UNIVERSITÄT PASSAU

Lehrstuhl für Mathematik mit Schwerpunkt Digitale Bildverarbeitung

Master Thesis

An analysis of the OSPRay Tracing framework on Rendering Techniques

Jobin Jothi Prakash

1. PRÜFER	2. PRÜFER
Prof. Dr. Tomas Sauer	Prof. Dr. Michael Granitzer

27/05/2021

Supervisor Contacts

Prof. Dr. Tomas Sauer

Lehrstuhl für Mathematik mit Schwerpunkt Digitale Bildverarbeitung

Universität Passau

E-Mail: Tomas.Sauer@uni-passau.de

Prof. Dr. Michael Granitzer

Lehrstuhl für Data Science

Universität Passau

E-Mail: Michael.Granitzer@uni-passau.de

Contents

Abstract

Visualization is considered an integral part of image processing techniques. As new visualization pipelines are introduced, the process becomes more complex for any new user to come to terms with the new updates.

According to [1], delivering strategies generally dependent on rasterization experience the ill effects of execution and quality limits, especially in HPC conditions without committed delivering equipment. As there has to be much more sacrifice in performance, new rendering technologies have been evolved in the current trend.

Although there have been various literature regarding the new rendering technologies, which method to employ for what kind of data has been less analyzed. This research presents the ray tracing method, which has better performance than the traditional rasterization technique and provides a high-quality solution for typical visualization frameworks.

The research is made possible by an open-source tool, Intel's OSPRay tracer, a CPU ray tracing framework. This is one of the rendering techniques, which produces high-quality and high-fidelity images by tracing the path of the light rays.

After evaluating the provided data, we classify some exciting results based on specific parameters and understand the impacts of the OSPRay tracer in the rendering technique.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my thesis supervisor, Prof. Dr. Tomas Sauer, for the continuous support of my research, for his patience, enthusiasm, and immense knowledge. His guidance helped me thoroughly during the time of my research and also in writing this Thesis.

Besides my supervisor, I would like to acknowledge all the faculty, staff members of the chair "Mathematik mit Schwerpunkt Digitale Bildverarbeitung" for their encouragement and insightful comments.

I also Thank my friends Ashish, Melbin, and Raja for all the sleepless nights and all the fun we had for the past three years and for helping me by providing their personal PCs to complete my Thesis.

My sincere thanks to the University of Passau and the FIM department for all the privileges and opportunities.

Last but not least, I would like to thank my family: my parents, for having the trust in me in the first place and supporting me throughout my life.

Thank You all

Listings

List of Figures

List of Tables

List of Abbreviations

OSPRay	Open, Scalable, and Portable Ray
HPC	High Performance Computing
API	Application Programming Interface
GPU	Graphics Processing Unit
AMR	Adaptive Mesh Refinement
MLRTA	Multi level ray tracing algorithm
VTK	Visualization Tool Kit
SIMD	Single instruction, multiple data
IDE	Integrated Development Environment

Chapter 1: Introduction & Motivation

1.1 Context

Visualization in High-Performance Computing (HPC) is generally carried out using Rendering techniques. Rendering images are considered an integral part of the image processing technique, making the images visibly better. That is, it displays three-dimensional objects on a two-dimensional surface. Some new scientific insights are gained through several visual methods during the visualization process. The use of visualizations has become crucial in the interpretation of generated data. Various rendering algorithms are usually applied to accomplish the process. Traditional visualization rendering entails rendering data represented as geometry or volumes and typically relies on industry-standard graphics APIs (E.g.OpenGL)with hardware acceleration to enable interactive rendering. The traditional or the primary algorithm method is the "Rasterization" technique which is the imperative drawing of a scene content. This CPU based rendering method allows running the entire visualization pipeline on a CPU even in the absence of a GPU. Here, in this research, we are concentrating on the more advanced and higher-end rendering method called the "Ray Tracing" technique. This technique is generally capable of creating photorealistic images from 3D scenes. Here, the light transport is generally a physically-based simulation. The working of how the light transportation takes place can be seen in Fig 1.1. Even though the Rasterization technique produces a good result, it does not give a clear picture as that of the Ray Tracing technique, which has ample details than that of the Rasterization.

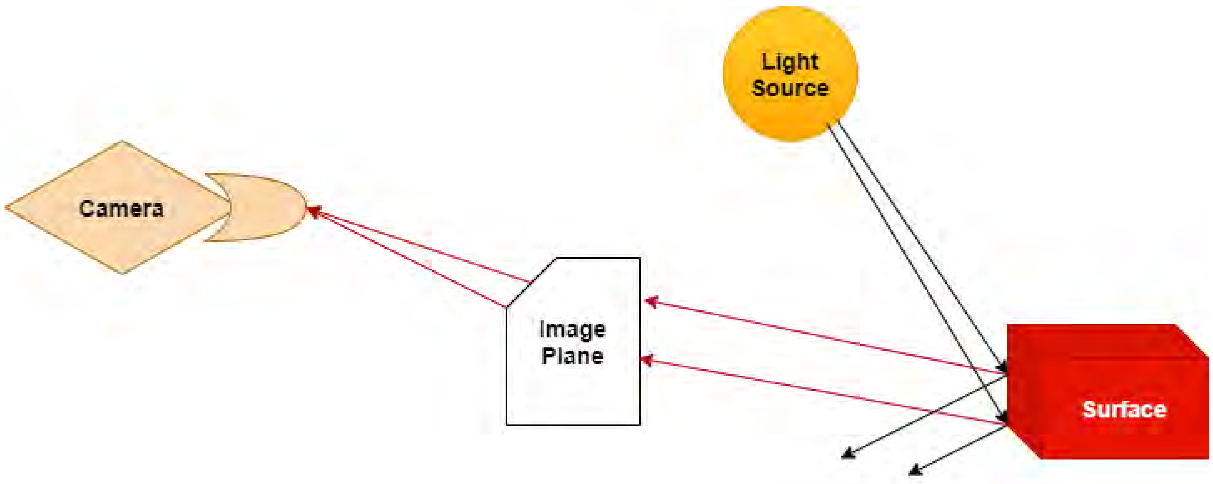

Figure 1.1: Work model of light transportation in ray tracing

1.2 Motivation

Ray Tracing is one of the State-of-the-art technique, in which it calculates the path of every ray of light and follows it throughout the scene until it reaches the Camera. So it is quite obvious that it could create much accurate reflections and refractions. In general, ray tracing works by creating a ray for each pixels that is displayed on the screen. Then the path of the each ray is traced from the camera, back through the scene to the original light source. Along these lines, essentially, ray tracing can create reasonable pictures by mimicking light rays. Few characteristics of ray tracing are:

- ***Automatic, simple and intuitive:*** Straightforward and execute and conveys sensible outcomes.
- ***Robust and efficient:*** Able to work in both parallel and distributed environments.
- ***Inherited light rays:*** Used in optics and in the designing of lens.

In particular, "OSPRay" is another quick open-source ray tracing framework for Intel CPUs for strong surfaces and volumes that replaces conventional rasterizers. This OSPRay tracing framework enables high-quality, efficient CPU-based solutions that are already integrated into many common visualization applications which is literally a part of the Intel oneAPI Rendering Toolkit.

Simultaneously, it also provides some extra details of the rendering process where several features of the Intel OSPRay are discussed as follows:

Interactive CPU Rendering: [1] Scientific Visualization applications rely on OSPRay's scalable CPU rendering capabilities. Also some advanced shading effects such as shadows, transparency and ambient occlusion can be simulated interactively to enrich massive data sets.

Volume Rendering: [1] In addition to high-fidelity, interactive direct volume rendering, the OSPRay platform makes use of a number of cutting-edge visualization features.

Global Illumination: [1] OSPRay is furnished with a path tracer prepared to do interactive rendering delivering photorealistic illumination utilizing solid-based types of equipment.

MPI Distributed: [1] Generally it decreases the render time and increase the total scene size by running on a large scale distributed-memory systems with high-performance MPI (Message Passing Interface).

Industry Adoption: [1] The improvement unit of OSPRay is straightforwardly remembered for ParaView 5.x. Variants of OSPRay for VisIt, VMD, and other famous instruments are accessible for nothing.

1.3 Various light sources in ray tracing

Several materials and light sources could be applied in the ray tracing techniques where it produces few extra effects in the resulting images. For instance, a normal sphere or a cube could be visualized with effects in the ray tracing process.

The main feature of OSPRay tracing is that it can handle all polygonal and non polygonal materials, advanced shading effects and also volume rendering. Also OSPRay gives a practical option for current and impending supercomputers without GPUs. A significant part of OSPRay's volumetric material is that it can extract any volumetric type that can be examined utilizing a 3D space.

[1]`https://www.ospray.org/`

1.3.1 Light transportation

Here, we will discuss how the light travels in the ray tracer. At this point, the thought behind ray tracing is to discover numerical answers for process the convergence of this ray with different sorts of materials: triangles, quadrics. As seen in Fig 1.1, ray tracing figures the shade of pixels by following how light would take if it somehow managed to go from the eye of the watcher through the virtual 3D scene.

Various light sources can generate shadows by tracing the rays of the light, which are called as **Ray-traced shadows**. Various light sources influence the ray tracer to generate shadows of different perspective in the resulting images and the light sources are namely **Directional**, **Point**, and **Rectangle**.

The **Directional light** [1] uses the angle of the sun to produce shadows in the resulting image, where the size of the sun is set in degrees. For instance, the angle of the sun is set to 60°. It controls the quantity of rays that utilizes per pixel. When the pixel quality is high, it can produce accurate shadows to a certain level. The denoise makes the shadows more clearer to any viewer.

The **Point light** [1] evaluates the point of light from a single point and produces ray-traced shadows. This also controls the quantity of rays reused per pixel, and the radius of the sphere is set by itself, where it is employed in evaluating the shadows. The area that emits light is increased, and also here, the denoise makes the shadows more visible by avoiding any available noise.

The **Rectangle light** [1] is added to the ray tracer where the resulting images has a much better shading effect which makes the images looks very photorealistic. The ray-traced shadows produced are more brighter and of low pixel breakage.

[1]`https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.1/manual/Ray-Traced-Shadows.html`

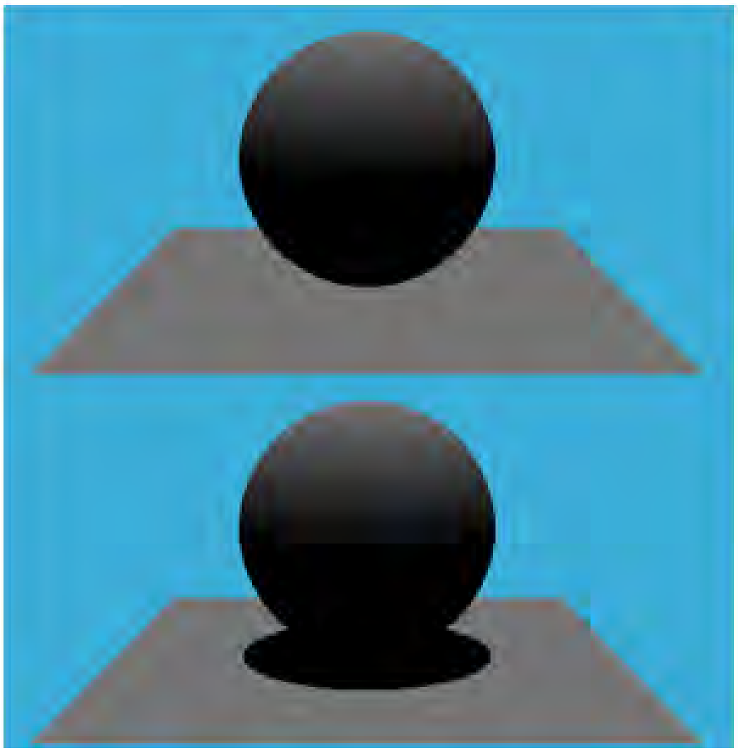

Figure 1.2: Ray-traced shadow in the resulting image [1]

In fig 1.2, the ray-traced shadow effect is clearly visible, where the image with shadow is much more realistic than the one without the shadow. Various light sources influenced in ray tracer produces a different clarity to the shadows in the resulting images.

1.4 Research Theme

The main goal of this research is to understand how the OSPRay works on any voxel image datasets and discuss their main parameters. This research can be framed as:

- **Working on the voxel image datasets to generate it into a 3D or 2D image and understand the features of OSPRay tracing and its impacts on rendering images.**

[1] `https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/light-and-shadows`

1.5 Architecture

The below figure 1.3 gives an overview of the process of vtkOSPRay in the research.

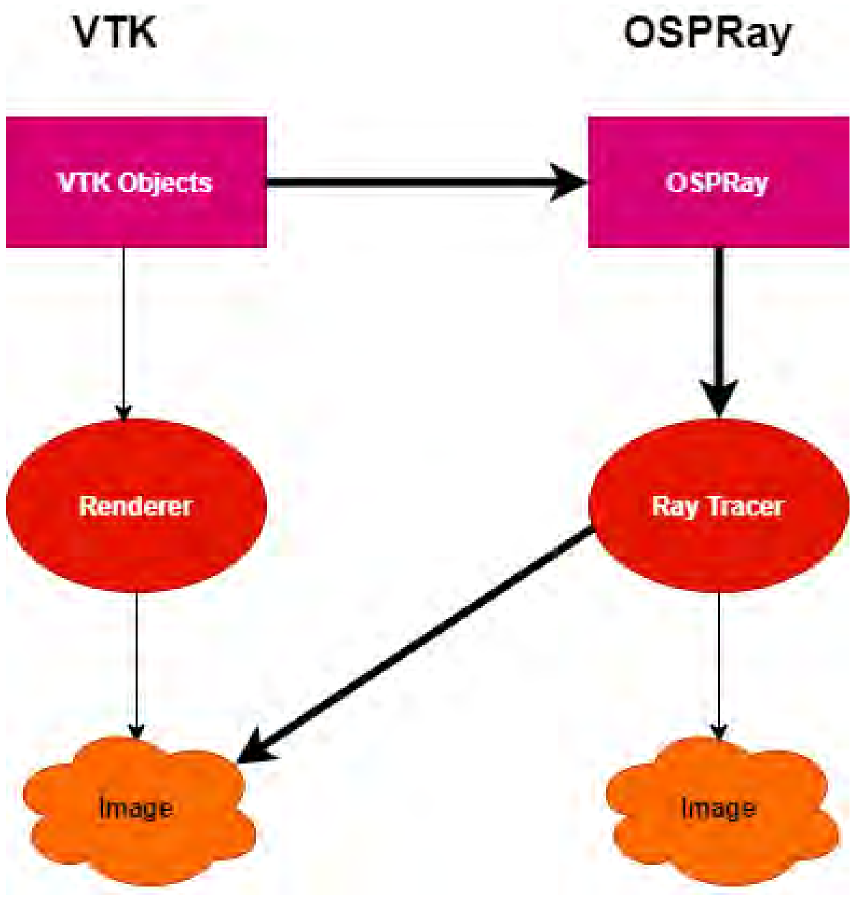

Figure 1.3: A visual outline of the VTK OSPRay workflow [2]

This research is coordinated as follows. Chapter 2 gives the subtleties on scarcely any accessible writing works identified with this exploration. Chapter 3 gives a short prologue to the related devices. In chapter 4, we investigate explicit objectives and difficulties looked by the OSPRay. Chapter 5 shows the assessment approach dealt with in this exploration. Chapter 6 sorts out the got results and examines them, and features the restrictions. Chapter 7 finishes up the exploration by featuring the outcomes.

Chapter 2: Literature review

Several related works similar to our research area have already been published and we discuss some of the recognized works in this chapter.

Advanced shading effects can enormously help fit as a fiddle of complex spatial information yet isn't inconsequentially upheld by conventional rasterization techniques. Ray tracing method is engaging in that it conceivably addresses this issue in a brought together way: it effectively scales to huge polygon checks and loans itself typically to non-polygonal geometry, volume information, and advanced shading. Wald et al. in [1] proposed the possibility of OSPRay, a ray tracing structure for excellent representation delivery. OSPRay will probably go past confirmation of-idea ray tracing frameworks for visualization and offer a total turn-key solution for existing production visualization programming bundles that can run proficiently on current equipment while offering particular gadget support for future equipment structures. It is involved a lightweight gadget-free API for perception situated ray tracing. Additionally, it also had a special CPU-oriented implementation of the API, making it possible for general-purpose CPU workstations to render graphics using varying SIMD widths. Specifically, they have shown OSPRay's reasonableness to drive certifiable visualization through mix into ParaView, VisIt, and VMD. Their execution upholds both polygonal and non-polygonal information, volume rendering, elite shading effects, and specifically, can utilize accessible memory. In any case, their outcomes demonstrate that ray tracing is now an intriguing expansion to programming and hardware rasterization, and they accept that OSPRay is a significant advance towards more pervasive utilization of ray tracing in visualization rendering [1].

Wang et al. demonstrated a volume renderer that empowers high-loyalty interactive visualization of enormous volumes on multi-core CPU designs in their work [3]. This specific renderer permits clients to have an information outline inside the space of seconds rather than the current CPU-based visualization systems, which may require minutes or hours. For enormous datasets, the IO inactivity caused by the information stacking measure is restrictive in the current volume renderer, regardless of whether all the information fits into memory. The long IO hanging time for the colossal dataset corrupts the client experience. In this work, they presented a solution for enormous scope volumes on multi-core CPU designs. Their solution permits clients to get an outline of the vast data in seconds rather than minutes or hours utilizing existing visualization structures. A bricktree module was built to facilitate large-scale data visualization on multi-core workstations, as an addition to the OSPRay tracing framework, which already contained various methods for visualizing scientific data. Considering how OSPRay has been incorporated into Paraview and Visit, this bricktree module is prepared to join universally valid visualization systems. Their solution is a considerable improvement contrasted with the current OSPRay volume renderer, which for the most part, requires minutes or then again hours to stack the information before rendering the primary edge. Propelled by numerous new renderers, they presented a various leveled information structure – a Bricktree – with a substantial expanding factor and generally low overhead [3].

Han et al. in [4] introduced an overall elite procedure for ray tracing generalized tube primitives. Using this technique, tube primitives with fixed and variable radii can be efficiently generated. Also, general acyclic graph structures with bifurcations can be constructed and can implement internal transparency correctly by removing the interior surface. This paper deals with a superior and high-devotion rendering of information addressed as 3D line primitives. Such line primitives are utilized to address information in the scope of logical areas, like fluid dynamics. Line primitives have been generally utilized in representation, and a few open-source applications exist for ray tracing them, with differing levels of help for bifurcations and changing radius. Their strategy outfits first-class rendering with low memory overhead for up to a large number of primitives. Besides, their strategy is sufficiently general to insert into any ray tracing system, like Nvidia Optix. Carrying out this method inside the OSPRay ray tracing system, they assess it as a robotized representation apparatus for fixed and shifting radius measures, pathlines, complex neuron constructions, and cerebrum tractography [4].

The idea of an Interactive ray tracing model was proposed by Bigler et al. in [5]. They introduced the software architectural model of Manta interactive ray tracer and used them in scientific visualization and engineering. Regardless of the enormous number of ray tracing programming bundles that exist, a large part of the customary plan shrewdness should be refreshed to oblige intelligence, high levels of parallelism, and current packet-oriented speed increase. They proposed a two-piece programming model to exploit the current and forthcoming equipment plan. The primary piece is a multi-strung adaptable equal pipeline. The second is an assortment of programming instruments and information structures, given ray bundles, for misusing parallelism and execution advancement from segment-based rendering code. Manta is a ray tracing framework applied to various designs and perception issues, from triangle network rendering to time-fluctuating multi-modular circle glyph and volume rendering. Manta's data structures are inspired by the design of modern microprocessors that can exploit at the instruction level. As Manta was initially executed on SGI Origin and Itanium2 based multi-processor frameworks, the Origin originated before the far-reaching selection of SIMD units, and on the Itanium2, these units didn't give a convincing improvement. Thus, Manta's execution encountered a considerable change when workstations and x86 multi-processor frameworks with SIMD units turned out to be broadly accessible. Ray packet information must be coordinated upward for SIMD directions while new accessors were added to oblige heritage horizontal code [5].

The idea of Texture-based volume rendering was proposed in [6] by Behrens et al. which is a strategy to envision volumetric information utilizing surface planning equipment effectively. The algorithm exploits quick frame buffer tasks present-day designs equipment offers; however, it doesn't rely upon any particular equipment. Though the calculation doesn't perform lighting estimations, the subsequent image has a shaded appearance, which is a further obvious prompt to spatial comprehension of the information and allows the images to show more sensibilities. All in all, they created and executed an algorithm that fuses shadows in a texture-based volume renderer to expand the authenticity of the subsequent pictures. Here, since the calculation of shadows could be applied to any volume rendering process, whether two- or three-dimensional, it makes the most sense with volume rendering algorithms based on texture mapping hardware [6].

Ahrens et al. [2] examined the assessment of how to envision petascale data. In this paper [2], they assessed the rendering execution of multi-core CPU and GPU-based processors. The information understanding cycle is made out of various exercises, including examination and insights, visualization, and rendering. In petascale platforms, visualization and rendering may be suitable for running on the supercomputer platform as opposed to a visualization cluster. For achieving high-performance rendering on multi-core processors, they used raytracing engines optimized for multi-core processors. They have interfaced these rendering motors with VTK and ParaView for testing under genuine conditions and getting ready for petascale visualization undertakings. It was much evident from the results that apparently, rendering software optimized for multi-core CPU processors results in improved performance.

Wang et al. in [7] proposed the Adaptive mesh refinement (AMR) method. However, despite the widespread adoption of these techniques in high-performance computing simulations, visualizing the output interactively and without artifacts or cracks has remained challenging. AMR procedures have been comprehensively received to tackle complex enormous scope simulation issues in high-performance computing. While visualizing AMR information, a primary advance is to perform fitting insertion of qualities to recreate the field. Their visualization approach is based on a novel remaking technique for inserting across level limits called Generalized Trilinear Interpolation (GTI). It guarantees the recreated field is smooth and empowers representations that more precisely address the fact than the current AMR remaking draws near. It was demonstrated that their approach results in artifact-free isosurface and volume rendering and that output images are of higher quality than existing methods [7].

DeMarle et al. presented distributed shared memory to image parallel ray tracing on clusters in their work [8]. Since any handling machine can require admittance to any piece of the scene, image parallel rendering has customarily been restricted to scenes that are adequately little to find a way into the memory of every node. Graphics and visualization experts ordinarily require the ability to deliver more considerable information than accessible memory and handling assets. To resolve this issue, parallel processing could be conducted in order to maximize the available memory and computing power. As part of their work, they develop software distributed shared memory layers that allow a cluster's memory to be utilized as needed. For the interactive rendering of multi-gigabyte datasets, this mechanism works well with gigabit ethernet connections [8].

The idea of some innovative approaches to ray tracing that significantly reduce the number of operations required can be significantly reduced while maintaining strict geometrical accuracy was proposed by Reshetov et al. in [9]. This makes constant ray tracing and Global Illumination (GI) alluring for execution on work area machines. In this paper, they center around the most major assignment in ray tracing, in particular, discovering the convergence of rays with a given calculation. The main focus of this paper is the MLRTA which is a robust approach to the high-performance ray tracing method. This is refined by giving a different section point into the worldwide speed increase structure (kd-tree) for each ray gathering. For comparing different algorithms, it is most appropriate to consider reducing the average number of operations per ray. It is a hierarchical process, that allows for geometric anti-aliasing that is optimally suited to the complexity of the geometry of a particular view direction. Suppose the cost of partially traversing the tree is amortized across all rays in a beam. In that case, it is possible to achieve time savings up to an order of magnitude and enable interactive graphics on an ordinary desktop computer. The outcome gave another appealing property of the MLRTA calculation that it gives a characteristic proportion of the mathematical complexity of explicit view headings [9].

Wald et al. in [10] proposed another method that included navigating frustum-limited bundles of coherent beams through a uniform framework to perform interactive ray tracing of moderate-sized enlivened scenes. The coherent grid traversal algorithm brings to grids the same performance components that have made KD-trees so successful, such as packets, SIMD extensions, and frustum traversals, while preserving trivial computation required for incremental grid steps. Packet tracing makes gatherings of spatially intelligent rays that are followed together through a kd tree, where all rays play out every crossing cycle in lock-step. This paper proposed another crossing plan for lattice-based acceleration architecture that considers navigating and converging bundles of coherent rays utilizing an MLRT-propelled frustum-traversal plan. Since the grid can be built efficiently on every frame, even for highly dynamic scenes, which are typically a challenge for interactive ray tracing systems, this performance is possible. This algorithm carefully considers moving the new benefits in quick ray tracing—specific, ray parcels, frustum testing, and SIMD expansions — to lattices. These methods had previously not been accessible [10].

The plan to figure anisotropic shading for direct volume rendering was introduced by Ament et al. in their work [11] to improve the impression of surface-like designs' direction and shape. Volumetric shading with nearby light models can give significant obvious prompts to improve the view of direction, curvature, and state of surface-like designs in volumetric informational indexes. Through analysis of its ambient region, they were able to determine a shading point's scale-aware anisotropy. Fundamentally, they planned an information-driven, continuous change from isotropic to unequivocally anisotropic volume shading. Further, their algorithm was not dependent on the transfer function, allowing them to compute each shading parameter once and store them in the data set. They have shown tentatively that their anisotropy assessment technique with a bit of radius is firmly identified with the vital arch directions. The result demonstrated that anisotropic rendering produces better perception than isotropic rendering because of the data-driven local illumination [11].

A study by Beyer et al. in [12] examined methods for interactive visualization of large volumes. They analyzed interactive visualization with gigabytes, terabytes, and petabytes of volume data that could run on GPUs. The pivotal property that empowers these techniques to scale to outrageous scale information is their yield affectability. They put forth both the computational and the visualization attempt relative to the measure of noticeable information on the screen. In addition to using parallel GPU processing power in conjunction with out-of-core methods and data streaming, another factor leading to interactive experiences is the proportionality of computational and visualization effort based on visible information. The parallel (distributed) visualization methodology based on clusters for this application is seen as an orthogonal set of techniques. They explore that the information won't ever found a way into GPU memory entirely. In this manner, it is pivotal to decide, store, and render the functioning arrangement of noticeable blocks in the current view effectively and precisely [12].

Chapter 3: Associated tools

3.1 Outline

In this chapter, we discuss the tools and technologies associated with the OSPRay and the important methods connected with them. First, we discuss about the associated visualization toolkit and then look into rendering toolkits.

3.2 vtkOSPRay

The visualization toolkit is a significant rendering tool that handles the initial parameters of the rendered images efficiently. It uses a hidden information construction to address and handle an assortment of data types. The **vtkOSPRay** [1] produces images with more advanced quality or high-resolution images without any pixel breakage. The visualizing protocol is made much easier by this. In this research, this particular library acts as the baseline for OSPRay plugins.

3.2.1 ParaView and VisIt

Visual Molecular Dynamics (VMD) is another well-known tool that is a widely used program for molecular visualization. Few other tools which fit the best for general scientific visualization are **ParaView** [2] and **VisIt** [3].

Both ParaView and VisIt are open source with many users and developed on the VTK. In this research, they make use of the indirect visualization for rendering using the available geometric data.

[1] http://tacc.github.io/vtkOSPRay/index.html
[2] https://www.paraview.org/overview/
[3] https://wci.llnl.gov/simulation/computer-codes/visit

3.3 Intel® oneAPI Rendering Toolkit

This toolkit comprises some handy ray tracing and rendering libraries which has the capability to produce some high-quality and high-performance visual experiences. The **Intel® oneAPI Rendering Toolkit** [4] improves the visual performance using the ray tracing technique with global illumination. Even though the data is humongous, it provides interactive and high-quality performance.

3.3.1 Intel® Embree

It is no longer uncommon for graphic card and processor manufacturers to offer platform-specific solutions for fast ray tracing. Intel released **Intel® Embree** [5] where ray tracing is being increasingly adopted for both offline and interactive rendering. In this exploration, the Intel® Embree version 3.12.0 or latest is required where the location of this directory is found using the variable:

```
embree_DIR
```

3.3.2 Intel® Open Volume Kernel Library

Intel®Open VKL [6] is an assortment of elite volume calculation parts created at Intel. Several functionalities such as volume traversal and sampling functionality that help improve the performance of volume rendering applications are included in Intel® Open VKL. In this research, the Intel® Open VKL version 0.12.0 or latest is required where the location of this directory is found using the variable:

```
openvkl_DIR
```

[4]https://software.intel.com/content/www/us/en/develop/tools/oneapi/rendering-toolkit.html#gs.1ktr4y

[5]https://www.embree.org/

[6]https://www.openvkl.org/

3.3.3 Intel® Open Image Denoise

Intel® Open Image Denoise [7] is an open-source library maintained for an array of high-performance, high-quality denoising algorithms for images rendered using ray tracing. The main idea behind this denoising library is that it enables rendering times to be substantially reduced in ray-tracing applications. To perform Intel Open Image Denoise, a CPU with SSE4.1 backing or Apple Silicon is required. In this research, the Intel® Open Image Denoise version 1.2.3 or latest is required where the location of this directory is found using the variable:

```
OpenImageDenoise_DIR
```

3.4 CMake

CMake [8] is an open source tool that helps in developing and testing software. In order to facilitate the development process, OSPRay applies a CMake Superbuild script, that can bring down OSPRay's dependencies and develop OSPRay on its own. This CMake Superbuild can be run using the following command:

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" [<OSPRAY\_SOURCE\_LOC>/scripts/
    superbuild]
cmake --build
```

Listing 3.1: CMake command

Table 3.1 below shows a few commands to be looked into, which is helpful to run the tool successfully. There are ample commands, but I have covered only the commands which are helpful for this research.

[7]https://www.openimagedenoise.org/
[8]https://cmake.org/

Commands	**Uses**
CMAKE_INSTALL_PREFIX	It will be the root registry where everything gets introduced
INSTALL_IN_SEPARATE_DIRECTORIES	It flips establishment of libraries in discrete or a similar catalog
BUILD_EMBREE_FROM_SOURCE	Set to OFF will compute a pre-assembled form of Embree
BUILD_OSPRAY_MODULE_MPI	Set to ON to fabricate OSPRay's MPI module for information repeated and appropriated equal rendering on numerous nodes

Table 3.1: Few CMake commands to be noted [13]

Chapter 4: Goals and challenges

4.1 Outline

This chapter outlines details on the certain goals and the corresponding challenges followed in the research and gives an explanation of the same.

4.2 Effective visualization on HPC systems

In recent days, there is a growing fact that several HPC systems do not have GPUs connected to them. Most HPC systems are CPU-based, which leads to the situation where the effective visualization in those systems becomes significantly faster rasterization. Although dedicated visualization resources often have GPUs, they have a large memory capacity and a powerful CPU. Even when renders are done on GPUs, most of the analysis is carried out on the CPU. Therefore, software rendering can run smoothly on systems with GPUs [1].
It is pretty evident that the systems with dedicated GPUs produce high-quality photorealistic images than the systems which do not have dedicated graphical units. Apart from that, another factor is the quality of images obtained after rendering.

4.3 Challenging rasterization-based visualization pipelines

In the current trend, rasterizing many triangles per second is easily achieved using GPUs. Present-day GPUs can rasterize billions of triangles each second yet accomplish top execution just under explicit conditions. Standard triangle rasterization could not inconsequentially render non-polygonal calculation utilized in like manner visualization modalities neither advanced shading effects [1].

4.4 Solution to tackle these challenges

Many techniques solve these challenges. One of the most promising technique is the ray tracing, where it can address these issues in a combined way. Several features make this program ideal for handling polygonal and non-polygonal surface data, as well as surface and volume data.

This program scales well to large amounts of data. Moreover, it is inseparable from cutting edge shading impacts, and furthermore it is feasible with both GPUs and CPUs. These benefits of ray tracing are by, and ample surely knew and progressively pervasive in visualization [1]. As OSPRay is a CPU based ray tracing framework for visualization, in this research we are primarily concentrating on the following factors by comparing with the different processors. The factors include:

- Computation time of ray tracing
- Parameters influencing the ray tracing
- Quality of images

These factors are some of the exciting things to be covered, which also determines the performance of each processor. They also help us to understand the comparison of processors.

Chapter 5: Evaluation techniques

5.1 Outline

This chapter explains a few of our novel methods considered in the evaluation and how the provided voxel datasets explain the features of OSPRay tracing. The dataset provided was some of the low-resolution voxel image datasets, which are the raw images. Also, we were provided with the dimensions of every image.

5.2 Dataset for evaluation

Data collection and the entire evaluation process were run on different processors, including the processors with dedicated graphical units and the processors without dedicated graphic cards. At first, we ran and executed the entire process on a system with an Intel(R) Core(TM) i5-4300U processor running at 1.90 GHz, using 4GB of RAM, and running Windows 10 Pro is of the 64-bit operating system. Then it has been compared with another high-end processor, which is of configuration AMD Ryzen 7 5800H with Radeon Graphics processor running at 3.20 GHz using RAM of 16GB. Now, we will discuss the previously mentioned factors in 4.4. But, at first, we will discuss the tools and libraries used in the evaluation process.

Voxel image datasets that were low in resolution were used in this research. The dataset format is simply a single block of memory of the dimensions provided in the header file (.txt file) and provided voxel images were simply of raw file format (.raw). In this research, we have used the NumPy library to read those raw image datasets. It is evident that running the datasets depends on the file size, and each image was of varying sizes with different dimensions.

5.3 Evaluation method

Jupyter Notebook, which is an open-source IDE, is used in the process where it creates an interactive python notebook, which are then shared with live codes. In this research, the Jupyter Notebook version 6.0.0 is used predominantly to perform the code to view the voxel raw data files. As mentioned, the dimensions of each raw image were provided in the format of **(width*height*depth)**.

The libraries which were helpful in running the process were:

- NumPy
- Matplotlib
- Scikit-image

NumPy library is useful in handling multi-dimensional arrays and in this research, the NumPy version 1.19.2 is employed. This is used as:

```
np.fromfile(filename, dtype='H', count=(width*height*depth)
```

Listing 5.1: Numpy command

through which the format of the voxel raw files can be easily readable. To obtain the 3D array, it has to be reshaped with the NumPy **reshape** function, which can be written as;

```
np.reshape(depth, height, width)
```

Listing 5.2: Numpy reshape command

Matplotlib library is used in viewing any desired slices of the images in any different angles, and using the function **imshow()**, we can plot an individual volume slice. Generally, this, in turn, allows us to kind of scroll through the volume along one axis. For instance, if we have to plot the Z value out of the X, Y, and Z, we have to plot it as **imshow(volume[z,:,:], ...)**. Also, we could view the voxel plot in 3D using the axes-specific voxel functions. In this research, Matplotlib version 3.3.4 is employed, which helps produce quality images.
Three functions of Matplotlib are used in this research. They are listed as below:

```
plt.imshow()
plt.colorbar()
plt.plot()
```

Listing 5.3: Matplotlib functions

where plt.imshow() plots the individual volume slice, plt.colorbar() plots the colorbar on the axes of the displayed images, and plt.plot() plots the desired 2D images.
Skimage library is the one that is useful in performing image processing techniques. In this research, it helps in resizing the obtained image, which is given as:

```
resized = resize(A, (IMG_DIM, IMG_DIM, IMG_DIM), mode='constant')
```

Listing 5.4: Skimage

where the obtained images can be resized as required. The image dimensions are the main factor in resizing the images, and the resized images depend on the given values.

Chapter 6: Results and discussion

In this chapter, we discuss the obtained OSPRay tracing images and describe them with the three main factors of this research, that is (1) The Computation time of ray tracing, (2) Parameters influencing the ray tracing, and (3) Quality of images. For this research, we were provided with a dataset Zip file which consisted of four raw images. Each image files were of different sizes, and the above factors also depend on the size of these images.

Here, we will discuss the results of obtained ray tracing image of every raw image one after another. First voxel data is the **Ford Fiesta spring** (5.07MB). In the below figure 6.1, we can see the 2D image of the spring from the top angle.

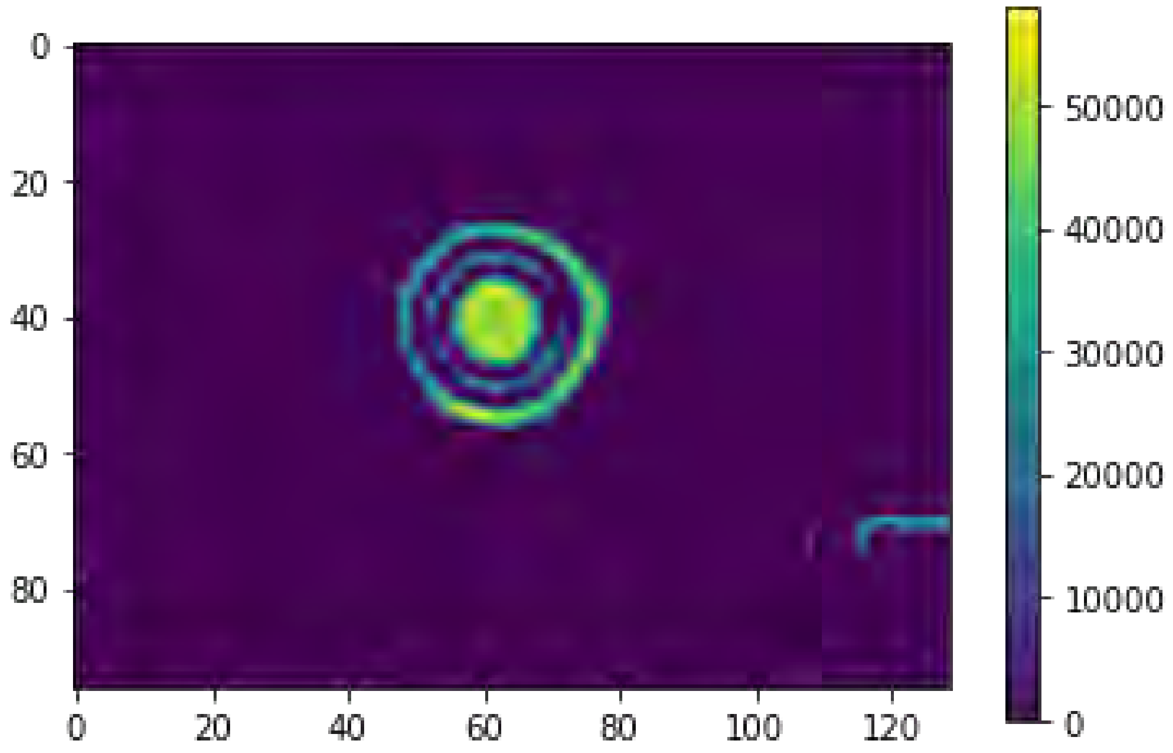

Figure 6.1: 2D image of the Ford Fiesta spring with image dimension = 75

The computation time to obtain the 2D image was almost the same in both the processors. The main parameters were the image dimension and the memory, which tend to differ slightly in both the processors. Also, the quality of the image depends on the dimension of the image. For this particular file, when the image dimension was reduced from 75, there seemed to be a high pixel breakage where the image was completely blurred.

3D images of this can be seen below with different dimensions, where the quality of images is entirely different. Images with dimensions from 50 to 150 are produced where the pixel breakage occurs.

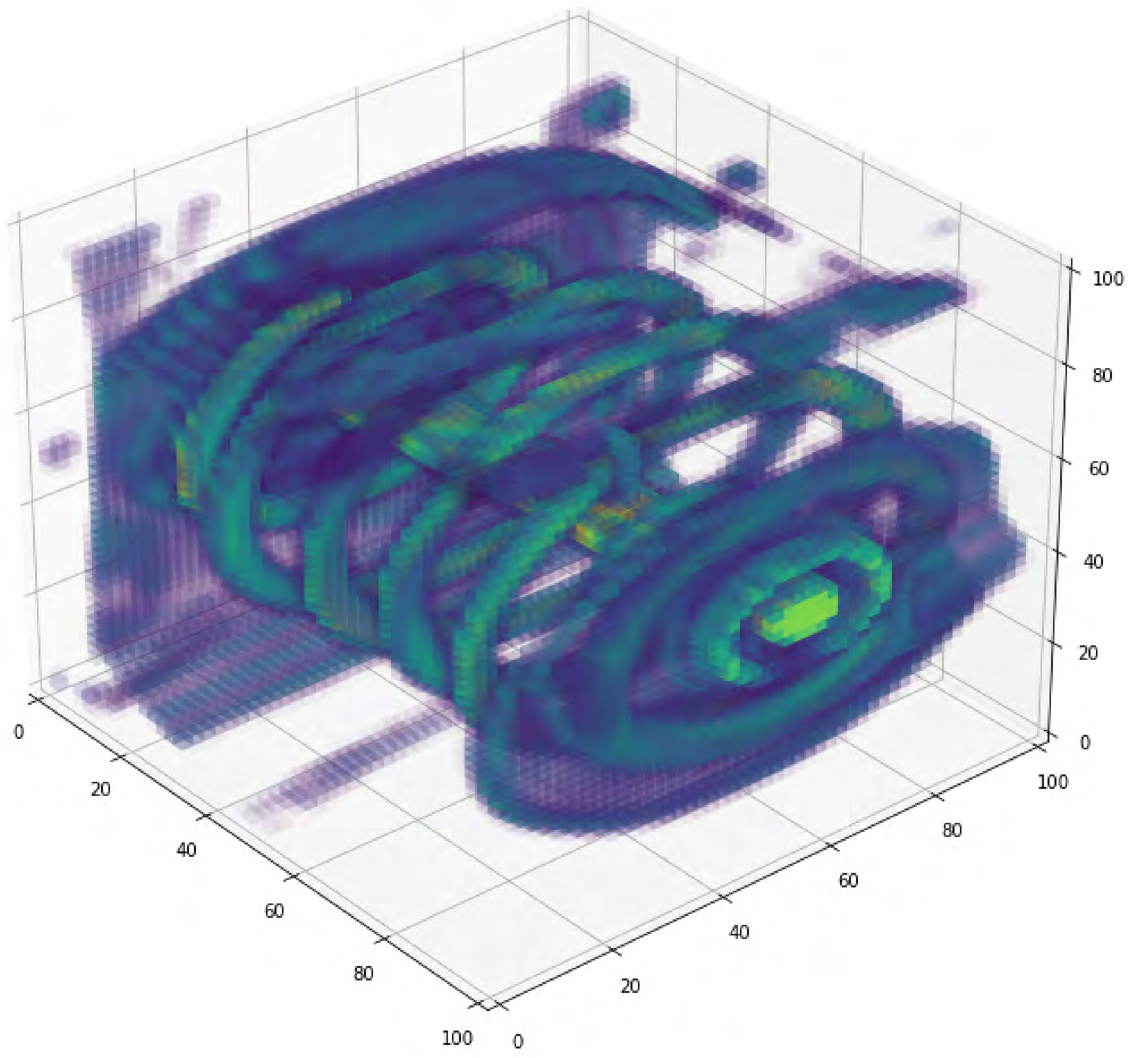

Figure 6.2: 3D image of the Ford Fiesta spring with image dimension = 50 at an angle of 320°

Here, with IMG_DIM = 50, the OSPRay image quality is inferior compared to the images that have IMG_DIM = 125 and also IMG_DIM = 150.

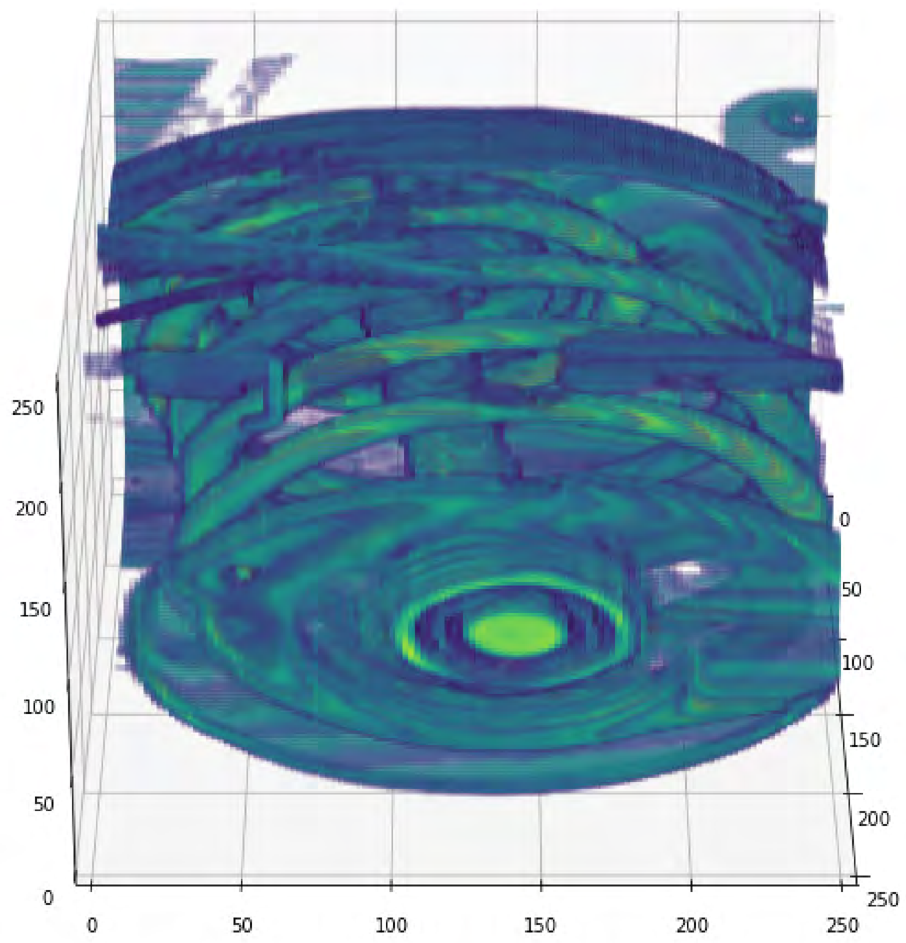

Figure 6.3: 3D image of the Ford Fiesta spring with image dimension = 125 at an angle of 360°

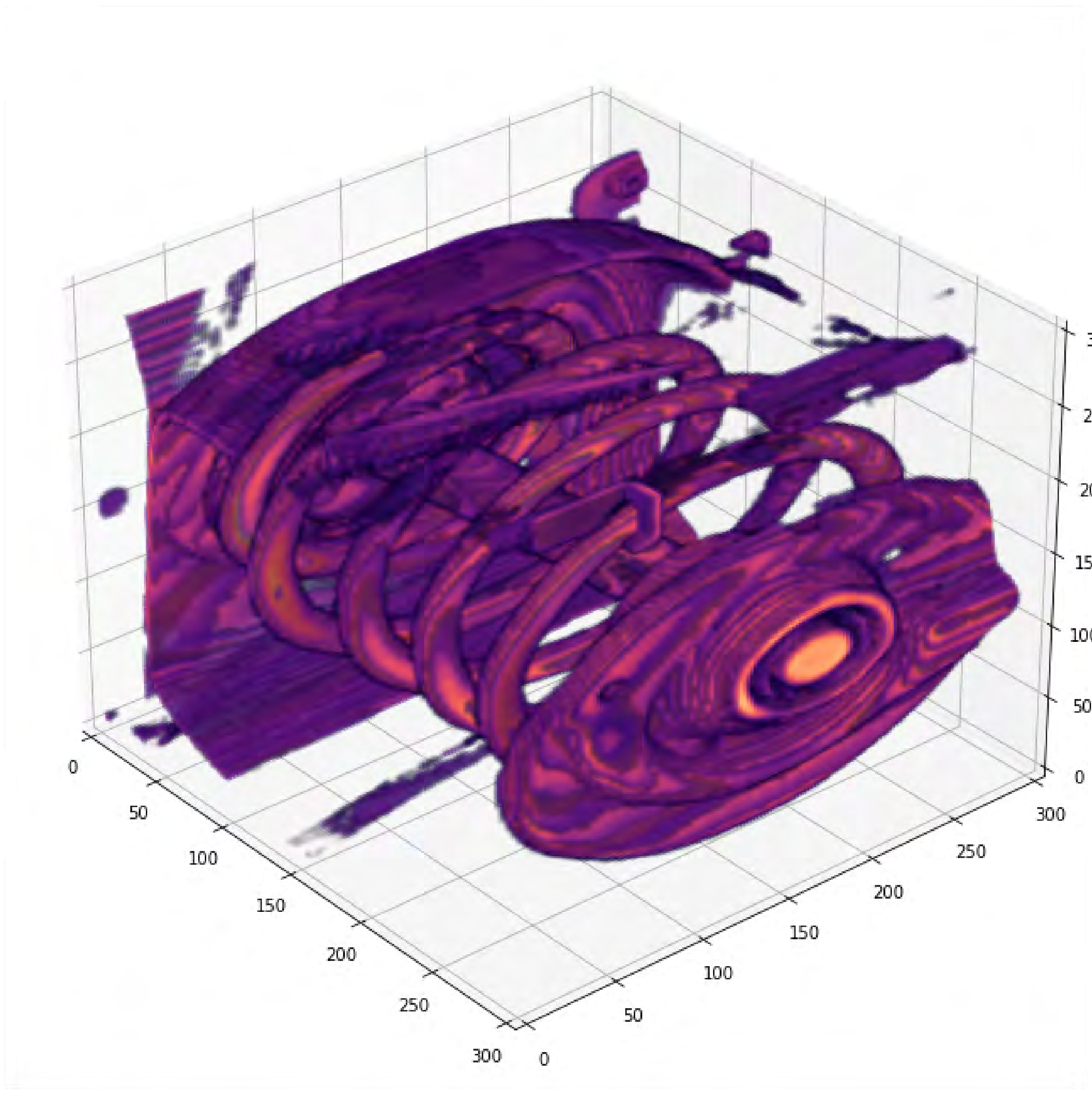

Figure 6.4: 3D image of the Ford Fiesta spring with image dimension = 150 at an angle of 320°

The second voxel data is the **Frosch2 shrunk4** (14.2MB) which has the image size slightly more significant than the previous file. So, generally, the computation time required is more when compared to the previous file. In the below figure 6.5, we can see the 2D image of the Frosch from the bottom angle.

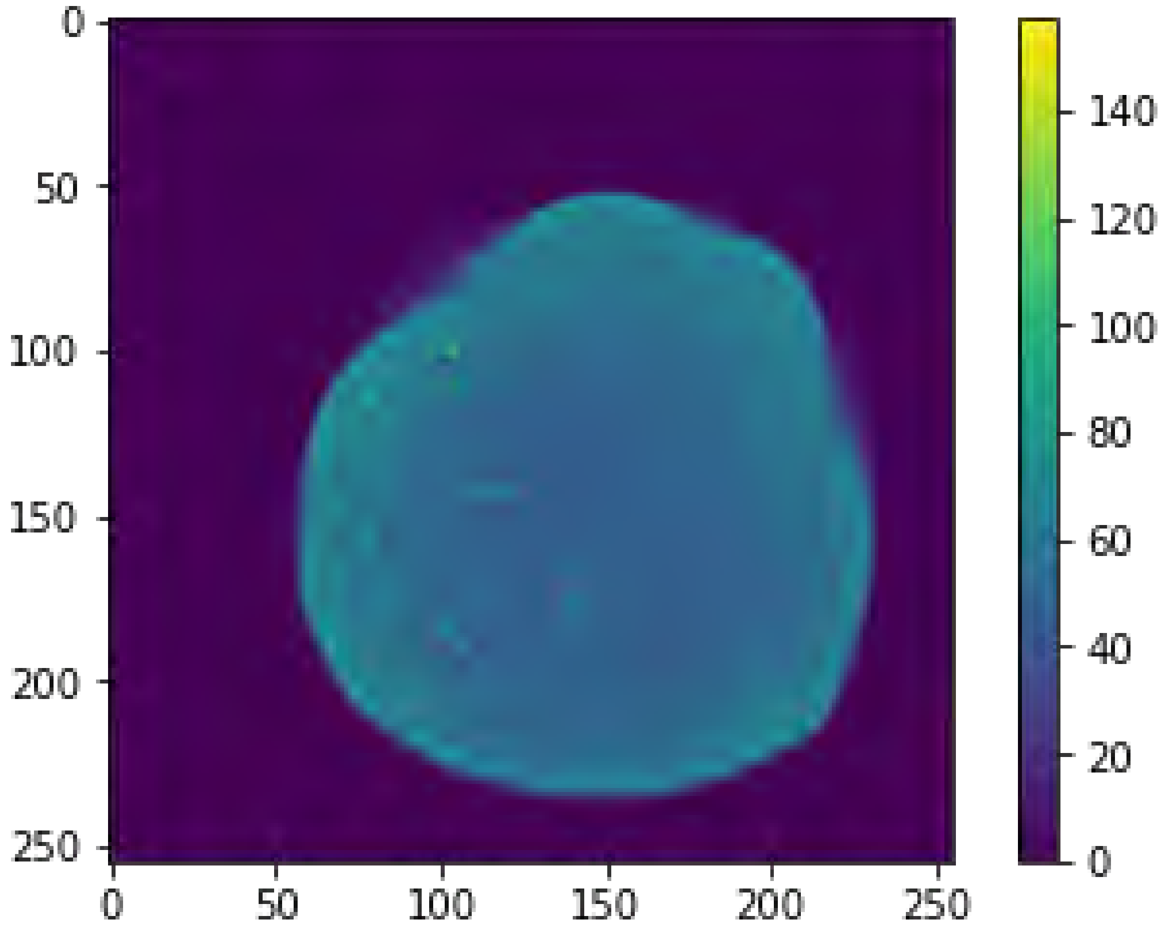

Figure 6.5: 2D image of the Frosch with image dimension = 50

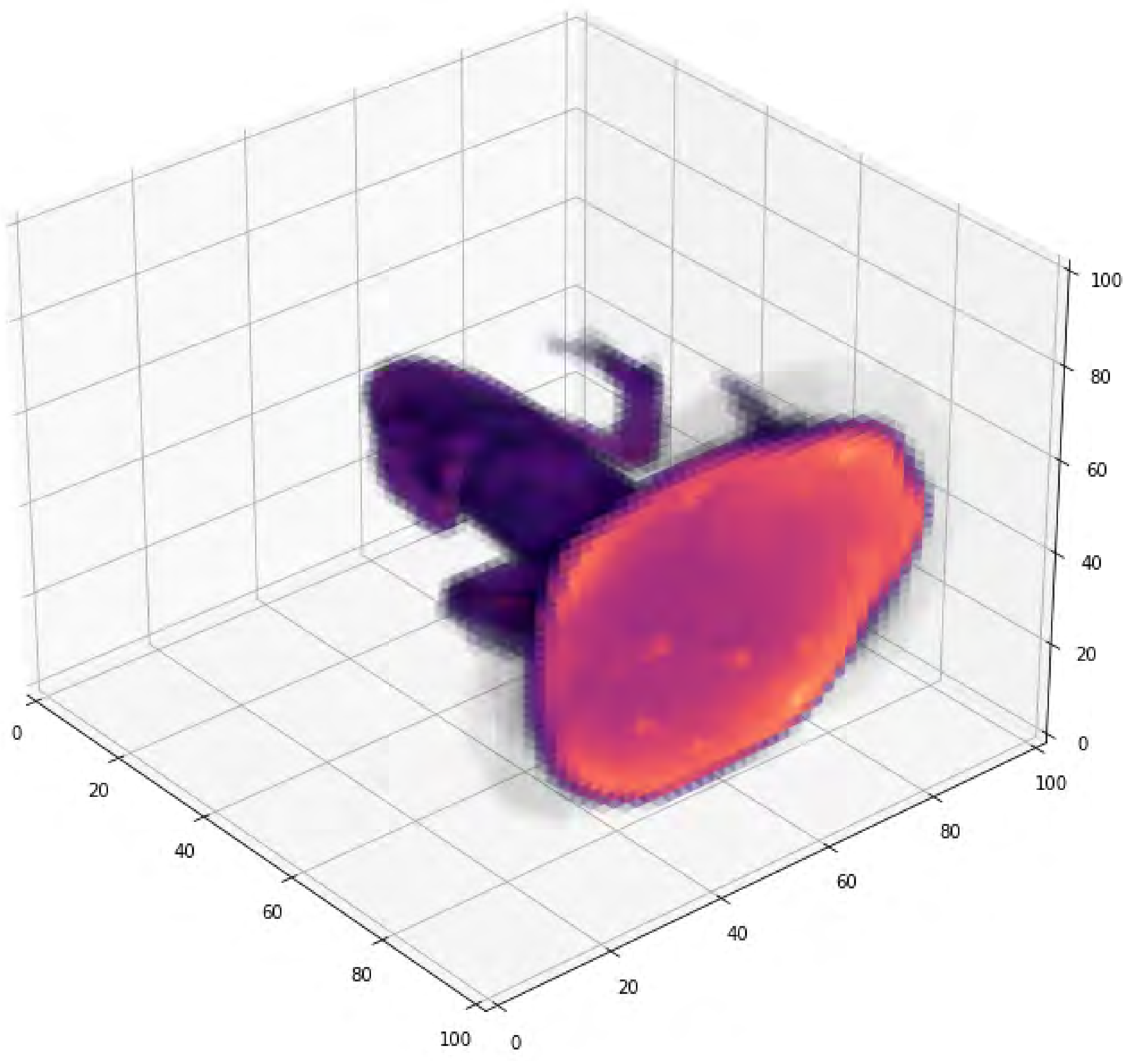

Figure 6.6: 3D image of the Frosch with image dimension = 50 at an angle of 320°

In this case, when the image dimension is set at 50, the 3D image appeared seemed to be of low-quality OSPRay image, where the image is a blurred image that is not clearly visible with slight shading effects.

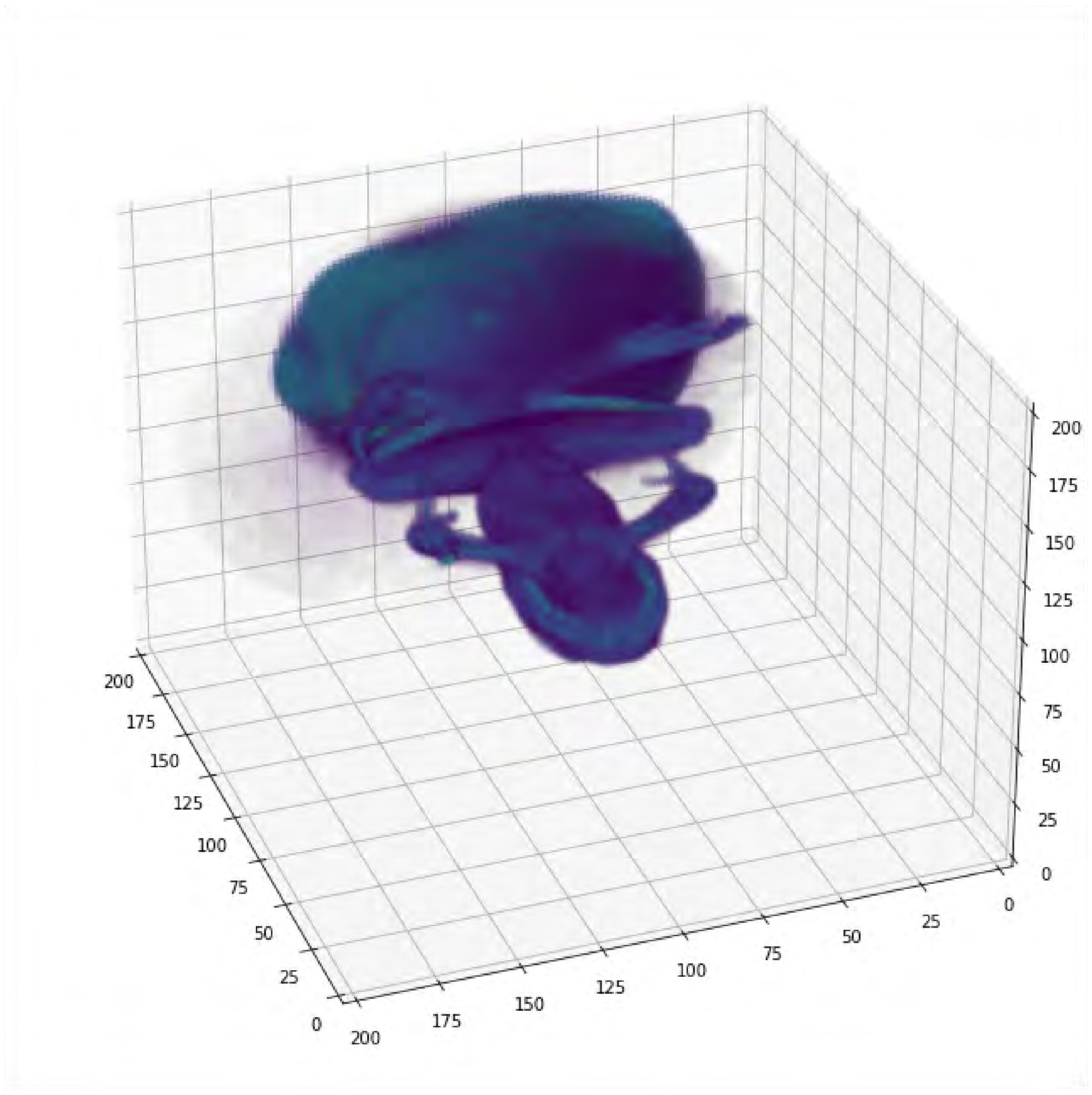

Figure 6.7: 3D image of the Frosch with image dimension = 100 at an angle of 160°

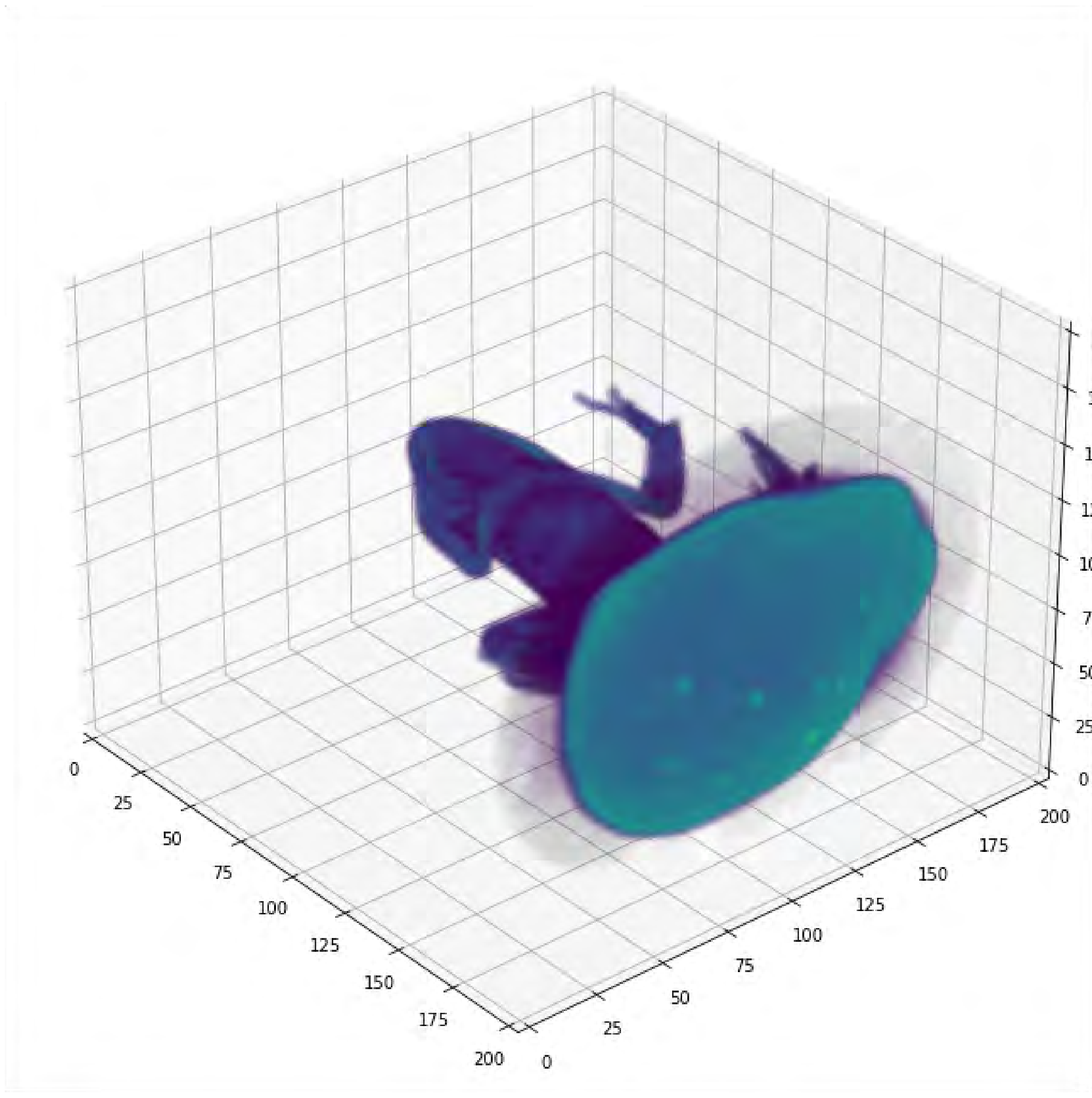

Figure 6.8: 3D image of the Frosch with image dimension = 100 at an angle of 320°

Here, the picture quality of 3D images obtained differs by that of the change in dimensions, and the computation time and memory used were high compared to the previous image file. So, it is pretty evident that when the size of the raw file increases, the computation time and the memory used increase, whereas the main parameter influencing the ray tracing is the Image dimension which plays a significant role in the quality of ray tracing images produced.

The third voxel data is the **Frosch2 shrunk2** (114MB), which is a very huge file when compared to that of the previous files. This file was run in a processor with high configuration as the processor with low configuration was not definitely suitable.

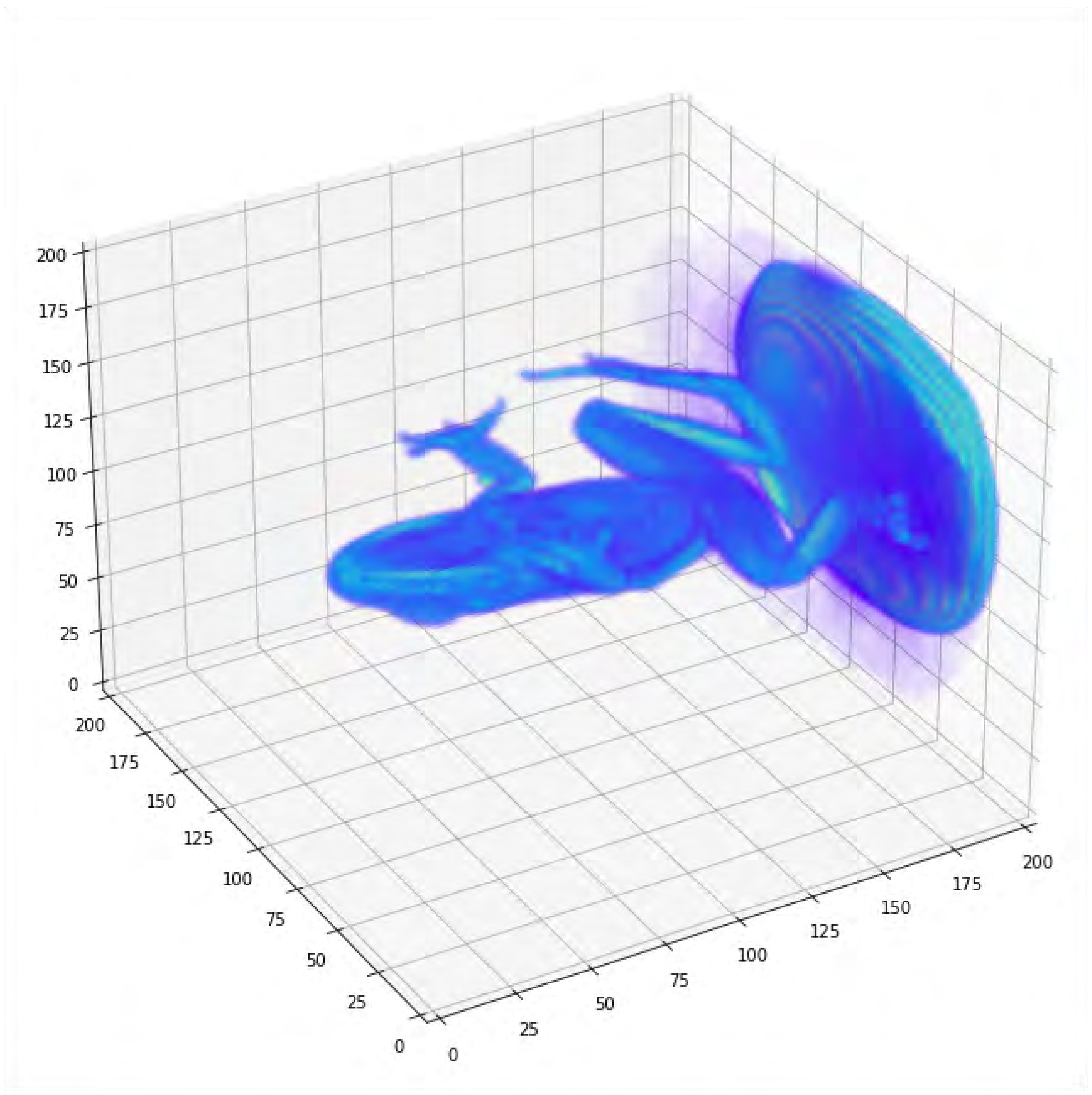

Figure 6.9: 3D image of the Frosch with image dimension = 150 at an angle of -480°

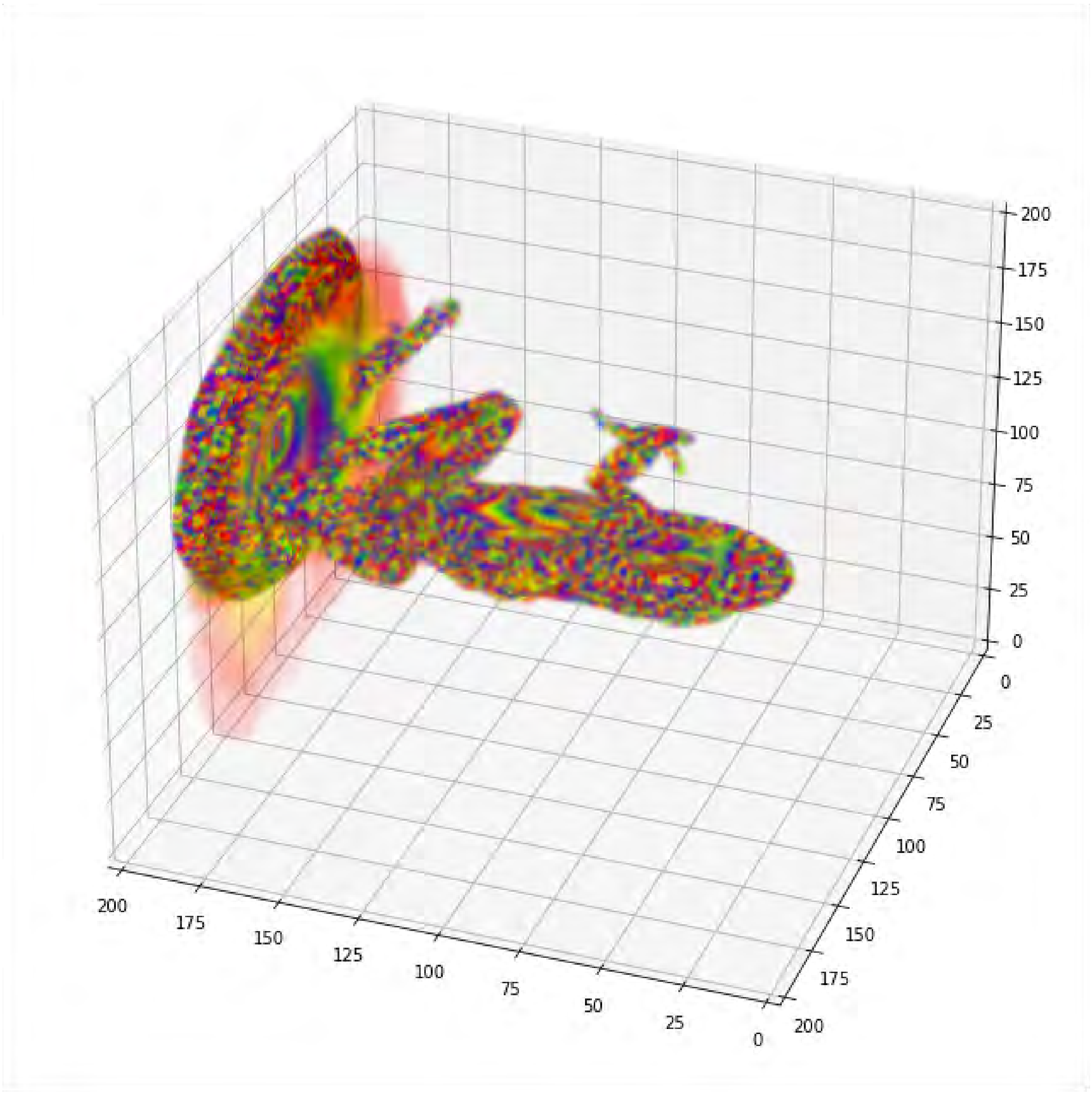

Figure 6.10: 3D image of the Frosch with image dimension = 150 at angle of 110°

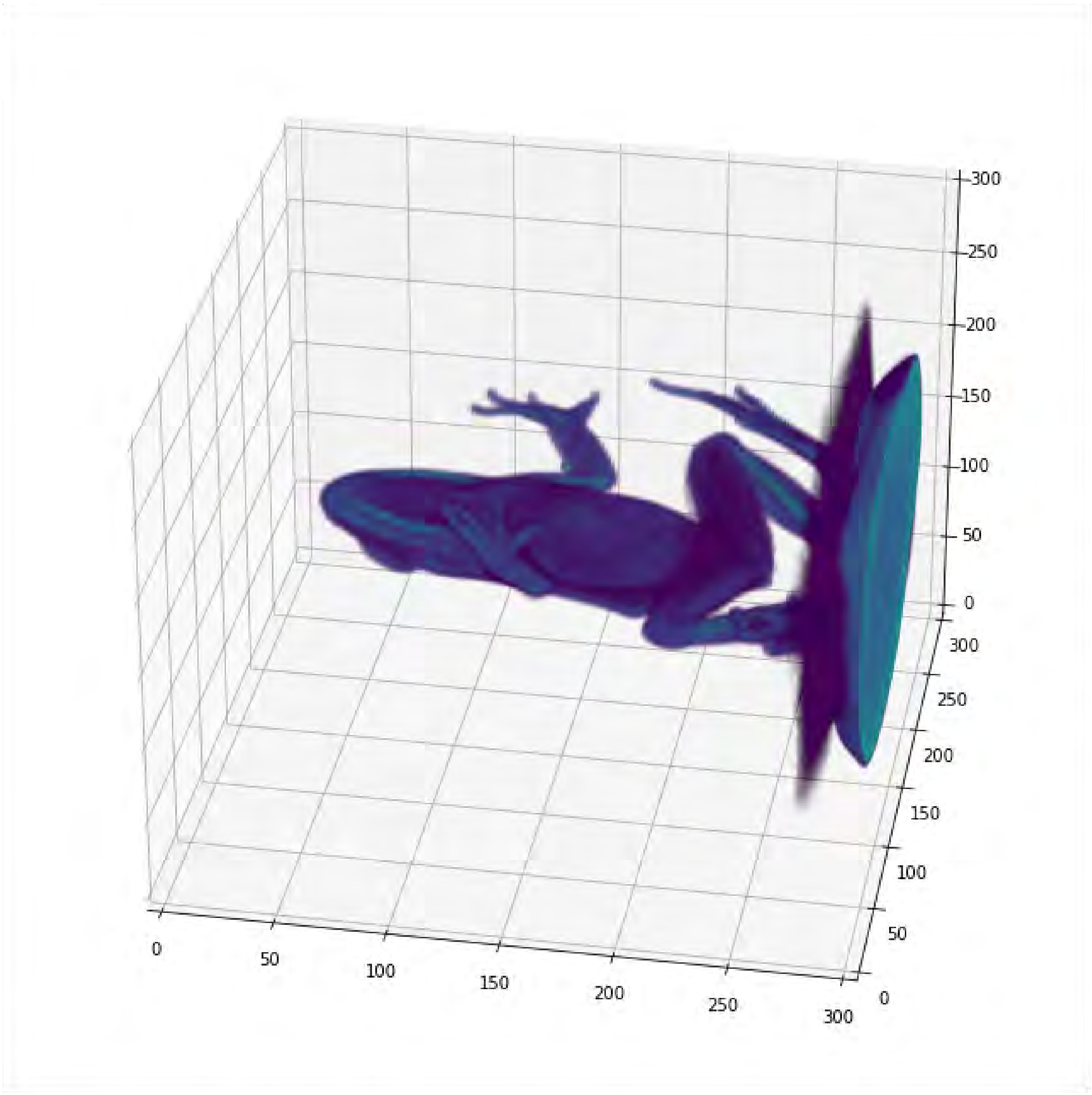

Figure 6.11: 3D image of the Frosch with image dimension = 100 at an angle of 640°

From this result from 6.11, we can say that the obtained OSPRay image is clearly visible in every tried angle with almost a shading effect. The computation time and the memory used were high than that of the previous images.

The fourth voxel data is the **Piston** (170MB, which is also a huge file and also of large provided dimensions. In this case, the 2D image at IMG_DIM = 50 is obtained without any technical hassles, as shown in fig 6.12.

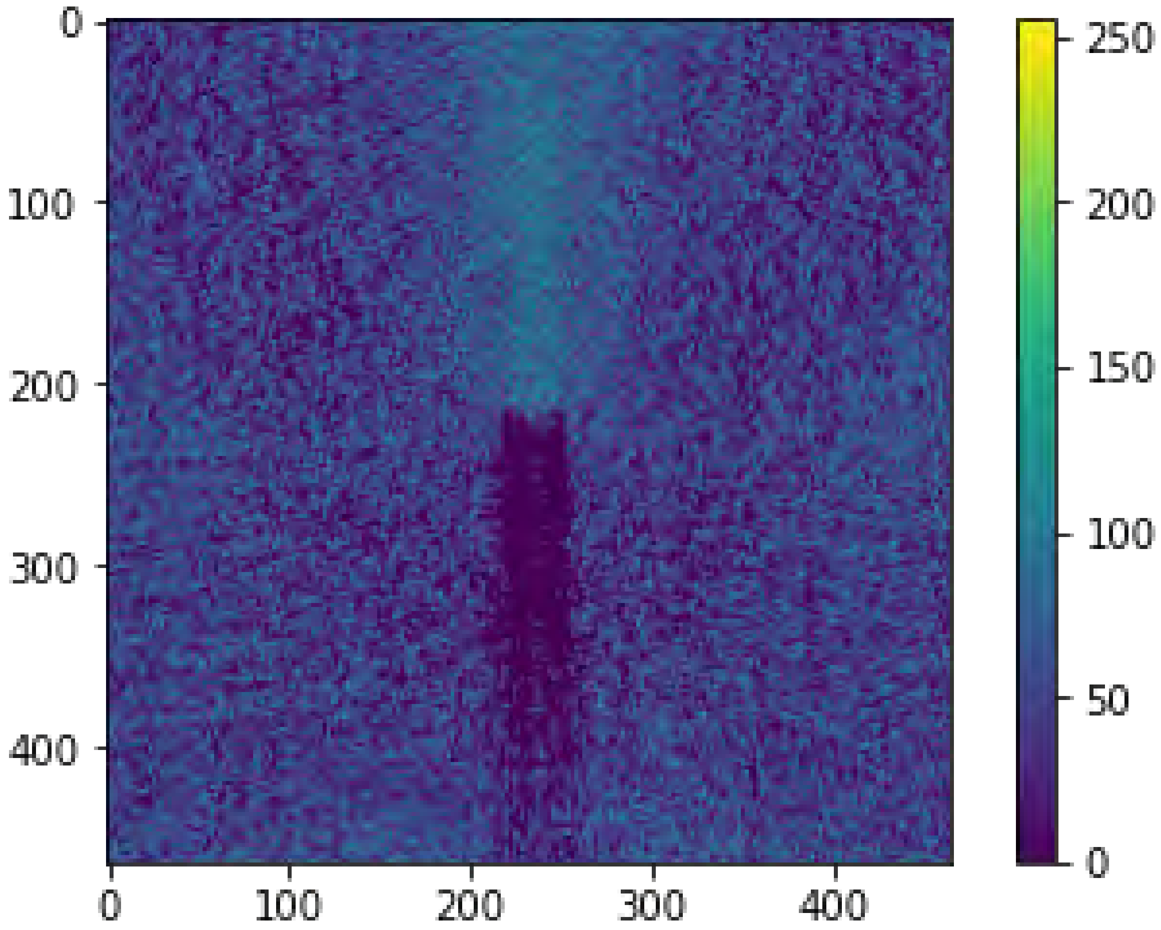

Figure 6.12: 2D image of the Piston with image dimension = 50 at an angle of 320°

Here, even the 2D image is not totally visible even though it was executed in a high processor with 32GB of RAM and 4GB of dedicated GPU. The produced OSPRay image is of low quality because of the shortage of processor requirements.

In the case of 3D image, the result is similar to that of the 2D image. As the file is enormous, it could be executed on an even better processor, maybe with 64GB of RAM and 8GB of the dedicated graphical unit. The OSPRay image of Piston is not at all visible, but only with some shadows and scattered lights, as shown in fig 6.13 and when run on a high configured system, we can see few visible parts of the piston from different angles, as shown in 6.14.

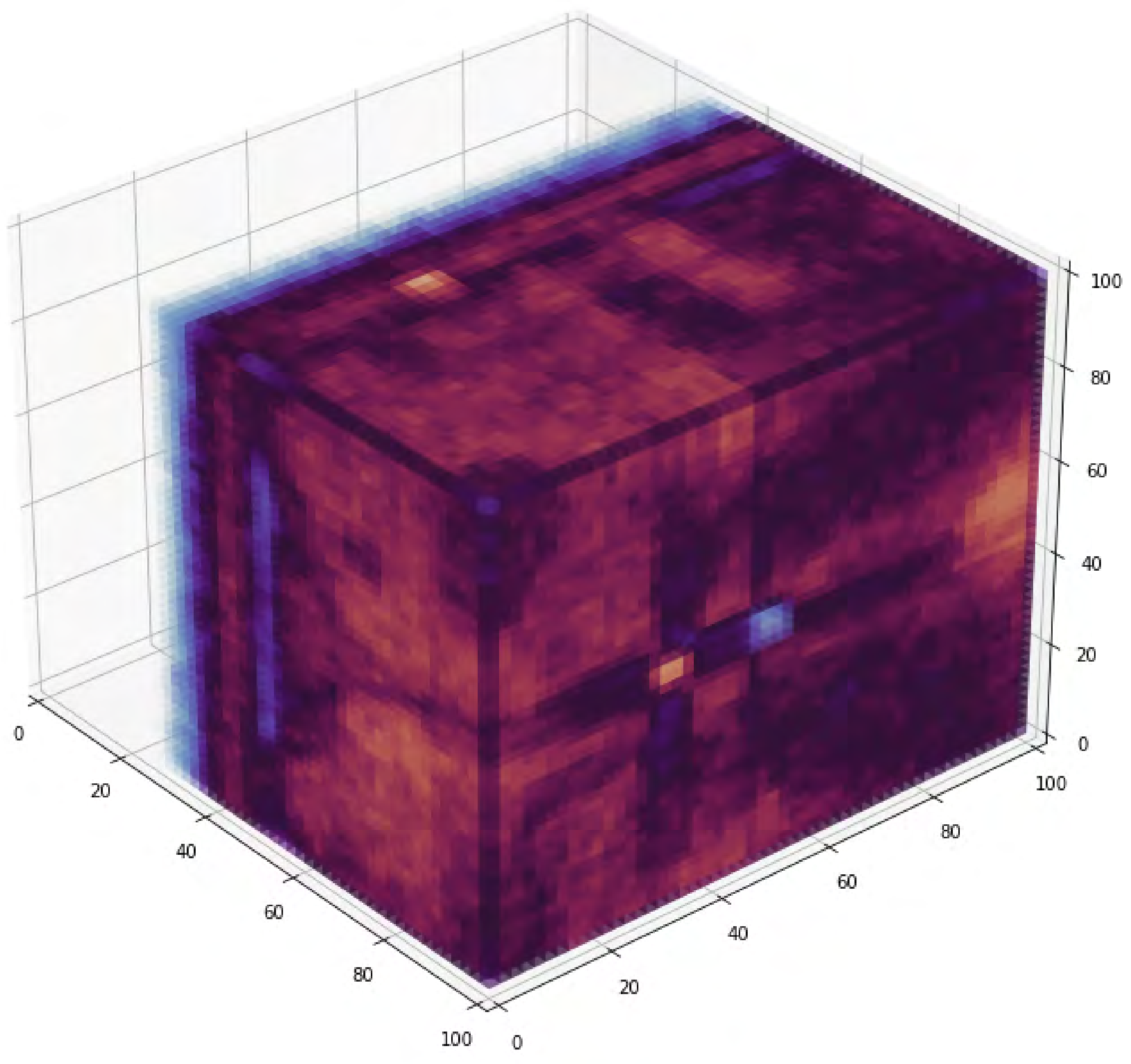

Figure 6.13: 3D image of the Piston with image dimension = 50 at an angle of 320°

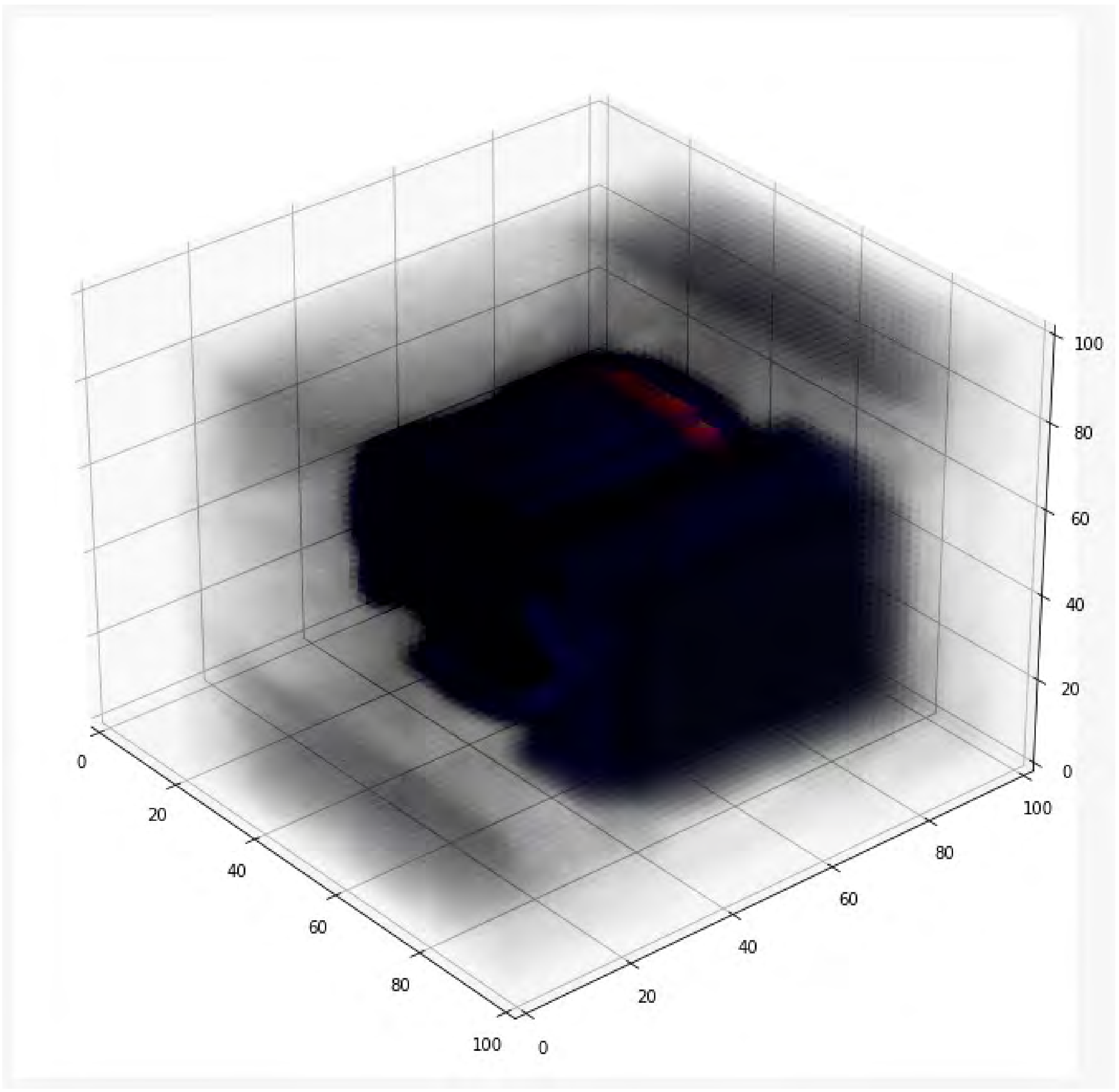

Figure 6.14: 3D image of the Piston with image dimension = 50 at an angle of 320°

As seen here, the Piston is not visible when the dimension of the image is kept at 50. The image might be more accurate if the dimension has been increased to 100. When the dimension was kept at 100 and executed, it crashed the processor with 8GB of RAM and 4GB of the dedicated graphical unit, probably because the system was running out of memory.

Here, we have noticed few elements regarding the three main factors of OSPRay, which we are discussing in this research. The computation time of the OSPRay tracing completely differs among high configuration and low configuration processors, whereas the main parameters influencing the OSPRay are the image dimension and pixels calculated in an image plane. The third factor, the quality of the obtained images, merely depends on the image dimensions and the pixel rate. A brief discussion of these factors comparing the performance with high configuration and low configuration processors is seen below 6.1.

Data	**Computation time of OSPRay tracing**	**Parameters influencing the OSPRay tracing**	**Quality of the images**
Ford Fiesta spring	Very low	Image dimension and the pixels generated	Equal
Frosch2 shrunk4	Moderate	Image dimension and the pixels generated	Equal
Frosch2 shrunk2	Very high	Image dimension and the pixels generated	Equal
Piston	Very high	Image dimension and the pixels generated	Equal

Table 6.1: High configuration processor with dedicated GPU

Here, we can see the comparison between the factors discussed where the **computation time** varies from low to high depending on the file size and also the processor. Then the main parameters influencing the OSPRay tracing are the **Image dimension** and the **Pixels generated** which provides the lighting and shading effects in ray tracing. When it comes to the **quality of the images**, both the processors generate the same quality irrespective of the pixel breakage and resolution. When you change the angle of the image, the resolution dips to a certain extent where still it can be visible.
Processors with high memory are required to execute large files without any hurdles and possibly reduce the computation time of the execution. This is considered as a limitation where it is not mandatory that everyone would own high-end processors.

Chapter 7: Conclusion

This study demonstrates the importance of one of the rendering techniques: ray tracing on images and especially the impacts of OSPRay tracing on any voxel image data. We have applied a novel approach in understanding the usage of Intel OSPRay tracing and how the various factors influence this process.

We have shown the influence of raw image files and how those images can be read in different dimensions, which empowers the ray tracing technique.

We also discussed the importance of light rays in ray tracing and how the transportation of light rays takes place. We likewise examined how the images are produced by following the path of the light as pixels in an image plane. We believe that further work on examining this in-depth would yield more and better results.

This study also showed shading effects in the ray tracing techniques and how they affect the polygonal and non-polygonal images. Besides this, further research works in the future can provide an insight into some state-of-the-art models in ray tracing techniques.

We believe our research can be a stepping stone to classify further models based on OSPRay tracing, and also, as this is open-source, it easily paves the way for upcoming future researches.

References

[1] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navrátil, "Ospray-a cpu ray tracing framework for scientific visualization," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 931–940, 2016.

[2] J. Ahrens, L.-T. Lo, B. Nouanesengsy, J. Patchett, and A. McPherson, "Petascale visualization: Approaches and initial results," in *2008 Workshop on Ultrascale Visualization*, IEEE, 2008, pp. 24–28.

[3] F. Wang, I. Wald, and C. R. Johnson, "Interactive rendering of large-scale volumes on multi-core cpus," in *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, IEEE, 2019, pp. 27–36.

[4] M. Han, I. Wald, W. Usher, Q. Wu, F. Wang, V. Pascucci, C. D. Hansen, and C. R. Johnson, "Ray tracing generalized tube primitives: Method and applications," in *Computer Graphics Forum*, Wiley Online Library, vol. 38, 2019, pp. 467–478.

[5] J. Bigler, A. Stephens, and S. G. Parker, "Design for parallel interactive ray tracing systems," in *2006 IEEE Symposium on Interactive Ray Tracing*, IEEE, 2006, pp. 187–196.

[6] U. Behrens and R. Ratering, "Adding shadows to a texture-based volume renderer," in *IEEE symposium on volume visualization (cat. no. 989EX300)*, IEEE, 1998, pp. 39–46.

[7] F. Wang, N. Marshak, W. Usher, C. Burstedde, A. Knoll, T. Heister, and C. R. Johnson, "Cpu ray tracing of tree-based adaptive mesh refinement data," in *Computer Graphics Forum*, Wiley Online Library, vol. 39, 2020, pp. 1–12.

[8] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker, "Memory sharing for interactive ray tracing on clusters," *Parallel Computing*, vol. 31, no. 2, pp. 221–242, 2005.

[9] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-level ray tracing algorithm," *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, pp. 1176–1185, 2005.

[10] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker, "Ray tracing animated scenes using coherent grid traversal," in *ACM SIGGRAPH 2006 Papers*, 2006, pp. 485–493.

[11] M. Ament and C. Dachsbacher, "Anisotropic ambient volume shading," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 1015–1024, 2015.

[12] J. Beyer, M. Hadwiger, and H. Pfister, “A survey of gpu-based large-scale volume visualization,” in *Eurographics Conference on Visualization (EuroVis)(2014)*, IEEE Visualization and Graphics Technical Committee (IEEE VGTC), 2014.

[13] Intel, *Intel® ospray*. [Online]. Available: `https://www.ospray.org/`.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß bernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 27/05/2021

Jobin Jothi Prakash