POLITECNICO
MILANO 1863

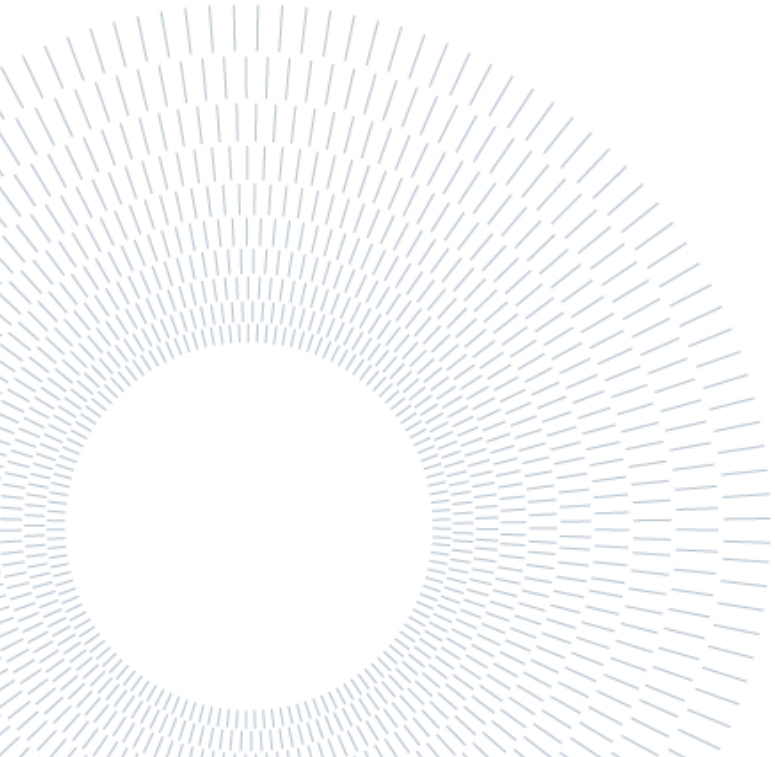
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Ultrasound Simulation with Deformable Mesh Model from a Voxel-based Dataset

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING – INGEGNERIA INFORMATICA

Author: **Nicolo' Stagnoli**

Student ID: 964718
Advisor: Marco Gribaudo
Academic Year: 2022-2023



Abstract

Ultrasound is a widely used, low-cost imaging modality which plays an important role in clinical diagnosis. Today, technology has developed a lot and it is possible to use computer simulations to train in the medical field. The aim of this work is to explore volume rendering techniques and develop a three-dimensional representation of the human body so that it contains also "internal" texture sampled from the AustinMan dataset. This internal texture is then used to build new images from the intersection of a plane with the body. The power of this concept lies in the fact that is possible to build slices of the body in any orientation, not just horizontal. The body can also be set in different poses and the position of the internal points is recalculated.

Regarding voxelizations, two algorithms are presented. The first, very simple, is a voxelization that builds meshes triangulating cubes. It is used to build simple models for the human skin and for every internal layer of the human body. Then, a slightly more complex algorithm, Marching Cubes, is presented. It works creating a polygonal surface mesh from a 3D scalar field by "marching" (looping) through the 3D space and determining each configuration for the given cube. It builds a smoother human skin mesh that, once bone-armored, is used to generate the colliders essential to capture the deformation of single body parts, like hands, arms, feet, etc., that is one main point of this simulation.

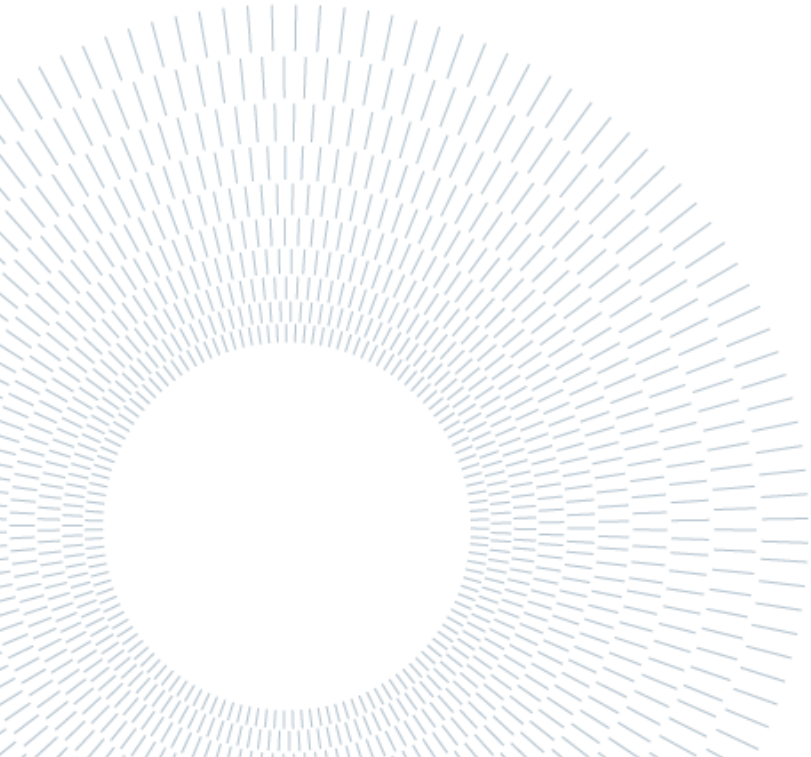
Keywords: Ultrasound Simulation, Voxelization, UVW mapping, Volume Rendering, Mesh generation, Mesh-Volume deformation.

Abstract in Italiano

L'ecografia a ultrasuoni è uno strumento di analisi medica a costo relativamente basso e largamente utilizzato, ed assume un ruolo di importanza nella diagnosi clinica. Oggi, grazie allo sviluppo della tecnologia, è possibile sviluppare simulazioni a computer per l'addestramento nell'utilizzo. L'obiettivo di questo lavoro è di esplorare le tecniche di rendering volumetrico e di sviluppare una rappresentazione tridimensionale di un corpo umano in modo tale che esso contenga anche una texture interna campionata dal dataset AustinMan. Questa texture interna è utilizzata per creare nuove immagini dall'intersezione di un piano con il corpo. La forza di questo concetto consiste nel fatto che sia possibile creare sezioni del corpo in ogni orientamento, non solo orizzontalmente. Il corpo può essere posizionato in diverse pose e la posizione dei punti interni viene quindi ricalcolata.

Riguardo alla voxelizzazione, sono qui presentati due algoritmi. Il primo, molto semplice, implementa una voxelizzazione che crea mesh triangolando cubi. Verrà utilizzato per costruire i modelli del corpo e di ogni tessuto interno. Successivamente, un algoritmo più complesso, Marching Cubes (Cubi Marcianti), è presentato. Esso crea una superficie poligonale da un campo scalare tridimensionale, facendo scorrere un cubo nello spazio, e determinando la configurazione del cubo in ogni luogo. Verrà usato per costruire un modello del corpo più liscio che, una volta armato per la deformazione, sarà usato per generare i collider essenziali a catturare la deformazione delle singole parti del corpo, come mani, braccia, piedi, ecc., che è uno dei principali punti di questa simulazione.

Parole chiave: Simulazione ecografia, Voxel, Coordinate UVW, Rendering Volumetrico, Generazione 3D, Deformazione 3D-Volume.



Contents

1 Introduction

Ultrasound is a widely used, low-cost imaging modality which plays an important role in clinical diagnosis. Image acquisition and diagnosis is, however, difficult to learn and mainly practiced with volunteers, phantoms, or patients. A practical learning phase is required to be able to perform it correctly. However, due to the high cost of the medical equipment and the necessary presence of a patient, the training process is complicated.

Today, technology has developed a lot and it is possible to use computer simulations to train in the medical field. In this case the advantages are many, first the possibility of testing individual skills in a simulated environment without these having consequences in the real world. In addition, another important feature is that of repeatability and reuse of the same simulation to evaluate different aspects of the ultrasound examination.

1.1. MRI Scan

Since its development in the 1970s and 1980s, Magnetic Resonance Imaging, or MRI has proven to be a versatile imaging technique. MRI is a noninvasive medical imaging test that produces detailed images of almost every internal structure in the human body, including the organs, bones, muscles, and blood vessels. MRI scanners create images of the body using a large magnet and radio waves. No radiation is produced during an MRI exam, unlike X-rays. These images give your medical personnel important information in diagnosing your medical condition and planning a course of treatment.

The MRI machine is a large, cylindrical (tube-shaped) machine that creates a strong magnetic field around the patient and sends pulses of radio waves from a scanner. Some MRI machines look like narrow tunnels, while others are more open. The strong magnetic field created by the MRI scanner causes the atoms in your body to align in the same direction. Radio waves are then sent from the MRI machine and move these atoms out of the original position. As the radio waves are turned off, the atoms return to their original position and send back radio signals.

These signals are received by a computer and converted into an image of the part of the body being examined. This image appears on a viewing monitor.

MRI may be used instead of computed tomography (CT, or X-Ray) when organs or soft tissue are being studied. MRI is better at telling the difference between types of soft tissues and between normal and abnormal soft tissues.

MRI provides exquisite detail of brain, spinal cord, and vascular anatomy, and has the advantage of being able to visualize anatomy in all three planes: axial, sagittal, and coronal.

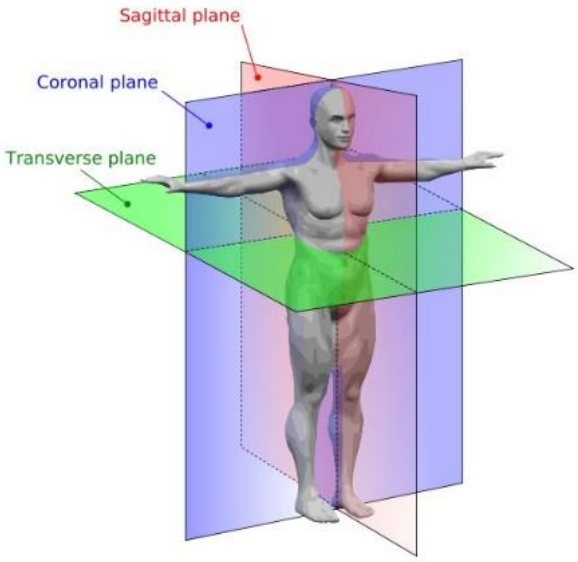


Figure 1: Cutting planes of MRI.

1.2. The AustinMan Dataset

AustinMan is a voxel model of the human body that is being developed for simulations. This dataset was developed from a real MRI scan by segmenting the color cross-sectional (transverse plane) anatomical images. It contains 1878 horizontal (only) slices of a whole human body in different resolutions. The grayscale value of each pixel corresponds to a different layer of the body, such as tissues, bones, and organs, for a total of 64 layers.

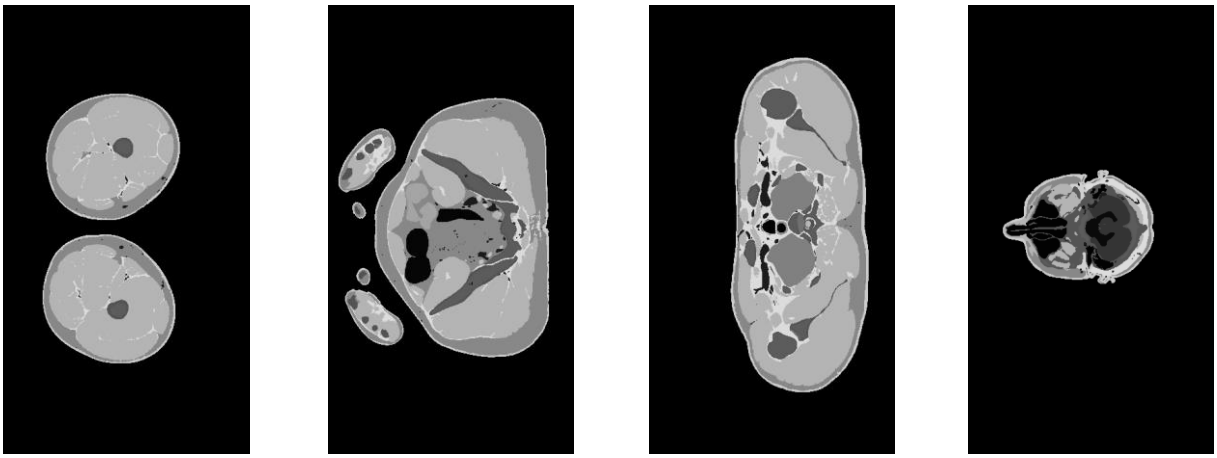


Figure 2: Sample images from the AustinMan dataset

1.3. Voxelization

The word *voxel* originated analogously to the word "pixel", with *vo* representing "volume" (instead of pixel's "picture") and *el* representing "element". A similar formation with *el* for "element" is the word "texel". The term *hypervoxel* is a generalization of voxel for higher-dimensional spaces.

In 3D computer graphics, a voxel represents a value on a regular grid in three-dimensional space. As with pixels in a 2D bitmap, voxels themselves do not typically have their position (i.e., coordinates) explicitly encoded with their values. Instead, rendering systems infer the position of a voxel based upon its position relative to other voxels (i.e., its position in the data structure that makes up a single volumetric image). Voxels are typically stored in three-dimensional matrices, but also smarter data structures can be used.

Voxelization is the term used to indicate the process of making three-dimensional model meshes by creating polygonal surfaces which properties depend on the voxel values.

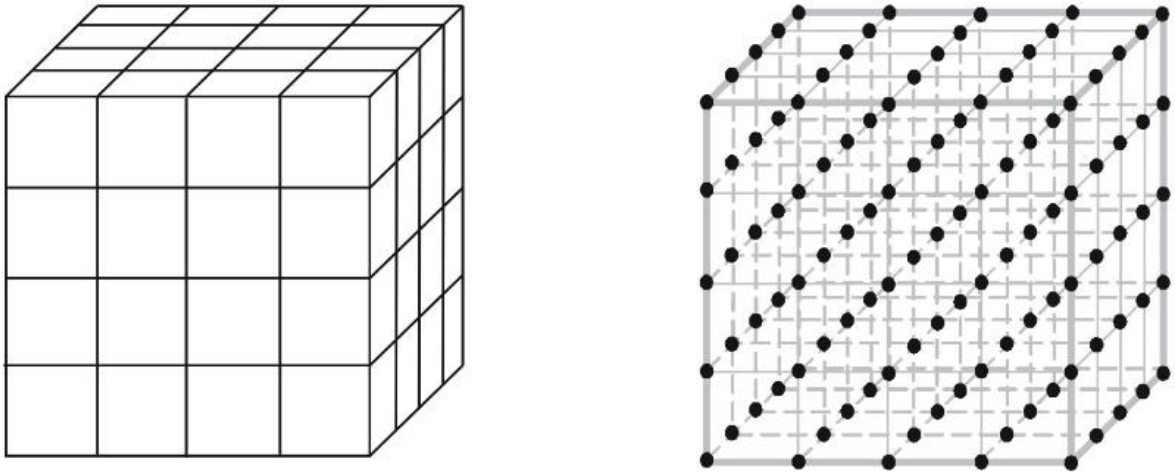


Figure 3: Two different models of a volumetric (voxelized) data set

In contrast to pixels and voxels, polygons are often explicitly represented by the coordinates of their vertices. Polygon points lie in continuous space, while voxels lie in a discrete space. A direct consequence of this difference is that polygons can efficiently represent simple 3D structures with much empty or homogeneously filled space, while voxels excel at representing regularly sampled spaces that are non-homogeneously filled.

A volume described as voxels can be visualized either by direct volume rendering or by the extraction of polygon iso-surfaces that follow the contours of given threshold values. The marching cubes algorithm is often used for isosurface extraction, however other methods exist as well.

Both ray tracing and ray casting, as well as rasterization, can be applied to voxel data to obtain 2D raster graphics to depict on a monitor.

1.4. Volumetric Rendering

Volume rendering is a technique for visualizing sampled functions of 3D data by computing 2D projections of a colored semitransparent volume. It involves the following steps: the forming of an RGB-Alpha volume from the data, reconstruction of a continuous function from this discrete data set and projecting it onto the 2D viewing plane (the out- put based on screen space) from the desired point of view.

An RGB-Alpha volume is a 3D four-vector data set, where the first three components are the familiar R, G, and B color components and the last component, Alpha, represents opacity. An opacity value of 0 means totally transparent and a value of 1 means totally opaque. Behind the RGB-Alpha volume an opaque background is placed. The mapping of the data to opacity values acts as a classification of the data one is interested in. Isosurfaces can be shown by mapping the corresponding data values to almost opaque values and the rest to transparent values. The appearance of surfaces can be improved by using shading techniques to form the RGB mapping. However, opacity can be used to see the interior of the data volume too. These interiors appear as clouds with varying density and color.

A big advantage of volume rendering is that this interior information is not thrown away, so that it enables one to look at the 3D data set as a whole. Disadvantages are the difficult interpretation of the cloudy interiors and the long time, compared to surface rendering, needed to perform volume rendering.

1.4.1. Volume Rendering Techniques

Volume rendering techniques are clustered into two categories, indirect volume rendering and direct volume rendering. Indirect volume rendering, where in a

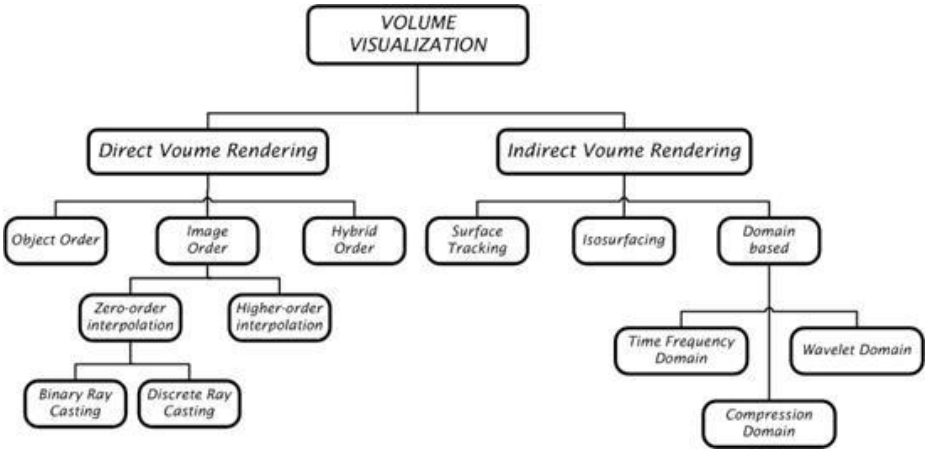


Figure 4: Volume Rendering Techniques Classification

preprocessing step the volume is converted to an intermediate representation which can be handled by the graphics engine. In contrast, the direct methods process the volume without generating any intermediate representation assigning optical properties directly to the voxels.

1.4.1.1. Indirect Volume Rendering

Indirect volume rendering technique extracts polygonal surface from volume data and represents an isosurface, it is also known as 3D contours. Indirect methods aim at the visualization of iso-surfaces defined by a certain density threshold. The primary goal is to create a triangular mesh which fits to the isoregions inside the volume. This can be done using the traditional image processing techniques, where first of all an edge detection is performed on the slices and afterwards the contours are connected. Having the contours determined the corresponding contour points in the neighboring slices are connected by triangles.

The most popular algorithm for indirect volume rendering is marching cube algorithm [13]. The "marching cubes" isosurface reconstruction marches through all the cubic cells and generates an elementary triangular mesh whenever a cell is found which is intersected by an iso-surface. Since the volumetric data defined in the discrete space is converted to a continuous geometrical model, conventional computer graphics techniques, like ray tracing or buffering can be used to render the iso-surfaces.

Another indirect volume-rendering approach is known as 3D Fourier transform (3D FT), where the intermediate representation is a 3D Fourier transform of the volume rather than a geometrical model [12][14][5]. This technique aims at fast density integral calculation along the viewing rays. Since the final image is considered to be an X-ray simulation, this technique is useful in medical imaging applications.

1.4.1.2. Direct Volume Rendering

Direct volume rendering techniques render images of an entire 3-dimensional scalar, a volume, without concentrating on, or explicitly extracting a surface corresponding to certain features of interest or a certain isovalue. In order to directly display the data stored in a volumetric data set an optical model is needed that describes how the scalars representing the volume interact with light, e.g. emission, absorption, reflection, or refraction [9]. In general, each scalar value contained in the volumetric data set is mapped to optical properties, like color and opacity, during rendering. Mapping scalar values to optical properties is usually achieved by evaluating a transfer function for each occurring sample value of the volume.

The application of mapping scalar values to optical properties is also called classification. In general, a distinction is drawn between two types of classification. Pre-classification is done prior to reconstruction of the volumetric data set, whereas post- classification evaluates the transfer function on each reconstructed value.

The optical properties, and their optical effects respectively, are then integrated along viewing rays into the volume. In order to generate a projected image of a volumetric data set, many viewing rays have to be integrated with each ray corresponding to a pixel of the final image. This integral is also referred to as the volume rendering integral. The volume rendering integral is contingent upon the underlying optical model. Many different techniques have been developed to approximately solve the volume rendering integral [26].

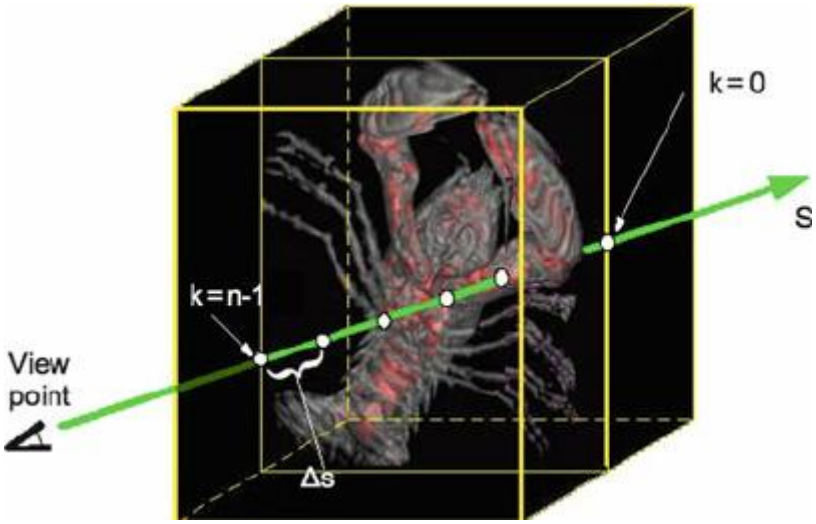


Figure 5: Conceptual model of direct volume rendering

1.4.1.3. Maximum Intensity Projection

Maximum Intensity Projection, MIP for short, is a variant of direct volume rendering. Instead of compositing optical properties, Maximum Intensity Projection determines the final color of each pixel in the final rendering by the maximum value encountered during traversing of each corresponding ray.

Visualizing medical three-dimensional data sets obtained by Magnetic Resonance Imaging scanner is an important application of such a rendering mode. Such volumetric data sets typically exhibit a significant amount of noise that can make it hard use other rendering modes, as it is difficult to extract meaningful isosurfaces, or define a transfer function that aids the interpretation of the data set. However, data values of vascular structures acquired by MRI scanners are higher than the values of the surrounding tissue. This can easily be exploited by Maximum Intensity Projection volume rendering for visualizing such medical data.

1.4.2. The Volume Rendering Integral

With respect to volume visualization, direct volume rendering techniques produce projected images directly from the volumetric data set. These techniques do not involve construction of intermediate representations, such as a polygonal surface, of the data stored in the volume. Therefore, direct volume rendering techniques require

some model of how the scalar values generate, reflect, scatter, or occlude light. As the volumetric data set is usually obtained by sampling a function regularly or irregularly at discrete sampling locations, it has to be interpolated to be used with one of the continuous optical models described here.

A comparison of different interpolation techniques for volumetric data sets can be found in [8]. Here the interpolation is done somehow to give a scalar function *f(x)* defined for all points *x* in the volume. Optical properties, such as color and opacity, can then be assigned as functions of the interpolated value *f(x).* Usually, optical models used for real-time rendering of volumetric data, that deliver a visual compelling visualization of the volume, are quite simplistic, as they do not consider effects like single or multiple scattering. However, real-time methods taking such effects into account are currently becoming available [18]. In each optical model a ray denoted by *x(t)* and parameterized by the distance to the eye *t* is cast into the volume. A brief survey of the most important optical models for direct volume rendering is here given.

Absorption only: The simplest participating medium has cold, perfectly black particles, which absorb all light that impinges on them. None of the particles emits or scatters light. The rate at which incoming light is occluded depending on the value of the particle is called the extinction coefficient or absorption coefficient.

Emission only: The volume is assumed to consist of particles that only emit light, but do not absorb any, as the absorption is negligible.

Absorption plus Emission: In general, particles both occlude incoming light as well as add their own glow. This optical model is commonly used in volume rendering applications as it mimics the light transport most realistically compared to the other simplistic approaches. However, this is also a rather simplistic model since it does not take optical effects, such as single or multiple scattering, i.e more sophisticated interaction between light and particles contained in a volume, into account.

Scattering and Shading: The next step toward greater realism is to include scattering of illumination external to a particle in the volume, i.e indirect illumination. The "Utah approximation" model, a simplified model, assumes external illumination reaches a particle unimpeded by any intervening objects or volume absorption.

Shadows: In order to take shadows into account, the transparency of the volume density between the light source and the point *x(t)*, as well as from *x(t)* to the ray source, should be taken into account

Multiple Scattering: Multiple scattering calculations are important for realistic rendering of high albedo media but are expensive in performance and are, usually, overkill for most scientific visualization applications.

A good trade-off between rendering performance and a visual appealing visualization of a volumetric data set can be achieved when the absorption plus emission optical model is used. The term used to evaluate light transport in this model is often referred

to as the volume rendering integral. The evaluation of this term, that integrates optical effects such as color and opacity, is common to all direct volume rendering techniques. In general, the evaluation of the volume rendering integral can be thought of as being evaluated along viewing rays cast into the volumetric data set, even if no explicit rays are actually employed by the underlying volume rendering technique. Therefore, ray casting could be seen as the most direct approach.

1.5. Ray-Casting

Ray-casting [5] can be seen as the most straight-forward approach for numerical evaluation of the volume rendering integral. Therefore, ray-casting is considered a direct volume rendering technique. But ray-casting can also be used for rendering non-polygonal isosurfaces as well as for Maximum Intensity Projection.

For each pixel in the final image, a single ray is cast from the camera center through this pixel into the volume. The volume is then resampled at certain intervals along a particular ray. The distance between two adjacent resampling locations is commonly referred to as the sampling distance. Usually, the sampling distance is equidistant along each ray, but some methods have been proposed that use non-equal distance between two adjacent sampling locations. For example, one technique jitters the sampling locations to eliminate patterned sampling artifacts [21], whereas another technique increases the sampling distance in order to efficiently skip empty regions of the volumetric data set [1].

The sampling, or reconstruction, of the volumetric data set is commonly done using trilinear interpolation. However, lower-order reconstruction filters, like nearest-neighbor, or higher-order reconstruction filters, like tri-cubic, can also be employed. After resampling, the interpolated scalar value is mapped to optical properties by evaluating the current transfer function for this value. This can be done efficiently, when precomputing the transfer function and storing the results in a lookup table. Instead of evaluating the transfer function for each scalar value, the scalar value is used

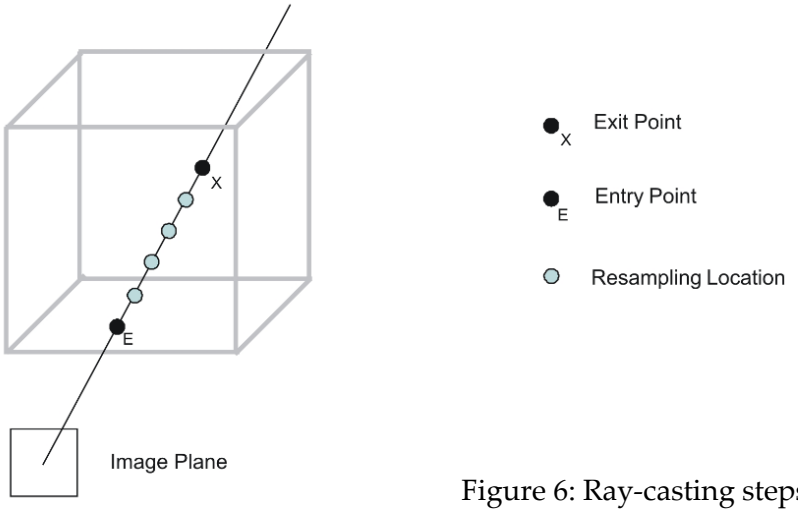


Figure 6: Ray-casting steps

as an index to access the precomputed values stored in the corresponding lookup table. Both methods, evaluating the transfer function on-the-fly or indexing into the lookup table, yield an RGBA color value for the corresponding location inside the volume that subsumes the respective emission and absorption coefficients [15]. The volume rendering integral is numerically approximated numerically by compositing these values in either front-to-back or back-to-front order.

1.6. The Rendering Pipeline

A pipeline consists of a sequence of stages operating in parallel and in a fixed order. Each single stage receives data from its prior stage as input and sends its output to the next stage in the sequence after it is finished with its work.

The rendering of virtual scenes on today's graphic hardware is implemented in such a pipelined fashion. The order of operations necessary for turning the geometry of a virtual scene into pixels that can be displayed on a screen or a portion of it is called the rendering pipeline and used to be implemented in a fixed way. Earlier developments have led to replacing the fixed function pipeline, specifically parts thereof, with a programmable one.

Input to the rendering pipeline is a stream of vertices describing the scene that can be joined to form geometric primitives, typically lines, triangles, quads, or polygons. It then computes a raster image of the virtual scene. The rendering pipeline can roughly be divided into five different stages.

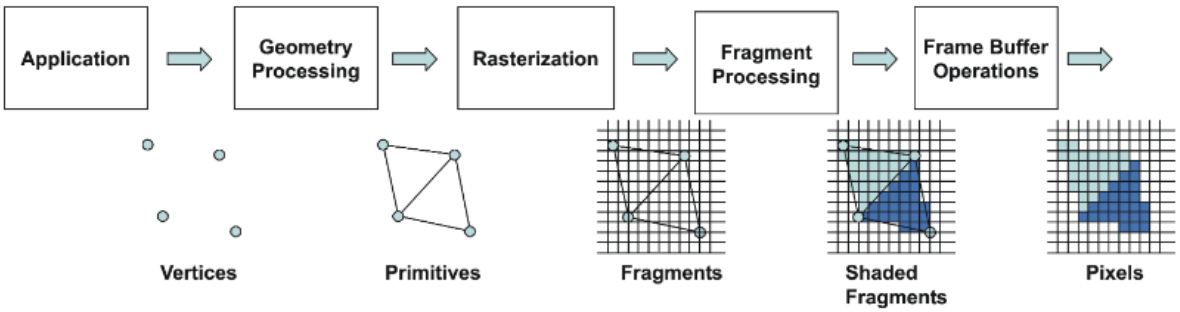


Figure 7: The fixed-function rendering pipeline

1.6.1. Geometry Processing

The geometry processing stage's or simply the geometry stage's tasks include the majority of the per-geometric primitive or per-vertex operations, responsible for modifying the incoming stream of vertex data. The geometry engine part of a modern GPU computes linear transformations, projective transformations, and evaluates local illumination models on a per-vertex basis.

Model and View Transform: On their way to the screen, all models are transformed from their own model-coordinate space to world-coordinate space by the model transform. The purpose of the view transform is to place the camera origin at a specific location and aim it in a certain direction dependent on the underlying 3D application programming interface to facilitate subsequent projection and clipping operations. After a vertex has been transformed to this so-called view-coordinate space or eye-coordinate space it is sent to the next stage of the geometry stage. Model and view transformations are specified as a 4 × 4 matrix using homogeneous coordinates usually concatenated into a single matrix, often called the model view matrix, for performance reasons.

Lighting: After all model and view transformations have been applied a local illumination model, e.g. Phong Lighting [7], is evaluated for each vertex. As this requires information about normal vectors and viewing direction it must be computed after the model and view transformations. It also requires that all objects, the camera, and all light sources, reside in the same coordinate space, thus re- quiring a transformation of a scene's light sources to eye-coordinate space as well. This is made possible because all relative relationships, such as normals and distances, between light sources and models are preserved by the linear transformations from model-coordinate space to eye-coordinate space, for models, and from world- coordinate space to eye-coordinate space, for light sources.

Projection: Next, geometric processing engines perform a perspective transformation, transforming the view volume, also called the frustum, into usually a unit cube with its extreme points at (−1; −1; −1) and (1; 1; 1), called the canonical view volume. Still it is considered to be a projection because after display the z-coordinate is not stored in the image generated. There are two different types of projections, namely orthographic projection and perspective projection. After either projection, vertex coordinates are stored in normalized device coordinates.

Primitive Assembly: Geometric primitives are generated from the incoming stream of transformed vertices. Vertices are connected to lines and lines are joined together to form triangles. Arbitrary polygons are usually considered to be tessellated into triangles to ensure planarity.

Clipping: Primitives fully inside the viewing frustum are passed as is to the next stage of the pipeline. Primitives completely outside are discarded. Only primitives that are partially inside the viewing volume are subject to clipping against the unit cube. For example, for a line where only one vertex is considered to be inside a new vertex has

to be generated where the line intersects the viewing volume while the vertex outside can be discarded for further processing.

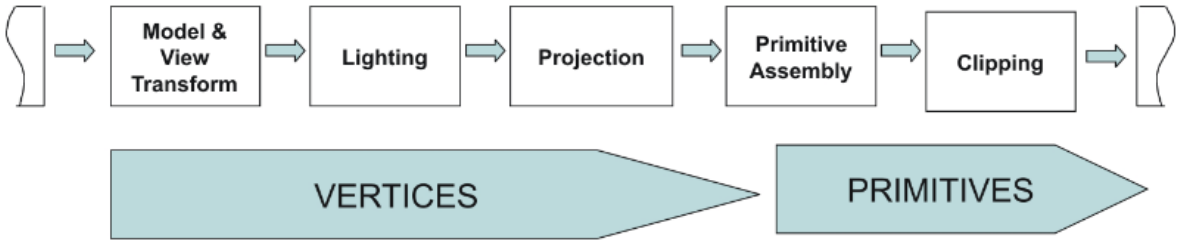


Figure 8: Geometry processing stage

1.6.2. Rasterization Stage

Rasterization is the process of determining the set of screen pixels covered by a geometric primitive. The result of the rasterization are a set of pixel locations as well as a set of corresponding fragments. It is important to distinguish between pixels and fragments. Pixels, short for picture elements, are parts of the final image that is displayed on a screen. Instead one can think about a fragment as a ”potential pixel”. A fragment is the same size as a pixel, is associated with the same screen location as the pixel it corresponds to, and has a set of parameters such as color, depth value, and one or more sets of associated texture coordinates. These parameters a generated by the rasterizer during fragment generation through interpolation of the corresponding vertices of the geometric primitive. Whether a fragment becomes a real pixel or not is decided through various tests at the end of the rendering pipeline.

1.6.3. Fragment Processing

This stage can further be divided into two different tasks as shown in figure 3.3. Once a geometric primitive has been rasterized into a set of zero or more fragments it enters either the texture fetching stage or the fragment shading stage. It may skip the texture fetching stage regarding by the current state that is set by the underlying 3D programing interface, if no texture lookup is to be executed.

Texture Fetching: Textures are 1-dimensional or multi-dimensional images that can be ”glued” to a 3-dimensional object. They are mapped onto geometric primitives in correspondence to the texture coordinates interpolated in the rasterization stage. This process yields an interpolated color value fetched from the texture. The order of interpolation depends on the dimension of the texture target and the graphic hardware’s capabilities. Current generation GPUs support the simultaneous fetching of multiple textures for each fragment without a hit in performance. Furthermore, these GPUs allow for enhanced controlling of the texture lookup itself. It is possible,

for example, to use the color value returned by the first texture fetch as texture coordinates for consequent texture lookups. This is known as dependent texturing.

Dependent texturing is important to implement different sorts of transfer functions for volume rendering. Other fragment attributes can be used as texture coordinates as well.

Fragment Shading: The fragment shading stage applies further color operations on a given fragment to compute its final color. This stage is also capable of applying different math operations on a fragment's values. It may choose to change nearly every value of a fragment, like the depth value, except for its screen location. Even allowing for the possibility that this stage may completely discard a fragment, thus preventing the fragment's corresponding screen pixel from being updated. The fragment shading stage emits one or zero completely colored fragments for each input fragment it receives.

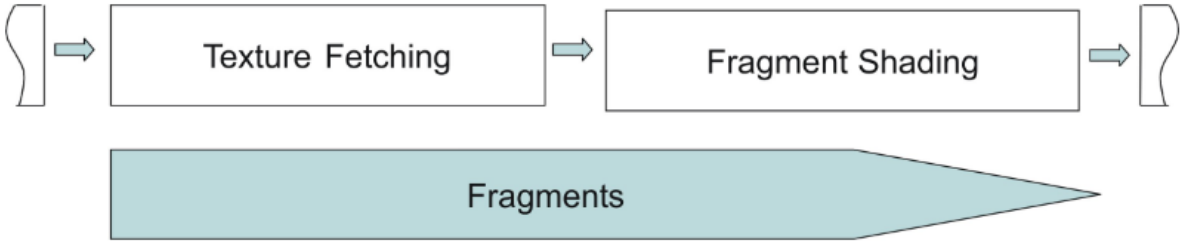


Figure 9: Fragment processing stage

1.6.4. Frame Buffer Operations

The frame buffer operations stage performs a set of per-fragment operations right before the fragment is turned into an actual pixel. The incoming fragment is at first checked based on the number of different tests. If any of these tests fail the pixel operations stage immediately discards the specific fragment without updating its corresponding pixel's value stored in the frame buffer. All tests can be enabled or disabled by the programmer, though it is not possible to change either their order of sequence nor their functionality. If a fragment passes all the tests another set of operations is performed to update the values stored in the associated buffers. Thus, the fragment has finally advanced to being a pixel. These operations can also be enabled or disabled.

Scissor Test: The scissor test is used to restrict drawing of pixels to a rectangular portion of the frame buffer. If a fragment lies inside this rectangle, it is further processed by the subsequent tests.

Alpha Test: The alpha tests compare the incoming fragment's opacity, its alpha value, with a reference value. The fragment is accepted or rejected based on the outcome of this comparison.

Stencil Test: The stencil test is typically used to mask out an irregularly shaped region of the frame buffer to prevent drawing from occurring within it. The pixel locations drawing is allowed or rejected on values stored in the stencil buffer, that is part of the actual frame buffer. Therefore, it resembles the frame buffer in width and height. The stencil buffer is essential to the application of the stencil test, without it every fragment passes the stencil test automatically. The stencil test itself involves a comparison of the fragment's associated pixel's stencil value stored in the stencil buffer with a reference value. Optionally this comparison can also take the associated pixel's depth value into account. If fragment passes the stencil test it may choose to update the value stored in the stencil buffer as well.

Depth Test: The distance between the camera origin and an object, the z-coordinate inside the view volume of an object, currently occupying a pixel location is stored in a specific buffer, namely the depth buffer. The depth buffer is also part of the frame buffer, therefore extending to the same dimensions as the frame buffer. The depth test decides whether an incoming fragment is occluded by a previously drawn pixel, by comparing the incoming fragment's depth value to the associated pixel location's depth value already stored in the depth buffer. If a fragment passes the depth test it may choose to update the depth buffer value with its own. The depth buffer together with the depth test therefore provide a convenient mechanism for depth ordering either partially or fully occluded objects on a per-fragment level.

Blending: After a fragment has passed all the pixel tests its color values are then combined with the color values already stored in the frame buffer at the corresponding location. This combination is referred to as blending. Different blending operations can be applied, e.g. replacing or modulating depending on the stored alpha values, thus allowing for semi-transparent objects.

Logical Operations: The final operation on a fragment is a logical operation, such as OR, XOR, and NEGATE. This operation is applied before the fragment is written to the frame buffer, thus becoming a pixel, to the incoming fragment's values and/or the values currently stored in frame buffer.

Figure 10: Frame buffer operations

1.6.5. Pipeline Programmability

Traditionally the rendering pipeline was implemented as a fixed-function sequence of stages as described before. An application developer only had the choice to enable or disable certain stages, functionalities, of the rendering pipeline, like lighting or texturing. Additionally 3D programming interfaces along with the underlying

hardware allowed for the possibility to set parameters for different stages, thus giving the programmer more configurability. But this made it nearly impossible for a programmer to implement other functionalities than what was implemented in hardware, such as the Gouraud Shading model [19], Phong's specular highlighting equation [7], or applying textures to surfaces [22]. Many of these algorithms were invented over 30 years ago but still were the mainstay of graphics hardware for years. Therefore, effects not implemented in hardware had to be computed using the regular CPU.

In order to free up CPU time for other computations than graphics processing, graphics subsystems had to offer true programmability. As the traditional rendering pipeline was not assigned for programmability its design had to be extended. This resulted in the inclusion of two distinct programmable processors, namely the programmable vertex processor and the programmable fragment processor. A Program written for either of the programmable processors is referred to as *Vertex Shader* or *Fragment Shader* respectively. Programs can be written through vendor-specific extensions to the 3D programming interface, using vendor-specific assembler code, or using one of the available high-level shader languages. The capabilities of different shader versions are comprised in a specification called shader model.

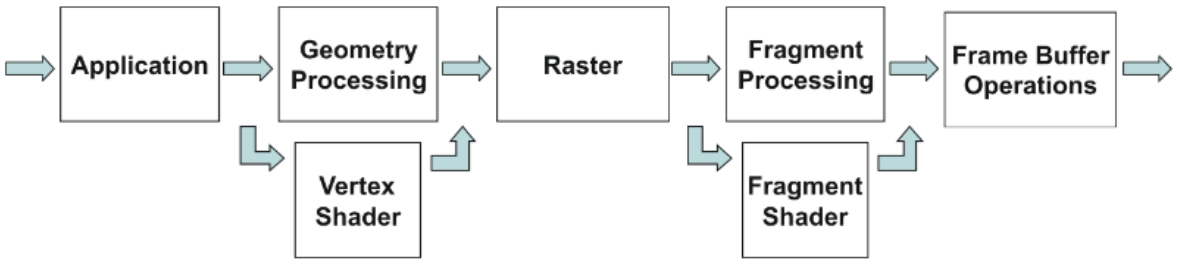


Figure 11: The programmable rendering pipeline

1.6.5.1. Vertex Shader

As their name and their place in the rendering pipeline suggests vertex shaders provide a way to modify parameters associated with each vertex, like its color, normal, texture coordinates, and position. A vertex shader processes each vertex that is passed to it separately. It is not possible to pass results generated by one vertex to another vertex. Neither can a vertex shader create additional nor destroy superfluous vertices. A vertex shader's output must always at least consist of the vertex's homogeneous clip coordinates. But many other values, like diffuse color or texture coordinates, can be modified beyond this. As the complete shader architecture is tailored toward graphical computing many of the supported functions can be executed most efficiently on three- and four-element vectors.

Vertex Shaders are designed to increase the computation speed of more sophisticated lighting models [10]. They are able to compute values that slowly change over a surface, as further down the rendering pipeline these values will then be interpolated. By design, and as their name implies, they are not capable of shading fragments.

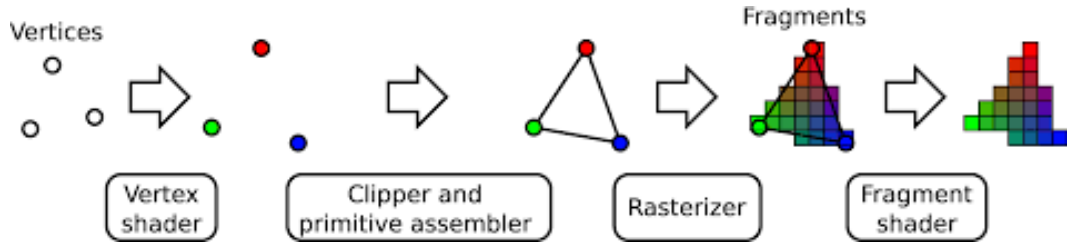


Figure 12: Programmable pipeline stages

1.6.5.2. Fragment Shader

In contrast to vertex shaders, fragment shaders, also called pixel shaders, are executed on a per-fragment basis during a rendering pass. Fragment shaders are essentially an evolutionary extension of the fixed function multitexture fragment processing stage of the rendering pipeline. Both operate on constants and interpolated values. Also, both are capable of multiple texture retrievals to produce a pixel color, and optionally an associated alpha value. However, fragment shader capabilities surpass the limited functionality of the fixed function multitexture fragment processing stage.

Fragment shaders support many of the same math operations that vertex shaders do, as well as texturing operations. The interpolated diffuse and specular colors and alphas, constants, and sets of texture coordinates are the three sets of inputs passable to a fragment shader. Each of these can be a vector of up to four values. Inputs are treated as RGBA data. A fragment shader consists of definitions of constants, a number of arithmetic instructions, and a number of texture address operations.

Fragment shaders are capable of using any result of a computation during their execution, even color values of a texture fetch, as texture coordinates for subsequent texture retrievals. Such a texture lookup is called dependent texture lookup. Dependent texture lookups are an essential part of volume rendering applications on consumer graphics hardware.

1.7. Ultrasound Simulator

Medical ultrasound is a medical procedure used for therapeutic or diagnostic purposes that makes use of ultrasound waves. In diagnostic cases, the reflection of these waves is recorded and used to create an image of a patient's internal body structure, measure certain features, or record audible sounds.

This is made possible by ultrasound waves generated by piezoelectric crystals in the probe itself. When these are electrically excited, they begin to vibrate, generating waves with frequencies of more than 20 kHz that act on the patient's internal structures. The reflected waves are called back echoes, and by scanning them with a computer it is possible to obtain different images depending on the intended use of the ultrasound system.

An Ultrasound Simulator is a medical simulation training tool that enables educators and learners to practice diagnostic, therapeutic, and surgical applications as they relate to imaging interventions. A key advantage of using ultrasound simulation is that the practice makes advanced visualization, case databases and automatically generated feedback possible.

Figure 13: Example of Ultrasound Simulator
software and tools for Augmented Reality

Overall, the main advantage of ultrasound simulators is that they provide new, innovative ways for learners to build up mental models. Yet, the most important feature of an ultrasound simulator is to provide feedback, which conforms to the concept of a mental model that must be updated. Another aspect that has been found critical across the realm of medical simulators is the ability to provide a range of difficulty levels for users with different skills.

Unfortunately, there are some disadvantages associated with this simulation systems.

First, these systems are now very expensive, and it is not possible to provide one to every student to train. These systems are generally quite bulky and require a very high computational power, which leads to being forced to use them in places set up for this purpose.

Furthermore, many of these systems consist in a "static" simulation, meaning that the models involved in the simulation are fixed and cannot be deformed. These models are often hand-crafted. A lot of effort is made to produce and validate these models, increasing the total cost of the work.

1.8. Purpose and motivation

The solution chosen to overcome the disadvantages described above is to provide a reliable simulation of an ultrasound activity, using an ultrasound probe to perform the examination. This simulation works in a lightweight environment that can potentially run also on portable devices.

All the three-dimensional models of the human body skin and its internal tissues and organs involved in this simulation are generated with voxelization algorithms. These voxelization algorithm in general performs triangulations of surfaces by sampling values from a discrete field of three-dimensional points.

When the ultrasound probe intersects the body, the "internal" texture of it is shown on the screen in the scene.

1.8.1. Mesh Model Deformation

The main point of the developed simulation is the possibility to deform the human body to set it in a general pose. Arms and legs can be oriented in any direction. Doing this, a technique to map every part of the body to the "internal" texture, and deform it, accordingly, is required.

1.9. Tools

To develop the proposed solution, the *Unity* game engine framework is used, along with C# as programming language.

Unity is a cross-platform game engine developed by Unity Technologies; it can be used to create three-dimensional, two-dimensional, Virtual Reality and Augmented Reality simulations and games.

C# is an object-oriented programming language developed and maintained by Microsoft, and it is used as a scripting language in Unity.

Models generated with the voxelization algorithms are post-processed using Blender, which is a free and open-source 3D computer graphics software used for creating animated films, visual effects, art, 3D printed models, motion graphics, interactive 3D applications, Virtual Reality and computer simulations and games.

The Unity project folder of this work can be found at:
https://github.com/nicolostagnoli/AustinMan-Voxel

1.10. Thesis Outline

Here a brief overview of this thesis content:

- Chapter 2 - State of the art in Ultrasound Simulation

  Brief review of recent techniques employed in Ultrasound Simulation and motivation of this work.

- Chapter 3 - Voxelization Algorithms

  Description of the presented voxelization algorithms and about their implementation.

- Chapter 4 - Ultrasound Echography simulation

  Description of the techniques employed for the developed Ultrasound simulation.

- Chapter 5 – Conclusions

  Comparison of the developed solution with the previous work, analysis of the criticalities about its functioning, and some proposals for further research.

2 State of the Art in Ultrasound Simulation

The evolution and development of technologies and computer knowledge in recent years have radically changed the way we look at things in many ways and have brought many benefits to the working environment.

One of the sectors that has been affected by these benefits, and that is still evolving today, is the medical sector, especially the field of medical simulations. These are used in many areas of medicine, such as medical ultrasound, to identify or reproduce specific case studies that can be used for learning.

2.1. Ultrasound Simulation Methods

In achieving educational outcomes, computer-based simulators mimic the ultrasound image produced within a computer. The methods to simulate ultrasound images can be categorized into:

- Interpolative
- Generative image-based
- Generative model-based

The interpolative approach uses prerecorded three-dimensional ultrasound volumes and slicing techniques, that can be combined with postprocessing like deformations and artificial shadow insertion in the final image. [24] [20]

Generative image-based models rely on Machine Learning and Deep learning techniques, Particularly, almost all the approaches to realistically simulate ultrasound images in this way are based on generative adversarial networks (GANs).

The generative approach simulates ultrasound images using geometry from imaging systems like computed tomography (CT) and magnetic resonance (MRI), or it is based on mesh models. Generative model-based ultrasound simulators create an image by extracting a bi-dimensional slice from the model and texturizing it. Although this method is very appealing for generating and depicting cases involving different pathologies, modeling, and confirming that the model is correct can present challenges. The model creation for this approach is more complex than for the interpolative approach because each model needs some preprocessing. [17] [16]

Another generative model-based technique is based on ray tracing. Multiple ray emissions from the probe are simulated and the intersection points with the three-

dimensional mesh model are calculated. Artifacts like shadowing and refractions can be added with postprocessing.

2.2. Related works

Despite the numerous proposed approaches and the current presence of very accurate commercial applications, current implementations require a lot of computational power and/or a lot of previous manual work for the creation and validation of the models.

We therefore want to investigate whether it is possible to obtain similar results in a computationally lighter environment, also accessible from portable devices such as laptops, tablets, and smartphones. For this reason, the Unity game engine has been chosen as a working environment.

In the past year, here at Politecnico di Milano, Diego Zucca presented a lightweight reliable simulation of an ultrasound activity, using an ultrasound scanner and an ultrasound probe to perform the examination, running on VR (Virtual Reality headset). Thanks to this simulation, the sensation of immersion of the environment in which the user finds himself and the fidelity of the simulation of an ultrasound examination achieve a good result of realism compared to the cost necessary for the use of the application, thus finding a good quality-price ratio. [27]

2.3. Objective

The aim of this work is to enhance the previous work by Diego Zucca to account for model deformation. Setting the model in different poses could be interesting to see what happens to internal body part when the pose is changed, or to see what the effect of involuntary body movements are, like breath, on the internal texture. For this reason, a technique to map every part of the body to the "internal" texture, and deform it, accordingly, will be investigated.

This work led to the realization of a hybrid method for ultrasound simulation. Since images on the screen are built by pre-recorded images, from the AustinMan dataset, it is an Interpolative method. The concept of mesh model deformation, and its application to the interpolated image, however, derives from a three-dimensional model. We can categorize this method as an Interpolative Model-Based.

3 Voxelization Algorithms

In this chapter the voxelization algorithms used to generate the model meshes of the human body are presented.

3.1. Simple Voxelization

3.1.1. Instantiating cubes

The simplest method to construct a three-dimensional model from the stack of images of the dataset is to instantiate a cube for each colored pixel, since black color corresponds to air. However, due to the high number of pixels in the dataset, this simple approach is not feasible because memory overload. The number of vertices is too high, and most of them are inside the body, so not even rendered.

3.1.2. Algorithm

The idea is to algorithmically create vertices and triangulate them to make simple cubes. Each pixel in the dataset images corresponds to a cube. If one face of a cube being drawn is covered by another cube, the face is inside the mesh, and it doesn't need to be rendered. Doing so, only the external part, the skin, of the human body is drawn, while the inside is empty.

This simple approach is also used to build each layer of the human body. Different meshes are built iterating each time on the whole dataset, taking only pixels with a specific value of the grayscale value.

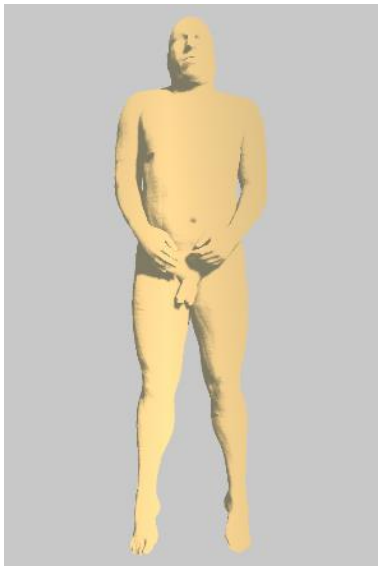


Figure 14: The mesh generated with the first algorithm.

3.1.3. Implementation

The implementation of this algorithm can be found in the *Layers_Voxelization* scene in the Unity project. The *AustinMan* game object is set with an initial empty MeshFilter a standard MeshRenderer, and a script component called *Voxel_Grid_Layers*.

This script initializes the three-dimensional voxel matrix with the grayscale pixel values of the AustinMan dataset images, with values from 0 to 255. After this, an array of Unity mesh is initialized, one for each grayscale. By looping all the voxels, a cube is added to the mesh of the corresponding layer. Each cube is set with world coordinates corresponding to the position of the voxel in the matrix. Cubes are built by creating faces with the *AddQuad* function. For each mesh (layer), a face of a specific cube is currently added only if there is no other cube (of the same layer) directly next to it. In this way, only visible faces are added to the mesh and rendered, making the voxelization feasible.

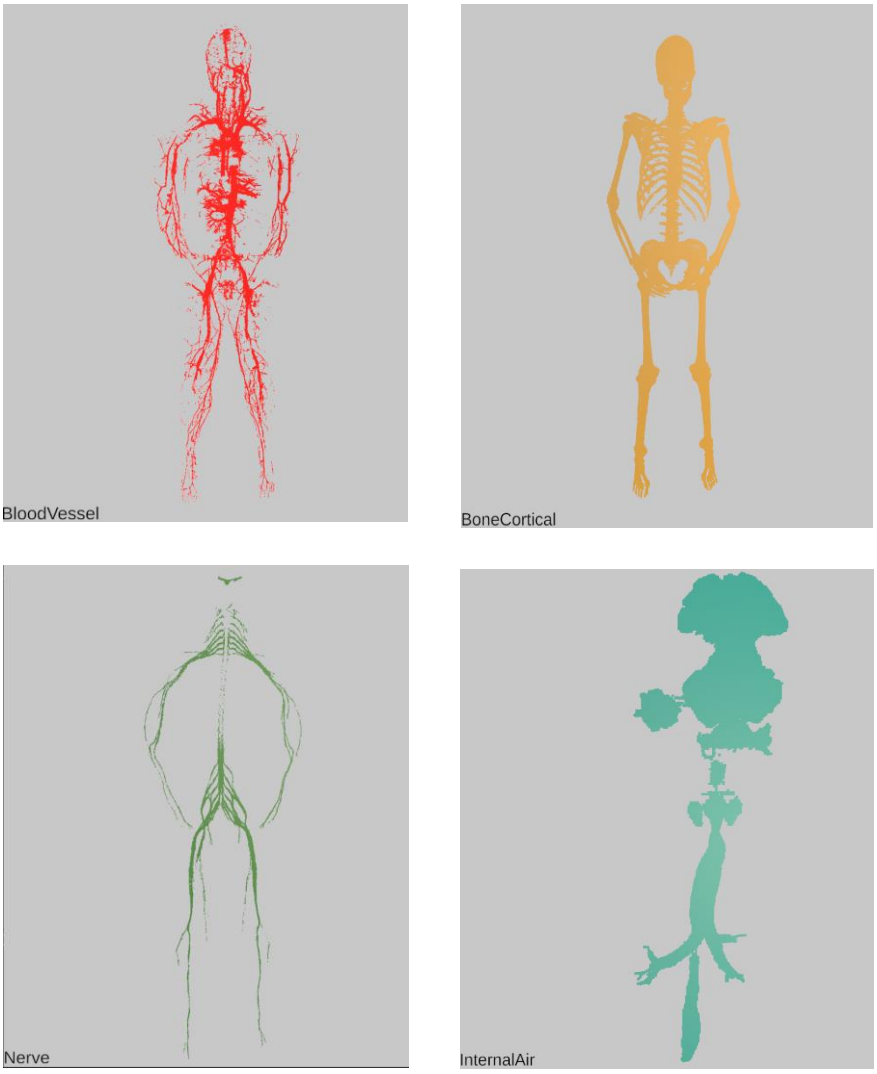


Figure 15: Some of the layers generated with the first algorithm.

3.2. Marching Cubes

The Marching Cubes algorithm, developed by Lorensen and Cline in 1987 [13] is used to approximate an isosurface by subdividing a region of space into a three-dimensional array of rectangular cells, which is the most popular method for isosurface rendering. It is mainly used in medical applications as indirect volume rendering technique and in video games to procedurally generate terrains (meshes, in general) from a discrete field of values, which can be randomly generated from noise or taken by image maps. The marching cubes algorithm creates a polygonal surface mesh from a 3D scalar field by "marching" (looping) through the 3D space and determining each configuration for the given cube.

The basic idea of Marching Cubes is that voxel could be defined by the pixel values at the eight corners of the cube. If one or more pixels of a cube have values less than the user specified isovalue, and one or more have values greater than this value, we know the voxel must contribute some component of the isosurface. By determining which edges of the cube are intersected by the isosurface, we can create triangular patches which divide the cube between regions within the isosurface and regions outside. By connecting the patches from all cubes on the isosurface boundary, we get a surface representation.

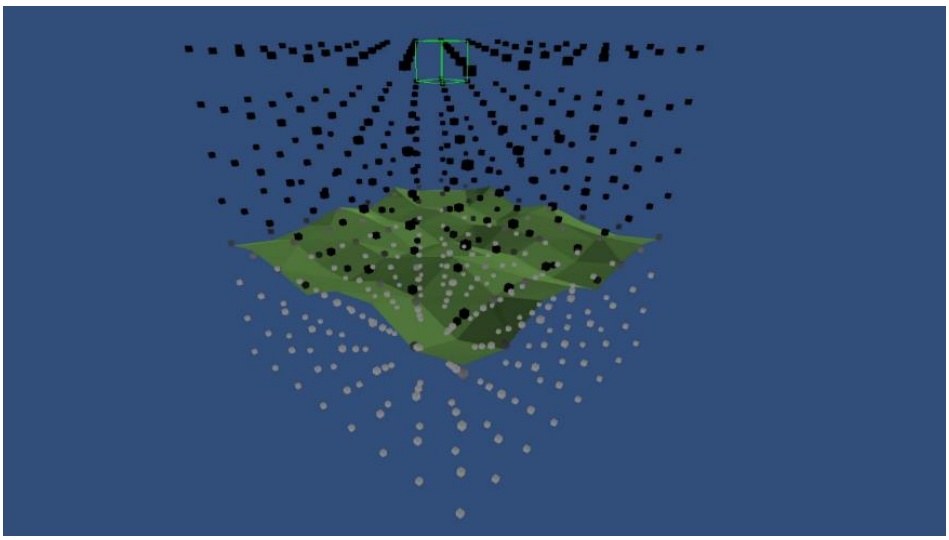


Figure 16: An example of a terrain generated with the marching cubes algorithm from a noise generated 3D volume.

In the eighties volume-rendering research was mainly oriented to the development of indirect methods. At that time no rendering technique was available which could visualize the volumetric data directly without performing any preprocessing. The existing computer graphics methods, like ray tracing or z-buffering [6] had been developed for geometrical models rather than for volume datasets. Therefore, the idea of converting the volume defined in a discrete space into a geometrical representation seemed to be obvious. The early surface reconstruction methods were based on the

traditional image processing techniques [3][4][23], like edge detection and contour connection. Because of the heuristic parameters to be set these methods were not flexible enough for practical applications, they lack detail and introducing artifacts. Lorensen and Cline [13] came up with the idea of creating polygonal representation of constant density surfaces from a 3D array of data. Existing methods of 3D surface generation by Wyvill et al. [2] trace contours within each slice then connect with triangles (topography map), create surfaces from voxels, perform ray casting to find the 3D surface using hue-lightness to shade surface and gradient, and then display density volumes. There are some shortcomings of Wywill et al's techniques. One thing is that they throw away useful information in the original data. Another thing is that these methods lack hidden surface removal, and volume models display all values and rely on motion to produce a 3D sensation.

Thus, the Marching Cubes algorithm is introduced. Marching Cubes algorithm uses all information from source data, derives inter-slice connectivity, surface location, and surface gradient, also the result of Marching Cubes can be displayed on conventional graphics display systems using standard rendering algorithms and requires only one parameter which is a density threshold defining the isosurface. In summary, marching cubes creates a surface from a three-dimensional set of data as follows:

1. Read four slices into memory.
2. Scan two slices and create a cube from four neighbors on one slice and four neighbors on the next slice.
3. Calculate an index for the cube by comparing the eight density values at the cube vertices with the surface constant.
4. Using the index, look up the list of edges from a precalculated table.
5. Using the densities at each edge vertex, find the surface and edge intersection via linear interpolation.
6. Calculate a unit normal at each cube vertex using central differences. Interpolate the normal to each triangle vertex.
7. Output the triangle vertices and vertex normals.

3.2.1. Algorithm

Every point in the 3D world is a value from 0 to 1, where 0 is black and above ground, and 1 is white and underground, or vice versa. We march a single cube through the 3D space and construct a mesh. When the value at a vertex is below a given threshold, also called isosurface, we can say that this vertex of our cube in the terrain is underground, and we want to hide it by drawing a face. A configuration is chosen by determining which of those vertices are below the isosurface, and which are not. In total there are 256 such combinations that can be formed by looking at the values of our vertices since cubes have 8 corners with each 2 possible states. These 256 configurations can be reduced to only 15 since most cases are symmetries.

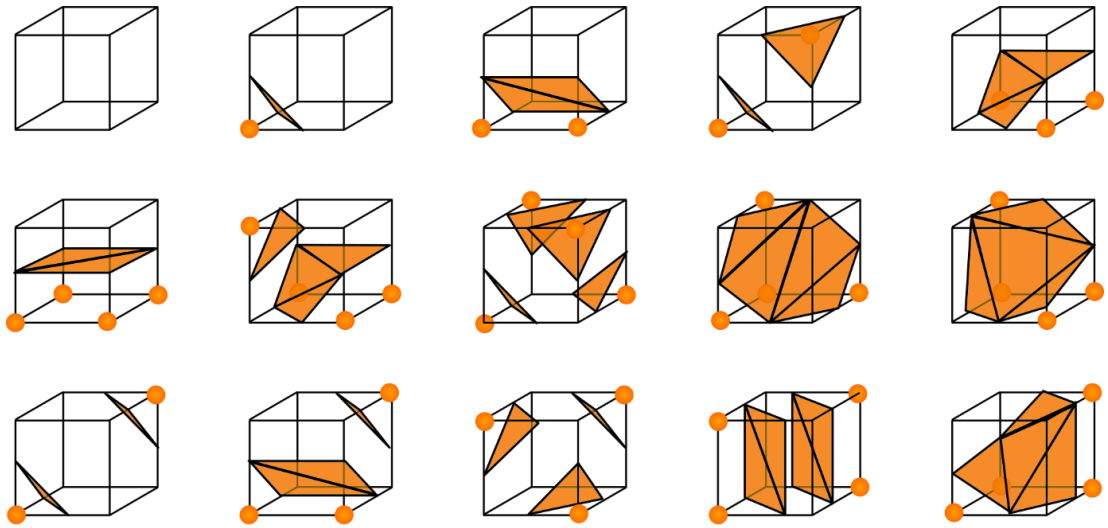


Figure 17: Configurations of a single cube in the marching cubes algorithm.

The algorithm begins by determining the configuration of the cube, by comparing the value of our cube at every corner vertex with the isosurface level. Cube configuration is then found with a lookup table, which contains lists of edges for each of the 256

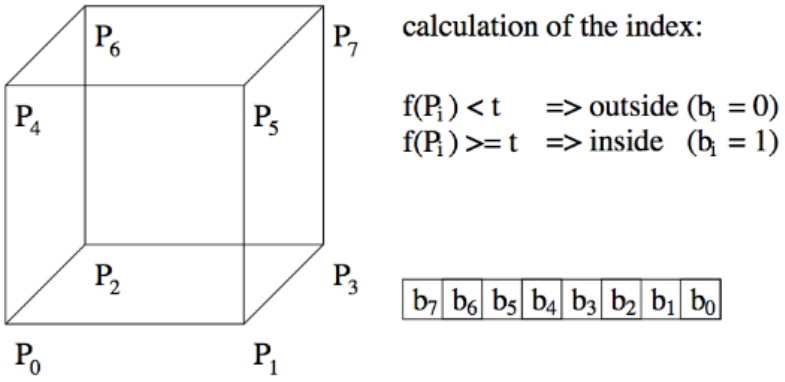


Figure 18: Calculation of the cube index.

possible vertex configurations. The configuration corresponds to an 8-bit word, one bit for each vertex. Bits corresponding to vertices below the isosurface level are set to 1.

The cube configuration gives us the list of "active" edges on the cube. The vertices of the triangles generated lie on these edges. Given the edges indexes, another lookup table is used to find the two cube vertices that this edge is between. So, there will be two cube vertices for each "active" edge.

The index of these two cube vertices is then used with another lookup table to get the local 3D coordinates of the cube vertices.

The last step consists of interpolating between those found vertices to estimate where along the edge the final vertex is, and this is done to give a smoother look. Interpolation is done considering the isolevel value of each vertex. Interpolation is done with the following formula:

$$\overrightarrow{v_t} = \vec{a} + \frac{(\mathrm{k} - v_1) * (\overrightarrow{b} - \overrightarrow{a})}{(v_2 - v_1)}$$

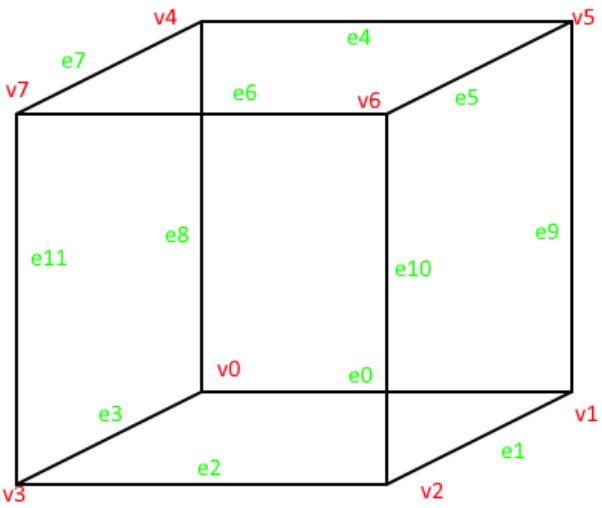


Figure 19: Numbering of vertices and edges on a single cube.

Where $\overrightarrow{v_t}$ is the final triangle vertex, $\vec{a}$ and $\vec{b}$ are the 3d coordinates of the cube vertices to interpolate, v_1 and v_2 are the isolevel values of the cube vertices, and k is the considered isosurface level. Notice that, since v_2 corresponds to the second cube vertex of the "active" edge, its isolevel value is always greater than v_1.

3.2.2. Example

For example, if vertex 0 has a value of 0, and all other vertices have a value of 1.0, given that the isosurface level is 0.5, we can conclude that since vertex 0 is the only vertex below the threshold, so we want to "hide" this vertex by creating a triangle in front of it by connecting edges 0, 3 and 8.

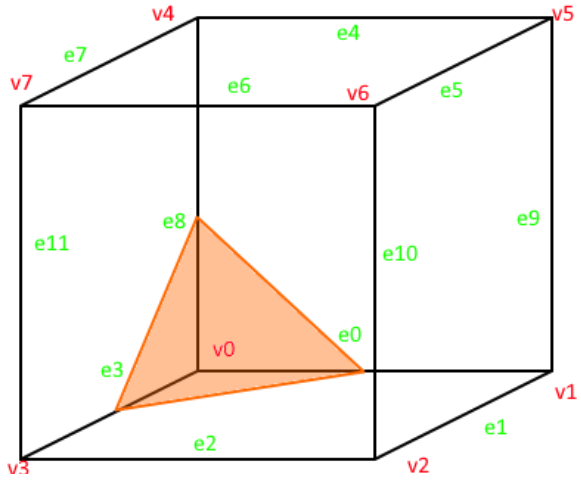


Figure 20: Example of a triangulated face with the marching cubes algorithm.

The cube configuration index is built by setting the least significant bit of the 8-bit word to 1. So, the cube configuration index we get is 1. From the first lookup table at this index, we get these edge indexes: {0, 8, 3}. This means that the vertices of the triangle lie on e0, e8, and e3.

From the second lookup table at these indexes, we get the following edge connections: {0, 1}, {0, 4}, {3, 0}. Indeed, v1 is the other vertex of e0, v4 of e8, and v3 of e3.

With these vertex indexes, we get the local cube coordinates of the vertices, which are then interpolated with the previous formula to get the final triangle vertices.

3.2.3. Implementation

The implementation of this algorithm can be found in the *Marching_Cubes* scene in the Unity project. The *AustinMan* game object is set with an initial empty MeshFilter a standard MeshRenderer, and a script component called *Marching_Cubes*.

As in the previous scene, this script initializes the three-dimensional voxel matrix with the grayscale pixel values of the AustinMan dataset images, this time with floating point values from 0 to 1. Voxel values corresponding to the air surrounding the body (all the black color in the dataset images) are set to 1, while all the voxel values corresponding to body parts are set to 0. The Isolevel value is set to 1. In this way, only the outer skin of the body will be generated by the algorithm.

This smoother mesh for the outer skin will be subsequently used in the ultrasound simulation as a “container” for the “internal” texture.

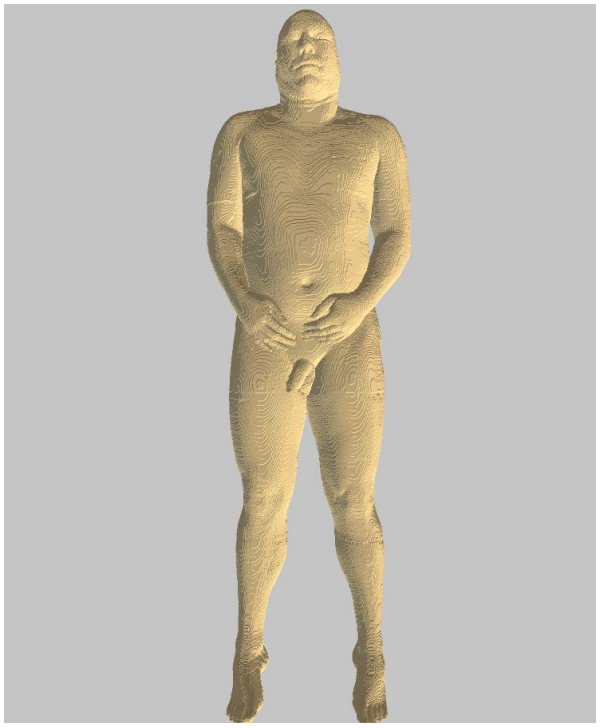


Figure 21: The mesh generated with the marching cubes algorithm.

4 Ultrasound simulation with Deformable Mesh Model

This chapter describes all the techniques used to make the simulation. First, the realization of the ultrasound simulation with 3D texture, the techniques experimented to make the mesh model deformable and to remap the texture coordinates to follow the deformation, are presented. The argument will continue with the details about the mesh model rigging and the setup of the Unity's scenes.

4.1. 3D Texture Shader

The possibility of being able to perform ultrasound scans in distinct parts of the patient's body is one of the objectives of this project.

A 3D texture is a bitmap image that contains information in three dimensions rather than the standard two. 3D textures are commonly used to simulate volumetric effects such as fog or smoke or to approximate a volumetric 3D mesh. In this work, a 3D texture is made by building a three-dimensional matrix from the dataset images. Each horizontal slice of the matrix corresponds to a slice image in the dataset.

3D texture can be rendered just like normal 2D textures, with the same types of filtering and interpolations to calculate final pixel values; the only difference is, obviously, the presence of one extra dimension for texture coordinates: there will be UVW coordinates. 3D texture can be applied to any mesh.

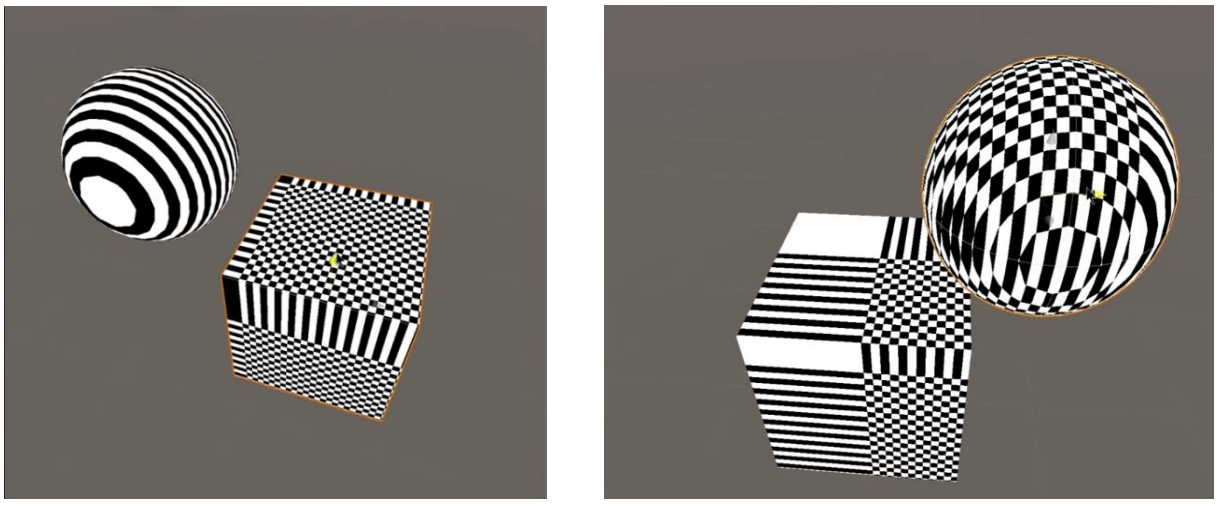


Figure 22: Example of a three-dimensional checkerboard texture applied to a cube and to a sphere.

4.1.1. UVW Mapping

A component necessary to assign a 3D texture to an object is a Shader. A Shader is a set of algorithms which describe how contiguous polygons and images must be processed until they are displayed on the screen.

The two main components of which it is composed are the Vertex Shader and the Fragment Shader. In the Vertex Shader the position of the object is transferred from object space to the camera's clip space in homogeneous coordinates. The most important part takes place in the Fragment Shader, where the UVW values are used as coordinates to indicate a section of the 3D texture from which extract the color to be assigned to the material.

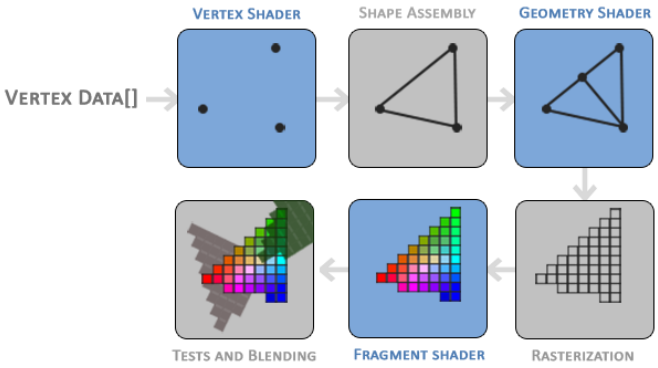


Figure 23: A basic shader pipeline

UVs are two-dimensional texture coordinates that are associated with each vertex of a mesh. They provide the link between a surface mesh and how an image texture gets applied onto that surface, like marker points that control which pixels on the texture corresponds to which vertex on the 3D mesh. As a convention, the U and V values vary between 0 and 1 along the horizontal and vertical axes of the texture.

This concept can also be extended with three-dimensional textures and is called UVW mapping. In this case there is a W value that varies between 0 and 1 along the depth axis of the texture. Inside the shader are passed the position and UVW values of the object the shader is associated with.

The idea is to exploit the possibility to use the position of the vertex of the invisible plane of the probe as UVW coordinates during the sampling phase of the 3D texture.

4.1.2. Ultrasound Probe and Screen

In this work, to simulate the ultrasound probe, a plane is used. This plane is set in the world origin. Every time the plane position or rotation changes, the UVW coordinates of each vertex of the plane are set to the world position coordinates of the vertex itself. By moving the plane in the space, we obtain a static mapping of the 3D texture.

The screen is the same plane used for the probe, the same UVW coordinates are applied to vertices of the screen.

The plane mesh is realized in Blender, by creating a simple plane mesh and subdividing it until reaches an acceptable "resolution". In this way, the plane is made by a higher number of vertices, not just 4 as in standard plane meshes. The idea is to exploit this higher "resolution" of the plane to have the possibility to set cluster of vertices mapped to a specific part of the 3D texture dynamically.

This concept is essential to realize the UVW mapping with the deformable mesh model. Differently from the static case, different parts of the 3D texture need to be rendered on the plane, and these parts in general are no longer aligned (in the UVW space), due to the possibility of deforming the model rig. This mapping would not be possible if the plane had only 4 vertices.

4.1.3. Vertex Shader Exploiting

In this work, the vertex shader step is exploited to make use of the rendering pipeline in a clever way. Each of the plane vertices is assigned with specific UVW coordinates.

Every fragment within these vertices will be automatically sampled and interpolated by Unity's built in rendering pipeline. In this way, there is no need to implement specific interpolation algorithms and filtering of the textures.

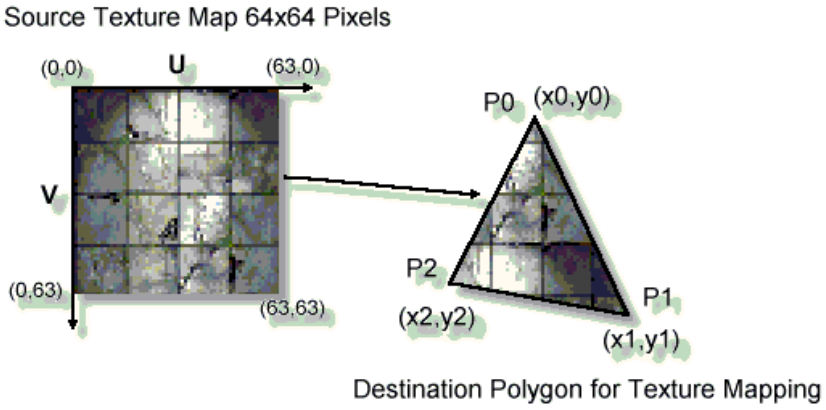


Figure 24: Texture interpolation for triangle vertices

4.2. Mesh Model Deformation

One of the main points of this work is to make the human model deformable, (just rotation and translation of body parts), with the mesh carrying the "internal" texture along the way. To do this, two approaches are presented.

Collision detection is the computational problem of detecting the intersection of two or more objects. Collision detection is a classic issue of computational geometry and

has applications in various computing fields. In Unity, collision detection is realized with components called Collider. Colliders can be used for collision detection only if their shape is convex.

4.2.1. Sphere colliders

The first approach consists in manually placing spheres along arms and legs. These spheres are attached to the model rig. The initial position of these spheres is stored at the start. The idea is that, whenever the model rig is deformed, each point position of the cutting plane is compared with all the spheres: if the point is inside one of them, the inverse of the sphere rotation and translation is applied to the UVW coordinates of that vertex. If the involved sphere is still in the original position, there is no rotation and translation, and the result is the same of the "static" version.

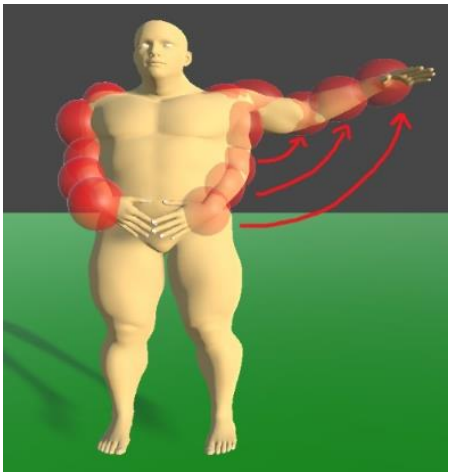


Figure 25: The approach with Sphere Colliders

This method works, but results applied to this precise human model is not satisfying, due to the nonstandard base position of the mesh (so also of the 3D texture), the initial overlapping of some body parts, and the high inaccuracy of the spheres to approximate body parts. The approach, however, is working.

4.2.2. Mesh collider

To improve the accuracy of collision detection with colliders, the idea is to exploit the human model geometry instead of the spheres to make more accurate colliders. However, Unity's built-in Mesh Collider component can only create convex colliders for the whole mesh, so this is not useful.

The final approach consists in automatically generating small convex mesh colliders for each part of the body to better approximate its bounding. Exploiting the bone weights of the human model rig, the model is subdivided into sub-meshes by considering which vertex is attached to which bone. These sub-meshes colliders are regenerated every time the model rig is deformed. As in the previous method, the initial position and orientation of each collider is stored at the start to subsequently apply the inverse transform to plane points.

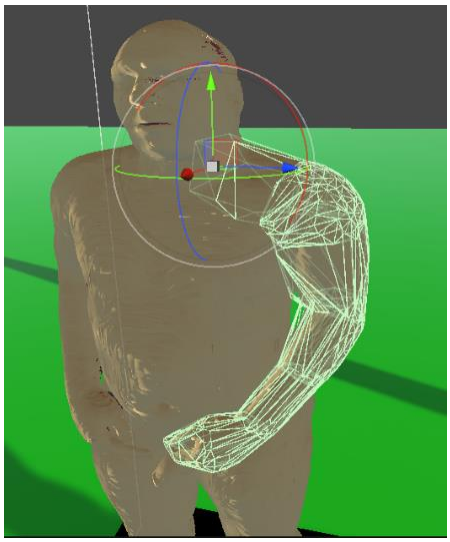


Figure 27: Colliders of the left arm generated from the bone weights.

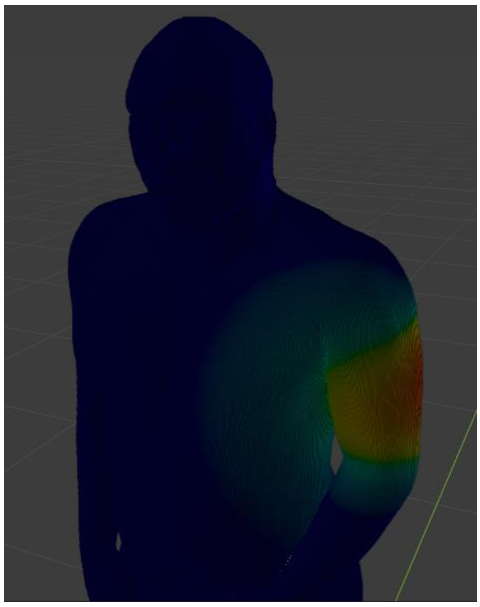


Figure 26: An example of bone weighting.

4.2.3. Model Rigging

Rigging is a technique used in skeletal animation for representing a 3D character model using a series of interconnected digital bones. Specifically, rigging refers to the process of creating the bone structure of a 3D model. This bone structure is used to manipulate the 3D model like a puppet for animation.

After a 3D model has been created, a series of bones is constructed representing the skeletal structure. For instance, in a character there may be a group of back bones, a spine, and head bones. The vertices of the mesh are associated with bones using the so called Bone Weights. Bone weights can be manually set and sometimes automatically generated with 3D editing software. Bones can be transformed, meaning their position, rotation, and scale can be changed. By recording these aspects of the bones along a timeline (using a process called keyframing) animations can be recorded.

The human model mesh generated by the Marching Cubes algorithm was manually rigged using Blender. This mesh wasn't in the usual T-Pose from which animators usually rig and deform meshes but was in the same pose of the human of the AustinMan dataset. This nonstandard pose led to some difficulties.

Figure 28: The model rigged in the base pose.

First, some geometry between the arms and the body was overlapping, causing the geometry to break when the pose of the arms was changed. This problem has been solved by manually deleting all the unwanted edges.

Second, it was difficult to infer the position of the bones given the base pose of the previously generated model. However, after some experiments, you have chosen to insert many bones to optimize the collider generation. More and smaller colliders mean more accuracy for the model deformation.

4.3. Breath simulation

To simulate the breathing movement of the body, the chest bone is animated. For a period of about 4 seconds, its scale increases, and decreases. Applying scale to a bone result in scaling all the vertices assigned to that bone.

By applying scale deformation to a bone, the same scale is applied also to all the child bones, so also the head and the arms are enlarged. To prevent this, the same animation,

but with inverse scale factor, is applied to all the child bones that doesn't need to be enlarged.

Colliders are automatically recalculated in each frame. The local scale of each bone/collider is also used to calculate the inverse transformation to be applied to plane points.

4.4. Scene Setup

The developed ultrasound simulator consists of two Unity's scenes.

In the first one the mesh model is not deformable, but layers of the body can be activated or deactivated. The probe plane is also bigger, it simulates the cutting effect of the MRI.

In the second one, the probe plane is smaller to resemble ultrasound echography system. A cone-shaped mask is also applied to the screen. Here the model mesh can be deformed and set in a different pose by clicking on body parts and using the gizmos to rotate them.

Both the scenes are setup equally, so only the details about the second one, most important, are described. In the scene there the following objects:

- Main camera
- Screen
- Probe Plane
- Texture Grid
- Human model

The screen plane and the probe plane's mesh filter are set on the same mesh. The plane object contains a script to control its movement.

The texture grid object contains the script checking collision between the plane and all the colliders of the body. At the startup, it loads the 3D texture from the dataset images and stores the initial position and rotation of every collider. In each frame update, it loops every plane point, check the intersection with the colliders, and sets UVW coordinates of plane points accordingly.

The human model object contains the model mesh and its rig. It also contains the script responsible for the generation of all the colliders by considering sub-meshes based on the bone weights.

5 Conclusions

In this final chapter, the proposed solution is compared to the previous one and the critical points are analyzed. Then, possible improvements are discussed along with possible topics for further research.

5.1. The developed application

The proposed solution consists of an application where the user can freely move in the scene and interact with the environment. The body mesh generated with Marching Cubes algorithm from the AustinMan dataset is placed in the center, and the screen for visualizing the ultrasound image is placed behind. The produced imaged is colorized with a pre-defined color for each body layer. This is because the grayscale version of the image can create confusion if the user assumes that dark areas are the densest, which is not the case since the dataset color encoding didn't follow this assumption. The probe plane position and orientation can be set by the user. Layers of the body, such as skin, muscles, bones, organs, can be set to be visible or not and be closely inspected. Body parts can be selected to set the orientation of the corresponding bone as the user wishes. There is also the possibility to simulate the breathing movement of the body, and its effect can be seen on the screen.

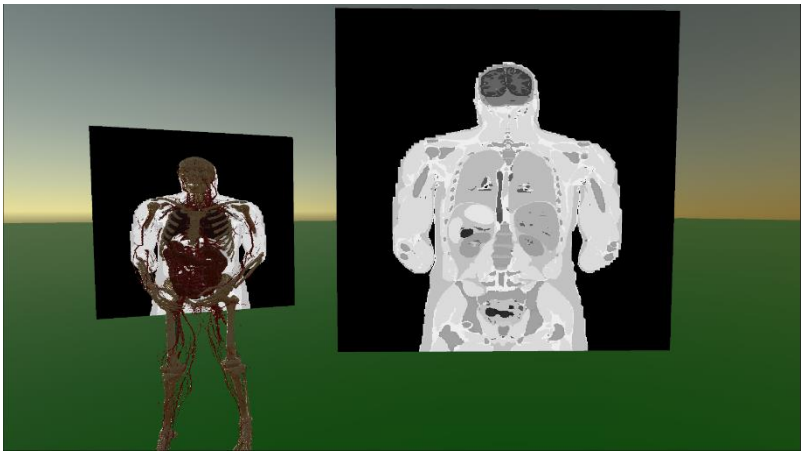


Figure 29: Screenshot of the developed application.

5.2. Comparison

In comparison with the previous work, there are significant improvements.

The first is about the model mesh geometry. In the previous work, a third-party model was manually set in position to be in some way aligned with the 3D texture. This

model, however, could not fit fully and exactly the texture because its shape didn't correspond to it. Its function was only to be like a placeholder.

In this work, the model mesh shape used to represent the human body corresponds perfectly with the shape of the 3D texture, because it is generated from it. This is also essential to make the collider subdivision, that is needed to map points when the model is deformed.

Another improvement consists, indeed, in the possibility to change the body position to see what happens to the internal texture when the body is deformed. This feature was not present in the previous work.

Last, the proposed solution contains a full body scan, while in the previous work only some parts were available. Also, there is no more need for model registration: the mesh model is already placed in position and aligned with the 3D texture. By simply placing the probe in the desired position we get a scan of the corresponding body part. In the previous work, to do this, a specific configuration scene called "Developer Mode" was set to align the body mesh model with the desired part of the texture.

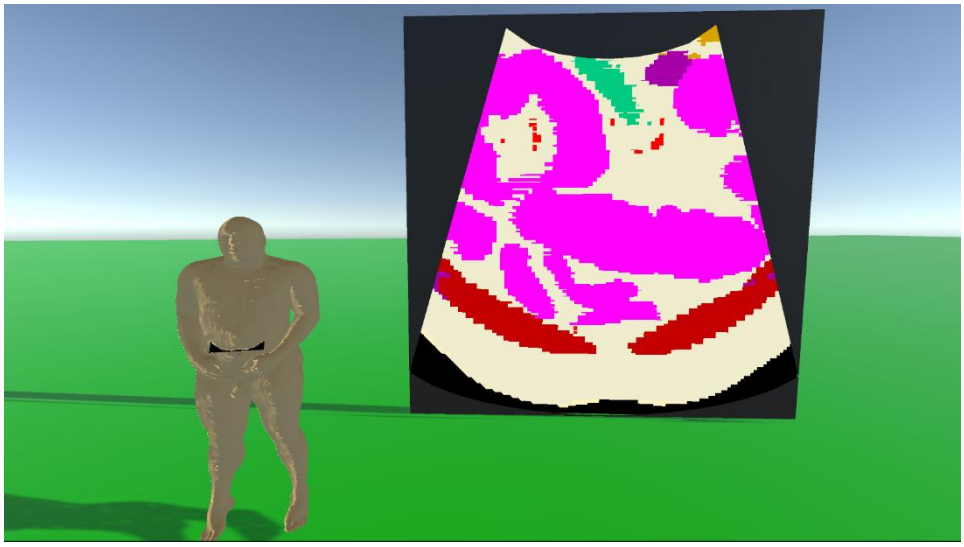


Figure 30: Ultrasound Ecography application screenshot

5.3. Critical points

Despite the proposed solution is working well, there are still some criticalities about some cases in which the approach is not working correctly, showing a distorted or segmented images on the screen.

The most important critical point is about the texture sampling derived from the model mesh deformation. When the body is in the standard pose, the 3D texture is sampled perfectly. However, when the body pose is changed, there could be some regions in which the 3D texture is not sampled correctly. These critical regions arise in body parts

corresponding to colliders boundaries. Colliders are automatically generated by considering the bone weights. Bone weights are also generated automatically, but the weighting strongly depends on the model rig, which is in this case done manually. Increasing the number of bones of the rig helped to reduce the size of these problematic regions, but in some cases, despite being smaller, these regions increased in number.

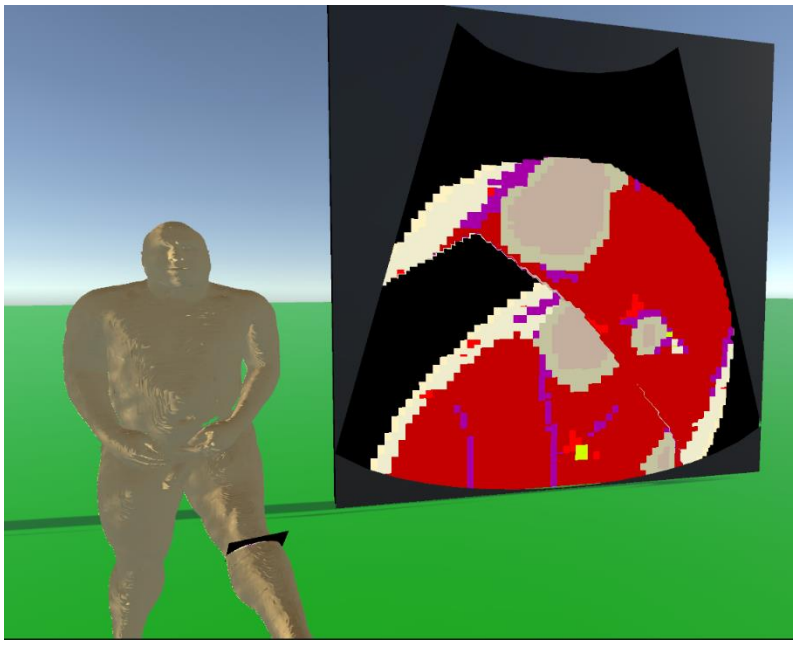


Figure 31: An example of wrong re-mapping of the texture after model deformation

Another important problem that arose during the development of this work was the illness about the position of the human body in the dataset images, which came from a real MRI analysis. This led to difficulties during the rigging of the model because of overlapping geometry between hands and body, and because automatic bone-weight tools work much better when the base pose of the model is the T-pose. The previous criticality about collider boundaries could depend on this also.

5.4. Further Reasearch

This work can be enhanced and improved in more ways.

First, a better subdivision of the mesh, rig, and colliders could be done to improve the inaccuracy encountered when scanning near colliders boundaries.

Also, a faster version of the proposed algorithms could be implemented. Now, the model mesh deformation and calculation of the UVW coordinates is entirely made with the CPU, with the GPU working only on fragment color interpolation. Research could be made to find if it is possible to exploit the GPU to directly apply the model deformation to every fragment UVW coordinates, exploiting the fragment shader programmability.

5.4.1. Ray-Casting

This work led to an Interpolative Model-Based ultrasound simulation application. It would be interesting to apply the model deformation technique with the Direct Volume Rendering (thus, Generative approach), already introduced Ray-Casting technique. Rays could be cast in the common way, passing through the image plane. The same mesh models generated from the AustinMan dataset can be used for collision detection and model deformation. When a ray intersects the mesh, the ray origin can be recalculated as if it was intersecting the same collider but in the base pose, to simulate the ray pass through the correct portion of the volume, when the model pose is deformed.

Bibliography

[1] R. YAGEL and S. H. I. Z., "Accelerating volume animation by space-leaping", Visualization, 1993.

[2] G. Wyvill and C. McPheeters, Wyvill B., "Datastructures for soft objects", Visual Computer1986; 2(4): 227–34.

[3] J. K. Udupa, "Interactive segmentation andboundary surface formation for 3D digital images," in *Proceedings Computer Graphics andImage Processing*, 1982.

[4] S. S. Trivedi, G. T. Herman and K. J. Udupa, "Segmentation into three classes using gradients", *In Proceedings IEEE Transactions on MedicalImaging*, 1986.

[5] T. Totsuka and M. Levoy, "Frequency domain volume rendering," in *Computer Graphics, Proceed-ings SIGGRAPH '93*, pp. 271–278,.

[6] L. Szirmay-Kalos, "Theory of Three Dimensional Computer Graphics", Akademia Kiado,Budapest, 1995.

[7] B. T. PHONG, "Illumination for Computer Generated Pictures," *Communications of theACM*, vol. 18, p. 311–317, 1975.

[8] G. NIELSON and J. TVEDT, "Comparing methods of interpolation for scattered volumetric data," in *State of the Art in Computer Graphics*, Aspects of Visualization, 1994.

[9] N. MAX., "Optical models for direct volume rendering", IEEE Transactions on Visualizationand Computer Graphics, 1995.

[10] M. D. MCCOOL, A. N. G. J. and A. AHMAD, "*Homomorphic Factorization of BRDFs forHigh-Performance Rendering"*, SIGGRAPH, 2001.

[11] J. W. Massey and A. E. Yilmaz, "AustinMan and AustinWoman: High-fidelity, anatomical voxel models developed from the VHP color images," in *Proc. 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (IEEE EMBC*, Orlando, 2016.

[12] T. Malzbender, "*Fourier volume rendering." ACM Transactions on Graphics*, 1993.

[13] W. E. Lorensen, Harvey E.Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," in *SIG '87*.

[14] L. Lippert and M. H. Gross, "Fast wavelet basedvolume rendering by accumulation of transparent texture maps," in *Computer Graphics Forum, Proceedings EUROGRAPHICS '95*, 1995.

[15] M. LEVOY, "Display of surfaces from volume data", IEEE Computer Graphics andApplications, 1988.

[16] A. Karamalis, W. Wein and N. Navab, "Fast ultrasound image simulation using the westervelt equation," *Med. Image Computing Computer-Assist. Intervent,* vol. 13, pp. 243–250, January 2010.

[17] J. Jensen, "Ultrasound imaging and its modeling," in *Imaging of Complex Media With Acoustic and Seismic*, New York, Springer, 2000, p. 1–38.

[18] M. J. HARRIS and A. LASTRA, *Real-time cloud rendering, Eurographics,* 2001.

[19] H. GOURAUD, "Continuous shading of curved surfaces," *IEEE Transactions on Computers,C-20,* p. 623–629, 1971.

[20] O. Goksel and S. Salcudean, "B-mode ultrasound image simulation indeformable 3-D medium," *IEEE Trans. Med. Imag,* vol. 28, pp. 1657–1669, November 2009.

[21] J. DANSKIN and P. HANRAHAN, "Fast algorithms for volume raytracing," *Workshop onVolume Visualization,* 1992.

[22] J. F. BLINN and M. E. NEWELL, *Texture and reflection in computer generated images,Communications of the ACM,* 1976, p. 542–547.

[23] E. Artzy, G. Frieder and G. T. Herman, "The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm," in *Proceedings Computer Graphics andImage Processing*, 1981.

[24] D. Aiger and D. Cohen-Or, "Real-time ultrasound imaging simulation," *Real-Time Imag,* vol. 4, pp. 263–274, August 1998.

[25] Sascha Bettinghausen, "Real-Time GPU-Based Ultrasound Simulation Using Deformable Mesh ModelsBenny Bürger", *IEEE TRANSACTIONS ON MEDICAL IMAGING,* vol. 32, March 2013.

[26] M. MEI ̈SNER, J. HUANG, D. BARTZ, K. MUELLER, and R. CRAWFIS, "A PracticalEvaluation of Popular Volume Rendering Algorithms", Volume Visualization, 2000.

[27] D. Zucca, "Low-cost Virtual Reality Simulation of Medical Ultrasound," Politecnico di Milano, 2022.

List of Figures

Acknowledgments

Here you might want to acknowledge someone.