UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3044

VISUALIZATION OF THREE-DIMENSIONAL ULTRASOUND DATA

Helena Hrženjak

Zagreb, June 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3044

VISUALIZATION OF THREE-DIMENSIONAL ULTRASOUND DATA

Helena Hrženjak

Zagreb, June 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 11 March 2022

MASTER THESIS ASSIGNMENT No. 3044

Student:	**Helena Hrženjak (1191236479)**
Study:	Computing
Profile:	Computer Science
Mentor:	prof. Željka Mihajlović

Title: **Visualization of Three-Dimensional Ultrasound Data**

Description:

Explore visualization methods of three-dimensional volumetric data. For a series of cross-sections of ultrasonic data obtained on a metal sample, develop appropriate visualization method. Develop appropriate shaders for volumetric rendering. Develop different mechanisms of interaction in the created environment. Discuss the influence of parameters of the models on the output. Evaluate the results and the implemented algorithms. Implement the appropriate software solution. Use C# programming language and Unity game engine. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Submission date: 27 June 2022

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 3044

Pristupnica:	**Helena Hrženjak (1191236479)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Vizualizacija trodimenzionalnih ultrazvučnih podataka**

Opis zadatka:

Proučiti postupke vizualizacije trodimenzionalnih volumnih podataka. Za niz poprečnih presjeka ultrazvučnih podataka ostvarenih na metalnom uzorku ostvariti vizualizaciju volumnih podataka. Načiniti odgovarajuće sjenčare volumnih podataka. Ostvariti različite mehanizme interakcije u kreiranom okruženju. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski alat Unity i programski jezik C#. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 27. lipnja 2022.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

Volumetric rendering is a collection of very important techniques in the field of computer graphics, they are a powerful tool for data visualization, offering an easy and intuitive way to understand data since they create visual data representation and with images, we can often see more than with raw data. In this thesis we are going to take a closer look at some important visualization techniques that are used on the transformation path from two-dimensional stack of images to achieving its faithful copy in three dimensions. The one that is of special interest is direct volume rendering and it will be the focus of this thesis. Volume rendering is most notably used in medical imaging, for constructing three-dimensional models to help medical staff and to more accurately see inside of the body, the data this type of visualization uses are medical images obtained from procedures like CT or MRI imaging. But volume rendering is finding more and more uses in other industries such as those that produce 3D data sets for analysis, e.g., physics for fluids, disaster preparedness simulations and more.

In this work we are applying volume rendering and other techniques to ultrasonic beam simulation. The ultrasonic beam is used for non-destructive material testing of metal parts for nuclear power plants and tools for inspection. The main part and focus of this simulation was the visualization of the ultrasonic detector beam. The final 3D model was recreated from a series of 2D cross-sections using ray marching technique and marching cubes algorithm. The steps and all used techniques will be described in the thesis.

The practical problem to which the covered techniques were applied to in this thesis was done as a part of INETEC Institute's project. The main goal was visualizing defect detection on a certain trajectory to make real life probe positioning quicker and more efficient. The ultrasonic probes, models and probe data used in the implementation were all provided by INETEC and collected using their instruments. The goal was to simulate how a probe would move and detect defects as it would if it was moved the same way in reality and with that reduce time of scanning for defects.

2. Ultrasonic probes and sound pressure

Ultrasonic probes are used for investigating internal structure of a target object. Depending on the position of a defect, with a regard to the volume and position of the probe itself, the ultrasonic probe will either be or not be able to detect the defect below the volume's surface. The detection sensitivity of a probe can be closely represented 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 figure 2.1. As the rainbow colormap was used, the dark blue color represents lower sound pressure field values and thus lower detection sensitivity while the dark red represents highest values meaning high sensitivity. The series of this kind of images, generated with different settings, in this case - multiple UT probe angles and different focal depths, will be used for the visualization.

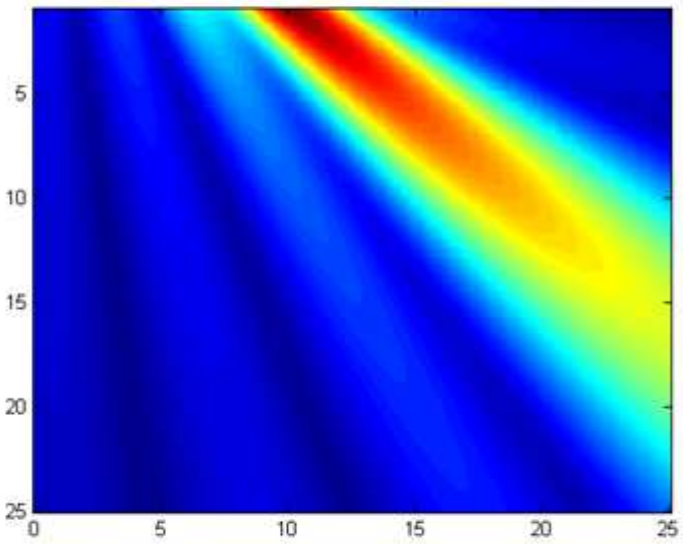

Figure 2.1: Representation of pressure field below the probe at a 45 degree angle

The sound pressure field can be computed using an implicit formula of the type:

$$p(\mathbf{x}) = \rho_1 c_1 v_0(\omega)\sqrt{\frac{2k_1 b}{\pi i}}\frac{1}{N}\sum_{n=1}^{N}\frac{T_{p0}(\theta_{10}^n)\exp(ik_1 b\overline{r}_{10}^n + ik_2 b\overline{r}_{20}^n)}{\sqrt{\overline{r}_{10}^n + \frac{c_2}{c_1}\frac{\cos^2\theta_{10}^n}{\cos^2\theta_{20}^n}\overline{r}_{20}^n}}D_{b/N}(\theta)$$

Figure 2.2: Formula for calculating pressure field

The formula 2.2 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. One downside of this approach is that it needs pre-generated images.

2.1. Ultrasonic sensors

Ultrasonic inspection is a collection of methods in which beams of ultrasonic waves are emitted into material for detection and properties of subsurface anomalies, flaws and structures of a said material. This way of detection offers a highly reliable and non-destructive way to discover internal structure. Ultrasonic sensor is an instrument that uses ultrasonic waves for flaw detection and inspection.

The sensors of this type work by emitting sound waves at frequencies in the high-frequency ultrasonic range, greater than 20 kHz, this range is higher than the normally audible range of human hearing. Ultrasonic sensors transmit ultrasonic waves toward a target and if there is an obstacle or an object on the way, sound wave will bounce back to the receiver. The distance of obstacle is then calculated by measured time the reflected waves took to return to the receiving sensor taking into consideration the speed of sound inside targeted object.

2.1.1. Working principle

To better understand how sensors like these work functional units they typically consist of will be described in detail in continuation. Pulser is an electronic device that can generate electrical pulse of high voltage. The transmitting transducer generates a beam that emits ultrasonic waves when triggered by bursts of voltage generated by pulser. Transducer contains important element that converts electrical energy into acoustical

one and vice versa. The generated energy once introduced into the materials propagates in the form of waves. Transducers can be one single element but phased arrays with multiple elements are more common nowadays.

A couplant is a material, usually liquid, that facilitates transmission of ultrasonic waves between the transducer and test surface. Couplant is necessary because of the large acoustic impedance mismatch between air and solids, this is the consequence of the speed of sound being varied in different materials, so when a sound wave strikes the border between the two, some of the wave reflects while some is transmitted into the second medium. The couplant helps in displacing the air and getting more sound energy into the test specimen, with more energy a more usable ultrasonic signal can be obtained.[10]

A receiver is tasked with accepting the output of ultrasonic waves. In most systems, transducer and receiver functions are integrated into single unit and ultrasonic element then alternates between emitting and receiving signals. Receiver and transducer can also be separated, in that case they are placed side-by-side as close together as possible, because when the receiver is close to the transmitter, sound travels in a straighter line on its way from the transmitter to the detected object and back to the receiver, yielding smaller errors in the final measurements, but errors are still going to be bigger than in same unit integration sensors. The reflected wave signal is transformed into an electrical signal by the transducer and is recorded on a display device for analysis.

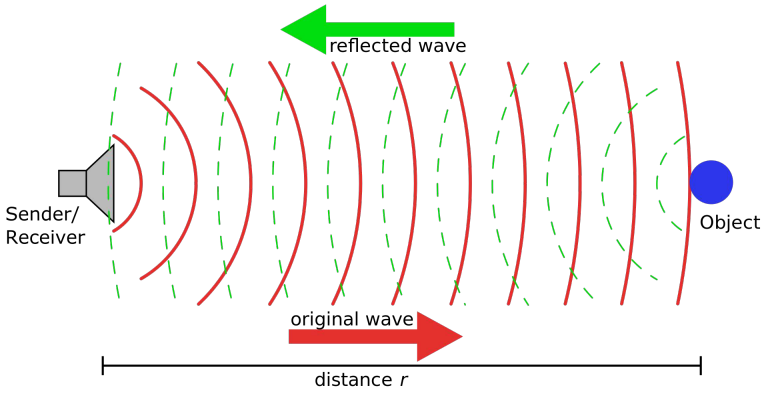

Figure 2.3: UT sensor working principle [7]

Besides these integral parts its some additional parts to highlight are display and an electronic clock. A display or some indicator is there to characterize and record the output from the test piece. The display device nowadays is most commonly a computer screen, which is an integral part of many units; a full copy of data is typically

recorded and kept for subsequent analysis. An electronic clock serves as a primary time reference point for the entire system. In many cases the pulser, receiver, display, and clock are combined into a single integrated unit.[2]

The mechanical waves travel through the material and interact with interfaces between different material and discontinuities, including flaws and other anomalies. Response signals are detected and then displayed and analyzed giving information used to define the presence, location, and characteristics of flaws.

Ultrasonic data can be collected and displayed in a range of formats but the three most common ones are known as A-scan, B-scan, C-scan and S-scan presentations as seen on image 2.4. Each mode provides a different view of looking at the material region being inspected.

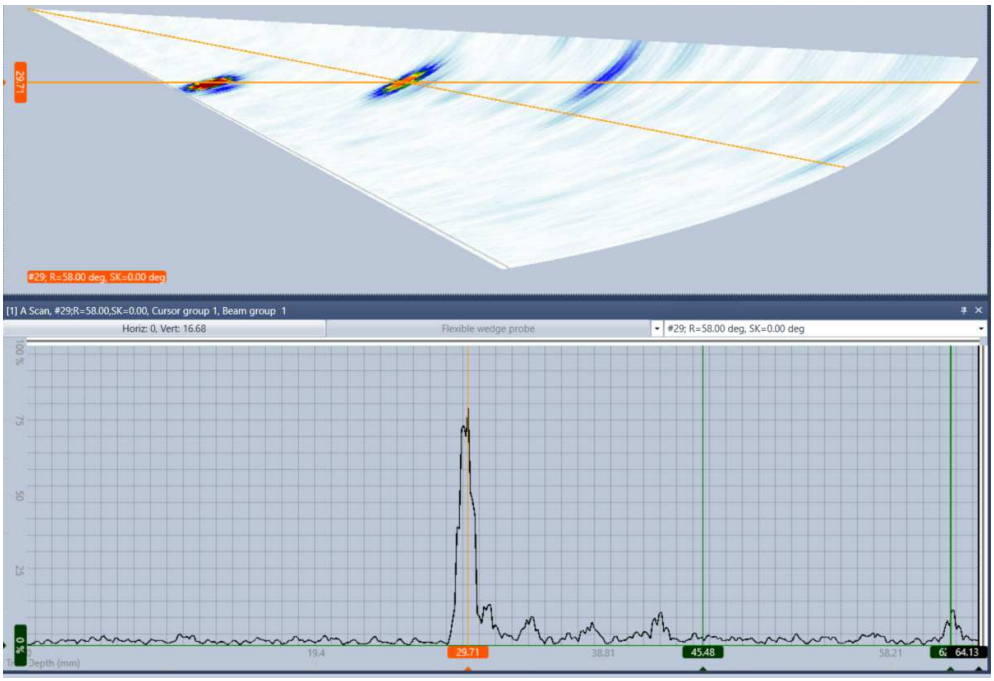

Figure 2.4: S-scan above and A-scan below

2.1.2. Real life applications

Ultrasonic inspection is one of the most widely used families of methods employed for nondestructive testing. The use of non-destructive techniques is necessary for the analysis of internal properties of structures without causing damage to the materials. In the last decade, ultrasonic techniques have shown to be very promising for non-destructive inspection offering advantages over other methods thanks to the ultrasound being safe and not damaging to the object being tested, great accuracy and high sensitivity for

detection of extremely small flaws are only some of the extra benefits. As such it finds its application in many fields, the most familiar usage is in prenatal medicine for medical screening of fetuses during pregnancy. Besides that, ultrasonic inspection is also widely used for quality control and material inspection in 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.[2] Accompanied by the rapid development of information processing technology, new fields of application, including factory automation equipment and car electronics, are increasing and should continue to do so.

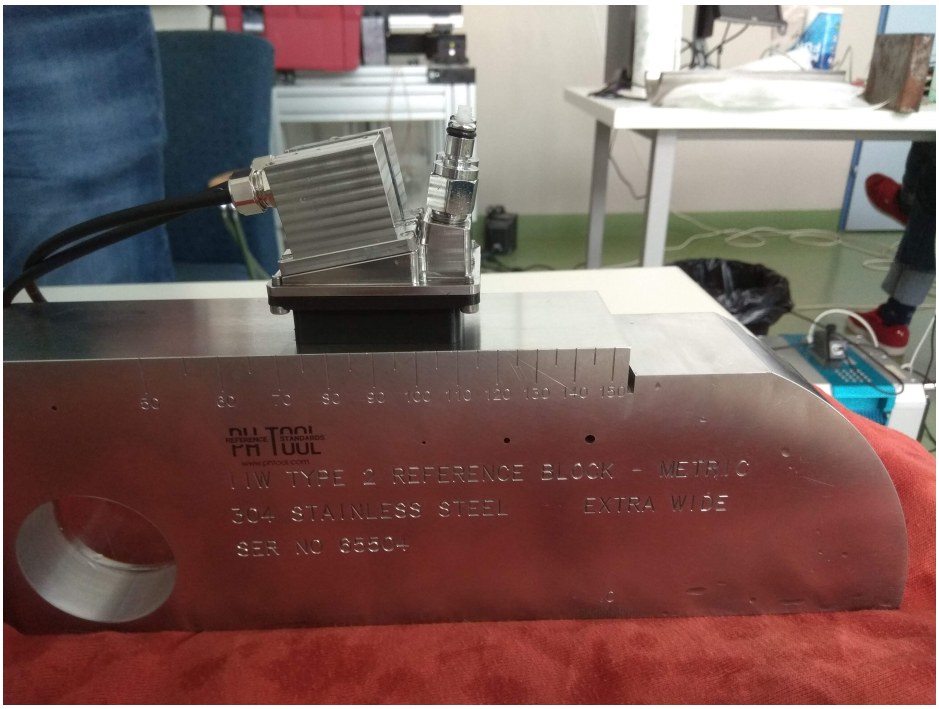

Figure 2.5: Example of UT scanning of a metal block with the INETEC's UT probe

3. Visualization techniques

Visualization is the process of representing data graphically and making the representations available for interaction in order to gain better insight into the data. Displaying data in visual form provides clear and easily understandable representation of data. Computer graphics is constantly updating mechanisms for creating, manipulating, and interacting with these representations.

The main problem in visualizing volume data is displaying three-dimensional data as a two-dimensional image while not losing too much and not losing valuable information. To effectively visualize volumes, it is important to be able to present them from different viewpoints and to shade them in a manner which brings out surfaces and subtle variations in density or opacity.[5]

Slicing, volume rendering and surface rendering represent the most important techniques applied for three-dimensional visualization. Slice is a 2D plane extracted from the 3D data volume and displayed as a 2D image, it shows detailed information about the selected plane being displayed. The slice may be either parallel to one of the axes of the volume or at an arbitrary orientation. Unfortunately, it is difficult to see the three-dimensional structure of the interior of volume by viewing individual slices. That is why the focus will be on other two techniques as they work better for needed requirements.

3.1. Surface rendering

Surface rendering method involves, in simplified terms, constructing polygonal surfaces in the dataset and rendering these surfaces. Surface-based methods work best with solid, nontransparent objects.

Surface rendering consists of constructing polygonal surfaces in the dataset and rendering these surfaces, while asserting an assumption that 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 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. The pixels are a part of one isosurface, which is the set of locations in the data where the scalar field equals same value. The geometric primitives are then fit to the data to form a 3D surface based on extracted values in the data. These primitives, polygon meshes or contours, can in the end be rendered for display using conventional geometric rendering techniques. [18]

In typical datasets from medical or scientific applications, the isosurface forms a connected surface, such as the boundary between skin and bone in a CT dataset. In the dataset present in the thesis, the isosurface will be detection ray density based on a density value selected through the application.

The process of determining surface is not perfect, as it must be determined whether the surface passes through every voxel volume element, this issue is especially apparent for datasets surrounding objects that are very small, blend into their surrounding or describe poorly defined features. 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 ones that do.[9]

If the volume is reduced to just surfaces it cannot faithfully display subtle surfaces and phenomena that occur at the transitions between materials and local variations in volumetric properties, such as light absorption or emission. That is why surface-based methods work best with solid, nontransparent objects. When surfaces are transparent or semi-transparent, geometric rendering techniques may not be the best choice, that is where the volume rendering comes in.

3.2. Volume rendering

Volume rendering is a collection of techniques used in computer graphics and scientific visualization to create a 2D projection from a discretely sampled 3D dataset. The most known use case would be medical dataset consisting of a stack of MRI or CT 2D image scans. Some other possibilities are physical simulations such as fluid dynamics or particle systems.

Volume visualization is a powerful technique for the representation, manipulation, and rendering of volume data. Unlike traditional graphics techniques, which represent 3D objects as geometric surfaces and edges approximated by polygons and lines, vol-

umetric datasets are 3D entities that can also contain data and information, not only on the surface, but also inside them, which is where surface rendering techniques fall short. Volume visualization techniques provide the mechanisms that make it possible to reveal and explore the inner or hard to see structures of volumetric data and allow visual insight into transparent and complex datasets.

A Volume is a regular 3D grid of voxels. A voxel or volume element is the 3D equivalent of the 2D pixel that represents fundamental volume element - cubic cell. Each voxel is characterized by its position in the 3D space, and can have a color, opacity, density or some other data value associated with it. Volume scalar field is a collection of values associated with each point in volume.

Volume rendering algorithms are also called direct rendering methods because they render every voxel in the volume raster directly, without explicit conversion to geometric primitives. They usually include an illumination model which supports semi-transparent voxels, this allows rendering where every voxel in the volume contributes to the final rendered 2D image. [1] This method is applicable to medical, seismic, atmospheric, and other scientific data e.g.from computational fluid dynamics simulations or measured tomographic technology sources. It is also applicable in cases when geometric surfaces for dataset are unavailable or too cost-ineffective to generate.

Volume rendering algorithms can be grouped into four categories: ray casting, resampling or shear-warp, texture slicing and splatting. [14]

3.3. Surface rendering vs. Volume rendering

Surface rendering methods produce hard surfaces at distinct field values. Volume visualization methods achieve soft surfaces by blending the contributions from multiple surfaces, integrating the contribution from the entire volume. Another advantage when using volume rendering is no need to determine surfaces in advance which removes need for a segmentation and polygon model representation.

Many visual effects are volumetric in nature. Fluids, clouds, fire, smoke, fog, and dust are difficult to model with geometric primitives. Volumetric models are better suited for creating such effects.

The main disadvantage of the volume rendering approach was a long computation time and a high-hardware requirement but due to a modern graphic card development achievements it is now possible to run the visualization in real time even with less powerful options.

4. Volumetric Ray Marching

Volumetric ray marching, or also known as volume ray casting, is the most commonly used technique for achieving image-based volume rendering. This method's main principle of operation is defining rays that represent light from a camera view.

The ray casting part of the name comes from similarity with traditional ray casting in that they only consider primary rays, the generated rays coming from the camera viewpoint, without spawning secondary one. But be sure not to confuse them as ray casting interacts only with surface data and stops as soon as it encounters an intersection between ray and surface in the scene while volume ray casting does not stop the computation at the surface level but passed through the object sampling it along the way. Ray casting assumes that the light only interacts with surfaces of objects. For the most scenes, this is a perfectly adequate assumption. To determine the light coming from a certain direction, one ray will simply be cast in a scene to find the first intersection and compute the luminance (power emitted by the light source) at the intersected surface.

Ray marching handles the other cases, where light interacts with objects not only at the surface of the object. This includes all 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 to the camera. The usefulness of this technique lies in the fact that no provided mesh data about objects in scene is needed, as it does not calculate intersections in the scene.

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 shadow are considered.

The Emission-Absorption Optical Model places two assumptions:

- Emission - the volume is assumed to consist of particles that emit light.
- Absorption - The volume is assumed to consist of black particles that absorb light. Meaning 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.

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

Where parts of the equations are:

- $I(s_0)$ is initial light intensity at point s_0, s_0 being a starting point
- $\mathrm{e}^{-\tau(s_0,s)}$ 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
- $\mathrm{e}^{-\tau(s_1,s)}$ is intensity reduction from s_1 to s
- τ is optical depth measure, measuring degree of absorption of light
- integral $\int_{s_0}^{s} q(s_1)\mathrm{e}^{-\tau(s_1,s)}ds_1$ 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. [25]

Each point in the volume is considered to emit and absorb light, according to the color and opacity specified by the transfer function. There are multiple compositing schemes for determining 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 set of equations also called *Under operator*:

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

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

These equations are know as Front-to-Back Compositing Equations, where C_{out} is the resulting accumulated color over n voxels seen from the front of volume. Variable α_{out} is the result opacity. 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. number of steps is exceeded.

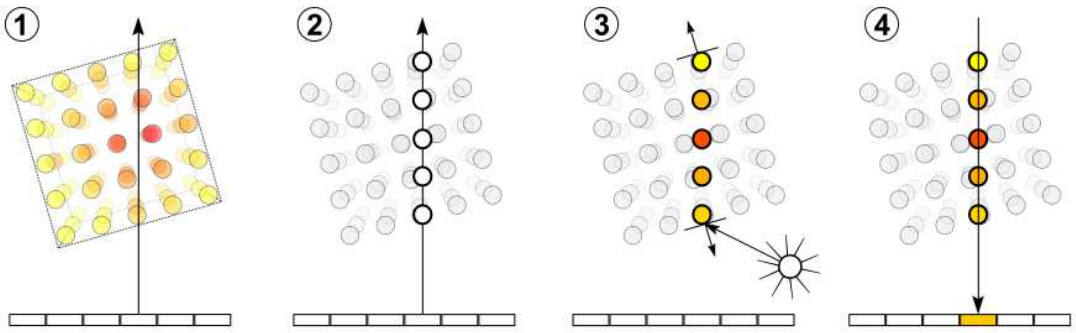

Figure 4.1: Ray marching through volume using compositing [24]

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. All the compositing schemes can be seen on figure below 4.2.

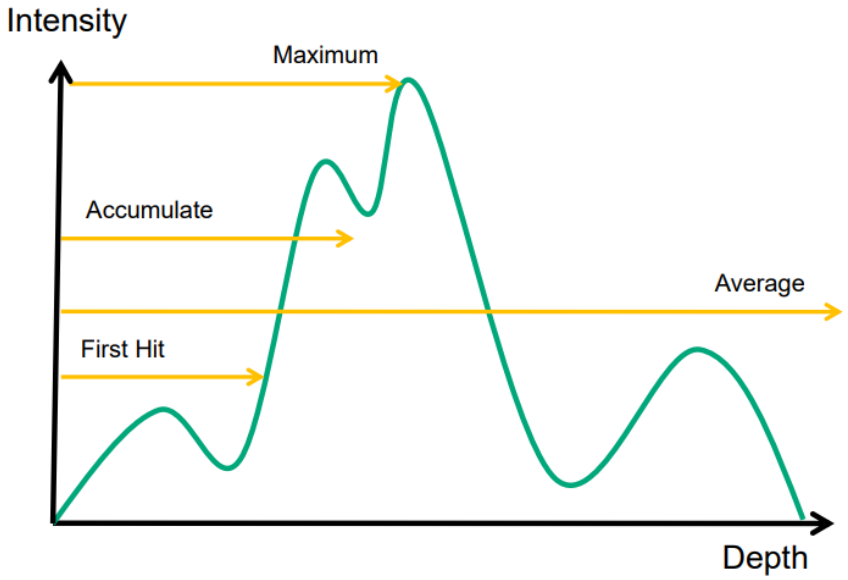

Figure 4.2: Volume rendering compositing schemes[25]

5. Marching cubes algorithm

Marching cubes algorithm is a computer graphics algorithm for creating polygonal representation of surfaces with constant density (isosurfaces) first presented in a paper *Marching cubes: A high resolution 3D surface construction algorithm* [13]. The algorithm processes the 3D data in scan-line order and calculates triangle vertices using linear interpolation. The desired density surface is specified as an input by user. Marching cubes algorithm belongs under surface rendering algorithms.

Its advantages are that it works on lookup tables so it's relatively high speed, also it maintains inter-slice connectivity - connections between slices so less detail that is present in the original data is lost. The most common usage is reconstruction of a surface from medical volumetric datasets.

The fundamental idea will be presented on a two-dimensional example first, since it is more clear and easier to understand, before moving on to the three-dimensional space.

5.1. Marching squares

Marching squares algorithm works on a divide-and-conquer principle, it divides problem of creating surface to small cells, it focuses on each cell and applies to it one iteration of the algorithm. First step of the algorithm would be splitting the space into a uniform grid of cells, in 2D that would be squares and in 3D - cubes. Secondly, the isovalue is defined, the value usually being the density of a desired isosurface, in this case the beam density, which serves as threshold value used to determine a pixel's relation to the isoline boundary. Next step is assigning a boolean value to each pixel according to its value in relation to the designated isovalue – value one if it is greater than the isovalue and value zero if its less than the isovalue. The former meaning the pixel is inside or lies on the surface and latter that its outside of the surface.

For each cell four pixels that make up cell corners are taken into evaluation. The sought case is found in the lookup table by forming 4-bit binary code based on the

associated Boolean vertex value. The code is composed by going through vertices in a clockwise direction, starting from the top-left corner and setting bit accordingly from left to right in binary code – so the top-left corner will be the bit position with highest value and bottom-left the one with the lowest.

Edges are line segments that connect two vertices. In this algorithm edge endpoints lie between vertices with opposing states. Basically, each edge that will be drawn represents boundary line that separates inner and outer pixels. For 2D case there are 16 possible boundary configurations 5.1.

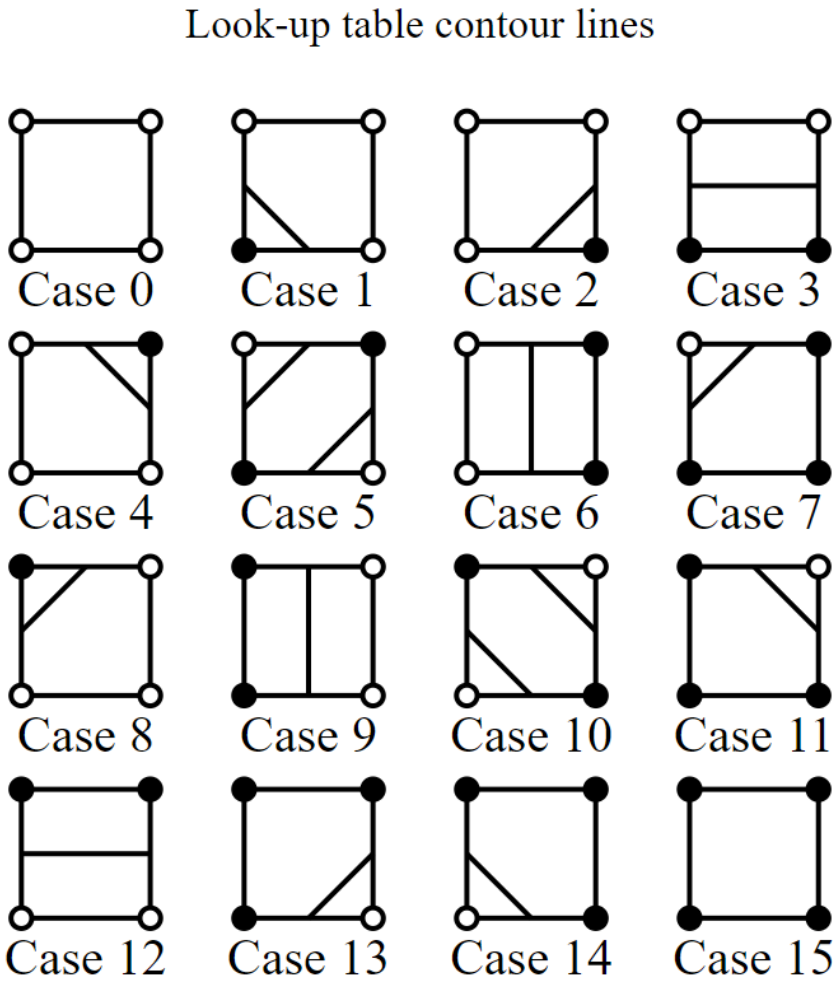

Figure 5.1: Possible marching squares combinations [16]

Once the calculation of intersected cell edges is complete, the exact points of intersection are calculated by linear interpolation. In mathematics, linear interpolation is a method of using linear polynomials to create new data points approximating the value of a given function at a given set of discrete points. If the two known points are given by the coordinates the linear interpolation result is a straight line between these points.

We can find edge endpoints by calculating the midpoint between two opposing vertices and assigning those points, midpoint being calculated with simple midpoint formula, where input points are two vertices of the edge that's being cut.

$$(x_m, y_m) = (\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2})$$

The formula calculation is very simple but not very accurate, it produces blocky results with lines angled at 45 degrees, consequently making all edges the same length. This can be improved by using linear interpolation to place the endpoints of the edges at a more fitting location based on the isovalue with respect to the original values of the cells.

First thing is the linear interpolation formula:

$$P_m(t) = (1 - t)P_1 + tP_2$$

Where P_m is one endpoint of N edge to be constructed and P_1 and P_2 are vertices of the cut edge, t is an interpolation parameter calculated by the following formula:

$$t = \frac{isovalue - V_1}{V_2 - V_1}$$

The interpolation parameter is calculated based on the isovalue and the voxel values and on both sides of the cut edge according to the equation.[15] V_1 and V_2 represent voxel values on both sides of the cut edge, while isovalue is the value chosen in earlier steps. Using this formula, the intersection will move in the direction of the voxel whose value is closer to the isovalue of the surface. Now the contours are smoother and more flexible and can represent even curved surfaces precisely. Once the algorithm repeats same procedure over all the cells, the boundaries join up to create a complete mesh even though each cell was considered independently.

5.2. Marching cubes

Now that we grasped the idea of the algorithm we can move on to higher dimension – using 3D space and cubes as units.

The good news is that this algorithm works almost identically in the three-dimensional case as it did in two dimensions. The space is divided into a grid of cubes, each cube considered individually in algorithm, adequate faces in each cube are drawn, and they will join up to form the desired boundary mesh.

Going cube by cube on a grid of cells algorithm locates the surface for each one by doing following: cube is defined with eight pixels that are its vertices, four belong to one slice (image) in input image sequence and four to another, adjacent slice, determines how the surface intersects current cube and then moves (or marches, hence the name) on to the next one. Algorithm finds surface-cube intersection by checking each vertex on the cube and determining if the data value at that vertex exceeds the threshold\value of surface user has assigned. The kind of vertices where data excess the threshold or is equal to it are located below (inside) or on the surface and are assigned label *1*. While vertices with values below the surface receive label *0* and are considered above (outside) the surface.

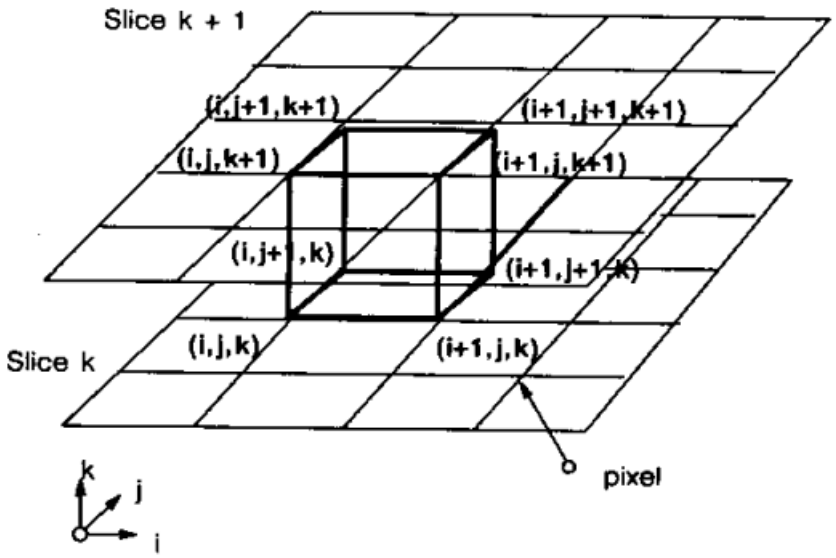

Figure 5.2: Space division to a grid of cubes in a Marching cubes algorithm[13]

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. This is where a look up table comes in, the 256 possible cases of surface-edge intersections are enumerated and put in the table. The states given to each voxel in previous step are now interpreted as 8 bit index, where each bit corresponds to a vertex, 0 or 1, depending if the voxel is outside or inside relative to surface, and composed together into an eight-bit index. That index is used to fetch data from the lookup table; each index corresponds to the one row in the table.

The 256 possible configurations can be grouped into fifteen equivalent classes that

generate same triangles. Complementary cases where vertices greater than the surface value are switched with those less than the surface value and vice versa produce same triangulation and can be grouped together thus reducing number of cases to 128. Taking into account rotation and symmetry properties, problem can be reduced to 15 patterns. The representation of these case groups can be seen of figure:5.3. Two of these cases are trivial: when all points are inside or when all points are outside the selected value, in that case no triangles are generated and cube does not contribute to the final isosurface. For all the other configurations the crossing point of isosurface needs to be determined along each cube edge, these edge intersection points are used to create one or more triangular shapes for the isosurface. Maximum number of triangles that can be added by one configuration is four.[13]

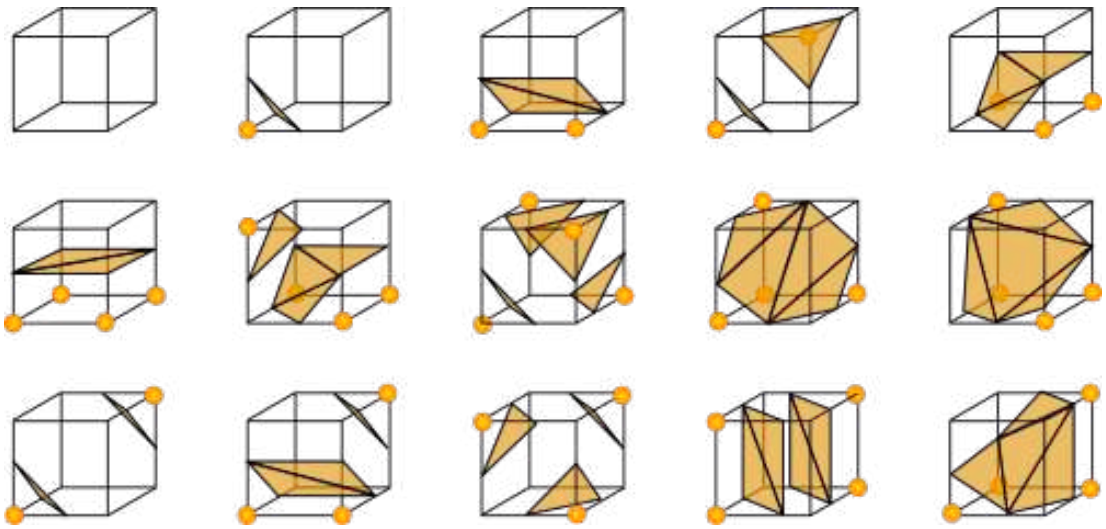

Figure 5.3: Possible marching cubes combinations[11]

Looking up by an index in the edge table returns a 12 bit number, each bit corresponding to an edge, zero meaning the edge isn't cut by the isosurface while one means it is cut by the isosurface. If table returns number zero, it means that no edges are cut, this can occur in the two trivial cases, where all the vertices have the same binary value. These cases correspond to cube indexes 0 and 256 (0b11111111 in binary notation).

The final step in the MC algorithm is usually calculating a unit normal vector for each triangle vertex. This information is later used for rendering algorithms. Since in implementation this algorithm will be used for the surface mesh construction and not for visualization this step will be skipped as it is not needed this time.

In general, to improve marching cubes results some kind of smoothing algorithm

can be applied, for example Gaussian smoothing. Considering that the result is only used as a collider this technique will not be used as less complex shape is sufficiently accurate and better for collision computing time (less complex shape equals faster computation). There are many applications for this type of technique, typical ones involving reconstruction of a surface or creating a 3D contour of a mathematical scalar field, example would be medical volumetric datasets as scans result in a volumetric data.

Algorithm iterates over all of cubes, adding triangles to a list, and the final mesh is the union of all of these triangles. The smaller the cubes, the smaller the mesh triangles will be, making approximation more closely match the target function. Each triangulation contains zero to four triangles and can be stored as a list of triangles where each triangle is a list of three numbers which are indexes of the vertices belonging to that triangle.

6. Unity

Unity is a game engine with integrated development environment developed by the game software development company Unity Technologies. It is counted among the most popular and most used game engines thanks to its ease of use and possibility to create 3D and 2D games and applications for multiple platforms - from mobile platforms to gaming consoles. Unity offers numerous options that are updated and added regularly. This development tool has a Personal Edition that is free for personal use, making it an appealing choice for aspiring game developers, and is available for download on Windows, MacOs and Linux operating systems. A numerous number of tools offer simplified and adjustable creation of games and game logic, as well as the other kinds of projects. A lot of user community created content can be found on the Unity Asset Store, with some of them being offered free for use. These user created tools and assets can be used to introduce some new functionality without needing to program it on your own or to expand what is already in the game. The version of Unity that was used in the making of this application is Unity 2020.3.14f1.

6.1. Interface

When opening the Unity Editor we are greeted by the Unity Editor interface. The aim of this short description is to help introduce names and functions of editor windows; knowing them is going to make things easier to understand and to follow along later on. This description assumes that the default layout arrangement is used.

The Toolbar is located at the top of the Editor, on the right side it contains buttons for manipulating objects such as Move, Scale and Pivot setup. In the middle - play, pause and step button control Play mode, pressing *Play* switches editor to the Game view and shows simulation of how the finished application will look like. Pause button is useful for pausing the application and then checking the state of objects or making a few changes in Scene mode and seeing how these changes pan out in the runtime, these changes are temporary and will reset once the Play mode is quit.

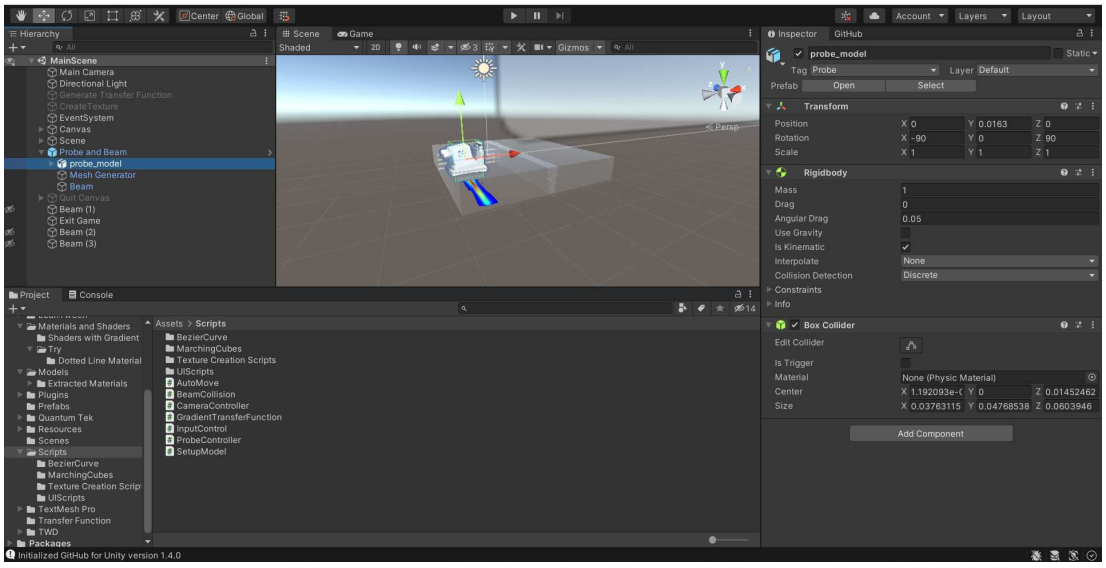

Figure 6.1: User interface in Unity editor window

The Hierarchy window, on the left, displays all GameObjects in the Scene and their mutual relationships. The GameObjects can be created by the right-click command in the window and selecting wanted object from the appearing menu. Parent-child hierarchy can be manipulated by dragging GameObject name representations. Child objects inherit movement and rotation of the parent objects. Additional options to help with organizing GameObjects at user's disposal include toggling object visibility, deleting, duplicating and sorting objects.

There are two views - Scene and, already mentioned - Game view. Scene view permits editing and interaction of the scene currently in a creation process. All the GameObject instances in the scene can be selected, manipulated and modified through this view, which is why it is the most utilized view in the course of the development.

Manipulation and easier search of Assets and other files in the project is possible through the Project window, usually found in the bottom middle part of the screen. On the right side it has folder hierarchy and on the left it lists all the Assets in the selected folder. On the window's top pane, on the right, there is a search bar and view settings and on the opposite side a button marked with a plus sign for creating new project Assets.

The Inspector window is used for overview and editing of properties and settings for almost everything in the Editor, including GameObjects, Unity components, Assets, Materials, and in-Editor settings and preferences. For accessing inspector window for object of type GameObject, it is enough to select one of the objects in the Scene

and the window opens on the right. In the window, components attached to the object can be altered as well as their properties, the ones that reference other objects can be assigned either by dragging and dropping object instances or Assets onto reference property fields or by using an Object Picker window. The inspector window for Assets (for example textures and materials) slightly varies as it displays import settings for the selected Asset instead.[20]

6.2. Shaders in Unity

A shader is a program that takes meshes, textures etc. as an input, calculates the color of the pixels on the screen and generates a rendered image as the output. Shader runs on a graphics processing unit (GPU) and it is an crucial part of the rendering pipeline, a concept that will be touched upon in more detail later; for now it is enough to know that it is a model that describes steps needed to take to go from a 3D scene to a 2D screen render.

Components needed for rendering an object in the scene in Unity are a 3D mesh, an associated material and a shader connected to that material. The 3D mesh describes model's geometry, it contains essential data about vertex positions in the model and in addition to that also provides other data like vertex normals and information for UV texture mapping. A material interacts with a shader by passing it pivotal data, namely textures and property values.

In Unity every shader — that belongs in the graphic pipeline category — is a Shader object, an instance of the Shader class, which acts as a wrapper for encompassing shader programs and specification info necessary for Unity to know how to use the shader in question. Shader objects can be created by writing code in a shader language or by using Unity's declarative programming visual tool — Shader Graph.

Shaders in Unity are made up of a combination of following languages:

- High-level shading language or **HLSL** for short, a programming language used for writing shader programs themselves. It is very alike to the Cg shader language. HLSL has two syntaxes: a legacy DirectX 9-style syntax, and a more modern DirectX 10+ style syntax; the difference mainly being in a way that functions for texture sampling work.[21]
- A Unity specific language called **ShaderLab** which is used to define aforementioned Shader objects. ShaderLab's most vital role is defining the structure of a shader and exposing its properties.

Two crucial parts that compose a shader structure are: `Properties` and `SubShader` sections.

In the `Properties` part of the shader, as per name, properties of a material are defined using ShaderLab language, but these property values are set outside of the shader; when a shader is linked to the material, properties are exposed in the inspector and users can view and edit them through that window. It should be noted that when using these variables within the shader code they also need to be defined in the body of the shader as well.

`SubShader` is comprised of compatibility information, shader Tags and most importantly, one or more Passes. A Pass block can be described as one run of a shader program resulting in a rendered image. Shader can have multiple passes, producing multiple images in that case. The shader Tags are key-value pairs found in the SubShader and describe important information to Unity to know when and how to use the SubShader in question.

`Queue` tag defines which render queue to use when rendering objects. It helps sort each material in a preferred order for rendering, otherwise the objects farthest from camera would be drawn first and closer ones later, which could cause problems in cases where there are transparent objects in the scene. The value can be defined with predefined queue name or with an integer offset from one of the queues, keeping the value in 0-5000 range. The materials with lower values will be drawn first while the ones with higher values will be drawn last, that is why for example default background value is one thousand as it is usually behind all the other objects.

The command `Blend` tells GPU how to combine an output of the fragment shader with the render target — color buffers that these fragments point to. The functionality of this command differs based on the blending operation, which can be set using the *BlendOp* command. `Blend` is a function that expects two input values called *factors*, based on the Blend operation it calculates the final color that will be shown on the screen. Blending occurs after the fragment shader step in a so-called Per-Sample Operations before writing generated color in the framebuffer. The blending equation is as follows:

$$\text{finalValue} = \text{sourceFactor} * \text{sourceValue operation } \text{destinationFactor} * \text{destinationValue}$$

In this equation:

- $finalValue$ is the value that the GPU writes to the destination buffer
- $sourceFactor$ is defined in the Blend command

- $sourceValue$ is the value output by the fragment shader
- $operation$ is the blending operation
- $destinationFactor$ is defined in the Blend command
- $destinationValue$ is the value already in the destination buffer[19]

There is one more important command — `Cull`. It decides which polygons are going to be drawn and which are not based on the direction that they are facing relative to the camera. Culling reduces GPU time to render things, as it is not rendering what would not be seen in the final image anyway. The possible values are Cull Back, Cull Front and Cull Off. By default, the GPU performs back-face culling; this means that it does not draw polygons that face away from the viewer.

6.3. Graphics pipeline

There is a lot to be said about graphics pipeline (or rendering pipeline), but this chapter will give just a brief overview for easier understanding of the implementation part concerning shaders. Graphic pipeline is a sequence of stages that leads from 3D scene described by meshes and vertices to a 2D image that is shown on the screen. It is named pipeline as each stage is operating in parallel in a fixed order and the output of one stage is an input for the following one. [6]

The programs that are written as part of the pipeline are called shaders. Whole GPU pipeline diagram can be checked on the figure 6.2, as it can be seen there are many stages, some optional, but this chapter will focus on programmable parts - vertex and fragment shader.

Vertex shader processes and performs transformations on individual vertices. Most important task for the vertex shader is transforming vertices' coordinates to clip space so that they can be used by rasterizer and be projected on the screen. The conversion from object space coordinates, which are inputs vertex shader receives, to the clip space is done by putting vertices through a sequence of transformation matrices. Object space or local space is a coordinate system tied to an object, as the coordinate system moves or rotates so does its object; generally, origin of an object's coordinate system is the object's pivot point.

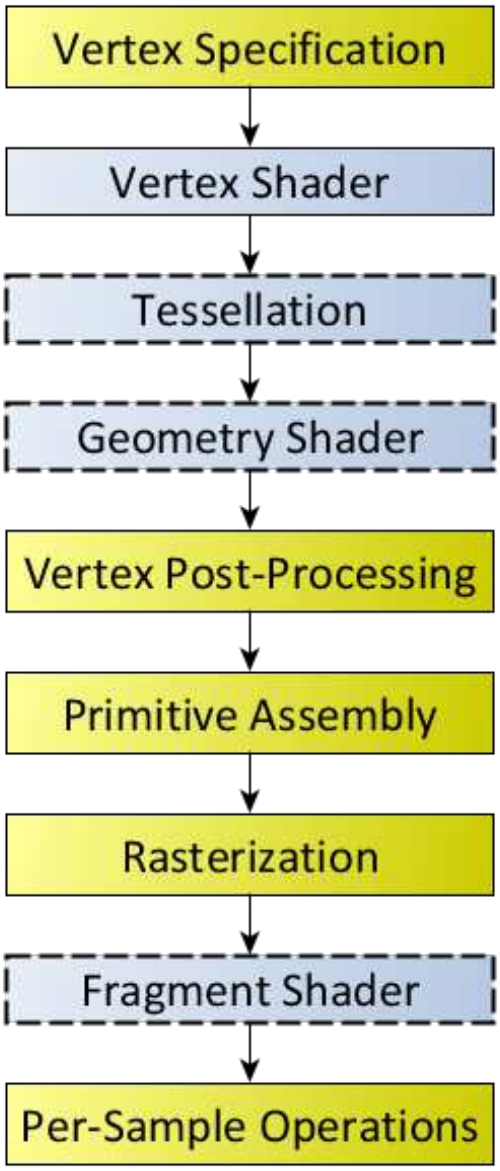

Figure 6.2: Graphic pipeline steps [8]

World space is reference frame for all the objects in the scene, it gives spatial relationship to all the objects or, more precisely, to their coordinate systems. World coordinate system origin is the center of the scene. The transformation matrix which transforms object space coordinates to world space coordinates is called *model matrix*. The transformation is done by multiplying the matrix with coordinates. The aim of these transformations is to overlap the origins and axes by using translation, rotation and scaling.

View space or camera space is the space as seen from the camera's point of view with camera as its origin. In order to transform coordinates from world space to view

space the *view matrix* is used. To determine what positions are actually going to end up in the image that will eventually get rendered the eye space is converted to clip space by using projection transform. Clipping determines which objects or object parts will be rendered. For instance, an object which is out of the desired coordinates will be discarded i.e. clipped. The clip space defines the allowed range inside the volume in which vertices should be if they are going to be rendered.

Projection matrix is a bit more complicated than the other two, but it is enough to know that it transforms coordinates from eye space into clip space which can then be rasterized and that perspective projection can be imagined as a truncated pyramid defined by perspective camera and two clip planes. That pyramid is, more technically, also called camera frustum and it represents part of space that camera sees and will eventually be rendered on user's screen. If it is a bit hard to visualize frustum at first, this is an image for a reference 6.3.

Instead of multiplying three mentioned matrices step by step, they can be combined into one matrix by the name **Model View Projection matrix** (MVP). Using the properties of matrices, several operations (translations, rotations, scaling, and projection) can be combined into a single matrix by multiplying them together, this composed matrix directly transforms vertices from object space to clip space. Just directly multiplying coordinates with MVP matrix is the same as if all three matrices were applied separately.

Luckily doing above steps and defining matrices one by one is not necessary as Unity has a built-in helper function `UnityObjectToClipPos(float3 pos)` that takes a point position and directly transforms it from an object space to the camera's clip space in homogeneous coordinates.

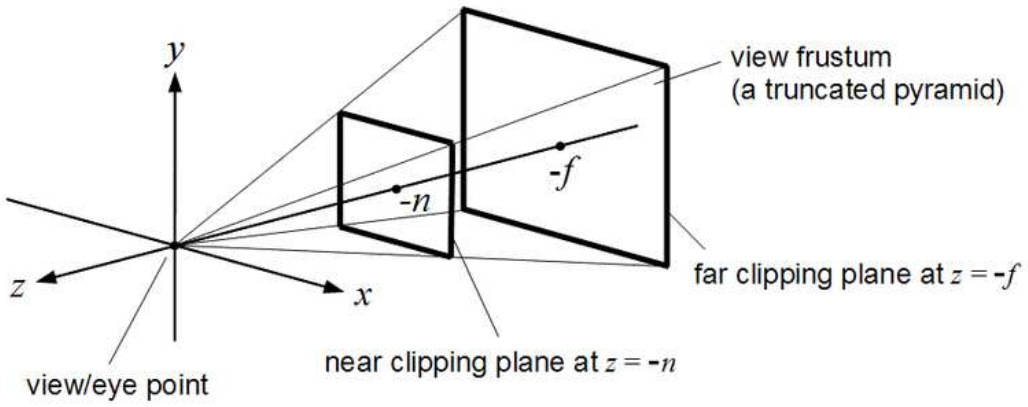

Figure 6.3: Perspective camera frustum [12]

In Vertex post-processing vertices go through variety of operations, most notable one being clipping. Rasterization converts primitives to fragments, by determining pixels covered by a primitive (e.g., a triangle) and interpolating the output parameters of the vertex shader for each covered pixel. The result of rasterizing a primitive is a sequence of fragments which is also an input for fragment shader.

Fragments are collections of data generated for each pixel covered by a primitive; each fragment contains attributes that are interpolated per-vertex values obtained from the rasterization process. It could happen that there is a pixel with multiple fragments mapping to it. Shader output is most often (unless only depth values are important and colors are left undefined) an array of colors for each color buffer. Fragment shader can be programmed to do a lot of things, from doing bump mapping, adding shadows, highlights and translucency to being used for simulating postprocessing effects.[8]

7. Implementation

The implementation of the problem presented in this thesis is made by using Unity 3D development platform. As defined, the goal of the problem is to move the probe as little as possible across the surface of the specimen so that the scanning for defects takes less time but to still collect all the needed information. The specimen is presumed to be flat and has a constant thickness throughout entire volume. It has welding defects, more specifically side drilled holes. The trajectory of the probe should be able to be either predefined or defined by the user inside the application.

Firstly, the project was prepared for development by importing needed data - 3D models and simulation images provided by INETEC. Secondly, the most important part of implementation was done - implementing the beam visualization using shaders and textures. The third part was setting up collision detection and mesh generation. As the last step the options for user controlling the movement of the UT probe were achieved with scripts and User Interface. On the UI there are also added buttons and input fields for control of the simulation. The programming language that Unity uses for scripts is C# so majority of the code is written using C# and shader language HLSL. Integrated development environment (IDE) used for development is JetBrains Rider as it has support not only for C# but also for Unity and HLSL shader development.

7.1. Model Import

The 3D models used in the creation of the application were provided by the courtesy of INETEC. The model set consists of a 3D representation of a UT detector probe and a 3D model of a test steel block. The block consists of multiple elements, some of them being parts of the block structure while some of them representing welding defects inside the block.

The models were made outside of Unity, with an external 3D modeling software from the Autodesk corporation; consequently, they are in the Autodesk's native format for the asset exchange - FBX.

They can be imported by just dragging model files into the Assets folder structure, this opens the Import Settings window in the Inspector offering different setting adjustments, the settings are going to be left unchanged in this case.

Next step is instantiating objects by adding each one to the scene, new necessary components can also be added in the inspector tab by clicking *Add Component* in the Inspector window on the right.

First essential component that needs to be added to the probe model and all of the defects is a `Rigidbody` with the kinematic setting set to True. The defects also need an another component - `Mesh Collider` with a Convex setting checked, this is going to be required subsequently for the collision detection. To remove the time-consuming and repetitive job of manually assigning components to all of the defects, which could be numerous, there is a script for making the job faster, it can be attached to the block model and automatically adds the above-mentioned components to all of its `GameObject` children tagged with a *Defect* tag.

7.2. Beam Visualization

Visualizing the beam itself was the most important part of the project. It was made using 3D textures, shaders and the ray marching technique. Each step is detiledly described in the following sections.

7.2.1. 3D texture and volumetric data storage

The beam volumetric data was stored in a Unity's 3D texture Asset Type. A 3D texture is a bitmap image that contains information in three dimensions: width, height, and depth, rather than the standard two dimensions. Texture coordinates are placed on a unit cube and texture data is accessed by three-dimensional texture coordinates. This kind of data type is fitting for storing and manipulating volume data.

The simplest way to imagine a 3D texture is like a series of 2D textures stacked on top of each other. They are a bit newer and rarer than 2D textures, but 3D textures are commonly used to simulate volumetric effects such as fog or smoke, scientific visualization data, or to store animated textures and blend between them smoothly. One downside is that the size of a 3D texture in memory and on disk increases rather quickly with the increase of resolution because of three dimensions.

The 3D texture containing volumetric data was built from 2D images through a script and will be one of the inputs for the shader. The images for creating 3D texture

are stored in folders, named based on the beam parameters; each folder has two sets of images, one with color for coloring the beam in later steps of the visualization, and separate B&W one for calculating density. The reason for needing two textures is the use of rainbow color scheme, in this color scheme colors are not ordered by brightness so we can't just calculate density by using color texture.

The script creates an instance of the `Texture3D` class and populates it with color data using built-in functions `Texture.SetPixels()` and `Texture.Apply()`, and then creates Unity Texture asset and saves it to the Resource folder of the Project with a command `AssetDatabase.CreateAsset()`.

`Texture.setPixels()` takes an array of colors and applies all of them to the 3D texture pixels, it is executed faster using this function than if every pixel is updated separately. `Texture.Apply()` actually applies all of the pixel changes that were made in previous step by sending them from CPU to GPU. This function is expensive as it uploads data to the graphics card, which is why it should be used at the end, when it is sure that there are no more pixel color changes to be made.

7.2.2. Shaders and Ray marching

Volume rendering has a few distinct color compositing types, in the project we will use Maximum intensity projection and Accumulation using Front-to-Back Compositing Equation.

To start writing the shader, first it needs to be created by right-clicking in the project window and selecting the following *Create -> Shader -> Unlit Shader*. Unlit shader is adequate as the beam does not need to be affected by lighting models. After naming the shader Unity assigns default code to it. Shaders can be opened and edited in JetBrains Rider IDE which offers support for shader development with HLSL and C#.

The Shader implements volume rendering using ray marching algorithm and gives as a result rendered volume representing the ultrasonic beam, the result looks three dimensional even though it was made from 2D images.

In the Properties section following data is defined: Maximum step size for the length of step taken when traversing the ray, two 3D textures, minimum and maximum density, minimum and maximum slice. Slice data is of Vector data type since it needs to store three values because cutting can be made along every axis, for example Vector3 value of (0.2, 0.4, 0) would mean we are looking at volume from slice at axis x = 0.2 and axis y = 0.4.

In ShaderLab `Queue` will be defined as Transparent and `RenderType` as Fade. Blend mode will be defined as standard `Blend SrcAlpha OneMinusSrcAlpha`.

In the Pass first keyword is CGPROGRAM, this defines CG shader block; it is very similar to HLSL but the difference is that it automatically includes Unity built-in shader include files among which are shader helper functions that will come in handy for writing the rest of the shader. Downside is that the built-in functions are only compatible with Unity's Built-in Render Pipeline, but that is acceptable in this case as that is the render pipeline that is used in project.

At the start of the shader there are two statements with `#pragma` directives, they are a type of preprocessor directives in HLSL which are instructions for the preprocessor when preparing code for compilation. Pragma directives indicate which shader function to compile for different shader stages. For example `#pragma vertex vert` would tell the preprocessor to compile function `vert` as the vertex shader.

Defined data structure `vertInput` will be the input data type for the vertex shader while `fragInput` will be the output data type from vertex shader and input for the fragment shader.

Semantics next to the variables indicate the information about data required by GPU for all variables passed between shader stages. Although on the modern GPUs, rules for assigning them are becoming more arbitrary, for example TEXCOORDn semantic can be used not only for textures, as could be assumed by its name, but also for anything that does not fit into other semantic labels. The exception being SV_POSITION in vertex shader output, which stores the final clip space position of a vertex, as GPU needs this data for correctly positioning pixels on the screen during rasterization.

Vertex shader runs for all of the vertices of the model. All the needed operations for manipulating the vertex position are executed in this part of the graphic pipeline. The vertInput contains vertex position in object space. To obtain the clip position required to be stored in SV_Position built in function `UnityObjectToClipPos()` is used, it takes a point in object space and transforms it to the camera clip space, its return type is float4 since result is in homogeneous coordinates. It is equivalent to multiplying MVP matrix with vertex position. Additionally the object space position will be passed to fragment shader. Since that are all vertex manipulations needed, this output will now be sent to fragment shader.

In fragment shader GPU calculates the final RGB color for every pixel. It is a stage after rasterization and runs for each pixel on the screen. In fragment part the actual ray marching with compositing will be done.

Before starting to ray march, for each pixel there has to be a ray created through it, that will be done by using pixel's local position as ray origin and ray direction vector from camera to pixel using helper function `ObjSpaceViewDir()` and inverting it since it returns 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 march step size by dividing distance between start point and end point 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 use.

Finally, the ray marching in executed in the loop that runs for a number of steps. In each step a density is read from B&W 3D texture and the color form colored 3D texture using adjusted local vertex position and saved for later processing. Now what is done with color information depends on technique chosen. If it is accumulation color is sent to the `BlendColor()` function where compositing formulas are used for collecting and interpolating colors through steps. If the pixel density does not satisfy threshold limits the pixel color is discarded from calculation. The following formulas are used in shader for compositing:

$$colorOut.rgb = colorIn.rgb + (1 - colorIn.a) * color.a * color.rgb$$

$$colorOut.a = colorIn.a + (1 - colorIn.a) * color.a$$

For optimizing calculation time, compositing along the ray can be terminated early if a currently accumulated opacity is too high, meaning contributions of future samples are insignificant.

In Maximum intensity projection (MIP) technique, as the name says, the maximum intensity voxel along the ray is saved and returned after checking all the pixels along the same ray. In both cases at the end of ray marching loop 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, based on choice the code differs a bit. The case of using textures was implemented in a way that a color for the voxel is read in ray marching algorithm, while when using transfer function first the density is calculated using ray marching and then at the end, based on that value the color is read from transfer function which is saved in a Texture2D format.

To apply shaders to objects, a material that will be linked to this shader needs to be created and then attached to the object.

7.2.3. Transfer function

The result of volume visualization algorithm is complete now, but it is in shades of gray. As the color is a vital component for a good visualization, the next logical step would be adding it. The volume can be colored by assigning color and opacity based on density, this is where the color transfer functions come into play. Transfer functions define mapping from voxel density to RGBA color based on user defined values. Using transfer functions provides a better overview and clarity to the data, they can be used to highlight important parts and omit unnecessary ones better emphasizing nuances inside the volume.

Transfer function takes the scalar value, in this case density and relates it to corresponding color and opacity/transparency, for example if it is desirable to highlight higher density parts higher opacity should be assigned to those densities. Deciding how a transfer function should look like is not an easy task, there is no fixed guide to perfect function because of variations from dataset to dataset and taking into consideration what kind features are wanted for particular visualization. Since original images are colored with rainbow heat map, the transfer function with same colors will be used for this assignment.

The script for generating transfer function can be executed in the inspector, without needing to put the scene in the play mode. Gradient is an input variable for the script stored in form of a Unity's `Gradient` class. The variable needs to have `[SerializeField]` decorator so it is exposed in the Inspector. Clicking the gradient class variable for editing opens up the *Gradient editor*, showing a gradient with set of keys at the top and at the bottom 7.1, the keys at the top control alpha value or transparency while the ones on the bottom determine color. New keys can be easily created by clicking below or above gradient and then selecting its alpha or color value with a slider. Gradient mode should be set to Blend if a smooth interpolation between colors is desired result. When the color choice is decided, script can be executed by clicking a button with *Generate Transfer Function* label.

In Unity transfer function can be created by using a 2D texture with height of one, this is enough since both needed information, color and opacity, can be stored in one pixel. The process of assigning color to pixels is similar to previous 3D texture creation except, of course, being in two dimensions. Texture color is read and set based on input gradient, important thing to note is that setting texture wrap mode to *Clamp* after the creation is needed otherwise edge colors will not give desired results. After texture creation it can be linked to 3D texture's properties.

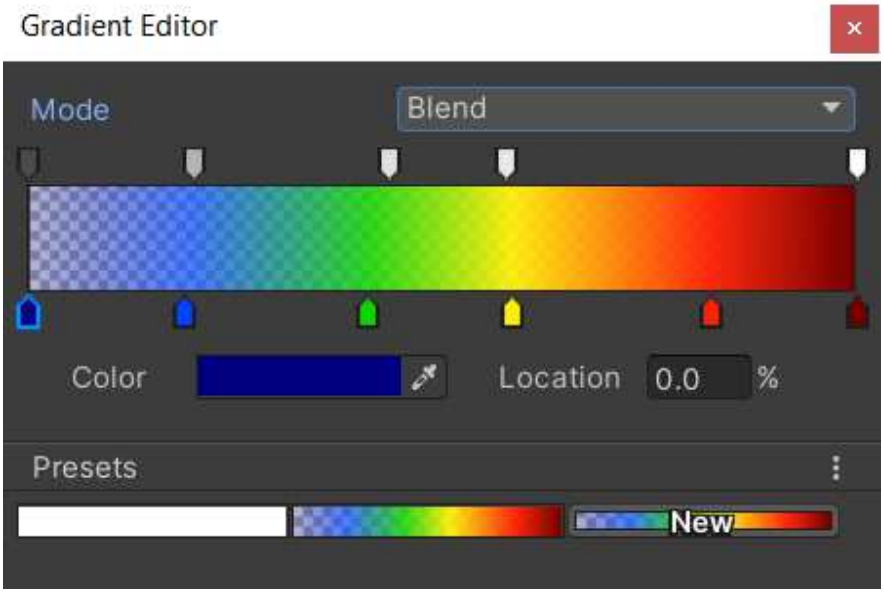

Figure 7.1: Gradient editor window

7.3. Collision

For detecting collisions Unity's default colliders were not a good enough of an approximation for the beam, so the mesh had to be reconstructed with surface rendering from the images. The reconstruction was accomplished with the marching cubes technique. The input is once again a stack of B&W images. The user defines minimum density value that is going to be considered as the threshold in the script; the script goes through each slice row by row, finding a first pixel that satisfies the equation in a way that its density is greater (or equal) to the threshold. After finding the first point the search continues until finding the last point that satisfies the equation, point after it and all the subsequent ones until the end of the row have lower density and will not be included in the mesh vertices. The pixels between the two found end points are going to be included inside the mesh, but only two edge points need to be sent to mesh creation function as other points are going to be included inside generated triangles.

Mesh creation function uses Marching cubes algorithm with interpolation. In this function the `Mesh` variable that will be populated with calculated data is defined, vertices are interpolated before finally being assigned and triangles are formed with the use of case table. (Triangle case table was taken from a reference site [3]) Having both vertices and triangles a mesh can now be created by populating instance of the `Mesh` class.

The script is set up by attaching it to the desired `GameObject` that must have

Mesh Renderer and Mesh Filter component and activates when application is run in the Play mode. After mesh creation the Mesh collider can be added to the beam, Unity automatically adapts Mesh collider to the selected Mesh - in this case the beam.

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

7.4. Input Controller

For a user to have a better overview of what is happening there is a `Camera Controller` script that controls camera movement by reacting on user's input from mouse and keyboard. The camera can move horizontally and vertically with the arrow keys, rotate around the test model with mouse and be zoomed in and out with the mouse wheel. Some sensitivity parameters can be adjusted in the Inspector like zoom in speed, camera damping and such. In `Update` function the script is listening for any input from the user in `Update()` function every single frame, here the Unity's Input System interface was used for checking input source, acquiring input values, and calling appropriate functions - `Move()`, `Zoom()` or `Rotate()` accordingly. The configuration of Unity's Input API can be set up in Input Manager, developer can edit keys, buttons and virtual axes, this menu is accessed by going to *Edit* > *Project Settings* > *Input Manager*. For example `Input.GetAxis("Mouse X")` will fetch action associated with name *Mouse X*, that action by default being mouse movement on x axis but it can be customized.

`GameObject` representing ultrasonic probe has two scripts for controlling its movement and trajectory attached. The first one, called `ProbeController` keeps the track of UT probe's position and whether it is currently controlled manually or with the simulation script. It contains references to the object's `Transform`, custom `AutoMove` script and fields on user interface meant for position setup input values.

7.5. User Interface

The User Interface is divided into three sections, which can be seen on the figure 7.2 The first section, on the left, consists of settings regarding beam visualization. Most important ones are Focal depth and Refraction angle, these settings 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 these particular settings.

For output color there are Max Intensity Projection and Compositing options, choosing either comes with different visualization results corresponding to a technique chosen. For the Compositing, the choice of current Z-slice is enabled, by using the slider user can select any cross-section on z-axis within the beam for a more detailed view. In addition to the beam options, this section also contains some general settings, like screen options and application exit button.

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. The unit in which field inputs are calculated are millimeters. Once the user has set all of the parameters, by pressing the button `Set Parameters` current values will be saved, play button will be enabled and automatic simulation will be ready to be played.

The middle section, in the bottom of the screen, controls movement of the UT probe model, the two tabs are designed for controlling the simulation parameters and manual user control, respectively. Animation tab offers buttons for playing or pausing and the slider for rewinding to an any point in the animation. Playback speed can be slowed down or accelerated with a drop-down menu with playback speed choices based on user's preferences. In 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.

All the menus on the interface can be collapsed (or expanded) by clicking on the related tab name for a better screen overview.

The components used in process of building the interface were either created from the scratch or extended from the Quantum UI package [23] which was downloaded from the Unity Asset store web page. Another custom package that was used was LeanTween for the UI animations.[17]

Figure 7.2 shows the simulation in the running mode, with completed user interface which offers the access to options and functionalities of the simulation.

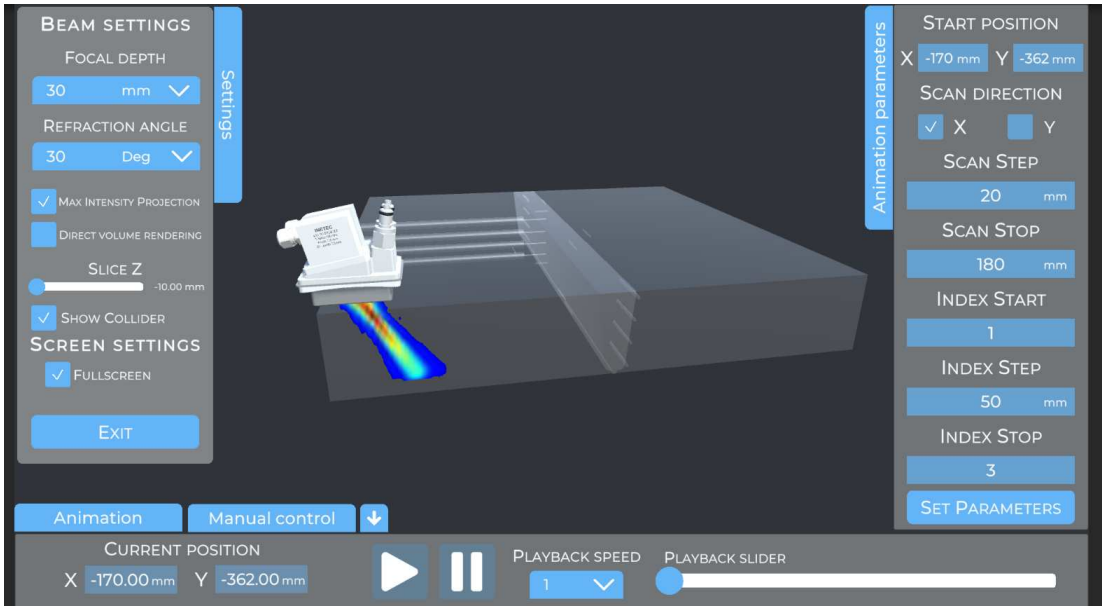

Figure 7.2: User Interface in the application

8. Results and Discussion

In this chapter, the final results of the implementation are presented and discussed. The comparison between technique types and two different color acquisition methods will be seen. Also, it should be noted that the results are shown on unit 1x1x1 cube (in Unity's measure units) for better observation while the beam in simulation is resized to match the real beam's dimensions.

As has been previously explained, the beam can be rendered with two techniques: Maximum intensity projection and Compositing. The differences between the techniques can be viewed on the figures 8.1 and 8.2 below. The images for comparison were generated using color acquisition from a colored 3D texture, texture used was the one with 30 degree refraction angle and 30 mm focal depth, and a cutoff density of 0.04 for examples on figure 8.1 and of 0.3 for figure 8.2.

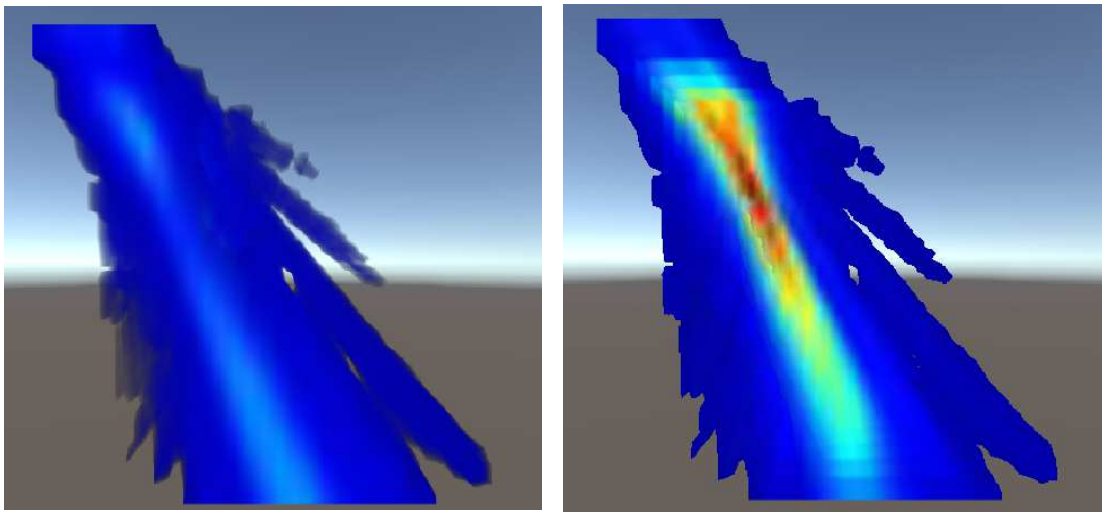

Figure 8.1: Comparison between MIP (left) and Compositing (right), using color from texture and cutoff density = 0.04

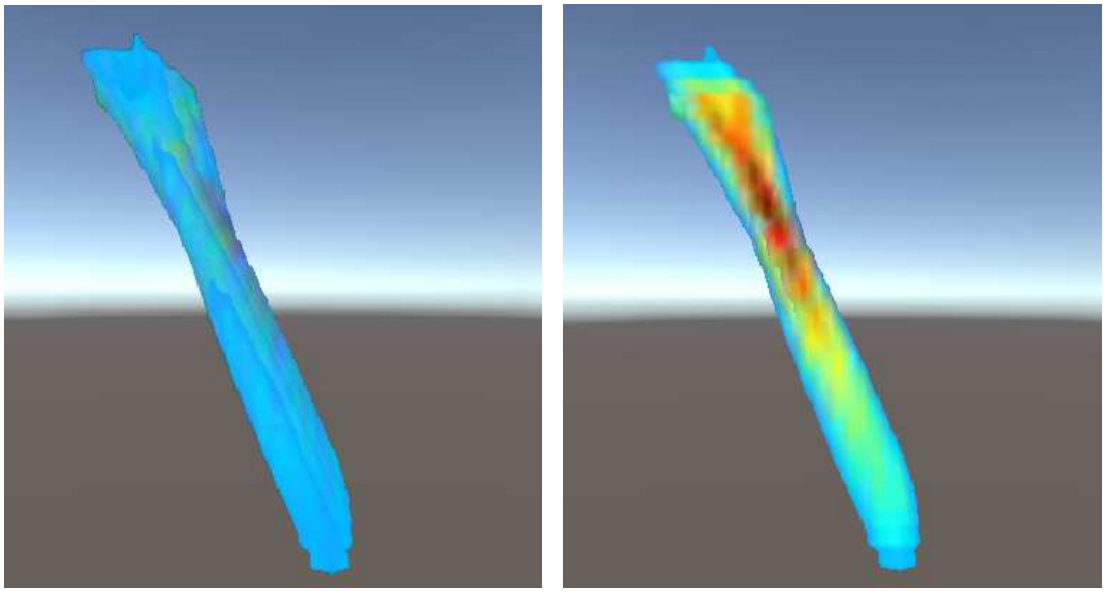

Figure 8.2: Comparison between MIP (left) and Compositing (right), using color from texture and cutoff density = 0.3

On the figure with Max projection, the highest density can be clearly seen from all of the angles (an effect more visible in the simulation, when moving and rotating the camera than on images), while with Accumulation we see color compositing effect, the most of the beam being blue as it is the color most present in the beam so it contributes the most, but also seeing some traces of red and yellow in the middle. The difference between two methods can be seen more clearly from the side view on figure 8.3.

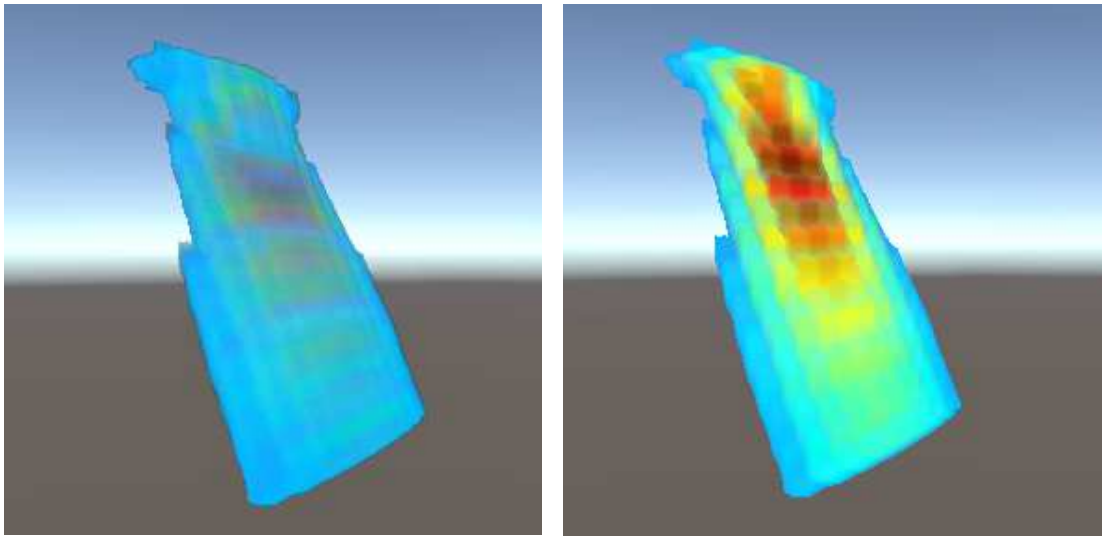

Figure 8.3: Comparison between sides MIP (left) and Compositing (right), using color from texture and cutoff density = 0.3 at a 30 degree angle

The cutoff setting difference is shown on the following sequence of images 8.4, with the cutoff parameter value increasing from left to right. As an example, the maximum projection visualization was chosen as the immutable part and cutoff density is the parameter that is changed. On figures, it can be seen how with the increase of cutoff threshold smaller number of pixels in the beam is considered for the final visualization, the ones that do not satisfy density requirement are excluded from final result. This setting helps with focusing on the high density/strong detection area of the beam.

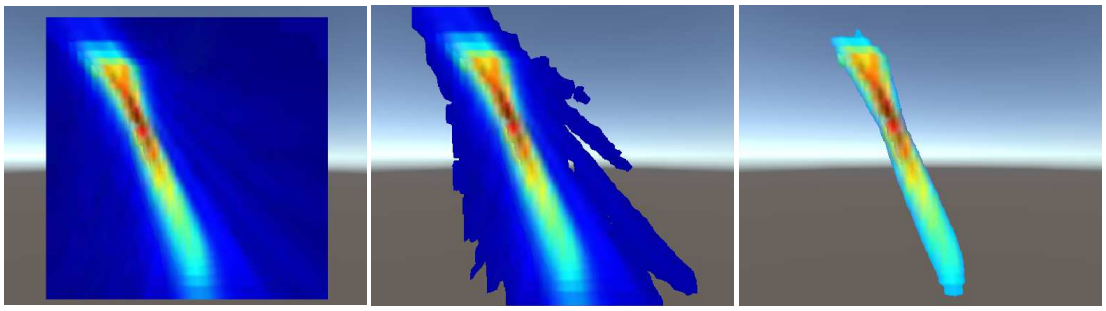

Figure 8.4: Comparison between different cutoff densities on example of MIP using color from texture, cutoff values are 0, 0.04, 0.3

Max intensity projection can be improved upon by adding transparency. With this feature enabled voxels with lesser density will be more transparent than the ones with higher values. This shifts focus to more important parts of the beam while still including all of the parts of the beam in the render unlike with cutoff setting. The result can be seen on figure 8.5 with outer blue rays being less visible than the middle ones in yellow-red range. Transparency is achieved by multiplying voxel density by an alpha multiplier parameter in the shader.

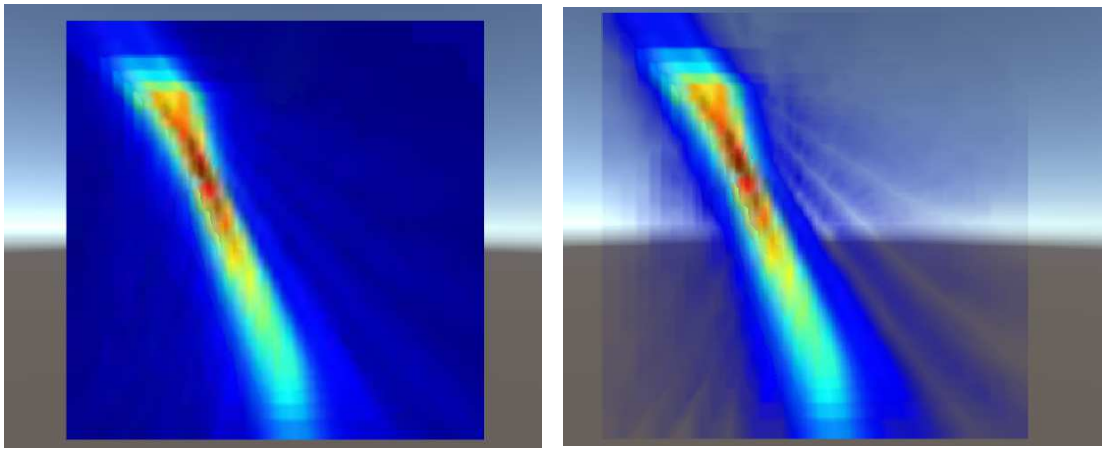

Figure 8.5: Comparison between MIP without and with transparency, alpha multiplier = 9

Previous images show that with the Composting color accumulation inner parts of volume are not clearly seen. To make inner parts more visible and to get better insight of inner structure there are two options: to remove parts outside of desired density segments or to slice off the part of the volume that is not needed.

Extracting particular 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. The result is a volume or an isosurface where its inner and outer sides are visible.

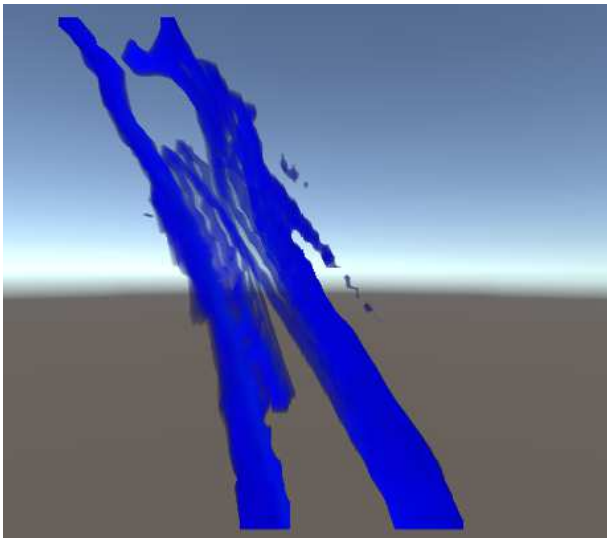

Figure 8.6: Compositing result beam with alpha value between 0.07 and 0.115

Another option is cutting the volume along one of the axes. For the example, z-axis, values are viewed along z axis and everything in front of the minimum z-value and behind the maximum z-value is removed. As the volume is drawn inside unit cube, possible input values for slices go from 0 to 1. While cutting the volume it is also possible to select particular slice by putting slice values for minimum and maximum as close as possible.

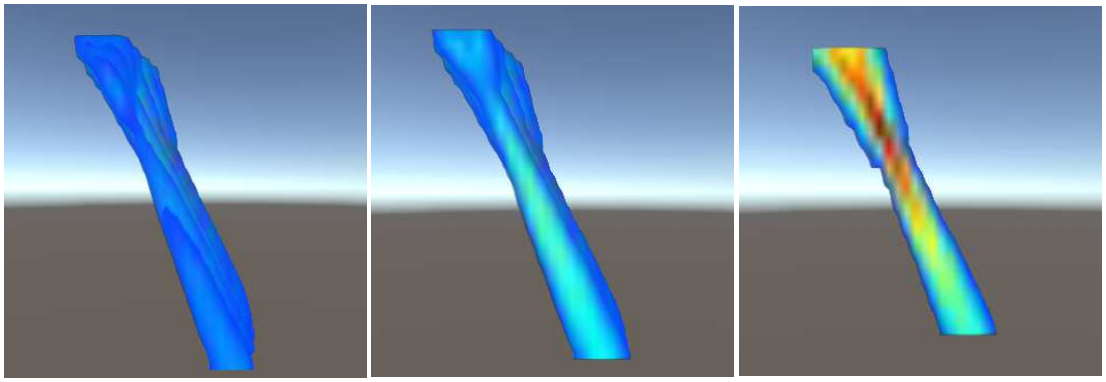

Figure 8.7: Comparison between different z slice values on example of MIP using color from texture, cutoff values are 0, 0.2, 0.5

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 transfer function is that two textures are no longer needed, saving time and space. 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, there is no need for transfer function, saving the time that would otherwise be needed to try to define perfect one for particular dataset.

On the result images 8.8 the transfer function render gives better results - colors are brighter and transitions between them are clearer. Even though the B&W texture used for density calculation was lower resolution the results are very smooth, 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 resolution the result would quite probably be significantly better and more similar to the transfer function result.

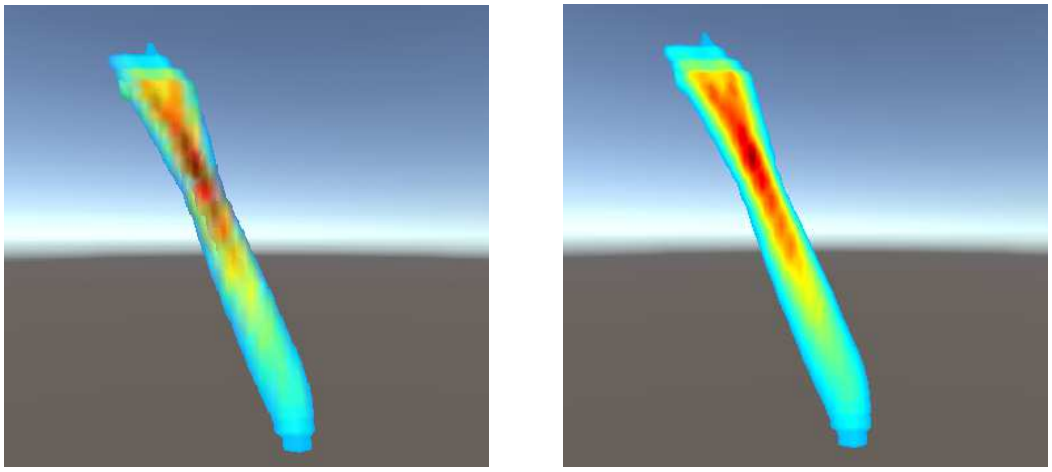

Figure 8.8: Comparison between color acquisition from a 3D texture and a transfer function

Two adjustable settings that substantially affect the beam material are focal depth and refracted angle. As this changes the parameters of the ultrasonic probe, the simulated field changes along with it which is reflected in implementation by swapping 3D texture sent to shader.

The focus depth change from 30 to 60 millimeters 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 is positioned in the top left corner and shoots rays at an angle. The possible angle options are in range of 30-70 degrees in 10-degree increments. On the figure 8.9, first two photos have the same angle and present the focal depth change difference, while second and third photo pair retains the same depth but shows an angle change effect.

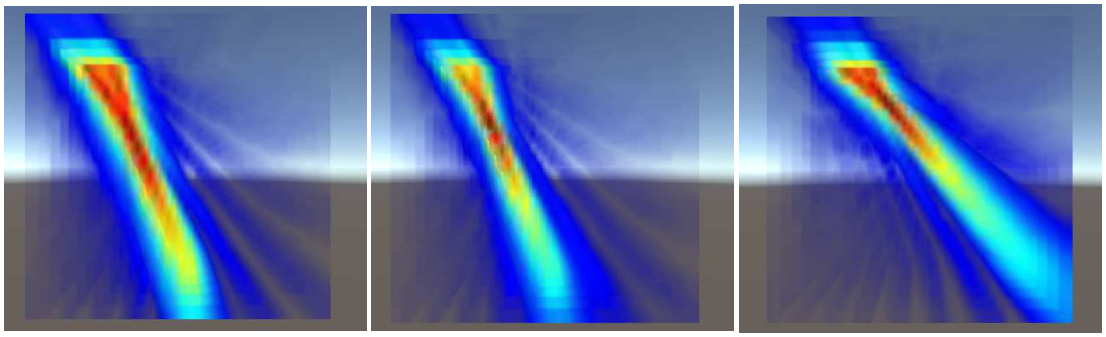

Figure 8.9: Settings from left to right: angle 30 depth deg 60 mm, angle 30 depth deg 30 mm, angle 50 depth deg 30 mm

Maximum step difference is an important parameter in the ray marching algorithm, too big of a step and the important data could be missed, too small and the computation could take a long time. Also, in some cases where step number is too small, the sampled steps in volume could be seen in a form of thin slices, effect slightly seen on the left image 8.10. For a reminder, bigger max step size equals smaller steps. The color is more blue on right photo because with smaller step more color data is gathered, while with the bigger step parts of blue colored volume were skipped and that is why the end result is less blue.

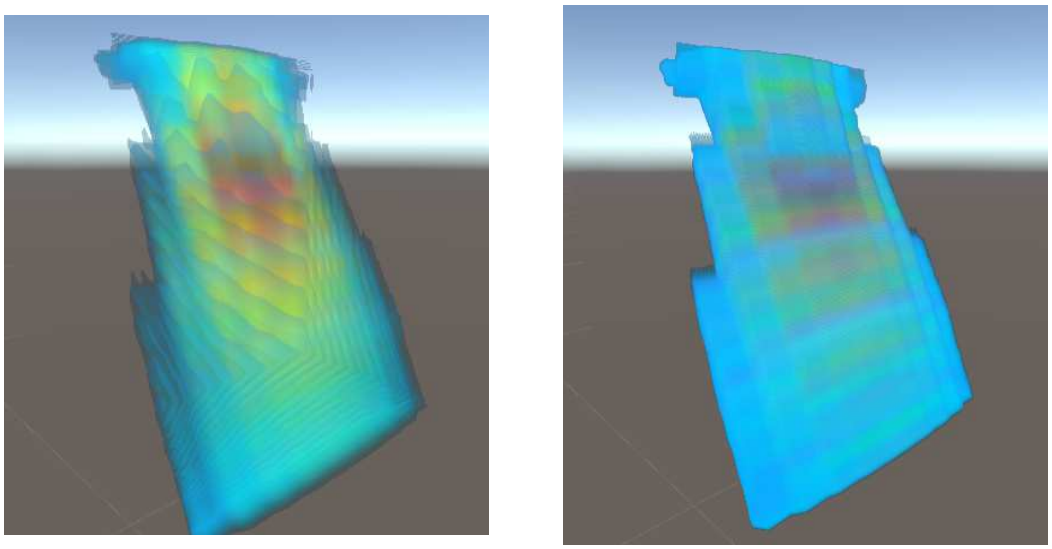

Figure 8.10: Difference between max step size of 30 (on the left) and 200 (on the right)

The marching cubes algorithm result is a mesh that approximates the beam. On the figure 8.11 we can see the comparison between the generated mesh and the beam using the density threshold of 0.3, the result resembles the high density portion of the original beam visualization really well. The mesh is lightly shaded on the image for comparison but it can be disabled from rendering in simulation if desired.

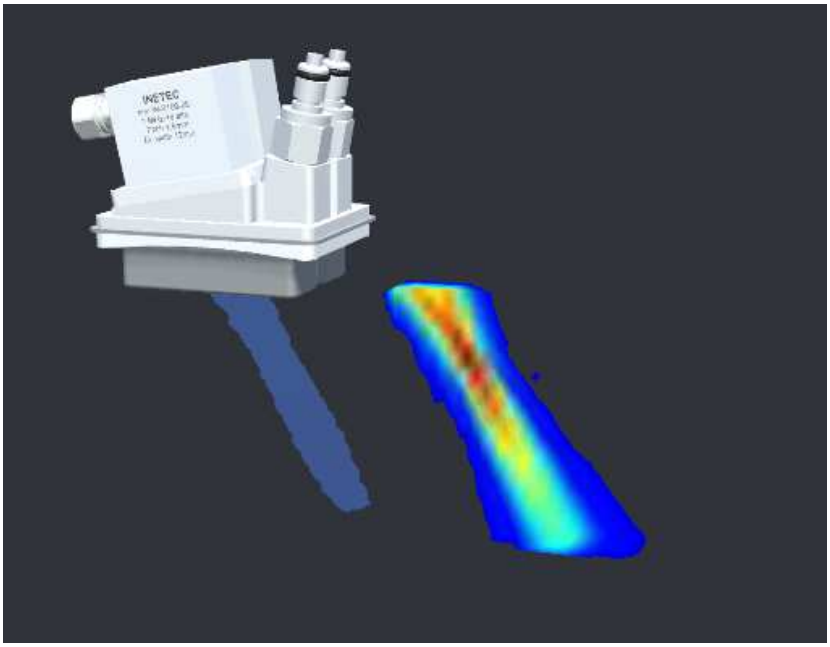

Figure 8.11: Mesh reconstruction for the density threshold of 0.3 and the original visualized beam

The final detection result can be observed on example case on figure 8.12. Defects whose colliders were in the beam range were detected and colored lightly green.

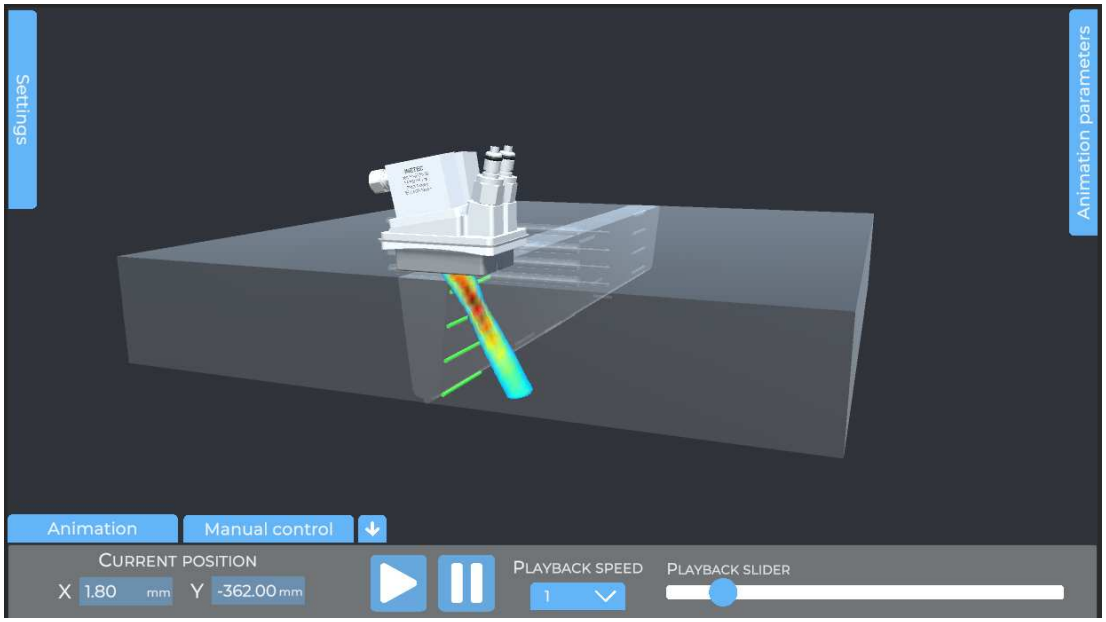

Figure 8.12: Highlighted defects after being detected

8.1. Possible improvements

Future work could improve upon offering more scanning path options, addition of possibility to simulate scanning of complex surfaces like the ones with curves or the ones with obstacles would make this program applicable for use in a wider number of cases since real objects that need quality scanning are not commonly shaped regularly, in addition they could have welds and other obstacles on trajectory. Implementing this can be possibly done by doing some calculations using probe and surface normals but calculating probe position would be way more complex than it is right now for flat and straight surfaces. Additional complication to path calculation would present obstacles that would prevent probe from moving freely.

One small issue, which can be solved easily, 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 as they were used as a placeholder for a course of development. If the simulation images can be re-generated in a bigger resolution, with their quality improved so would the quality results, the final visualization would be clearer and more accurate.

Another part the future research could explore is adding illumination and shading to volume rendered surfaces for a more 3D look of the volume. The improved illumination model would take into consideration ambient, diffuse and specular lighting component as per Phong Illumination Model. The beam visualization would not benefit that much from the three dimensionality given by illumination, since focus is more on area of detection rather than details of the beam's structure, and the beam is not a crude surface, but if its wished to display some other dataset where the structure consist of solid surfaces, like commonly used example of a head MRI with bones and tissues, this would be more useful in bringing that dataset to life.

9. Conclusion

Visualization techniques and computer graphic algorithms have come a long way and their usage is nowadays applied in many different areas and functionalities. These techniques are primarily used for scientific visualization in the field of medicine, since images gained from MRI or CT scanning are normally stored in a series of images, a format naturally suitable for 3D visualization. With constant upgrades and improvements of graphic equipment positively impacting computational power of computers, these techniques are now available in real time and are increasingly used in various industries for different use cases.

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 amount 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 were quite satisfactory and 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 solving that would probably be the best course for future work.

This paper aimed to highlight possibilities and advantages of visualizing data in three dimensions. Volume rendering offers a lot of adjustability, as it can be achieved using different technique variations and as such can be used for lot of different purposes. It is especially useful in cases when the dataset is already represented as volume or in a format that is easily converted into one. This thesis showcased the strengths and advantages of volumetric data visualization applied on a concrete problem, hopefully this set of techniques will be even more researched and used in future as it could improve efficiency and reduce errors in different science fields in general.

REFERENCES

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

[2] Leonard J. Bond. "Fundamentals of Ultrasonic Inspection[1]". In: *Nondestructive Evaluation of Materials*. ASM International, Aug. 2018. ISBN: 978-1-62708-190-0. DOI: 10.31399/asm.hb.v17.a0006470. eprint: https://dl.asminternational.org/book/chapter-pdf/514697/a0006470.pdf. URL: https://www.asminternational.org/documents/10192/22533690/05511G_SampleArticle.pdf/a8c3979c-3b35-f304-68f2-151271f2e3b8.

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

[4] T. D'Orazio et al. "Automatic ultrasonic inspection for internal defect detection in composite materials". In: *NDT E International* 41.2 (2008), pp. 145–154. ISSN: 0963-8695. DOI: https://doi.org/10.1016/j.ndteint.2007.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0963869507000904.

[5] 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.

[6] 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.

[7] modified by E.Zimbres derivative work: DJ Tricky Georg Wiora (Dr. Schorsch). *Sonar$_P$rinciple$_E$N*. URL: https://commons.wikimedia.org/w/index.php?curid=473189.

[8] Khronos Group. *Rendering Pipeline*. URL: https://www.khronos.org/opengl/wiki_opengl/images/RenderingPipeline.png.

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

[10] *Iowa State University : Nondestructive Evaluation Techniques*. URL: https://www.nde-ed.org/NDETechniques/index.xhtml (visited on 06/2022).

[11] Jmtrivial. *Marching cubes configurations image*. URL: https://commons.wikimedia.org/w/index.php?curid=1282165.

[12] GMartin Kraus. *Perspective view frustum*. URL: https://commons.wikimedia.org/wiki/File:Perspective_view_frustum.png.

[13] 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.

[14] TOM McREYNOLDS and DAVID BLYTHE. "Chapter 20 - Scientific Visualization". In: *Advanced Graphics Programming Using OpenGL*. Ed. by Tom McReynolds and David Blythe. The Morgan Kaufmann Series in Computer Graphics. San Francisco: Morgan Kaufmann, 2005, pp. 531–570. ISBN: 978-1-55860-659-3. DOI: https://doi.org/10.1016/B978-155860659-3.50022-6. URL: https://www.sciencedirect.com/science/article/pii/B9781558606593500226.

[15] Roger Ngo. *Marching Squares*. URL: https://urbanspr1nter.github.io/marchingsquares/ (visited on 06/2022).

[16] Nicoguaro. *Marching squares configurations image*. URL: https://commons.wikimedia.org/w/index.php?curid=39714681.

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

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

[19] Unity Technologies. *ShaderLab command: Blend*. URL: https://docs.unity3d.com/Manual/SL-Blend.html (visited on 06/2022).

[20] Unity Technologies. *Unity Editor*. URL: https://docs.unity3d.com/Manual/UsingTheEditor.html (visited on 06/2022).

[21] Unity Technologies. *Unity Shaders*. URL: https://docs.unity3d.com/Manual/Shaders.html (visited on 06/2022).

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

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

[24] Thetawave. *Ray Marching image*. URL: https://commons.wikimedia.org/wiki/File:Volume_ray_casting.png#globalusage.

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

Visualization of Three-Dimensional Ultrasound Data

Abstract

The aim of this thesis is exploring visualization methods of three-dimensional volumetric data and applying learned knowledge to a concrete problem. The problem in question is visualization of ultrasonic probe beam from a series of cross-sections. Additionally, it should be possible to simulate real-life quality scanning of metal objects.

The different methods of rendering three-dimensional data from a stack of two-dimensional images are presented and their advantages and disadvantages discussed. The final software was developed using Unity game engine Marching Cubes algorithm and Volume Ray Casting. The Volume Ray Casting algorithm was realized using Unity's shaders and HLSL language.

The results and options were presented and analyzed along with the possible improvements for the future work. The achieved simulation software can be used as a helping tool in minimizing the number of redundant scans in search of defects inside object structure during quality control phase.

Keywords: Volume rendering, Ray marching, Ultrasonic probe, Unity, Shaders

Vizualizacija trodimenzionalnih ultrazvučnih podataka

Sažetak

Cilj ovog rada je proučiti metode vizualizacije trodimenzionalnih volumnih podataka i primjena stečenog znanja na konkretni problem. Problem u pitanju je vizualizacija zrake ultrazvučne sonde iz niza poprečnih presjeka. Dodatno, trebalo bi biti moguće simulirati skeniranje metalnih predmeta pri kontroli kvalitete.

Predstavljene su različite metode prikazivanja trodimenzionalnih podataka iz niza dvodimenzionalnih slika te se raspravlja o njihovim prednostima i nedostacima. Konačni softver razvijen je korištenjem programskog alata Unity, Algoritma marširanja zraka i Volumnim prikazom podataka. Algoritam marširanja zraka je ostvaren korištenjem sjenčara u alatu Unity i jezikom za sjenčare – HLSL.

Dobiveni rezultati i mogućnosti programa su predstavljeni i analizirani zajedno sa mogućim poboljšanjima za budući rad. Postignuti softver za simulaciju može biti korišten kao pomoćni alat za smanjenje broja suvišnih skeniranja pri pregledu strukture objekata tijekom kontrole kvalitete.

Ključne riječi: Prikaz volumena, Marširanje zraka, Ultrazvučna sonda, Unity, Sjenčari