Visualization of Three-Dimensional Ultrasound Data

H. Hrženjak, Ž. Mihajlović
Faculty of Electrical Engineering and Computing, University of Zagreb, Republic of Croatia
helena.hrzenjak@fer.hr, zeljka.mihajlovic@fer.hr

***Abstract* – This paper presents visualization of an ultrasonic probe beam from a series of cross-section images. The goal was to simulate scanning with an ultrasonic probe as it would be done in real-life. Cross-section slices that represent the detection sensitivity of the probe were obtained by simulating a sound pressure field in the metal sample and were provided by INETEC. Different techniques for rendering the three-dimensional model of a sequence of images used for beam visualization in the making of the software are presented. The beam and its inner structure were rendered with the Volumetric Ray Marching algorithm while the polygonal mesh approximation was constructed with the Marching cubes algorithm. The final software was developed using the Unity engine. The results and visualization possibilities were presented and analyzed along with the possible improvements for the future work. The developed simulation software can be used as a helping tool in minimizing the number of redundant scans in search of defects inside an object structure during the quality control phase.**

Keywords - volume rendering, volumetric ray casting, ultrasonic probe, Unity, shaders, three-dimensional dataset

I. INTRODUCTION

Volumetric rendering is a collection of important techniques in the field of computer graphics. They are a powerful tool for data visualization as they offer a way to display data visually, offering easier and more intuitive ways to understand raw data.

Volume rendering is most notably used in medical imaging, for helping medical staff to see inside a body more accurately, using medical images obtained from procedures like computerized tomography (CT) or magnetic resonance imaging (MRI) three-dimensional models can be constructed. Besides medicine, volume rendering is finding more uses in other industries such as those that produce 3D data sets for analysis, e.g., physics for fluids, flood preparation simulations and more.

In this work, volume rendering and other techniques are used to achieve ultrasonic beam simulation. Ultrasonic beam scanners are used for non-destructive material testing of tools and metal parts for nuclear power plant inspections. The final 3D model of the beam was recreated from a series of 2D cross-section images using the ray marching technique and the marching cubes algorithm.

The main goal was to create a simulation that will visualize defect detection on a certain trajectory and make probe positioning in real testing quicker and more efficient. The models and data used in the implementation were provided by INETEC and collected using their instruments. The goal was to simulate movement of a UV probe during detection. Observing the best trajectory in the simulation could reduce the time to position the probe and scan for defects in practice.

II. ULTRASONIC PROBES

A. Representation of the pressure field

Ultrasonic probes are devices that are used for investigating the internal structure of a target object. The detection sensitivity of an ultrasonic probe can be closely estimated by the strength of the sound pressure field it generated in the specimen. The sensitivity field visualized as a heat map can be seen on the Fig 1.

The rainbow colormap was used for coloring the sound pressure field, the dark blue color represents lower pressure field density values and thus lower detection sensitivity while the dark red represents high values meaning high sensitivity. The series of this kind of images, generated with different settings - multiple UT probe angles and different focal depths, will be used as an input data for the visualization.

The formula for calculating pressure field is complex and therefore using it to calculate the pressure field multiple times for each setting would take some time. That is why visualization from images is a good option, there is no need for complex formula calculations to get good results. However, the downside of this approach is that it needs pre-generated images.

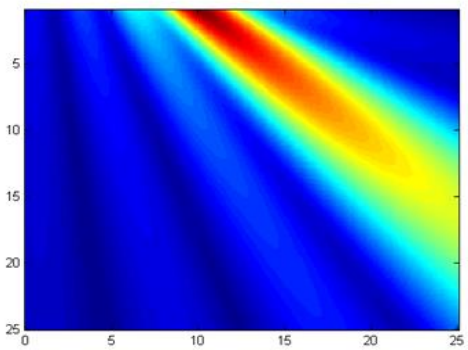

Fig. 1. Representation of a sound pressure field of the probe at a 45-degree angle

B. Ultrasonic sensors

Ultrasonic sensors are instruments that use ultrasonic waves for inspection of subsurface structures and detection of anomalies. Ultrasonic sensors work by emitting sound waves at frequencies in the ultrasonic frequency range - range greater than twenty kilohertz, higher than the audible range of human hearing. Ultrasonic sensors emit ultrasonic waves toward a target and if there is an obstacle or an object in the way, the wave will bounce back to the sensor's receiver. The distance of an obstacle is then calculated using the time the reflected wave took to return to the receiving sensor after being emitted.

Ultrasonic inspection is a family of methods most widely used for nondestructive testing, it offers a great accuracy, high sensitivity but, most importantly, it does not cause damage to an object being tested, as such it finds its application in many fields. The most familiar usage being in the field of prenatal medicine for medical screening of a fetus during pregnancy. Besides that, ultrasonic inspection is also widely used for quality control and material inspection across all major industries, including electrical and electronic component manufacturing, production of metallic and composite materials, and fabrication of structures such as airframes, engines, machinery and many others [1].

III. VISUALIZATION TECHNIQUES

Displaying data in a visual form provides clear and easily understandable representation of raw data. The main problem in visualizing volume data is displaying three-dimensional data as a two-dimensional image while not losing too much information and not losing valuable information [2]. Volume rendering and surface rendering represent the most important techniques for three-dimensional dataset visualization.

A. Surface rendering

Surface rendering method involves constructing polygonal surfaces in a dataset and rendering those surfaces. The assumption is that the original volume can be faithfully represented as a collection of polygonal surfaces. Surface rendering algorithms have a necessary preprocessing step, also called image segmentation, which involves determining a surface by extracting features from the volume data. More precisely - every pixel in an image is assigned a label such that pixels with the same label share certain visual characteristics such as color, intensity, or texture. These pixels are a part of one isosurface. The geometric primitives are then fit to the data to form a 3D surface based on the values extracted from the data in preprocessing step and rendered for display using conventional geometric rendering techniques [3].

The process of determining surface is not perfect, as it must be determined for every voxel how the surface passes through every cube of voxels. This issue is especially apparent for datasets surrounding objects that are very small, blend into their surroundings or describe poorly defined features [4]. A frequent problem that can occur is calculated surface containing voxels that don't really belong to the original object we are trying to visualize or skipping the voxels that do belong. Because of the mentioned issues, when surfaces are transparent or semi-transparent, geometric rendering techniques may not be the best choice, that is where the volume rendering comes in.

B. Volume rendering

Volume rendering is a powerful technique for the representation, manipulation and rendering of volume data. Traditional graphics techniques represent 3D objects as geometric surfaces and edges approximated by polygons and lines, but volumetric datasets can contain data and information not only on the surface, but also inside them, which is where surface rendering techniques fall short. Volume rendering techniques make it possible to explore the inner structures of volumetric data and allow visual representation of transparent and complex datasets.

The advantage of volume rendering algorithms is that there is no need to determine surfaces in advance which removes the need for the segmentation step and a polygon representation, instead they render every voxel in the volume raster directly, without explicit conversion to geometric primitives. Volume visualization methods can achieve soft surfaces by integrating the contribution from the voxels in entire volume towards final rendered image [5]. Therefore, this method is applicable even in cases where dataset can't be faithfully described by polygons or has a lot of small details that can't be lost. It is also applicable in cases when geometric surfaces for dataset are unavailable or too cost-ineffective to generate.

IV. VOLUMETRIC RAY CASTING

Volumetric ray casting, also commonly referred to as Volume ray tracing, is the most common technique for achieving image-based volume rendering.

The 'ray casting' part of the name comes from the similarity with the traditional ray casting in that they both define rays that represent beam from a camera's viewpoint and only consider these primary rays without spawning the secondary, reflected and refracted, ones. The difference is that the ray casting interacts only with surface data and stops as soon as it encounters an intersection between a ray and a surface in a scene while volume ray casting does not stop the computation at the surface level but passes through the object, sampling it along the way [6].

This technique shows the best results in materials where light goes through the object. Examples would include smoke, marble, and skin. When light hits one of these surfaces, a fraction of the light is reflected (ray casting only deals with this case) and a fraction is scattered into the object. Ray marching is a technique to compute the scattered light. Image projection is performed by simulating the absorption of light along the ray path to the eye and calculating light intensity when arriving at the camera. The advantage of this technique lies in the fact that it does not directly calculate intersections with the objects in the scene, so no mesh data of the objects needs to be provided.

Volumetric ray marching employs simplified Emission-Absorption Optical Model, taking into consideration, as per its name, emission and absorption while ignoring light scattering as it is too computationally complex while not affecting quality too much. This means it only deals with

primary rays, unlike some more common graphic rendering methods, no secondary rays like reflection or refraction or shadows are considered. The Emission-Absorption Optical Model places two assumptions: the volume is assumed to consist of particles that emit light. The volume is assumed to consist of black particles that absorb light [16]. Meaning the portion of light that passes through a particle will be absorbed. Volume rendering integral is an equation that computes light intensity at a point s along the ray, taking into consideration light attenuation.

$$I(s) = I(s_0)e^{-\tau(s_0,s)} + \int_{s_0}^{s} q(s_1)e^{-\tau(s_1,s)}\,ds_1 \qquad (1)$$

Where parts of the equations are:

- $I(s_0)$ is initial light intensity at point s_0, s_0 being a starting point
- $e^{-\tau(s_0,s)}$ is intensity reduction from s_0 to s
- $I(s_1)$ is light intensity (emission) at point s_1, s_1 being a point on the ray further down than point s_0
- $e^{-\tau(s_1,s)}$is intensity reduction from s_1 to s
- τ is an optical depth measure, measuring degree of absorption of light
- $\int_{s_0}^{s} q(s_1)e^{-\tau(s_1,s)}\,ds_1$ gathers contributions of all points on the ray between s_0 and s_1

There is no closed form solution to this integral, but it can be numerically approximated with a discrete sum of volume samples along the ray [7].

Each point in the volume is considered to emit and absorb light, and thus contributes to the final color and opacity. There are multiple compositing schemes for deciding the output color. The most common one is Volume compositing or accumulation. For the emission-absorption model, the accumulated color and opacity are computed according to a following set of equations:

$$C_{out} = C_{in} + (1 - \alpha_{in})\alpha C \qquad (2)$$

$$\alpha_{out} = \alpha_{in} + (1 - \alpha_{in})\alpha \qquad (3)$$

These equations are known as Front-to-Back Compositing Equations since we go from the front of the volume to the back, collecting color on the way. C_{out} is the resulting accumulated color over n voxels seen from the front of the volume [8]. Variable α_{out} is the opacity of the result. Marching from beginning of the ray towards the end accumulating color and opacity using these equations is repeated until bounding box is exited, or some other stopping criteria is defined e.g., predefined maximal opacity or number of steps is exceeded. Another scheme that is going to be mentioned in continuation is Maximum Intensity Projection. It simply takes color of the highest density point along the ray.

V. MARCHING CUBES ALGORITHM

The Marching cubes algorithm is a computer graphics algorithm for creating polygonal representation of surfaces with constant density – isosurfaces. It was first presented in the paper *Marching cubes: A high resolution 3D surface construction algorithm* [9]. The algorithm processes the 3D data in a scan-line order and calculates triangle vertices using linear interpolation. The density of the desired surface is specified as an input from the user.

The Marching cubes algorithm belongs to the surface rendering algorithms. Its advantage is that it works on lookup tables so it's relatively high speed. The algorithm works on a divide-and-conquer principle, it divides the problem of creating surface to the small cells, it focuses on each cell and applies to it one iteration of the algorithm. The first step of the algorithm would be splitting the space into a uniform grid of cells, in three dimensions that would be cubes. Secondly, the isovalue or a density of a desired isosurface is defined, in this case the beam density. Next step is assigning a Boolean value to each voxel according to its value in a relation to the designated isovalue – value one if the voxel's value is greater than the isovalue, meaning that the voxel is inside the isosurface or lies on the surface and value zero if it is lesser than the isovalue and the voxel is outside of the isosurface.

For each cube eight pixels that make up cube vertices are taken into polygon evaluation. Four vertices are pixels of one slice and the other four belong to adjacent slice. The surface intersects cubes at edges where one vertex is outside the surface and another one is inside, calculating the exact intersection location with interpolation in later steps of the algorithm. There are eight vertices in each cube and two possible states for each one, equating to $2^8 = 256$ ways a surface can intersect the cube [10]. These 256 possible cases of surface-edge intersections are enumerated and can be searched found in the lookup table. The states given to each voxel in the previous step are now composed into an 8-bit index, where each bit corresponds to a Boolean value assigned to the vertex. That index is used to fetch data from the lookup table. Algorithm iterates over all cubes, adding triangles to a mesh, and the final mesh is the union of all added triangles. The smaller the cubes, the smaller the mesh triangles will be, making approximation more closely match the target function.

VI. BEAM VISUALIZATION

Visualizing the beam is the most important part of the project. The visualization was realized in Unity with the ray marching technique using 3D textures and shaders. The shaders were written in High-Level Shader Language (HLSL). Results are three-dimensional even though inputs are two-dimensional images.

High-level shading language is a programming language used for writing programmable shaders in DirectX. It is very similar to the Cg shader language and will be used to write the shaders in the next steps. The Shader that will draw the ultrasonic beam implements volume rendering using ray marching algorithm and as a result gives a rendered volume representing the ultrasonic beam. Programmable parts of the graphics pipeline are vertex shader and fragment shader [15]. The vertex shader processes and performs transformations on individual vertices. Fragment shader is where GPU calculates the final RGB color for every pixel. The implementation is done in Unlit shader type as the beam doesn't need to be affected by lightning models.

Most important task for the vertex shader is transforming vertices' coordinates to the clip space so that they can be used by rasterizer and be projected on the screen. The vertex shader receives object space coordinates as its inputs, the conversion from object space to the clip space is done by putting vertices through a sequence of transformation matrices. To make things easier, all the transformation matrices can be combined into one matrix by the name Model View Projection matrix (MVP). In Unity there is a built-in helper function UnityObjectToClipPos(float3 pos) that does exactly that, it takes a position point and directly transforms it from an object space to the camera's clip space in homogeneous coordinates [11].

Fragment stage is where the ray marching will be done. There is a ray created through each pixel, pixel's local position is the ray origin, and the ray direction is the vector from camera to the pixel. Direction vector is calculated by using helper function ObjSpaceViewDir() and inverting it, since it returns opposite direction, from pixel to camera. Using this ray, the unit cube in which volume is drawn is intersected. The points of intersection are helpful for calculating marching step size. Distance between the start and the end point of intersection is divided by the defined maximum step size. The max step size limit is there because of performance concerns, if the number of steps is too great visualization would not run smoothly in real-time.

Finally, the ray marching is executed in the loop that runs for a number of steps. In each step the density is read from the B&W 3D texture and the color from the colored 3D texture (or from the transfer function) using adjusted local vertex position.

These 3D textures are a special Unity Asset type, they are bitmap images that contain information in three dimensions as opposed to the standard two. Texture coordinates are placed on a unit cube and texture data is accessed by three-dimensional texture coordinates. This kind of data type is fit for storing and manipulating volumetric data. The 3D texture containing volumetric data was built from a sorted sequence of 2D images through a script and is of the input properties for the shader. What is done with color calculation in the next step depends on the technique chosen.

In the compositing technique color is sent to the method where accumulation formulas are used for collecting and interpolating colors through the steps. If the pixel density does not satisfy threshold the pixel color is not taken into calculation. Compositing can be terminated early if the currently accumulated opacity is so high it makes contributions of future samples insignificant.

In Maximum intensity projection (MIP) technique the color of a voxel with maximum intensity along the ray is returned after checking all the pixels along the same ray.

In both cases the final pixel color is returned. The color of the beam can be chosen to either be read from a 3D texture or from a defined transfer function. The case of using textures was implemented in a way that a color for the voxel is read from the texture in the ray marching algorithm, while when using transfer function, the density is first calculated using ray marching and then at the end, based on that value the color is read from the transfer function.

A. Transfer function

Transfer functions define mapping from voxel density to RGB color and opacity based on user defined values. Using transfer functions provides a better overview of the data. Transfer functions can be used to highlight important parts and omit unnecessary ones thus better emphasizing nuances inside the volume. Transfer function takes the scalar value, in this case density and relates it to a corresponding color and transparency, for example if it is desirable to highlight higher density parts higher opacity should be assigned to those densities [15]. Deciding how a transfer function should look is not an easy task, there is no fixed guide to perfect function because of variations from dataset to dataset.

In Unity transfer function can be created by using a 2D texture with height of one, this is enough since needed information, color and opacity, can be stored in one pixel. Texture color is read and set based on input gradient stored in form of a Unity's Gradient class.

B. Collision

For detecting collisions Unity's default colliders were not close enough of an approximation for the beam, so the mesh had to be reconstructed from the images. The reconstruction was accomplished with the Marching cubes algorithm. The script inputs are a stack of B&W images and a minimum density value defined by the user, that value is going to be considered as the threshold in the script.

In the preprocessing step when assigning labels for each pixel the slices were traversed line by line, in each one the first pixel with equal or higher density than defined was assigned a label one and all subsequent ones until the pixel below the threshold. After labeling the pixels were used in mesh creation using the Marching cubes algorithm with interpolation. The resulting triangles were decided using the case table and together they formed the resulting beam mesh. The mesh data is then saved in an instance of Unity's Mesh class.

The script is set up by attaching it to the desired GameObject that must have a Mesh Renderer and a Mesh Filter component. After mesh creation the Mesh collider can be added to the beam, Unity automatically adapts the Mesh collider to the selected Mesh - in this case the beam.

Beam Collision script listens for the collision events and reacts by calling function that changes material color on a defect when it is under collision to a color that can be seen easily. After the beam collider is no longer in contact with the defect the material color is reverted to the original one. OnCollisionEnter() and OnCollisionExit() are physics engine specific functions that are triggered on update frames when the collision is detected and when the contact between colliders has stopped.

C. User interface

The User Interface is divided into three sections, which can be seen in Fig.2. The first section, on the left, consists of settings regarding beam visualization. Most important ones are Focal depth and Refraction angle, these settings

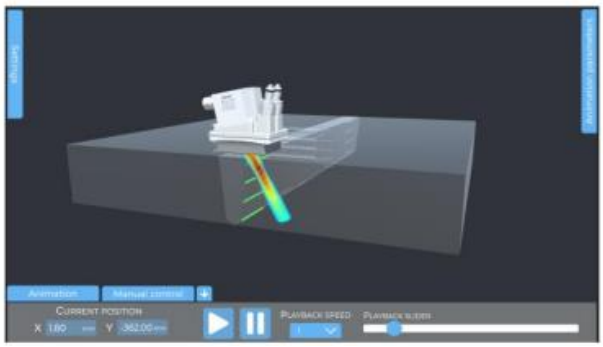

Fig. 2. Highlighted defects result.

can be chosen from their drop-down lists, each combination of the two maps to the one set of images and by extent textures that will be used for visualization. For output color there are Max Intensity Projection and Compositing options. For Compositing the choice of current Z-slice is enabled, by using the slider the user can select any cross-section on z-axis within the beam for a more detailed view. The panel on the right is tasked with handling the input of the auto-movement path parameters. The user can fine tune the course of simulation by inputting the values for the desired simulation output in the suitable fields. Once all the parameters are confirmed automatic simulation will be ready to be played.

The middle section, at the bottom of the screen, controls the movement of the probe model. Animation tab contains buttons for playing and pausing and the slider for rewinding the animation. Playback speed can be slowed down or accelerated. In the manual control tab user can either input exact probe position or use two sliders for selecting horizontal line position and sliding UT probe by a selected increment along it. The components used in the process of building the interface were extended from the Quantum UI package [12] which was downloaded from the Unity Asset store web page. Another custom package that was used during UI development for smoother UI transitions is LeanTween [13].

VII. RESULTS

In this chapter, the results of the implementation are presented and discussed. The comparison between beam visualization using two different technique types and two color acquisition methods will be shown. Since results are visualization, the described results are mostly observations. Numerical approximation of the result accuracy can be explored in the future work.

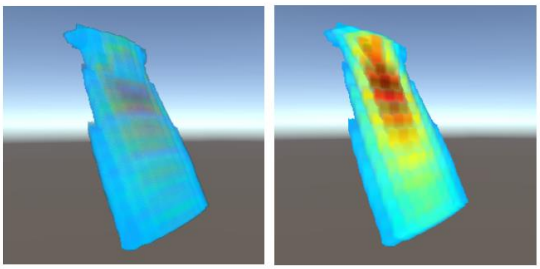

Fig. 3. Comparison between Compositing(left) and MIP (right) using color from texture and a cutoff density = 0.3 at a 30-degree refraction angle.

As previously explained, the beam can be rendered with two techniques: Maximum intensity projection and Compositing. On Fig.3. when using Max projection, the highest density can be clearly seen from all of the angles (an effect more visible in movement than on images) while with Compositing we can see color accumulation effect, most of the beam is in shades of blue as it is the color most present in the photos, so it contributes the most, but there are also some traces of red in the middle.

Max intensity projection results can be improved upon by adding transparency. With this feature enabled voxels with lesser density values will be more transparent than the ones with the higher ones. This makes important parts of the beam stand out more. The result is outer blue rays being less visible than the middle ones in yellow-red range. Transparency effect is achieved by setting alpha color value to depend on voxel density, in this case it is just multiplied by a parameter in the shader.

To get better insight of inner structure there are two options: removing parts outside of desired density segments or slicing off the parts of the volume that are not needed. Extracting segment from the beam is done by selecting minimum and maximum density value, all the other parts that are not included in this range are going to be transparent as can be seen on Fig.4. Cutting the volume along one of the axes makes it possible to select a slice or a part of the volume that is of interest. For example, on the z-axis if maximum and minimum are set, everything in front of the minimum and behind the maximum z-value will be removed.

Coloring the volume after calculating densities can be done in different ways. The most common coloring method for volume rendering is using transfer function, since it is easily adjustable and can define both color and transparency. Another advantage of the transfer function is that we do not need additional color texture, saving time and space. The advantage of reading color from texture is that everything is colored exactly like in the original dataset, there is no approximation and thus it is more similar to the source data. Also, it saves the time that would otherwise be needed for defining perfect transfer function for a particular dataset. Even though the B&W texture used for density calculation was lower resolution the transfer function gave very smooth results, which is not the case with a color from texture one where it caused a bit of a blocky look to colors. If the input images were of better

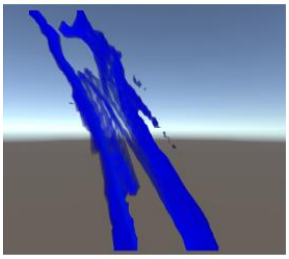

Fig. 4. Result beam with an alpha value between 0.07 and 0.115, only volume with density inside that range is visible.

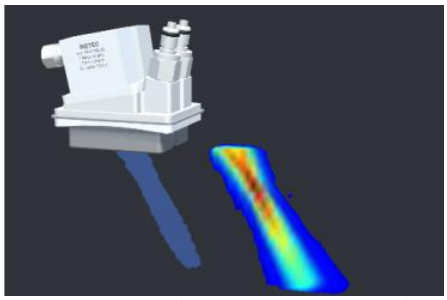

Fig. 5. Mesh reconstruction for the density threshold of 0.3 and the original visualized beam

resolution the result would quite probably be significantly better and more similar to the transfer function result.

The focus depth change from 30 to 60 mm causes the rendered result to consist of a more intense and larger surface of red area in comparison with the 30 mm one, this difference shows how the ray with depth change shifts focus to a deeper depth, around 60 mm deep from probe position. The refraction angle, as the name says, changes the angle of the probe and in extension angle in which rays enter the block. The probe can be imagined as if it was positioned in the top left corner.

The marching cubes algorithm result is a mesh that approximates the beam. On the Fig.5. the comparison between the generated mesh and the beam using the density threshold of 0.3 can be seen. The mesh is lightly shaded on the image for the comparison, but it can be disabled from rendering in simulation if desired.

VIII. Future work

Future work could improve upon offering more scanning path options. Possibility to simulate scanning of complex surfaces like the curved surfaces or the surfaces with obstacles would make this program applicable for use in a wider number of cases since real objects are not regularly shaped blocks and commonly have welds and other obstacles on trajectory. Implementing this feature would be more complex than it is for flat surfaces. Additional complications to path calculation would present obstacles as they would prevent the probe from moving freely. Another small issue is that the results with current images come out a little pixelated, the cause is that the current images were generated in dimensions of 43 x 31 pixels to save time for a course of development. If the simulation images can be regenerated in a bigger resolution, with their quality improved so would the quality results, the final visualization would be clearer and more accurate.

The results could be evaluated numerically, to calculate how close the simulation represents the real-life beam and detection.

IX. Conclusion

The possibilities and advantages of using volume rendering in displaying datasets have been explored by applying described techniques and algorithms to a practical problem of visualizing ultrasonic beam for quality inspection. The achieved simulation can be used as a helping guide in minimizing the number of redundant scans in search of defects inside object structure during quality control phase. The implementation was done using Unity and writing custom shaders, results achieved showcased one possibility of use of volume rendering in the quality inspection field. The biggest limitation of the final application is no support for complex trajectories and that could be a potential course for future work.

Acknowledgments

I would like to extend my gratitude to INETEC for supplying the materials, 3D models, image sequences and providing mentoring help in the creation of this paper.

References

[1] L. J. Bond. “Fundamentals of Ultrasonic Inspection[1]” In: Nondestructive Evaluation of Materials. ASM International, Aug. 2018.,URL:https://www.asminternational.org/documents/10192/22533690/05511G_SampleArticle.pdf/a8c3979c-3b35-f304-68f2-151271f2e3b8.

[2] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. “Volume Rendering”. In: SIGGRAPH Comput. Graph. 22.4 (June 1988), pp. 65–74. ISSN: 0097-8930. DOI: 10.1145/378456.378484. URL: https://doi.org/10.1145/ 378456.378484.

[3] Andreas G. Schreyer and Simon K. Warfield. “Surface Rendering”. In: 3D Image Processing: Techniques and Clinical Applicatio Ed. by Davide Caramella and Carlo Bartolozzi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 31–34. URL: https://doi.org/10.1007/978-3-642-59438-0_4.

[4] Heavy.AI. Volume Rendering. URL: https://www.heavy.ai/technical-glossary/volume-rendering (visited on 06/2022)

[5] Multiple authors. Volume Rendering. URL: https://www.sciencedirect. com/topics/computer-science/volume-rendering (visited on 06/2022)

[6] ETH Zurich, “Direct Volume Rendering “, https://cgl.ethz.ch/teaching/former/scivis_07/Notes/stuff/StuttgartCourse/VIS-Modules-06-Direct_Volume_Rendering.pdf

[7] Tino Weinkauf. Introduction to Visualization and Computer Graphics. 2015. URL: https://www.kth.se/social/files/565e35dff27654457fb84363/08_VolumeRendering.pdf

[8] M. Ikits, J. Kinss, A.Lefohn, C. Hansen, “GPU Gems, Chapter 39.Volume Rendering Techniques”, https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-39-volume-rendering-techniques

[9] William Lorensen and Harvey Cline. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. In: ACM SIGGRAPH Computer Graphics 21 (Aug. 1987), pp. 163–. DOI: 10.1145/37401.37422.

[10] Paul Bourke. Polygonising a scalar field. URL: http://paulbourke.net/ geometry/polygonise/ (visited on 06/2022).

[11] Unity Technologies. Unity User Manual. URL: https://docs.unity3d.com/Manual/ (visited on 06/2022)

[12] Quantum Tek. Unity Asset Store package - Quantum UI. URL: https : / / assetstore.unity.com/packages/tools/gui/quantum-ui-162077 (visited on 06/2022)

[13] Dented Pixel. Lean Tween. URL: https://assetstore.unity.com/packages/tools/animation/leantween-3595.

[14] L. Radisson, “Raytracing Studies“ https://observablehq.com/@makio135/raytracing-studies

[15] Randima Fernando and Mark J. Kilgard. “Chapter 1. Introduction, Chapter 4. Transformations”. In: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley Professional, 2003. URL: https : / / developer.download.nvidia.com/CgTutorial/cg_tutorial_frontmatter.html.

[16] Tino Weinkauf. Introduction to Visualization and Computer Graphics. 2015. URL: https://www.kth.se/social/files/565e35dff27654457fb84363/08_VolumeRendering.pdf