Inclusion Test for Polyhedra Using Depth Value Comparisons on the GPU

D. Horvat and B. Žalik

Abstract—**This paper presents a novel method for testing the inclusion status between the points and boundary representations of polyhedra. The method is executed entirely on the GPU and is characterized by memory efficiency, fast execution and high integrability. It is a variant of the widely known ray-crossing method. However, in our case, the intersections are counted by comparing the depths, obtained from the points and the models' surfaces. The odd-even rule is then applied to determine the inclusion status. The method is conceptually simple and, as most of the work is done implicitly by the GPU, easy to implement. It executes very fast and uses about 75% less GPU memory than the LDI method to which it was compared.**

Index Terms—**Inclusion test, GPU processing, computational geometry, polyhedron, point containment.**

I. Introduction

The inclusion test (also called the hit test or containment test) is one of the basic operations in computational geometry [1], [2]. It checks, whether the considered point is located within the boundaries of a given geometric object. Being an elemental operation, it is usually used in conjunction with more complex tasks and can, therefore, be found in various fields such as physics simulations, artificial intelligence, computer graphics or object modeling [3], [4]. Given that the problem is long-established, many different solutions have already been presented. However, new approaches still appear in scientific publications which solve the problem more efficiently.

In this paper, a novel approximation method is presented for the testing of inclusion in 3D scenes, meaning, between points $T = \{t_i\}$, where t_i is defined by the coordinate triple $t_i = (x_i, y_i, z_i)$ and the boundary representation (B-rep) of polyhedrons. While geometry, tested by the proposed method, can be arbitrarily shaped and its surface representation is not necessarily limited to triangle meshes, it is however, required to be manifold and watertight. The method is executed almost completely on the GPU and is most closely related to, the approach that uses layered depth images (LDI). LDI was first introduced by Shade [5] and can be used to establish the inclusion status of a point, based on the Jordan Curve Theorem [4], [6]. The layered structure of LDI, however, is not very memory efficient as the layers can contain many empty allocated values that are not needed to perform the inclusion test. Additionally, multiple geometry passes are required for the construction of LDI, which can represent a problem for objects containing many triangles or dynamic scenes with changing geometry where LDI is constantly recalculated. Motivated by these shortcomings, the following contributions of the proposed method can be outlined:

- **Memory efficiency:** Only a list of point depths and their index map is required, thus making the method very memory efficient.
- **Fast execution:** The method is almost completely executed within the rendering pipeline of the GPU. This enables real-time detection of inclusions on geometrically arbitrary or deformable objects.
- **High integrability:** If used in the right order, most of the code for inclusion can be executed along drawings of objects, which makes the method applicable in computer graphics or in various simulations.

The paper is divided into 4 sections. The following two subsections describe the related work and the background needed for the explanation of the developed method. In Section II, the presented method for the inclusion test is explained. Results are given and debated in Section III, while Section IV concludes the paper.

A. Related Work

The most known method for the inclusion test is the ray-crossing method [7], [8]. It works by casting a ray from the given test point in the random direction and counting the number of intersections with the object boundaries. Inclusion status is determined based on the number of intersections by using the odd-even rule, derived from the Jordan Curve Theorem [9]. A given point, consequentially, lies inside an object if the number of intersections is odd and outside, if it is even. Although the ray-crossing method, in its base form, does not require any preprocessing, the calculations can be accelerated immensely by preprocessing the scene using spatial subdivision structures such as octrees, BSP-trees or kd-trees. Preprocessing, in terms of memory and CPU time, represents the most intensive operation but it is done only once. Hierarchical organization of the data leads to the geometry queries being accelerated from $O(n)$ to $O(\log(n))$ [7].

In the method presented by Feito and Torrez [10], the inclusion is determined by subdividing the geometric object into a set of tetrahedra and calculating their signed volumes. By associating each tetrahedron with the values 1, -1 and 0, based on their volume signs, the inclusion status can be obtained with summation of those values for the tetrahedra within which the point is located. If the value is positive, the point lies inside the polyhedron. The basic version was later modified for use on models with Bézier surfaces [11] and speed up by approximately 10% with the use of a custom data

Manuscript received June 29, 2015; revised December 15, 2015. This work was supported by the Slovenian Research Agency under grants 1000-13-0552, P2-0041, and J2-6764.

Denis Horvat and Borut Žalik are with the Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia (e-mail: {denis.horvat, borut.zalik}@ um.si).

structure [7]. A similar concept is also used in the approach proposed by [12] with a model decomposition called multi-LREP.

Another common way for determining the inclusion status of a point is the voxel space method. The method works by voxelizing the scene and marking each voxel as "inside" or "outside" of the model. Voxelization can be performed very fast on the GPU [13], [14]. The inclusion test then simply consists of determining the voxel in which the point is located and checking the voxels value (this can be done in O(1)). However, partial inclusion states, where geometry partially lies inside a voxel, should be taken into account in which additional test are performed. To obtain as little partial inclusions as possible a high resolution grid is needed.

A unique testing method was presented in [15] where the Horns Theorem is used for the inclusion test, in which, a single determining triangle is required. The authors claim, their method is up to 10 times faster than the ray-crossing method, while the memory requirements remain the same. The layer-based decomposition of models proposed in [6], decomposes a given geometric model to the set of layers based on a fixed viewpoint. This enables that the inclusion tests are performed very fast using the binary search method without singularities as they are removed during the layers construction.

Programmable rendering pipelines of modern graphic cards have enabled the creation of layered depth images (LDI) [5], which can be used for the inclusion test [3], [4]. As the method proposed in this paper expands the concepts based-on LDI, they are explained in the next subsection.

B. Concept of LDI

Layered depth images (LDI), proposed by Shade [5] consist of a set of overlayed LDI layers. Each LDI layer is represented by an image, while its pixels contain normalized distances of the models' surface if observed through a discrete viewing grid. These distances are called depth values and each recorded depth is an intersection with the objects surface. The viewing grid is positioned in 3D space, while its size *width x height* determines the sizes of the LDI layers. To better illustrate the concept of LDI, consider a ray that is cast from the center of each cell in the viewing grid. The rays are cast in the same direction, while being perpendicular to the viewing grid. If a given ray intersects with an object, the depth value of the intersection is saved within a LDI layer. As multiple layers can exist, the layer number in which the given depth value will be saved is dependent on the number of the current intersection. Consequentially, if the *n-th* intersection occurres, the depth is saved into the *n-th* layer, within a pixel with the same coordinates (*x, y*) from which the intersected ray is cast. The number of layers in LDI is therefore equal to the maximum number of intersections between an object and any of the cast rays. The extent of this number depends on the object's complexity, and the orientation, size and position of the viewing grid. The concept of LDI is outlined within Fig. 1(b), where the depths of a dragon model from Fig. 1(a) are saved as LDI layers. When LDI is constructed, the inclusion status for a given point t_i consists of only calculating its cell position in the viewing grid and counting how many depth values in LDI at the same cell position are larger (or smaller) than the depth value of the point. As seen in Fig. 1(c), the odd-even rule can then be efficiently applied to determine the points inclusion status. Three cases are shown, where in 1) and 2) the point is located within the geometry and in 3) the point lies outside the geometry.

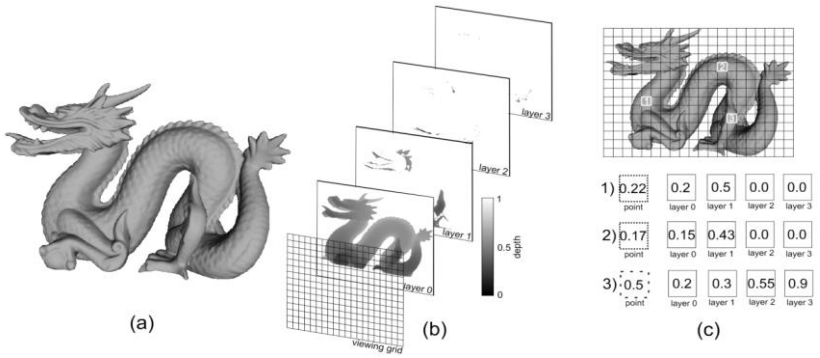

Fig. 1. Procedure for determining the inclusion status of points within a (a) 3D scene, from which (b) a set of depth layers is constructed and (c) 3 cases for testing the inclusion are demonstrated, where in 1) and 2) the point lies inside the model and in 3) it is outside.

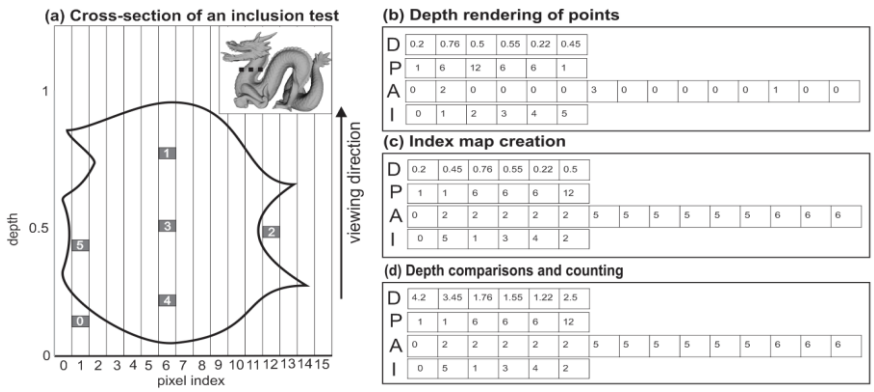

Fig. 2. An example of a 3D scene, where (a) the inclusion test for points, marked with their associating indexes, is shown on a cross-section of a model (the location and length of the cross-section is marked with the dotted line on a model in the upper right corner). The values within the arrays are shown after (b) depth rendering of points, (c) index map creation and (d) depth comparisons and counting of intersections.

The biggest advantage of the LDI method is that it can be easily implemented on the GPU using the programmable rendering pipeline available in modern graphical libraries such as OpenGL or DirectX. LDI layers can be created by simply rendering the object and saving the depths, required by the depth test, into a fixed set of 2D textures. Most of the work, such as rasterization, pixel index determination and calculation of the depth values is done implicitly by the GPU. The rendering parameters Ω, which include an orthogonal projection matrix and the viewport size, specify the viewing grid. The projection matrix defines the left, right, bottom and top clipping planes, which limit the area of the scene that will be rendered. The clipping planes are set to be slightly larger than the width and height of the bounding box that belongs to the tested object. The viewport size determines the number of pixels in the LDI layers and consequentially, the accuracy of the method. After the creation of LDI layers, the second step only consists of rendering the points using Ω, while comparing their depths with the same pixel positions within the LDI layers and incrementing a counter when these values are bigger.

Several problems are, however, apparent when using the LDI method. Namely, to construct LDI on the GPU, the object has to be rendered at least twice. Two rendering passes are required because it is not known how many depth layers will be generated and the memory used by the LDI has to be allocated beforehand. Therefore, during the first pass, the number of required LDI layers is calculated, while the second pass fills the actual layers with depth values. This two-step approach causes a bottleneck between the GPU and CPU as, after the first step, it is necessary to copy the information about the number of layers onto the CPU. The CPU then allocates the required LDI memory and initiates the second step. Copying data between the GPU and CPU is not particularly fast and should be avoided if possible. It is worth mentioning that there exists another way of creating LDI. This method is called depth peeling [4] and requires even more rendering passes as the described two-step approach (each LDI layer requires its own rendering pass). Regardless of how it is obtained, LDI is also not particularly memory efficient as many pixels within LDI layers are unused. The unused pixels generally increase with the number of layers (see layers 2 and 3 in Fig. 1 (b)). This can accumulate to large portions of empty allocated memory, which cause unnecessary memory deficiency when using higher viewport sizes. The memory problem only worsens if, to improve the accuracy [6], additional attributes like surface normals are attached to each pixel of the layer. The so-called layered depth normal images (LDNI) quadruple the memory consumption or at least double it if the layer data are, at the cost of the numeric accuracy, organized as proposed by [3]. To address these shortcomings, a new method is developed which is described in the next section.

II. Methodology

Similar to the previously described LDI method, the proposed method also uses the odd-even rule to determine the inclusion status of a point. It is primarily designed to be executed on GPU using the rendering pipeline during the rendering of objects. The latter means that the following assumptions are made: (1) Rasterization is implicitly done by the GPU, while fragment depths of the rendered objects and currently processed pixel positions are available during execution; (2) the method's logic is executed after rasterization and independently for each pixel; (3) the pixels are processed concurrently in an arbitrary order by the numerous cores available to the GPU which are orchestrated by the graphics driver.

The memory consumption problem is addressed by reversing the procedure of the LDI method. The depths of the points are therefore saved first, and the comparison with the objects' depths is done last. This way, LDI does not need to be created nor saved and the memory consumption is, consequentially, smaller while being mostly independent of the used viewport size. This also means that the tested object does not need to be rendered twice as the number of LDI layers does not need to be calculated. The proposed method generally operates using the arrays $D = [d_0, d_1, \ldots d_n]$, $I = [i_0, i_1, \ldots i_n]$, $P = [p_0, p_1, \ldots, p_n]$ and $A = [a_0, a_1, \text{and } a_m]$, where n represents the number of tested points and m the number of rendered pixels. $D \subset \mathbb{R}$ contains the depths of the tested points, $I \subset \mathbb{Z}$ the indexes to points to which the depths belong, $P \subset \mathbb{Z}$ the indexes to pixels where the points are located and $A \subset \mathbb{Z}$ the number of points contained within each pixel. As A is an 1D array, each processed pixel is always mapped to its 1D index px using $px = width * x + y$. The method consists of the following three steps:

- **Depth rendering of points** allocates the required GPU memory and renders the points during which D, A, P and I are filled.
- **Index mapping** step creates an index map by modifying D, I and A using P, such that points can be accessed directly from the pixels in which they are located.
- **Depth comparisons and counting** step renders the object, while comparing and counting the intersections based on depth values.

While individual steps are explained in the following subsections, an example in Fig. 2 demonstrates the idea and shows the values in the arrays after each step.

A. Depth Rendering of Points

Algorithm 1: GLSL code used by the fragment shader in OpenGL for the depth rendering of points.

```
layout (binding = 0, r32f) uniform imageBuffer D;
layout (binding = 1, r32ui) uniform uimageBuffer U;
layout (binding = 2, r32ui) uniform uimageBuffer P;
layout (binding = 3, r32ui) uniform uimageBuffer I;
uniform vec2 window;

void main( void )
{
      int pntIdx = gl_PrimitiveID;
      int pixIdx = int((window.x*gl_FragCoord.x)
                                    + gl_FragCoord.y);

      imageStore(I, pntIdx, uvec4(pntIdx,0,0,0));
      imageStore(D, pntIdx, vec4(gl_FragCoord.z,0,0,0));
      imageStore(U, pntIdx, uvec4(pixIdx,0,0,0));
      imageAtomicAdd(P, pixIdx, 1);
      discard;
}
```

The main purpose of this step is the GPU memory allocation and rendering of points while filling D, A, P and I. Firstly, the rendering parameters Ω, which remain constant

throughout all 3 steps, are initialized in the same way as when using the LDI method (described in subsection I.B). The required memory for all arrays is then allocated on the GPU. Points are rendered using Ω and during rendering *D*, *A*, *P* and *I* are filled with their associated values, which are provided implicitly by the GPU. Algorithm 1 shows the rendering code needed to execute the described step.

Note that the program at the end of the execution, if no actual rendering output is needed, can call the *discard* command. Consequentially, the fragment does not proceed to the next pipeline stage (i.e. per sample processing), which further speeds-up the inclusion test. At the end of the first step, four arrays reside on the GPU as depicted by the example in Fig. 2(b).

B. Index Mapping

In this step, a mapping scheme is created so that the points can be accessed from the individual pixels and then be compared in next step with the depths of the tested object. This is achieved by performing a prefix sum on *A* and sorting *P* in an ascending order, while modifying *D* and *I* along with it. The prefix sum and sorting can be done entirely on the GPU. They are consequentially executed very fast and no data is ever copied between GPU and CPU. Since several GPU algorithms exist for both sorting [16] and the prefix sum [17], they are not discussed here. Efficient implementations can be found within the AMD SDK (available at http://developer.amd.com/tools-and-sdks/). The state of *D*, *I* and *A* after the prefix sum and sorting can be seen in Fig. 2(c). The points can now be accessed from individual pixels (i.e. from *A*). This can be done by firstly calculating the number of points t_{px} in the current pixel *px*. t_{px} is obtained by subtracting the values in *A* at the location equal to the current *px* and one index before *px*, i.e. $t_{px} = A[px] - A[px - 1]$. If $t_{px} = 0$, no points are located within *px* and if $t_{px} > 0$, then the pixel contains t_{px} points. For example, in Fig. 2(c), the number of points in pixel $px = 6$ is 3 as $t_6 = A[6] - A[5] = 5 - 2 = 3$. These points can then be accessed through *D* and *I* on an interval $[A[px - 1] .. A[px])$. Expanding the described example, the depths and indexes to points for $px = 6$ can are located in *D*[2], *D*[3], *D*[4] and *I*[2], *I*[3], *I*[4]. Now that the points can be accessed from *px*, rendering of the geometry can be made, where the depths are compared.

C. Depth Comparisons and Counting

The number of intersections is counted in this step. This is done by rendering the geometric object using Ω. During rendering, the depths of the points, saved within *D*, are compared with the depth value from a part of the geometric object that is currently rendered within a given *px*. If the depth value is larger than the depth of the point, this counts as an intersection and the counter can be incremented. To further save the memory, the increments are done on *D* as depth values are normalized to values between *[0,1)* and only the decimal values of the number are needed. Therefore, the non-fractional part of *D* now contains the number of intersections for a given point, while the fractional part represents its depth. Algorithm 2 shows the rendering code needed to execute comparison and saving the results.

After the rendering of object, *D* is copied onto CPU where the odd-even rule is applied to determine the inclusion status for each point.

Algorithm 2: GLSL code used by the fragment shader in OpenGL for the counting of intersections between the points and the rendered object.

```
layout (binding = 0, r32f) uniform imageBuffer D;
layout (binding = 1, r32ui) uniform uimageBuffer I;
layout (binding = 2, r32ui) uniform uimageBuffer A;
uniform vec2 window;
void main( void )
{
        int px = int((window.x*gl_FragCoord.x)
                                + gl_FragCoord.y);
        int nIt = int(imageLoad(A, px).r);
        int npIt = 0;
        if(px > 0)
            npIt = int(imageLoad(A, px-1).r);
        for(int i = npIt; i < nIt; i++)
        {
            float pntDepth = fract(imageLoad(D, i).r);
            if(pntDepth < gl_FragCoord.z)
            {
                int pntIdx = int(imageLoad(I, i).r);
                imageAtomicAdd(D, pntIdx, 1);
            }
        }
        discard;
}
```

TABLE I: NUMBER OF VERTICES AND TRIANGLES ABOUT THE MODELS USED FOR TESTING

Model	Num. vertices	Num. triangles
Bunny	35947	69451
Horse	48485	96966
Dragon	100250	202520
Heptoroid	286678	573440
Brain	294012	588032
Budha	543652	1087716

TABLE II: THE MEASURED EXECUTION TIME AND MEMORY CONSUMPTION OF LDI AND THE PROPOSED METHOD ON VARIOUS MODELS FOR 100000 POINTS

Model	LDI method			proposed method	
	num. LDI layers	GPU memory [MB]	time[s]	GPU memory [MB]	time [s]
Bunny	13	53.2	0.013	**6.48**	**0.011**
Horse	10	41.2	0.013	**6.94**	**0.010**
Dragon	10	41.2	**0.014**	**8.8**	**0.014**
Heptoroid	11	45.2	**0.016**	**15.55**	0.019
Brain	15	61.2	0.017	**15.7**	**0.013**
Budha	15	61.2	0.015	**24.77**	**0.012**

III. RESULTS

In order to test the proposed method, the point's inclusions were tested on 6 different models containing various amounts of vertices and triangles. Information about the models is shown in Table I.

The tests were performed using a workstation with Intel® Core™ i5 CPU, 32GB of main memory and AMD Radeon R9 200. The implementation was done using C++ and a rendering pipeline provided by OpenGL 4.0. Memory consumption and execution times were compared with the LDI method described in subsection *I.B*. For each model, the inclusion test was performed on 100000 points which were randomly distributed within the models bounding box. A viewport of resolution 1000 × 1000 was used for rendering. As both methods are render-based and their execution times can vary based on the models' rotation. The execution times were therefore measured in the following way: the model was rotated around its *x*, y, and *z* axes for all combinations in 5 ° increments and the execution time of the inclusion test was measured before each increment. The average execution time

of all the measurements was then taken as the result. Memory consumption of the LDI method was calculated as the sum of the memory required by the LDI layers and the tested points. The memory required by the proposed method was equal to the sum of the memory occupied by the arrays *D, P, A, I* and the geometry of the model that was tested. The result for the execution time and the GPU memory consumption is given in Table II where the better results between the two comparisons are written in bold.

While the proposed method executes only slightly faster, it uses far less GPU memory than the LDI method (on average about 75% less). The proposed method also scales more practically in terms of memory because its memory requirements rise with the number of points and can therefore be better regulated based on the memory capacity available to the GPU. For example, if the method is used for particle simulation, fewer particles would be spawned if the graphics card lacked the required GPU memory. In the case of the LDI method, however, LDI layers occupy most of the required memory and would still have to be generated even if only one point is tested. As both described methods are approximate, the number of pixels in the rendering viewport is aimed to be as high as possible. The memory consumption of the LDI method however is, in contrast to the proposed method, highly dependent on the resolution of the viewport. This is because all LDI layers have the same resolution as the viewport and LDI can contain, in the case of very complex models, upwards of 20 layers or more, which requires a lot of GPU memory if the rendering resolution is high. The impact of resolution in terms of memory consumption is minimal when using the proposed method as only the index array *A* is affected. The memory is, consequentially, also unaffected by the complexity of the models. Additionally, the majority of the memory required by the proposed method is occupied with the geometry of the models. As the inclusion test is usually a part of more complex operations (e.g. physics simulation), the geometry will, in most cases, already be loaded onto the GPU (e.g. for visualization purposes). Consequentially, if the proposed inclusion test is executed in conjunction with drawing a shader code, then the memory, which is solely required by the inclusion test, is drastically reduced (in our case, only 5.2 MB of memory would be required for each test). This is not possible when using the LDI method as the model cannot be reconstructed out of LDI layers. Additionally to the relatively low memory consumption, the method also executes very fast. Note that no preprocessing is ever done, which means that the method is suitable for dynamic scenes with deformable objects.

IV. Conclusion

This paper proposes a novel method for testing the inclusion of points in polyhedra. It works on the principle of ray-crossing which counts the intersections between a ray cast from the tested point and the given geometric object. The logic is designed in such a way that it can be executed on the GPU, which enables very fast processing times. It does also not require any preprocessing or complex data structures as the majority of the work, such as rasterization and depth calculation is done by the GPU. Being influenced by LDI, its memory requirements are much lower, while its execution times are slightly faster. Although being approximate, it can regulate its accuracy by increasing the viewport resolution with low increase in GPU memory.

References

[1] B. Žalik and I. Kolingerova, "A cell-based point-in-polygon algorithm suitable for large sets of points," *Computers & Geosciences*, vol. 27, no. 10, pp. 1136-1145, 2001.

[2] J. Li, W. Wang, and E. Wu, "Point-in-polygon tests by convex decomposition," *Computers & Graphics*, vol. 31, no. 4, pp. 636-648, 2007.

[3] C. Wang, Y. Leung, and Y. Chen, "Solid modeling of polyhedral objects by layered depth-normal images on the GPU," *Computer-Aided Design*, vol. 42, no. 6, pp. 535-544, 2010.

[4] B. Heidelberger, M. Teschner, and M. Gross, *Volumetric Collision Detection for Derformable Objects*, Zurich, 2003.

[5] J. Shade, S. Gortler, L. He, and R. Szeliski, "Layered depth images," in *Proc. 5th Annual Conference on Computer Graphics and Interactive Techniques*, 1999.

[6] W. Wang, J. Li, H. Sun, and E. Wu, "Layer-based representation of polyhedrons for point containment tests," *Visualization and Cumputer Graphics*, vol. 14, no. 1, pp. 73-83, 2008.

[7] C. Ogayar, R. Segura, and F. Feito, "Point in solid strategies," *Computers & Graphics*, vol. 29, no. 4, pp. 616-624, 2005.

[8] D. Horvat, "Ray-casting point-in-polyhedron test," in *Proc. CESCG*, Smolenice, 2012.

[9] T. Hales, "Jordans proof of the Jordan curve theorem," *Studies in Logic, Grammar and Rhetoric*, vol. 10, no. 23, pp. 45-60, 2007.

[10] F. Feito and J. Torres, "Inclusion test for general polyhedra," *Computer & Graphics*, vol. 21, no. 1, pp. 23-30, 1997.

[11] A. Garcia, J. Miras, and F. Feito, "Point in solid test for free-form solids defined with triangular Bézier patches," *The Visual Computer*, vol. 20, pp. 298-313, 2004.

[12] F. Martinez, A. Rueda, and F. Feito, "The multi-LREP decomposition of solids and its application to a point-in-polyhedron inclusion test," *The Visual Computer*, vol. 26, no. 11, pp. 1361-1368, 2010.

[13] E. Eisemann and X. Decoret, "Single-pass GPU solid voxelization for real-time applications," *Proceedings of Graphics Interface*, Toronto, 2008.

[14] Y. Fei, B. Wang, and J. Chen, "Point-tessallated voxelization," *Proceedings of Graphics Interface*, Toronto, 2012.

[15] J. Liu, J. Maisog, and G. Luta, "A new point containment test algorithm based on preprocessing and determining triangles," *Computer-Aided Design*, vol. 42, no. 10, pp. 1143-1150, 2010.

[16] H. Peters, O. Hildebrandt, and N. Luttenberger, "Fast in-place sorting with CUDA based on bitonic sort," in *Proc. the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, 2007.

[17] M. Harris, "Parallel Prefix Sum (Scan) with CUDA," in *Proc. GPU Gems 3*, Addison-Wesley Professional, 2007.

Denis Horvat received his B.Sc. and M.Sc. in computer science from the Faculty of Electrical Engineering and Computer Science at the University of Maribor in 2011 and 2013, respectively and is currently a Ph.D. candidate.

He is a young researcher at the Laboratory for Geometric Modeling and Multimedia Algorithms at the University of Maribor. His areas of research include computational geometry and pattern recognition in remote sensing data.

Borut Žalik received the B.Sc. degree in electrical engineering in 1985, the M.Sc. and Ph.D. degrees in computer science in 1989 and 1993, respectively, all from the University of Maribor, Maribor, Slovenia.

He is a Professor of computer science at the University of Maribor. He is the head of the Laboratory for Geometric Modeling andMultimedia Algorithms at the Faculty of Electrical Engineering and ComputerScience, University of Maribor. His research interests include computational geometry, geometric data compression, scientific visualization and geographic information systems.