness analysis of intricate shapes, in Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics, 2005, 415–420. http://dx.doi.org/10.1109/CAD-CG.2005.86
[15] Razavi, M.; Talebi, H. A.; Zareinejad, M.; Dehghan, M. R.: A GPU-implemented physics-based haptic simulator of tooth drilling, International Journal of Medical Robotics and Computer Assisted Surgery, 11(4), 2015, 476–485. http://doi.org/10.1002/rcs.1635
[16] Scarpino, M.: OpenCL in Action, Manning Publication Company, New York, USA, 2012.
[17] Schwarz, M.; Seidel, H. P.: Fast Parallel Surface and Solid Voxelization on GPUs, ACM Transactions on Graphics, 29(6), 2010, http://doi.org/10.1145/1866158.1866201
[18] Song, C.; Pang, Z.; Jing, X.; Xiao, C.: Distance field guided L1 -median skeleton extraction, The Visual Computer, 2016, 1–13. http://doi.org/10.1007/s00371-016-1331-z
[19] Vankipuram, M.; Kahol, K.; McLaren, A.; Panchanathan, S.: A virtual reality simulator for orthopedic basic skills: A design and validation study, Journal of Biomedical Informatics, 43(5), 2010, 661–668. http://doi.org/10.1016/j.jbi.2010.05.016
[20] Vidim, K.; Wang, S.-P.; Ragan-Kelley, J.; Matusik, W.: OpenFab: a programmable pipeline for multi-material fabrication, ACM Transactions on Graphics, 32(4), 2013, 1–12. http://doi.org/10.1145/2461912.2461993
[21] Wang, F.: Composite model representation for computer aided design of functionally gradient materials, PhD Thesis, Missouri University of Science and Technology, Rolla, Missouri, 2016.
[22] Zhang, L.; Chen, W.; Ebert, D. S.; Peng, Q. S.: Conservative voxelization, The Visual Computer, 23(9–11), 2007, 783–792. http://doi.org/10.1007/s00371-007-0149-0
[23] Zheng, F.; Lu, W. F.; Wong, Y. S.; Foong, K. W. C.: Graphic Processing Units (GPUs)-Based Haptic Simulator for Dental Implant Surgery, Journal of Computing and Information Science in Engineering, 13(4), 2013, http://doi.org/10.1115/1.4024972