



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF ALGORITHMS AND THEIR APPLICATIONS

Octree-based Approach for Real-time Visualization of Surfaces Defined by Signed Distance Fields

Author:

Pedro Henrique Villar de Figueirêdo

Computer Science BSc

Supervisor:

Csaba Bálint

Doctorate Student

Supervisor:

Dr. Zsóka Viktória

Assistant Professor

Budapest, 2020

This page should be the original Thesis Topic Declaration.

Contents

1	Introduction	4
1.1	Applications	5
1.2	Context	6
1.3	Thesis structure	7
2	User guide	8
2.1	Installation	8
2.2	Basic Features	9
2.2.1	Auxiliary Window	9
2.2.2	Main Window	12
2.3	Advanced Features	14
2.3.1	Node List Tab	14
2.3.2	Automated Testing	18
3	Theoretical Background	20
3.1	Sphere Tracing	20
3.1.1	Distance Functions	20
3.1.2	Signed Distance Functions	21
3.1.3	Lipschitz continuity	22
3.1.4	Ray-Surface Intersection	22
3.2	Octree	23
4	Algorithms	25
4.1	Tree traversal	26
4.2	Octree Construction	26
4.3	Auxiliary Graph	28
4.3.1	Definition	29

4.3.2	Construction	29
4.3.3	Closest Triangles Subproblem	30
4.4	SDF Computation	32
4.4.1	Distance Values Computation	32
4.4.2	Inside/Outside Partition	33
4.5	Octree Optimization	37
4.6	Octree Serialization	39
4.7	Serialized Octree Lookup Algorithm	40
5	Implementation	43
5.1	Overview	43
5.2	Auxiliary Classes	45
5.2.1	AABB	45
5.2.2	Triangle	47
5.2.3	Importer	48
5.2.4	OctreeNode	49
5.2.5	Vertex	50
5.3	Octree	51
5.3.1	Data Structure	52
5.3.2	Traversal	55
5.3.3	Construction	55
5.3.4	SDF Computation	56
5.3.5	Optimization	57
5.3.6	Serialization	58
5.3.7	Lookup	59
5.4	Testing	61
5.4.1	User interface tests	61
5.4.2	Automated tests	63
5.5	Generated SDF Estimations	64
6	Conclusion	67
	Bibliography	69

CONTENTS

List of Figures	71
List of Codes	73

Chapter 1

Introduction

The generation of objects in simulated controlled environments has been an important tool for advancements in various applications in which the visualization is part of the development of products. It can be one of the ways of prototyping or output for real-life applications. Examples include simulation software for theoretical physics, modelling programs for architecture or medicine, and rendering engines for entertainment. Nowadays, with the constant improvement of processors both for general (CPU) and specific (GPU) workloads, that fit an increasing number of transistors in their many cores, three-dimensional simulations of objects and environments are being visualized and modified in real-time. The majority of software options for real-time visualization of objects rely on triangle-based representations, in which any shape can be described as a combination of triangles.

Although the triangle-based representation works well for most cases, there are some applications, such as group-theoretic operations (eg. union, intersection), which are not efficiently implemented in such approach, compromising the real-time capability that can be an essential feature. When describing objects via triangles, the rendering algorithm needs to insert or remove new primitives from the scene to apply any operation which modifies an object in a non-regular manner. This results in very inefficient operations. Therefore, a solution for dealing with such specific applications is to search for an alternative way of representing objects in a simulated environment.

Surfaces can be implicitly defined by a function which describes the distance from any point in space to such object, named *signed distance function* (SDF).

Many operations are applied to SDFs in an efficient manner [1], including the offset, union, and intersection. Composite objects can be represented via SDFs by group theory operations of well-defined functions of primitives.

Even with many benefits on certain operations, the SDF representation of objects lacks the compatibility with most three-dimensional models, which are triangle-based. Although complex objects can be constructed via group operations from simple shapes, the complexity of the geometry directly affects the rendering performance. As a result, it is infeasible to compute the signed distance function for typical triangle representations.

Considering such limitation of SDFs, we propose an efficient solution for the estimation of surfaces of triangle-based objects through the use of *signed distance functions*. The solution is based on an octree data structure, which acts as a recipient to store and manage distance values of a given object. It works for any closed triangle mesh, and allows real-time visualization of operations such as the offset, the union, and the intersection.

1.1 Applications

The software subject of this thesis has various application scenarios. Since it has a well-defined goal to estimate triangle-based objects using *signed distance functions*, it can be applied in cases needing operations that would benefit from the implicit representation of SDFs.

Operations that can take advantage of objects defined by SDFs include the offset operation, as well as the erosion and dilatation of complex meshes. The processing of *soft shadows* and the visualization of group-theoretic operations are also significantly faster for implicit surfaces.

Therefore, the visualization of specific operations applied on closed triangle meshes is a viable application of our software. It can also be used for integrating triangle-based surfaces into already existent scenes defined by SDFs.

By extending the range of efficient representations of SDFs to typical triangle meshes, our application allows the use of this alternative way of representing objects to previously infeasible scenarios. For instance, the visualization of simulations for

theoretic physics that require the extensive use of group-theoretic operations can use SDFs to represent all surfaces for improved performance, including complex objects made from triangle meshes.

1.2 Context

The context in which the application of this thesis was formulated had a working *Sphere tracing* renderer [2], which is an optimized variant of a *ray tracer* for scenes defined by distance functions. All the benefits of using the implicitly defined surfaces were already in place prior to the development of the octree-based solution.

It was referred that, to solve the estimation of the SDF for triangle meshes using an octree, the new solution would be integrated into an existing rendering engine that used *NVIDIA Falcor C++ API* and *DirectX High Level Shading Language*. However, due to inconsistencies in new releases of the *NVIDIA API* and the superior compatibility and portability of the *OpenGL shading language*, the latter was chosen for implementing the software of this thesis together with the *C++ programming language*.

As previously explained, the poor performance for triangle-based inputs in the SDF representation was a major drawback in adopting this type of surfaces description. Bærentzen [3] uses a simple grid to estimate the SDFs for triangle meshes. Our new approach using the octree data structure was inspired by the aforementioned inefficiency with SDFs and the software by Bærentzen.

It is important to note that our proposed solution only works for closed surfaces (defined by closed meshes of triangles), due to the inside/outside partitioning of the octree nodes during the construction. It would be possible to estimate the SDF of initially open meshes by utilizing algorithms [4] to deduce and add missing triangles, closing the open spaces, and resulting in a closed mesh that can be utilized by our software.

1.3 Thesis structure

This thesis is organized in six chapters. It starts with the introduction (Section 1), explaining the motivation, providing applications, and context for the reader. Afterwards, it presents a detailed user guide (Section 2) for the accompanying software, with basic and advanced manuals.

A theoretical background (Chapter 3) is provided, containing definitions needed for the understanding of following chapters. The algorithms chapter (4), which explains every major algorithm used in the proposed software solution is presented next. The implementation chapter (5) discusses structures and classes as well as examines efficiency and optimization choices for the implementation of the algorithms presented in chapter 4. The testing plan and some important SDF estimations are also investigated in chapter 5. Lastly, the conclusion of the work is formulated (Chapter 6).

The *developer documentation* is divided into many different chapters, providing better indexing of the work and ease for the reader. Specific aspects of our proposed solution are located by navigating directly to a part of the thesis for which the reader is most interested in.

Furthermore, the decision of explaining the main steps of the solution in an algorithmic pseudo-language intends to expand its applicability and readability, allowing an easier integration to various programming languages and paradigms.

Chapter 2

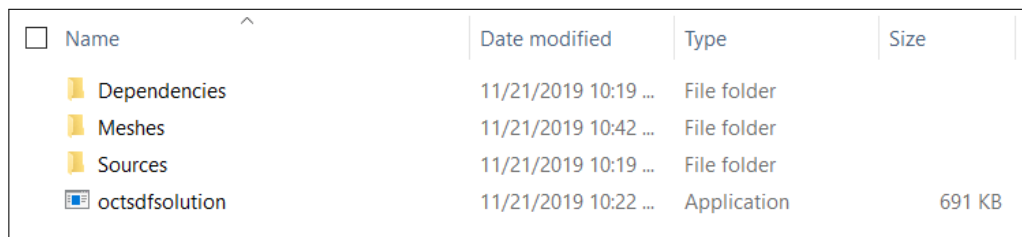
User guide

This chapter aims at explaining the functionalities of the software to a user. It starts by explaining the installation of the software (2.1), then presents a basic (2.2) and an advanced (2.3) features manual.

2.1 Installation

The installation of the software can be done by extracting it in any directory from the compressed file *octsdfsolution.zip*, available in the accompanying CD.

Once extracted, a folder containing the executable file *octsdfsolution.exe* and three subfolders (*Sources*, *Dependencies*, and *Meshes*) is available. Figure 2.1 shows the extracted folder. The *Sources* subdirectory contains all the source code files of the project. *Dependencies* hosts the packages and other external libraries needed for compiling the source code and executing the program. The *Meshes* subfolder includes various triangle mesh files that can be loaded and visualized by the program.







<input type="checkbox"/> Name	Date modified	Type	Size
 Dependencies	11/21/2019 10:19 ...	File folder	
 Meshes	11/21/2019 10:42 ...	File folder	
 Sources	11/21/2019 10:19 ...	File folder	
 octsdfsolution	11/21/2019 10:22 ...	Application	691 KB

Figure 2.1: Extracted software solution.

2.2 Basic Features

The user can start by executing the *octsdfsolution.exe* file. It will trigger the start of the application, which then loads two windows, as can be seen in Figure 2.2. The first and main window (on the left) shows a canvas in which objects will be rendered. The second window (on the right) is an auxiliary user interface, in which it is possible to load new mesh files as well as visualize and modify information regarding the execution of the program. Further information on both windows is presented in Sections 2.2.1 and 2.2.2.

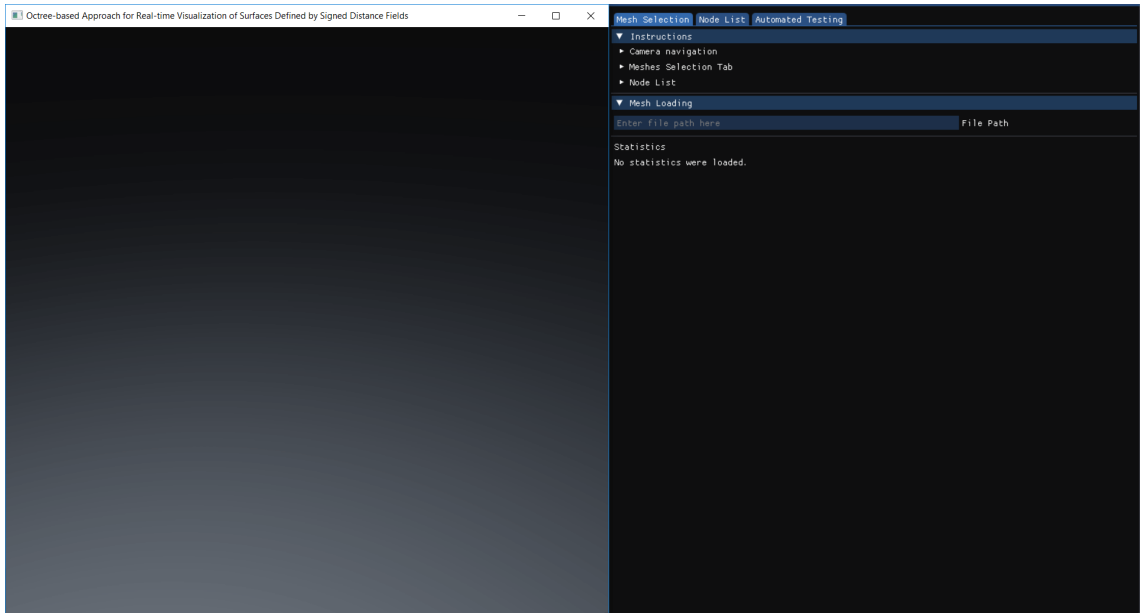


Figure 2.2: Starting screen of the program.

2.2.1 Auxiliary Window

The auxiliary window is composed of three tabs: *Mesh Selection*, *Node List*, and *Automated Testing*. It is possible to visualize such tabs at the top of the window. For this section, only the *Mesh Selection* tab will be explained. For further information on the remaining tabs, see Section 2.3.

With the *Mesh Selection* tab selected, it is possible to see that the auxiliary window is divided in three sections: *Instructions*, *Mesh Loading*, and *Statistics*. The *Instructions* section provides general guidelines on how to use the program. It has information about how to navigate the camera (explained in Section 2.2.2), it briefly describes the two other sections of this tab (*Mesh Loading* and *Statistics*), and

sums up essential information about the two other tabs (*Node List* and *Automated Testing*).

To interact with the *Instructions* section, it is sufficient to click on one of its topics, and it will show additional information for it (See Figure 2.3). By clicking again in the topic, it will hide the previously shown additional information. It is possible to hide all the contents of the *Instructions* section by clicking on its name. If the user wants to revisit the information of this section in a later stage, clicking on it again will show its content once more.

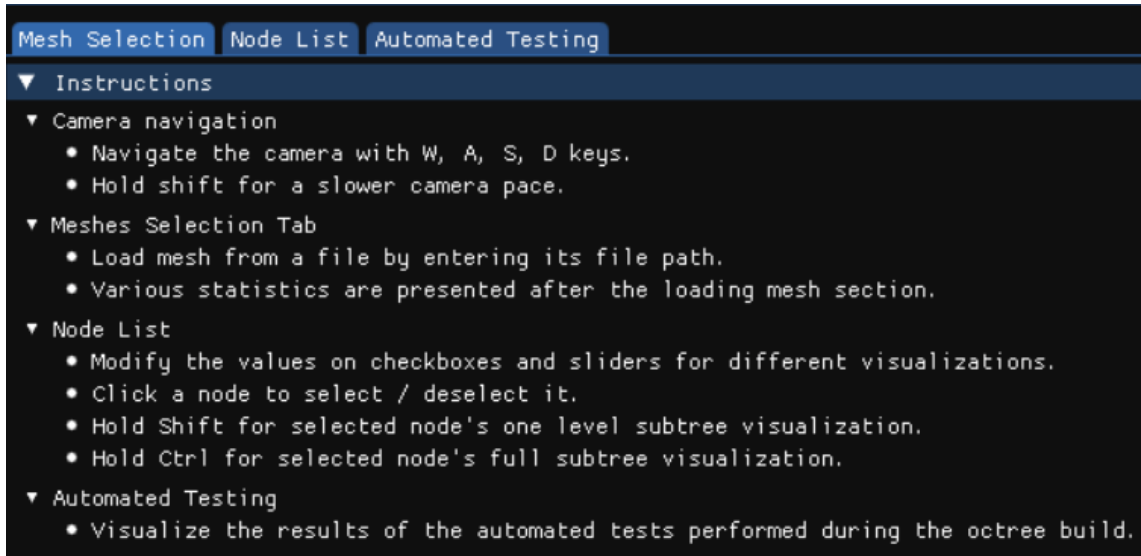


Figure 2.3: Expanded view of the Instructions section of the Auxiliary Window.

The *Mesh Loading* section has the same functionalities of hiding and showing its content as in the *Instructions* section. Initially, it only has one field. This input field accepts text, and it is needed for loading a triangle mesh file into the program. Once the user starts typing, a new button named *Load Mesh* appears beneath the text entry input field. To load a new mesh, the user should provide the full or relative path to the file containing it. Once the path is written into the input text field, the *Load Mesh* button needs to be pressed to start loading the mesh into the program.

If a valid file is given, a progress bar prompts the user about the stages of the octree building. If the file or the path is not valid, the user is presented with a message explaining the error and then the program waits for a next file entry. The mesh loading is executed in a new thread independent of the user interface. This allows a smooth experience for the user and provides instant feedback of the status

of such build through the progress bar. A representation of the *Mesh Loading* section before clicking the *Load Mesh* button is shown on Figure 2.4.



Figure 2.4: Expanded view of the Mesh Loading section of the Auxiliary Window.

The *Statistics* section presents the user with the data about the execution of the program. Initially, it only shows the informative message: "No statistics were loaded". This happens since the program had not loaded any triangle mesh; therefore, no meaningful data about its execution can be provided. Once a mesh is loaded, information about the loaded mesh (*Mesh* subsection), building times for different steps of the software algorithm (*Build Times* subsection), and the octree (*Octree* subsection) are provided. To show or hide the content of a subsection, it is sufficient to click on its name. An additional feature is that, since the loading of a mesh is asynchronous, statistics appear in this section as they are calculated.

In the *Mesh* subsection, it is possible to see information about the mesh of triangles provided as input. It includes the file name and extension, as well as the number of triangles in such mesh. *Build Times* subsection provides information about the time of execution of every main step of the octree realization, from its construction, to serialization. Additionally, a total building time is provided in the end as an easy way of comparing building times for different meshes. Lastly, the *Octree* subsection provides information about the octree. It includes the maximum depth and number of nodes of the octree; provides the number of vertices in the auxiliary graph, and the number of leaf and non-leaf nodes in the serialized arrays of the octree.

The data displayed in the *Statistics* section is useful for understanding how the octree data structure performs for specific meshes, and how costly every main step in the octree construction is in practice. Figure 2.5 shows the *Statistics* section for one of the mesh files provided as example in the *Meshes* subdirectory (Section 2.1), named *Suzanne.obj*.

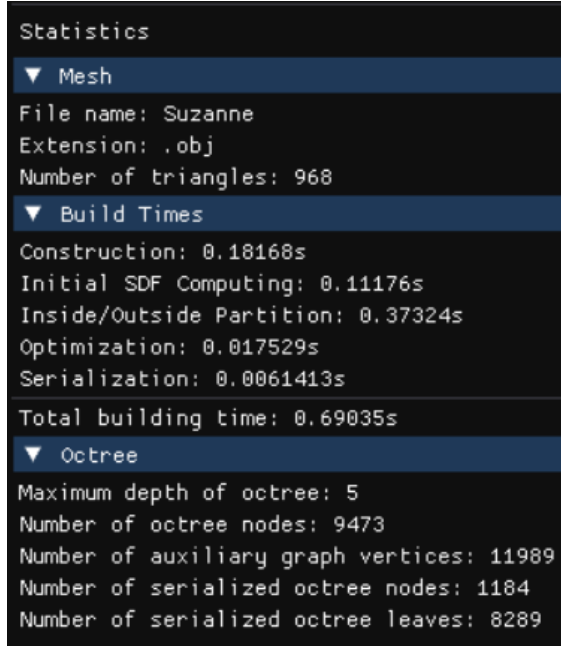


Figure 2.5: Expanded view of the Statistics section of the Auxiliary Window.

2.2.2 Main Window

The main window shows the result of the rendering of objects by the software. It presents an interactive three-dimensional simulated environment showing real-time results obtained through the use of a *sphere tracing* renderer (see Section 3.1 for further information about *sphere tracing*). What is seen through the *Main Window* is a reflection of the processing of the scene seen by the camera in the simulated environment.

It is possible to navigate the camera in the three-dimensional space to be able to visualize objects from different angles and at different distances. These movements involve the use of the keyboard and mouse. More specifically, the keys *W*, *A*, *S*, *D*, *Shift*, on the keyboard, as well as the left button and the wheel of the mouse. Table 2.1 presents information about each of the aforementioned input options for navigation, their intended type of interaction, and the impact on the movement of the camera within the rendered space of the *Main Window*.

Input key	Type of Interaction	Action
<i>W</i>	Click or hold	Move the camera forward
<i>A</i>	Click or hold	Move the camera to the left
<i>S</i>	Click or hold	Move the camera backwards
<i>D</i>	Click or hold	Move the camera to the right
<i>Shift</i>	Hold	Slower the pace of camera movement
<i>Mouse wheel</i>	Scroll	Change the velocity of the camera
<i>Left button of the mouse</i>	Hold and drag	Control the direction of the camera

Table 2.1: Mapping of input keys, their intended type of interaction, and the impact on the camera movement.

By combining the use of the left button and the other movement keys (*W*, *A*, *S*, *D*), the user can navigate in any direction and even set the pace of which the camera moves slower (*Shift*), to be able to investigate an object from a short distance. The additional option of scrolling the mouse wheel for adjusting to a specific velocity improves the experience for very small or large meshes. Examples of distinct viewing points of the same object that illustrate the movement of the camera can be seen in Figure 2.6. It shows *Suzanne.obj*, a well-known mesh used for various computer graphics applications, introduced by the Blender renderer project [5].

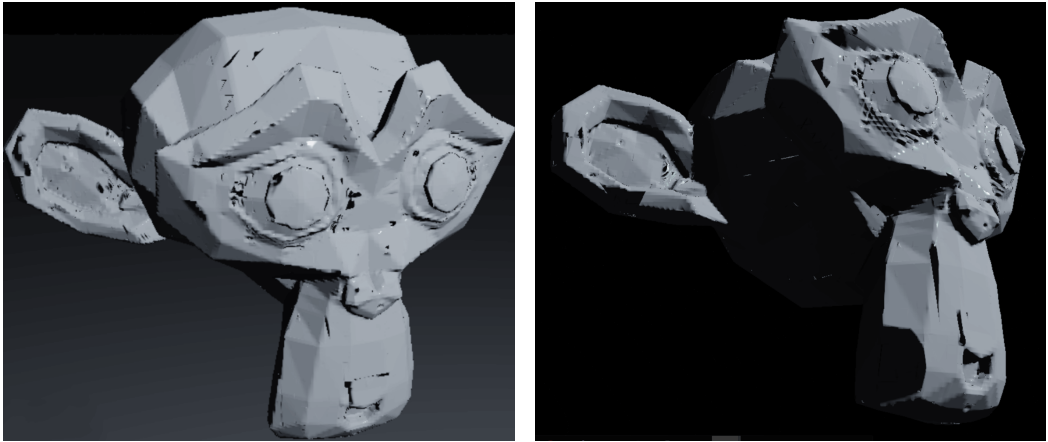


Figure 2.6: Different angles for rendering of the mesh included in *Suzanne.obj*.

2.3 Advanced Features

This section discusses of a more advanced features with the program, which involves knowledge on how the octree works and the main steps in its build process. The usage of both *Node List* (Section 2.3.1) and *Automated Testing* (Section 2.3.2) tabs of the *Auxiliary Window*, and its implications on the *Main Window* are discussed. To navigate to the aforementioned tabs, it is sufficient to click on their name.

2.3.1 Node List Tab

The *Node List* tab provides different ways of interacting with objects described by the octree structure of the proposed software solution. Its interface (Figure 2.7) contains multiple visualization options for highlighting certain aspects of an object and the octree that constitutes it. Additionally, the interface of the *Node List* tab allows the modification of parameters used in the octree build process, and the testing of different levels of the *offset* operation.

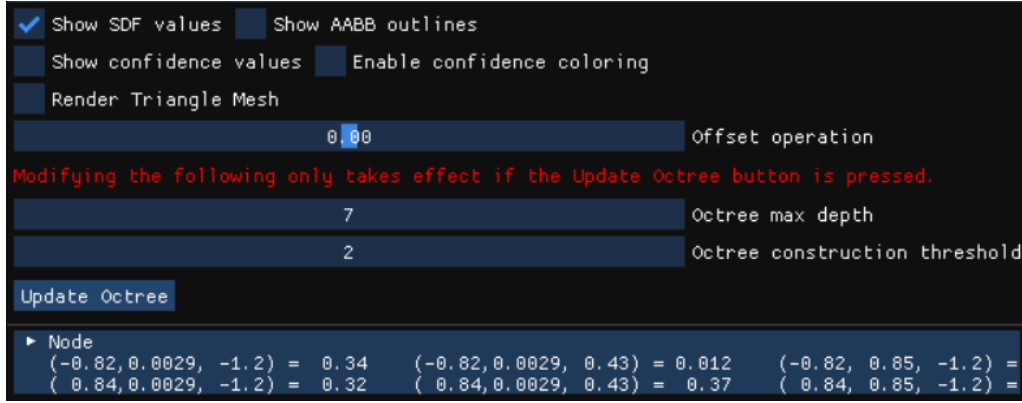


Figure 2.7: Expanded view of the Node List tab.

In the top part of the tab, there are checkboxes which enable different visualization options of objects in the software. It is possible to combine them to display several additional data to the triangle mesh rendered in the *Main Window*.

Show SDF values and *show confidence values* options write SDF and confidence values of corners of octree nodes next to their correspondent location in the simulated environment. It is helpful to see such key values next to the SDF representation to

be able to visualize if they are coherent (eg. smaller SDF values are closer to the surface, positive SDFs are outside).

The *show AABB outlines* checkbox allows an essential option of visualizing the axis-aligned bounding box of any octree node. It is especially important to understand the 3D representation of an octree node and the integration between the octree and the final rendered result. This feature also allows an easier navigation inside the object.

By default, the coloring of *AABB* lines corresponds to a debugging visualization of the validity of SDFs between vertices. When green, the line segment between two vertices indicates that they are of same sign, asserted by the relation further discussed in Section 8. When a segment line is yellow, it indicates that two vertices must have opposing signs, also due to the same relation. A white-colored line indicates that no assurances could be made and the signs were resolved by segment checking (see Section 5.3.4).

When the option *enable confidence color* is activated, the color code for the *AABB* changes to reflect visual debugging in relation to the confidence rather than the SDF. For the confidence color code, positive confidences (indicate positive signs of the SDF) are portrayed as shades of green. The intensity depends on the confidence value between vertices. Negative confidence values are shown as shades of blue, which also have their intensity dependent on their value. Figure 2.8 shows a visualization of a tetrahedron with the confidence coloring activated to show octree nodes both outside and inside.

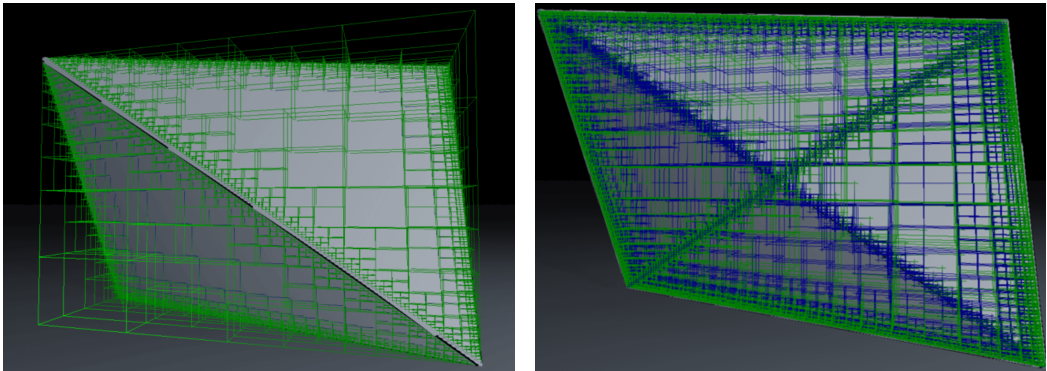


Figure 2.8: Outside (left) and inside (right) octree nodes of mesh *tetrahedron.obj*.

The *render triangle mesh* option uses the most common rendering technique

(rasterization) to render every triangle of the input mesh. It serves as a comparison standard of the SDF approximation model proposed by this thesis. It is possible to visualize the regions in the triangle mesh in which the the SDF approximation underestimates (painted in green), and overestimates (highlighted in red) the actual representation. Areas where the rasterization is not present show the accurate SDF estimation of the object.

Table 2.2 provides a summary of the visualization options, referencing example figures (2.9) for each option.

Visualization option	Brief description	Subfigure of Figure 2.9
Show SDF values	Displays SDF values of corners of octree nodes.	b), c)
Show confidence values	Displays confidence values of corners of octree nodes.	b), c)
Show AABB outlines	Draws the AABB lines for octree nodes.	a), b), c)
Enable confidence coloring	Switches coloring of AABB lines to reflect the confidence color code.	a)
Render Triangle Mesh	Renders original triangles and overlaps them with SDF estimation. Stress colors are applied for contrast.	d)

Table 2.2: Visualization options displayed as checkboxes in the *Node List* tab.

The slider bar after the checkboxes of visualization options modifies the parameter for the implementation of the *offset* operation, one of the important results of this thesis (see Section 5.5). Such slider allows both negative and positive offsets, which are shown in real-time.

Two additional sliders are implemented in the *Node List* tab which are responsible for modifying parameters used during the octree construction. *Octree max depth* and *Octree construction threshold* are applied once the *Update Octree* button, located right below the sliders, is pressed, triggering a full reconstruction of the octree

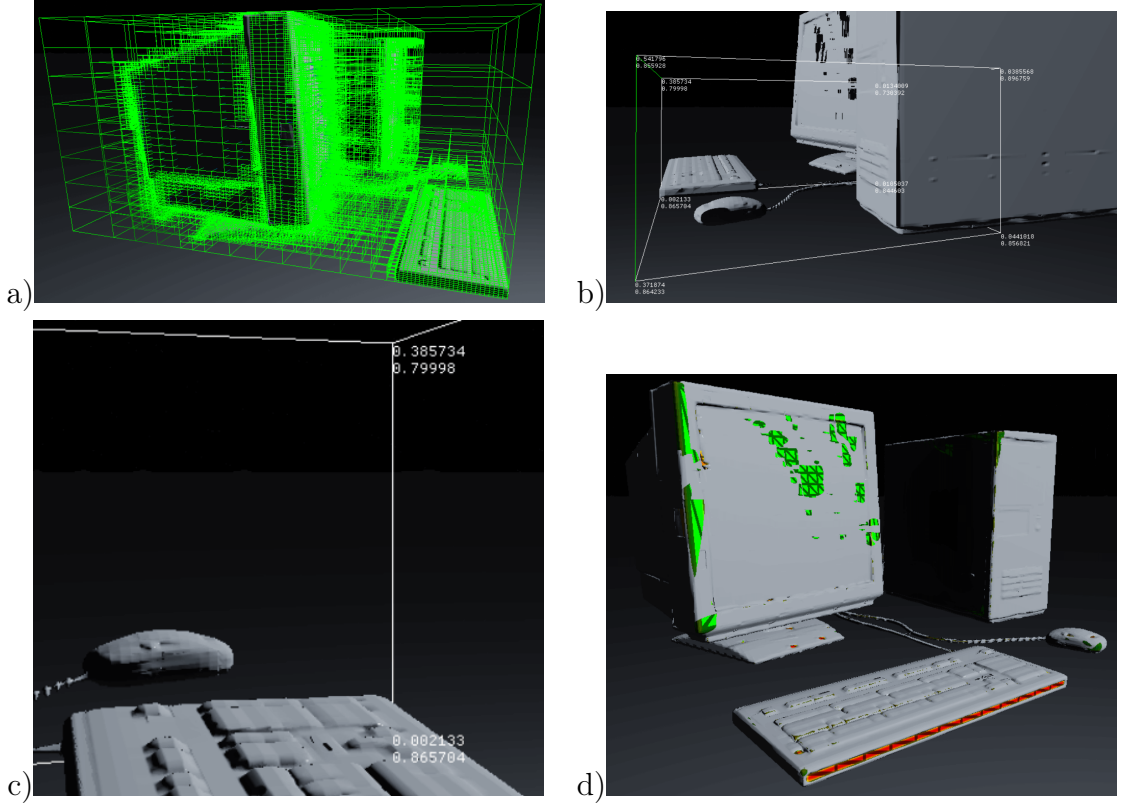


Figure 2.9: Visualization options applied to mesh *computer.obj*.

structure. Depending on the size of the input mesh and the chosen parameters, the user needs to wait until the new octree is generated.

The *max depth* slider alters the maximum height of the generated octree, limiting the level of details seen in the final rendered object. The *construction threshold* modifies the minimum number of triangles present in a leaf node in the octree. This parameter is essential to the *octree construction algorithm* (2), and changes the accuracy of the SDF estimation.

Lastly, a recursive list of octree nodes is shown at the end of the tab. It lists all the nodes of the loaded input mesh, distinguishing between non-leaf and leaf, showing its position in the three-dimensional space and its SDF. The user selects a node by clicking on it. Once selected, the aforementioned visualization options will be applied and available for such node in the *Main Window*. The same options are also applied for the node to which the user hovers the mouse upon, allowing a fast preview. By clicking on the selected node, or in another node, the user cancels or changes its selection, respectively. Figure 2.10 demonstrates an expanded list of nodes with a selected node and another node being hovered at the same time.

▼ Node	(1, 1, 1) = 0.86	(1, 1, 4) = -0	(1, 3
	(2, 1, 1) = -0	(2, 1, 4) = 0.86	(2, 3
▼ Node	(1, 1, 1) = 0.86	(1, 1, 2.5) = 0.43	(1, 1
	(1.5, 1, 1) = 0.43	(1.5, 1, 2.5) = 0	(1.5, 1
Leaf	(1, 1, 1) = 0.86	(1, 1, 1.8) = 0.64	(1, 1
	(1.3, 1, 1) = 0.64	(1.3, 1, 1.8) = 0.43	(1.3, 1
Leaf	(1, 1, 1.8) = 0.64	(1, 1, 2.5) = 0.43	(1, 1
	(1.3, 1, 1.8) = 0.43	(1.3, 1, 2.5) = 0.21	(1.3, 1

Figure 2.10: Expanded view of the *Node List* tab with selected (dark blue) and hovered (light blue) nodes.

It is also possible to apply visualization options to more than two nodes simultaneously. To do that, the user must hold either the *Shift* or *Ctrl* keys after selecting a node. *Shift* will apply the options to the immediate child nodes of the selected node (if non-leaf). *Ctrl* will apply to all child nodes of the selected node, recursively. Therefore, to select an entire octree and to apply different visualization options, the user can select the *root* node and hold the *Ctrl* key.

2.3.2 Automated Testing

The *automated testing* tab shows a testing plan for different aspects of the project. It allows the user to apply the automated test cases with a single click. Its interface is shown in Figure 2.11.

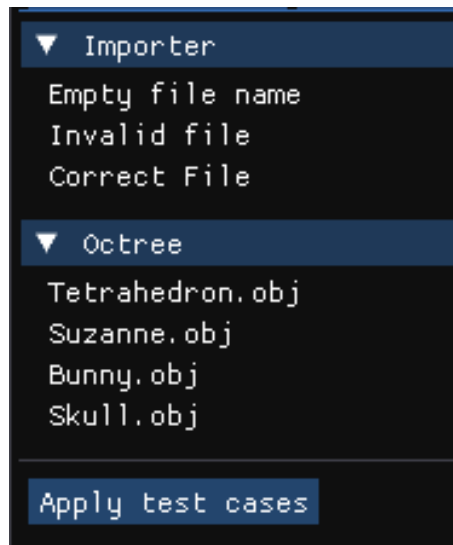


Figure 2.11: Expanded view of the *Automated Test* tab.

To interact with the interface, the user can click the *Apply test cases* button to start the test routine. As soon as each test is completed, its result will show if it was successful or not by changing the color of its name from white (ongoing) to green (success) or red (failure). The user is prompted with a progress bar to estimate the remaining test cases, as seen in Figure 2.12. The user cannot initiate a new test routine while one is in progress, and the interface handles this exception by hiding the button when a sequence of tests is being applied.

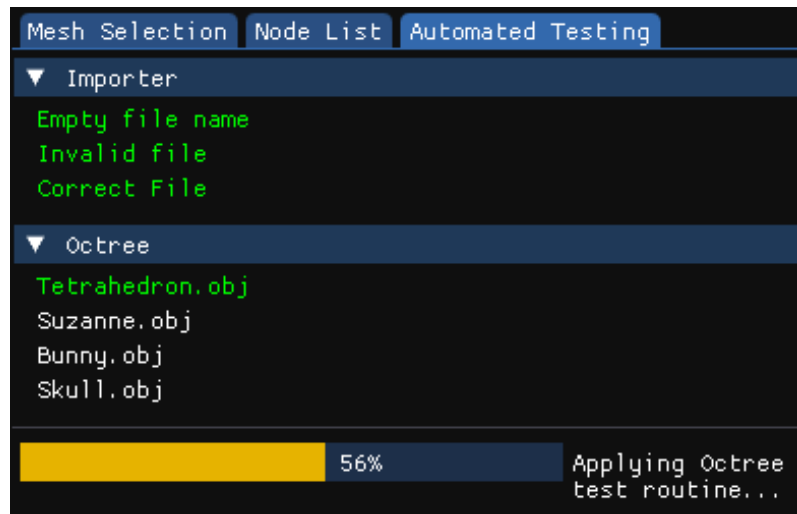


Figure 2.12: View of the *Automated Test* tab during the test routine.

The automated tests are covered as part of the testing scenarios presented in Section 5.4.

Chapter 3

Theoretical Background

This chapter provides theoretical background for the discussion of algorithms as well as implementation and testing of the proposed software. Concepts of sphere tracing (3.1), (signed) distance functions (3.1.1, 3.1.2), Lipschitz continuity (3.1.3), ray-surface intersections (3.1.4), and the octree data structure (3.2) are presented.

3.1 Sphere Tracing

Sphere Tracing is a technique which capitalizes on functions that return the distance to their implicit surfaces (3.1.2), intersecting rays (3.1.4) and creating a final image from these interactions [2]. It calculates ray-surface intersections, thus being a form of ray casting. It uses signed distance functions to represent objects, which allows important consequences explained in Section 3.1.2. Definitions are adapted from Hart [2] and Bálint [1].

3.1.1 Distance Functions

Definition 1. *The point-to-set distance defines the distance from a point $x \in \mathbb{R}^3$ to a set $A \subseteq \mathbb{R}^3$ as the distance from x to the closest point in A*

$$d(x, A) = \inf_{y \in A} \|x - y\|_2. \quad (3.1)$$

From Definition 1, a set A is inferred [2]. It is possible to define a function which returns the distance from a point to a surface, as seen in Definition 2.

Definition 2. $f : \mathbb{R}^3 \rightarrow [0, +\infty)$ is a distance function if

$$f(p) = d(p, \{f \equiv 0\}) \quad (\forall p \in \mathbb{R}^3) \quad (3.2)$$

where $\{f \equiv 0\} = \{x \in \mathbb{R}^3 \mid f(x) = 0\}$ is the level set surface of a set.

3.1.2 Signed Distance Functions

Definition 3. If $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ is continuous and $\|f\|$ is a distance function, then f is a Signed Distance Function (SDF).

An important aspect of SDFs is their continuity, since it allows the distance estimation of any given point $x \in \mathbb{R}^3$. Moreover, for the use case of this thesis, they not only provide the distance from a point to a surface, but also include the information of whether the given point is inside ($f(p) \leq 0$) or outside ($f(p) > 0$) the primitive whose surface is being taken into consideration. Both continuity and inside/outside checking are essential information needed for several algorithms included in Chapter 4.

As shown in Equation 3.2, distance functions depend on the primitive to return the distance of a given point p to its bound. For some primitives such as spheres, this function can be easily defined (e.g. for a unit sphere and $p \in \mathbb{R}^3$, $f(p) = \|p\|_2 - 1$). However, for complex shapes it is not simple to find such equation. This limitation can be managed by using a divide-and-conquer strategy, in which a complex shape is divided in multiple well-defined shapes, though the algorithm complexity and computing requirements may grow substantially. Such strategy is discussed in Section 3.2.

For the software solution proposed in this thesis, a discretization of signed distance functions is used. The result of this process is called *Signed Distance Fields*, which holds SDF values in a sparse way. By using interpolation, an approximation of SDF values can be recomputed at any point. In practice, signed distance fields are stored as an octree (3.2). Any given point in the \mathbb{R}^3 can be interpolated from SDF values stored in vertices of octree nodes to regain an approximation of the original signed distance value, as seen in Section 4.7.

3.1.3 Lipschitz continuity

Definition 4. Let the function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ be arbitrary. We define the set of Lipschitz constants as

$$\text{Lip } f := \{L > 0 : \forall x, y \in \mathbb{R}^3 : \|f(x) - f(y)\| \leq L \cdot d(x, y)\}. \quad (3.3)$$

The function f is Lipschitz continuous if $\text{Lip } f \neq \emptyset$.

Corollary 1. Every signed distance function is Lipschitz continuous and their smallest Lipschitz constant is 1. Formally:

$$\forall f : \mathbb{R}^3 \rightarrow \mathbb{R} \text{ SDF} : \inf \text{Lip } f = \min \text{Lip } f = 1. \quad (3.4)$$

Corollary 1 presents an important characteristic of signed distance functions. It is used in the octree painting algorithm to help categorize inside/outside partitions (Section 4.4.2). Note that dividing a function by its Lipschitz constant yields a SDF lower bound, crucial for creating SDF estimates.

3.1.4 Ray-Surface Intersection

The ray-surface intersection is achieved by finding the least positive root of $F(t) = f(\mathbf{p}_0 + t \cdot \mathbf{v}_0)$, in which p_0 represents the starting point, and v_0 the normalized direction of the ray. The smallest positive root of $F(t)$ can be found through the recurrence sequence defined by

$$t_{i+1} = t_i + F(t_i), \quad (3.5)$$

with initial point $t_0 = p_0$, $F(t_i)$ being the signed distance function value for a ray of length t_i . The sequence in Equation 3.5 converges if and only if the ray intersects the implicit surface [2].

The intersection process is better illustrated in Figure 3.1, in which there is an intersection of the ray starting at p_0 and direction v_0 with the surface. Dashed-lined circles represent steps taken by every iteration of the aforementioned recurrence sequence, stopping and converging once the intersection is completed (once ray r first intersects surface).

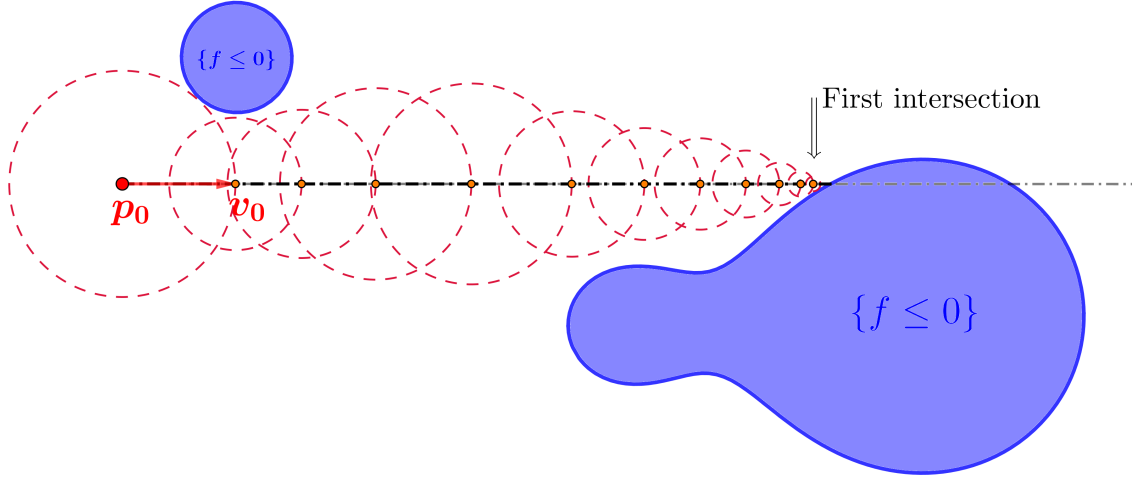


Figure 3.1: Illustration of a ray-surface intersection.

3.2 Octree

As mentioned in Section 3.1.2, one way of overcoming the need of formulation of implicit equations for complex shapes is by using parts of the original shape which can be described by well-known equations. Following this idea, it is necessary to choose an efficient way of subdividing the surface. The octree data structure is the option chosen for this software.

Octree is a data structure that subdivides the three dimensional space in eight equal sections in a recursive manner. It is a tree in which every node has eight children. A visual representation can be seen in the Figure 3.2. In such figure, the spacial subdivision of a cube is exemplified in the left side, while the tree diagram, showing eight children per node, is drawn to its right one.

Amongst the benefits of utilizing octrees, the high level of regularity and the considerably lower memory footprint are especially important for the rendering application subject of this thesis.

Seeing that the number of children per node is fixed, it is easier to serialize the octree structure, crucial step needed for working with graphical processing units (GPUs). Furthermore, since it is a tree, in the worst case, a low $\log(h)$ complexity is applied for accessing signed distance values stored at the leaves, where h is the height of the tree.

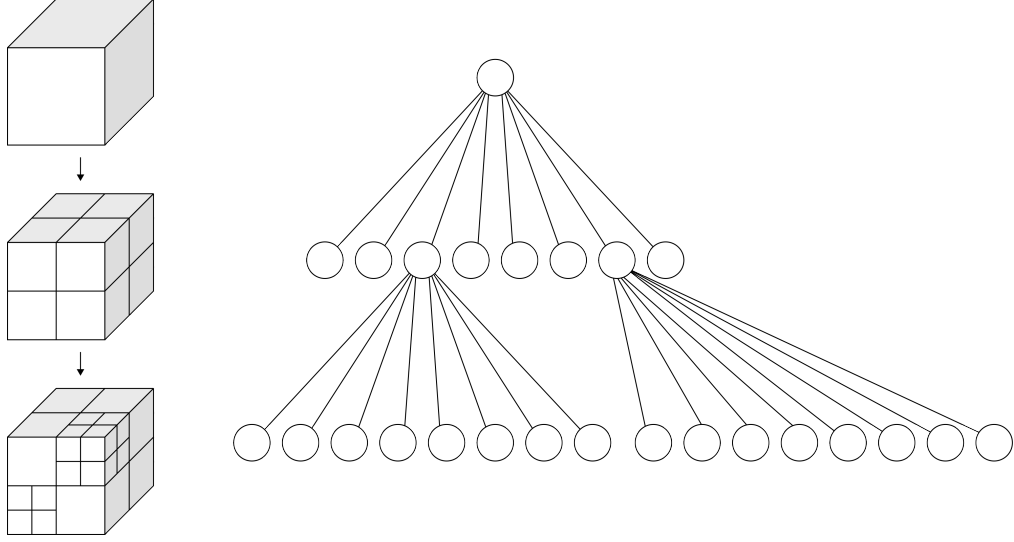


Figure 3.2: Illustration of an octree data structure.

In the proposed application, since a triangle mesh given as input consists of well-defined triangle primitives, the octree saves substantial memory by not storing all triangle information. Instead, it stores the distance from a corner of a leaf node to the closest triangle, making the model highly scalable for real-time visualization. In other words, the execution time of scenes, made from triangles, depends on the number of levels in the octree, not in the number of triangles in the mesh.

Chapter 4

Algorithms

This chapter presents and explains algorithms used in the implementation (Chapter 5). Sections are organized in the natural order of execution of the proposed software solution. It starts with a generic base function that traverses all nodes in a tree and applies routines (Section 4.1), and explains the construction (Section 4.2) of an octree. An auxiliary graph is defined (Section 4.3) as part of the signed distance function computation (Section 4.4). Furthermore, the optimization (Section 4.5) and serialization (Section 4.6) of an octree, and the lookup of a serialized octree (Section 4.7) are presented. For better visualization, a workflow diagram of the most important algorithms discussed in this chapter is displayed in Figure 4.1.

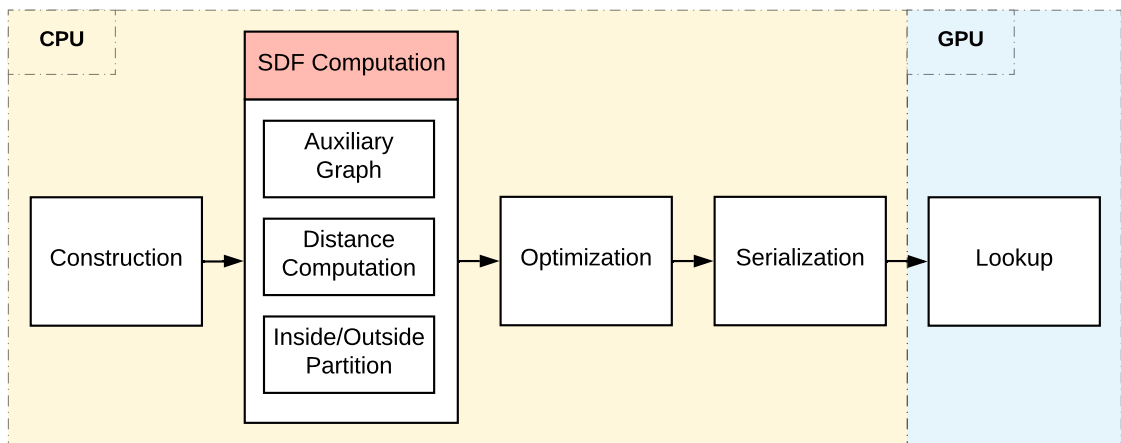


Figure 4.1: Workflow of octree algorithms following the natural order of execution.

The diagram partitions the algorithms into processing units (CPU or GPU) intended for implementation.

4.1 Tree traversal

The following algorithm is responsible for applying functions in both pre-order (**f1**) and post-order (**f2**) to every node for a (sub)tree whose **root** is given as a parameter of the recursive routine described in Algorithm 1.

Algorithm 1 Tree Traversal

Func TreeTrav(*node*, *f1*, *f2*)

```

1: f1(node)                                     //Pre-order function call
2: Let A be the set of children of node
3: for child  $\in$  A do
4:   TreeTrav(child, f1, f2)                 //Recursive call
5: end for
6: f2(node)                                     //Post-order function call

```

Algorithm 1 constitutes a generic function, which works for any kind of tree, serving as base function for complex algorithms applied to octrees that follow in this chapter. **TreeTrav** ends once all nodes have been visited. A leaf node does not contain any children, thus being the limit for the recursive call.

4.2 Octree Construction

The Octree construction utilizes the *generic tree traversal algorithm* (1) to apply a function whose purpose is to build new nodes until a condition is reached. The versatility of Algorithm 1 makes it possible that the traversal algorithm, generally used to visit or update nodes, is applied for subdividing the **root** node recursively until a full sized octree is formed. A visualization of the generated octree which describes the mesh in *Skull.obj* (rendered by our software) is shown in Figure 4.2.

Preceding the call of the *octree construction algorithm* (2), the **root** node of the octree has to be created large enough to fit all triangles of the scene to be rendered. Once the **root** is created and initialized, the aforementioned tree traversal algorithm is called with **root** as starting node, the construction algorithm (2) as the pre-order function, in addition to NULL as post-order function, since it is not used.

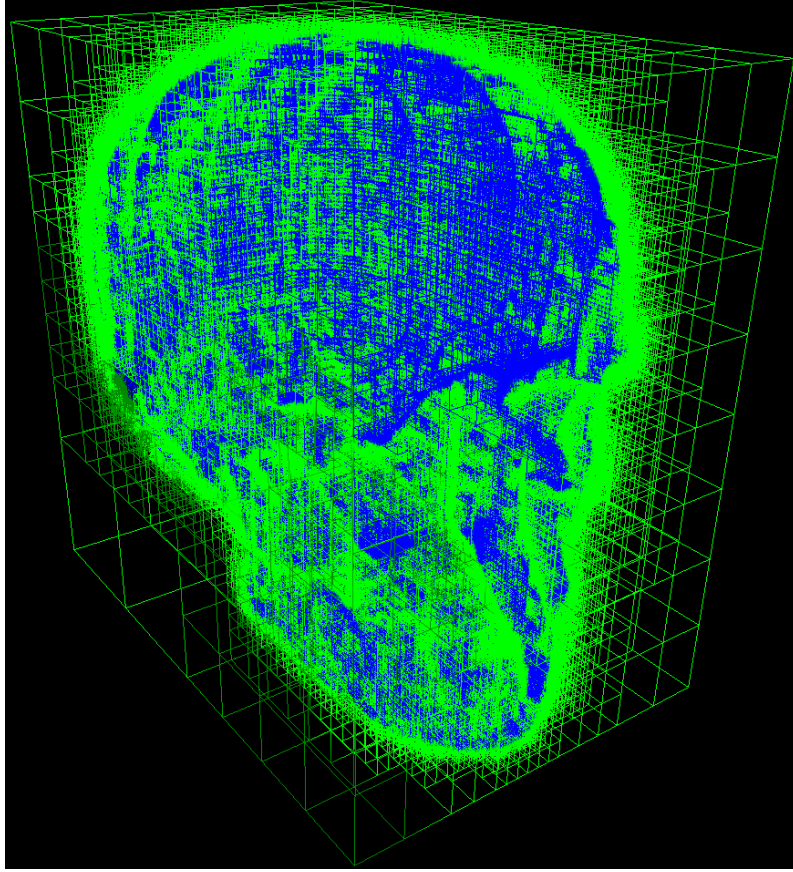


Figure 4.2: Visualization of octree describing the mesh contained in *Skull.obj*.

The octree construction algorithm (2) creates eight children for a node given as its only parameter and it adds them to the octree. It repeats this process recursively until the maximum depth of the octree (external parameter that can be fixed for simplicity) is reached or there is only one triangle in a node.

The maximum depth constraint is set to prevent the octree of reaching an excessive height when encountering meshes with large number of triangles or whose distribution of triangles include multiple sparse clusters of primitives. In the latter, an algorithm to separate the original mesh into multiple meshes, one per cluster, could greatly increase performance, given that it would reduce both the number of intersection tests and the number of nodes with no triangles inside.

The constraint which establishes the minimum of two triangles per node is placed to avoid unnecessary node subdivisions. Dividing a node that has one or zero triangles would contradict one of the goals of implementing an octree: to reduce the number of intersection tests between rays and primitives in a scene. This threshold for the minimum number of triangles in a leaf node can be set as a parameter to

better adapt to specific meshes, as seen in Sections 2.3.1 and 5.3.3.

Even though dividing a node with only two or three triangles might still prove inefficient by generating more nodes than necessary, there are algorithms which can revert many of the disadvantageous subdivisions. Thus, the default upper limit of one triangle per node in the octree creation was chosen to avoid trivial subdivisions, since the data structure will be optimized for nontrivial cases in a later stage (see Section 4.5).

For each child node, triangle-cube intersections are performed to determine which primitives are incorporated into it. Considering that there are multiple ways to solve the triangle-cube intersection subproblem, the strategy used in the software solution constitutes an implementation choice, being better explained in Section 5.3.3.

Algorithm 2 Octree Construction Algorithm

Funct Construct(*node*)

```

1: Let depth be the depth of node
2: Let nodetri be the triangles in node
3: Let maxdepth be the maximum depth of the octree
4: if depth < maxdepth and |nodetri| > 1 then                                //node is nonleaf
5:   for i = 1, ..., 8 do
6:     Let childnodei be a new octree node
7:     Let Ti be an empty set of triangles of childnodei
8:     for triangle ∈ nodetri do                                                //Triangle-cube intersection tests
9:       if triangle intersects childnodei then
10:        Ti := Ti ∪ {triangle}
11:      end if
12:    end for
13:  end for
14: end if

```

4.3 Auxiliary Graph

This section defines an auxiliary graph (4.3.1), it explains the construction for this structure (4.3.2), and it presents the closest triangles subproblem (4.3.3).

4.3.1 Definition

The auxiliary graph stores distance and *confidence values* for every *unique* vertex of the octree. For every leaf in the octree, there are exactly eight vertices, following the three-dimensional visual representation in which a cube (octree node) has eight corners.

A *unique* vertex is a single vertex instance referenced by one or more octree nodes. Scenarios that exemplify the existence of more than one octree node associated with the same vertex include neighboring octree nodes in the same level, as well as parent and children nodes, which are placed in different levels of the octree structure.

The importance of building a separate graph for vertices instead of using the already allocated octree graph is optimizing the SDF computation. This optimization comes from the *uniqueness* of the vertices. Since the octree is a regular structure, many vertices are shared by octree nodes of different heights. By processing only distinct vertices, the algorithm becomes significantly faster, it avoids redundancy, and it improves maintainability while using additional memory as a trade-off.

Every vertex in the auxiliary graph holds a distance value calculated in Algorithm 5, which represents the distance from such vertex to the closest primitive in the input mesh. The vertex also stores its confidence level calculated in Section 4.4.2, responsible for deciding if the vertex is inside or outside the input mesh.

4.3.2 Construction

The construction of the auxiliary graph is described in Algorithm 3. It receives two parameters: \mathbf{G} , which is a reference to an auxiliary graph, and \mathbf{O} , a reference to an octree. Following the definition of auxiliary graph, eight vertices are created for every leaf node in the octree. Then, they are connected by edges to its neighbors of the same leaf node according to axes \mathbf{X} , \mathbf{Y} , and \mathbf{Z} . The last edge links vertices in a diagonal fashion to decrease the length of the path between such vertices. The diagonal link significantly improves the outcome of Algorithm 6.

Algorithm 3 Auxiliary Graph Construction

Funct AuxGCTr(G, O)

```

1: Let  $L$  be the set of leaf nodes in  $O$                                 //  $O$  represents an octree
2: for  $node \in L$  do
3:   for  $i = 1, \dots, 8$  do
4:     Create  $v_i$  new vertex in  $G$                                 //  $G$  represents an auxiliary graph
5:     Create edges with correspondent vertices according to axes  $X, Y, Z$ 
6:     Create edge with correspondent vertex according to the cube diagonal
7:   end for
8: end for

```

To obtain the set of leaf nodes in the octree, needed in Algorithm 3, one could use Algorithm 1 to traverse the whole octree, executing the tasks only when the node is a leaf. However, the creation of vertices in the auxiliary graph can be better optimized by incorporating Algorithm 3 into Algorithm 2. It can be done by adding an **else** clause to the **if** statement of line 4 of Algorithm 2. With the simple addition of such clause, it would be possible to process the non-leaf nodes for the *octree construction algorithm*, and the leaf nodes for the construction of the auxiliary graph. This way, both graphs are efficiently constructed simultaneously in a single traversal. We chose to use the aforementioned strategy in the implementation of the proposed software solution.

4.3.3 Closest Triangles Subproblem

The closest triangle subproblem queries all triangles (organized in an octree) in the proximity of a vertex of the auxiliary graph. To achieve that, it creates new points $p \in \mathbb{R}^3$ in the proximity of **vert** and searches the octree for triangles containing such points. The *closest triangles algorithm* (4) takes two parameters: **vert**, representing a vertex in the auxiliary graph, and **O**, which references an octree to be searched. Such procedure is used as part of the distance values computation (Section 4.4.1).

The algorithm starts by calculating the smallest step (**step**) taken in an octree lookup to not skip a leaf node of maximum depth, which, for an octree, can be

trivially calculated. The diagonal of the smallest leaf is $\|\text{rootdiag}\| \cdot \frac{1}{2^{\text{maxdepth}}}$, where rootdiag is the diagonal of the `root` node.

Then, a set of triangles `T` is created to allocate the closest triangles of the given vertex `vert`. Computing such triangles is done by searching the octree for leaves storing point `p`, which is computed by moving the `vert` position by vectors of size $\|\text{step}\|$ and directions of cube diagonals. The idea of this lookup is to access the neighboring nodes by querying diagonally shifted positions.

In case the searched leaf has no triangles, the algorithm adds the triangles from its parent to set `T`, since the closest primitive to `vert` might be located in its neighbor. Otherwise, it adds the triangles from the leaf to set `T`. In the end, `T`, returned by the algorithm, contains all triangles from the nodes in the proximity of `vert`.

Algorithm 4 Closest Triangles Algorithm

Funct ClosestTri(*vert*, *O*)

```

1: Let diag be the diagonal of O                                // O represents an octree
2: Let maxdepth be the maximum depth of O
3: Let step =  $\|\text{diag}\| \cdot (\frac{1}{2})^{\text{maxdepth}}$                     // Smallest step in O
4: Let cubeDiagonals be the set of normalized vectors with directions equal to the
   diagonals of the cube
5: Let T =  $\emptyset$ 
6: for diagonali  $\in$  cubeDiagonals do
7:   Let coordv be the coordinates of vertex
8:   Let p = coordv + diagonali · step
9:   Let leaf  $\in$  O | p  $\in$  leaf                                // Finding leaf that contains p
10:  Let leaftri be the triangles in leaf
11:  if  $\|\text{leaftri}\| = 0$  then                                    // If leaf is empty, add parent's triangles
12:    Let parenttri be the set of triangles of the parent node of leaf
13:    T := T  $\cup$  parenttri
14:  else
15:    T := T  $\cup$  leaftri
16:  end if
17: end for
18: return T

```

4.4 SDF Computation

The SDF computation process encompasses the distance value computation (4.4.1) and inside/outside partitioning (4.4.2), described in this section.

4.4.1 Distance Values Computation

The computation of the distance value for a vertex is done by executing a *minimum search* of the distances between said vertex and all the primitives around it. This can be achieved by comparing every vertex in the auxiliary graph (Section 4.3) to all the primitives in the scene to be rendered.

However, it is not efficient to take into consideration all the primitives in every distance calculation. Since it constitutes of a *minimum search* subproblem, it is sufficient to ignore the ones furthest away from a vertex. First, Algorithm 5 uses Algorithm 4 to query the closest triangles to a vertex. Then, it calculates the distance value of such vertex by taking the minimum of the distances to the previously queried triangles.

The algorithm receives two parameters: G , a reference to an auxiliary graph, and O , a reference to an octree. The procedure iterates through all vertices in G . For every iteration, it queries the closest triangles to a vertex, storing them in T . Then, a simple minimum search is performed to determine the distance value of such vertex to the closest triangle.

Algorithm 5 Distance Values Computation

Func DFComp(G, O)

```

1: Let  $V$  be the set of vertices of  $G$            //  $G$  represents an auxiliary graph
2: for  $vertex \in V$  do
3:   Let  $T = ClosestTri(vertex, O)$            //  $O$  represents an octree
4:   Let  $df$  be the distance function of  $vertex$ 
5:   for  $triangle \in T$  do // Minimum search of distance values of triangles in  $T$ 
6:     Let  $trisd$  be the SDF of  $coordv$  and  $triangle$ 
7:      $df = min(df, trisd)$ 
8:   end for
9: end for

```

Even though a signed distance value can be calculated for a given point $p \in \mathbb{R}^3$ and a triangle, it is not possible to determine whether such point would receive negative (inside) or positive (outside) value in relation to the whole mesh, only to the single triangle taken by the SDF. This is the reason why it is called a distance value and its sign is not considered. The inside/outside partition of the values in the auxiliary graph is discussed in Section 4.4.2.

4.4.2 Inside/Outside Partition

Deciding the sign of distance functions for a complex mesh is not trivial. The software solution relies on an iterative algorithm which uses mathematical consequences of SDFs (3.1.3) to set signs of some distance functions of vertices with certainty, while handling the rest by using a weighted average of its neighbors. A visualization of nodes of opposing signs of the SDF representation of the mesh in *Skull.obj* can be seen in Figure 4.3.

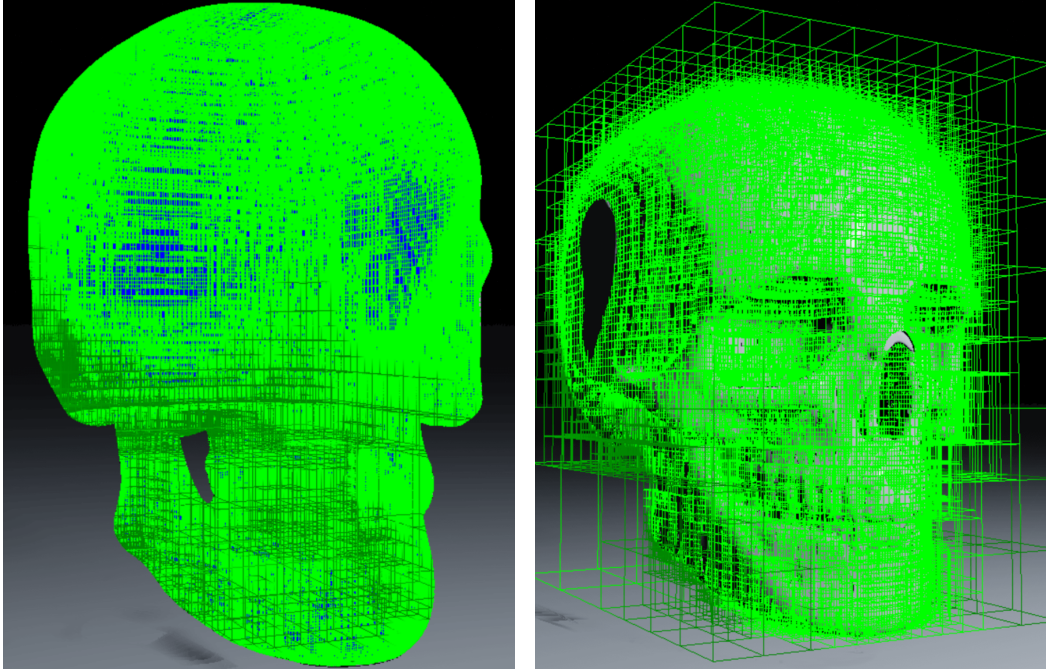


Figure 4.3: Octree of mesh described in *Skull.obj* with inside (left) and outside (right) nodes highlighted.

There are two main attributes of vertices essential for this algorithm: **confidence** and **status**. Confidence values are of the form $-1.0 \leq \text{v.confidence} \leq 1.0$, a

negative confidence meaning the vertex appears inside the mesh of triangles, and a positive being the opposite. It varies in a range to allow estimation of uncertain vertices. A visualization of the inside and outside of mesh *Tetrahedron.obj* has been presented in Section 2.3.1, and can be seen in Figure 2.8.

The status attribute represents the state of each vertex going through the robust algorithm. It is of the form $v.status \in \{ \text{unprocessed}, \text{processed} \}$. All vertices are initialized with an **unprocessed** status, and change it to **processed** when the sign of its distance function (and confidence) has been determined.

Algorithm 6 starts by creating an empty set V which will contain vertices of **unprocessed** status. Next, it processes an initial vertex $v_0 \in G$, guaranteed to be outside the triangle mesh, by assigning it the maximum confidence of 1.0 and a **processed** status. Then, it updates the neighbors of v_0 (all vertices connected to such vertex) and the set V . Lastly, it handles all other vertices in G by repeating the procedure described in Algorithm 7 until $V = \emptyset$. Since G is a connected graph, it is guaranteed that Algorithm 6 reaches all vertices in G .

Algorithm 6 Inside/Outside Partition Algorithm

Funct Partition(G)

```

1: Let  $V = \emptyset$ 
2: Let  $v_0$  be an outside vertex in  $G$ 
3:  $v_0.confidence = 1.0$ 
4:  $v_0.status = \text{processed}$ 
5: for  $n \in v_0.neighbors$  do //  $v_0.neighbors$  is the set of vertices connected to  $v_0$ 
6:   Update confidence of  $n$ 
7:    $V = V \cup \{n\}$ 
8: end for
9: while  $V \neq \emptyset$  do // Ends when all vertices are processed
10:   PartGraph( $V$ )
11: end while
```

Algorithm 7 receives a set V of vertices as a parameter. A vertex v of maximum confidence is extracted from the given set V and marked as **processed**. Then, following the steps of Algorithm 6, it updates the confidence of neighbors of v as well as

the set. Since the maximum search inside a set can be costly, implementation choices can be made for the data structure of V to increase efficiency of this operation, as seen in Section 5.3.4.

Algorithm 7 Partition Graph Auxiliar Algorithm

Funct PartGraph(V)

```

1: Let  $v = \max(V)$                                 //Retrieve vertex with maximum confidence
2:  $V = V \setminus \{v\}$ 
3:  $v.status = \text{processed}$ 
4: for  $n \in v.neighbors$  do           //v.neighbors is the set of vertices connected to v
5:   if  $n.status = \text{unprocessed}$  then
6:     Update confidence of  $n$ 
7:      $V = V \cup \{n\}$ 
8:   else
9:     if  $n \in V$  then                   //n is being processed already
10:      Update confidence of  $n$ 
11:   end if
12: end if
13: end for

```

On both Algorithms 6 and 7, the confidence of a vertex is calculated. In the software solution, a confidence of a vertex is defined as a weighted average of the confidence of its neighbors. In such average, the weight is composed by a floating number dependent of a vertex and its neighbor having the same or opposite sign, as described in Algorithm 8.

Has Same Sign Algorithm (8) receives two vertices (\mathbf{va} , \mathbf{vb}) as parameters and returns a floating point number $-1.0 \leq \mathbf{f} \leq 1.0$. First, it uses the Corollary 1 (Section 3.1.3) to check whether the *Lipschitz continuity* can be sufficient to define the sign of the two vertices. It checks the relation between their SDFs and their distance in the three-dimensional space ($\|f(va) - (-f(vb))\| > \|va - vb\|_2$). If it is true, the two vertices must be of same sign, and the algorithm can return with maximum certainty ($\mathbf{f} = 1$), since the aforementioned relation is proved for SDFs.

On the other hand, when the calculation is not conclusive, a segment check is performed on all triangles in the proximity of both vertices. Such operation aggre-

gates the nearby leaves and builds the union of their triangles. Then, a counting is performed on the number of intersections between the line segment that connects the two vertices (\mathbf{rab}) and the aforementioned triangles.

If the counting results in an odd number, it means that the vertices are in opposing sides of the mesh, thus it can be implied that they have opposite distance function signs. For an even number of intersections, the vertices share the same SDF sign. Since this operation is not as robust as using Definition 4 and Corollary 1, the returned weights are slightly smaller in absolute value ($\|\mathbf{f}\| = 0.99$).

Even though the segment check is robust, in practice, it is still best used in closed meshes, which is a limitation for the proposed software. For open meshes, there are algorithms [4] which can deduce and add missing triangles resulting in a closed mesh that can be utilized by our software.

Algorithm 8 Has Same Sign Algorithm

Func HasSameSign(va, vb)

```

1: if  $\|f(va) - (-f(vb))\| > \|va - vb\|_2$  then
2:   return 1           //Relation using Definition 4 was enough to decide the sign
3: end if
4: Let  $count = 0$ 
5: Let  $rab$  be the line segment between  $va$  and  $vb$ 
6: Let  $T = va.leaves \cup vb.leaves$  // $v.leaves$  is the set of leaves that share vertex  $v$ 
7: for  $triangle \in L.triangles$  do // $L.triangles$  is the set of triangles of set  $L$ 
8:   if  $triangle$  intersects  $rab$  then
9:      $count = count + 1$ 
10:  end if
11: end for
12: if  $count$  is even then
13:   return 0.99
14: end if
15: return  $-0.99$            //count is odd

```

4.5 Octree Optimization

The optimization of a tree can have different approaches, depending on the structural limitations of the resulting object. A tree can be optimized by maintaining the same number of nodes and reorganizing its components in a process known as *tree balancing*, first introduced in *AVL Trees* [6]. However, in the case of our software solution, the resulting tree needs to maintain the structure of eight children per node and its construction is tied to the positioning of triangles in a scene. Thus, it is not possible to balance the octree without changing the input scene.

A way of optimizing an octree, in our case, arises from its practical use. It is possible to delete unnecessary leaf nodes, since an octree serves as structure for *signed distance fields*, and, for $x \in \mathbb{R}^3$, a SDF value can be calculated by using interpolation (see Section 3.1.2). A leaf node is considered unnecessary if its SDF values can be computed, with a small error tolerance, by interpolating the SDF values of its parent node. The result is a much smaller octree classified as an *Adaptively Sampled Distance Field* [7]. A visualization of the effect of the optimization process can be seen in Figure 4.4.

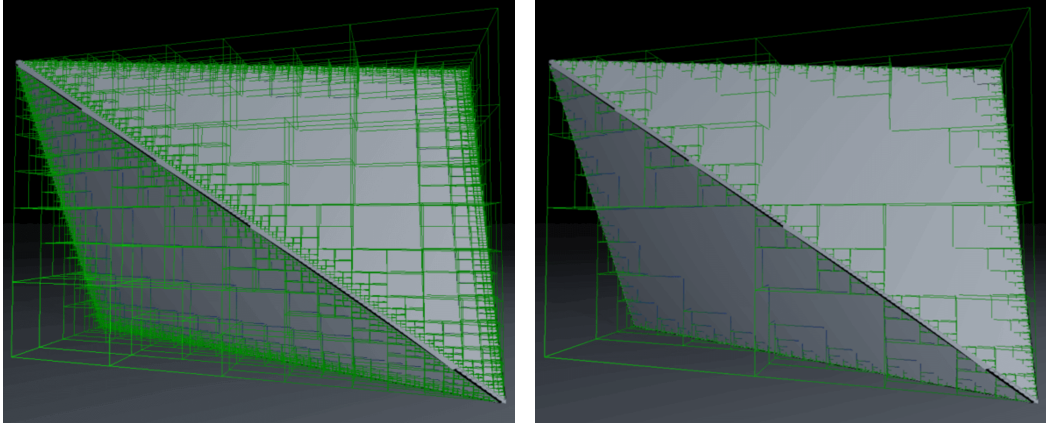


Figure 4.4: Octree of mesh described in *Tetrahedron.obj* before (left) and after (right) its optimization process. It features a reduction of 50% on the number of octree nodes, in this case.

Algorithm 9 describes how an octree node given as a parameter is optimized. It starts by ignoring such parameter if the subtree whose `root` node is `nd` is of height different than one ($h \neq 1$). It is equivalent to state that the height cannot be zero or bigger than one, since $h \in \mathbb{N}^*$. These cases are ignored since we can only remove

nodes that are parent to eight leaves. It is not possible to remove children nodes if \mathbf{nd} is already a leaf ($\mathbf{h} = 0$). Furthermore, it is not possible to interpolate SDF values of non-leaf nodes, given that only leaf nodes store SDF values ($\mathbf{h} > 1$).

Upon confirmation that the shape of \mathbf{nd} is a subtree of height one, Algorithm 9 calls *error estimation algorithm* (10) to calculate *absolute and relative errors*. Algorithm 10 does so by comparing SDF values of vertices of leaf nodes and their interpolation from SDF values at \mathbf{nd} . The maximum of the *absolute errors* and the sum of the *relative errors* are calculated for all vertices of the children of the input node and returned.

Algorithm 9 continues by checking if both errors are smaller than a predefined optimal threshold. If so, \mathbf{nd} is turned into a leaf by having its children nodes deleted from the octree. An optimal threshold is a value chosen by the implementation that best limits the error and estimates whether a leaf node is dispensable or not.

To apply Algorithm 9 to all nodes in an octree, it is sufficient to use it as a parameter of post-order function ($\mathbf{f2}$) for Algorithm 1 in conjunction with the \mathbf{root} node of the octree: $\mathbf{TreeTrav}(\mathbf{root}, \mathbf{NULL}, \mathbf{Opt})$. It is important that the optimization algorithm is passed as a post-order function to be able to solve the recursive case of consecutive optimizations. This way, nodes that are transformed into leaves by Algorithm 9 can trigger an additional optimization of their parents which can be also processed by the same function in a later stage.

Algorithm 9 Octree Optimization Algorithm

Func $\mathbf{Opt}(\mathbf{nd})$

```

1: Let  $h$  be the height of subtree with root node =  $\mathbf{nd}$ 
2: if  $h \neq 1$  then
3:   return false                                //Cannot be optimized if not in correct shape
4: end if
5: Let  $\mathit{absErrMax}, \mathit{relErrSum} = \mathit{ErrorEst}(\mathit{node})$ 
6: if  $\mathit{absErrMax}$  and  $\mathit{relErrSum}$  are small enough then
7:   for  $\mathit{child} \in C$  do                          //nd is now a leaf
8:     Delete  $\mathit{child}$ 
9:   end for
10: end if

```

Algorithm 10 Error Estimation Algorithm

Funct ErrorEst(*nd*)

```

1: Let absErrMax = 0.0
2: Let relErrSum = 0.0
3: for child ∈ nd.children do
4:   for i = 1, ..., 8 do
5:     Let sdfi be the SDF of vertex i of child
6:     Let sdfProji be an interpolation of nd to vertex i
7:     Let absErri =  $\|sdfProj_i - sdf_i\|$ 
8:     absErrMax =  $\max(absErrMax, absErr_i)$ 
9:     relErrNum = relErrNum + (absErri/sdfi)
10:  end for
11: end for
12: return absErrMax , relErrSum

```

4.6 Octree Serialization

The process of serialization is done to allow the use of *graphical processing units* (GPUs) to provide real-time visualization for the software solution. Since GPUs do not support pointers to memory locations nor natively allow recursion (there are no stacks on GPUs), algorithms such as Algorithm 12 have to adapt to these hardware specifications. The most important adaptation, in our case, is the serialization of the octree data structure, since it contains all the distance values in a given input scene and feeds the rendering program executed in the GPU.

Algorithm 11 explains how the serialization can be performed by porting the octree from a tree structure into the array of nodes (**nodesArray**) and array of leaves (**leavesArray**). **nodesArray** is responsible for carrying information on the structure of the octree by storing positions of other nodes and leaves. **leavesArray** will contain only SDF values which are initially stored in leaf nodes in the octree. Both arrays will be later copied to the GPU and used for the serialized octree lookup, as seen in Section 4.7.

The algorithm can be called initially with **node** being the **root** node of the

octree and have empty arrays for both `nodesArray` and `leavesArray` parameters. The procedure starts by checking whether `node` is a leaf, which is a recursion base case for recording the SDF value of such node into `leavesArray`.

In case `node` is a non-leaf, the same algorithm is recursively called for all its children. Additionally, each child has its position recorded in the `nodesArray`.

Algorithm 11 Octree Serialization Algorithm

Funct `Serialize(node, nodesArray, leavesArray)`

```

1: if node is leaf then
2:   Record SDF from node into leavesArray
3:   return
4: end if
5: Record position of node into nodesArray
6: for  $i = 1, \dots, 8$  do
7:   Let  $child_i$  be the  $i_{th}$  child of node
8:   Serialize(childi, nodesArray, leavesArray)
9:   Record position of  $child_i$  in nodesArray
10: end for

```

For simplicity, we have presented Algorithm 11 as an equivalent alternative to do one used in our implementation. Even though Algorithm 11 provides a valid way of serializing an octree, the strategy and code structure of the software solution uses Algorithm 1 whenever another procedure needs to execute in the totality of the octree. This pattern was seen previously in Sections 4.2 and 4.5. Since this is the case for the serialization of the octree, Section 5.3.6 displays a way of adapting it to use Algorithm 1 to navigate the octree structure.

4.7 Serialized Octree Lookup Algorithm

The lookup of a serialized octree is needed for accessing relevant SDF values stored in such data structure. This is an algorithm that needs to comply with GPU limitations and be heavily optimized, given that it called repeatedly for the whole duration of the execution of our software solution. Together with hardware specifica-

tions, the quality of the *serialized lookup algorithm* has great impact on the real-time capability of the software.

The idea of this procedure is to estimate the SDF value at \mathbf{p} by interpolating SDF values stored at the leaf node of the octree containing or closest to \mathbf{p} . At this stage, there is no information concerning triangles, their locations or distance values to \mathbf{p} . Rather, the octree only stores distance-to-mesh values in each of its leaf corners.

Algorithm 12 receives a 3D coordinate $\mathbf{p} \in \mathbb{R}^3$ and arrays `nodesArray` and `leavesArray` representing the serialized octree as parameters. Firstly, a position variable `index` is set to the location of the `root` node in `nodesArray`, usually being the first record in the array of nodes. Then, since there is no support for recursion, a loop with upper bound being the maximum depth of the octree is used to limit the number of times the lookup algorithm will navigate the array. This restriction is important for the GPU compiler since it is translated to a more efficient assembly equivalent code than if there was no upper bound in the loop (eg. `while(true)`).

Inside the loop, `index` receives the position of the child node which contains or is closest to \mathbf{p} . This way, `index` serves as a reference to the current node being evaluated. Furthermore, it is important to stress that the algorithm finds an approximation for the SDF value of any $\mathbf{p} \in \mathbb{R}^3$, even if it is not located inside of the octree structure (\mathbf{p} is outside the octree bounding box). This is secured by the fact that interpolation can be performed in relation to the closest leaf node.

Once the `index` variable is set to the child node, it is checked whether such child is a leaf. If it confirms to be so, the lookup found the correct leaf node for interpolating. Otherwise, the loop continues to the next level in the octree structure represented by `nodesArray`.

Finally, once a suitable leaf node is indicated by `index`, it is sufficient to interpolate the values stored in the position `index` of `leavesArray` to estimate the SDF value of \mathbf{p} returned by the algorithm.

Algorithm 12 Serialized Octree Lookup Algorithm

Funct Lookup($p, nodesArray, leavesArray$)

```
1: Let  $index$  be the position of the root node in  $nodesArray$ 
2: Let  $maxDepth$  be the maximum depth of the octree
3: for  $i = 1, \dots, maxDepth$  do
4:    $index$  receives the position of its child node in  $nodesArray$  containing (or
     closest to)  $p$ 
5:   if  $index$  is of a leaf node then
6:     Exit from loop
7:   end if
8: end for
9: return Interpolation of  $p$  by SDF values at  $leavesArray[index]$ 
```

Algorithm 12 concludes this chapter. Next, we discuss the implementation of the software.

Chapter 5

Implementation

This chapter explains the implementation of the proposed software solution. Firstly, a general project overview is shown (Section 5.1). Then, all the auxiliary classes are presented (Section 5.2), preparing the way to the description of the main class: `Octree`, in Section 5.3. It also discusses the testing plan (Section 5.4), and it shows important SDF estimations of triangle-based objects (Section 5.5) generated by our software.

5.1 Overview

The implementation of the software solution was added to an already existent *Sphere Tracing Renderer* which could handle the processing of objects described by SDFs. The new contribution allowed the renderer to efficiently work with triangle meshes, which are the most common way of representing 3D objects, by transforming them to *signed distance fields*. Essentially, octrees are handled as a special kind of primitive that is given as input to the renderer.

Although octrees are treated as a primitive by the renderer, they need additional processing in both CPU and GPU. In addition, they occupy memory equivalent to the data structure created and later optimized by the algorithms described in Chapter 4.

To better visualize the integration between the already existing renderer and the software solution, Figure 5.1 shows a simplified *Constructive Solid Geometry* [8] constructed from the union of three primitives given as input to the renderer: a

plane and a fractal, both described by well-defined SDFs, and an octree representing the *Signed Distance Field* of the *Suzanne.obj* mesh.

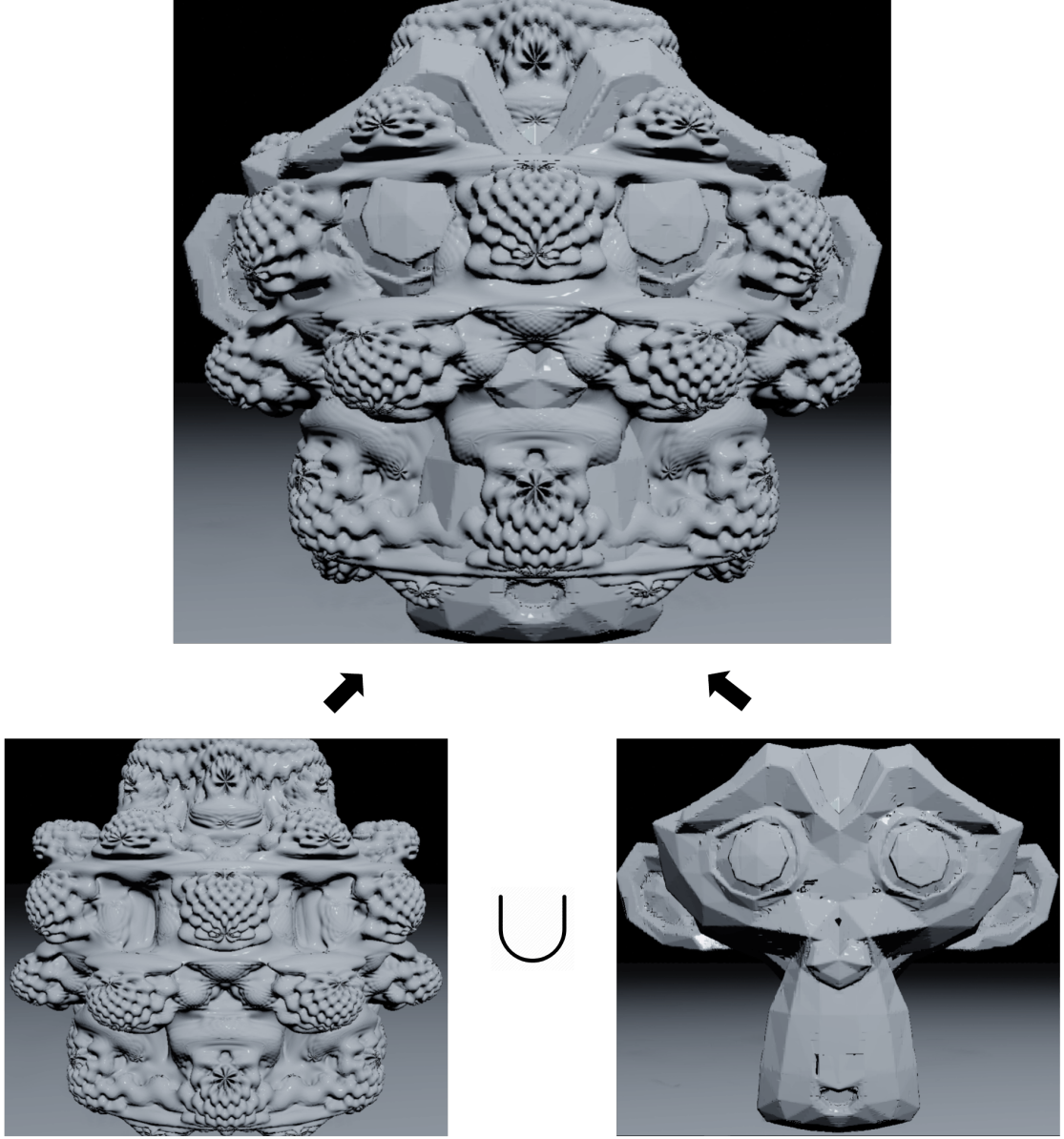


Figure 5.1: Constructive Solid Geometry of plane, fractal, and octree. The addition of the plane to the intermediate images was implied, for simplicity. The fractal is a primitive featured by the previously existent *Sphere Tracing* renderer. The octree representation of mesh included in *Suzanne.obj* is a contribution of our software.

The programming language chosen for CPU processing, responsible for all the operations up to the serialized lookup of the octree (Section 5.3.7), is the *C++* programming language. It has the benefits of an object-oriented language that can

be very useful for large projects, especially for the maintainability and scaling. At the same time, it allows optimizations which can be checked against the generated *Assembly* code, making the impact of a modification in the code very transparent to the developer. The versatility of *C++* together with its transparency makes it the default choice for larger projects focused on efficiency, such as the software solution of this thesis.

For the GPU part of the software, the chosen API was the *Open Graphics Library* (*OpenGL*). It provides a portable interface designed for rendering 2D and 3D graphics. It is the industry standard, being the most used rendering API, with a well defined documentation and active community of developers. Its portability in being an open multi-platform standard is an important characteristic, since its main competitors (*Direct3D*, from Microsoft, and *Metal*, from Apple) run only on specific environments. The *OpenGL shading language* (*GLSL*) is the high-level shading language used for programming according to the *OpenGL* API. Its syntax is based on the *C* programming language, thus being a natural extension for projects using the *C++* programming language.

The software is implemented in the object-oriented paradigm, describing modules as classes. The class diagram shown in Figure 5.2 describes how the various classes are organized, their respective attributes, operations and relationships. Due to the high number of operations and attributes of the `Octree` class, only its main methods and attributes were displayed.

5.2 Auxiliary Classes

This section provides explanation for the auxiliary classes used by the `Octree` main class. After the explanation for each class, the prototype (contents of header files) is provided in *C++* syntax. The auxiliary classes are the `AABB` (5.2.1), `Triangle` (5.2.2), `Importer` (5.2.3), `OctreeNode` (5.2.4), and `Vertex` (5.2.5).

5.2.1 AABB

The `AABB` class represents the *bounding box*, which is a cube-like structure that encompasses primitives in a 3D scene. It stores the minimum and maximum points

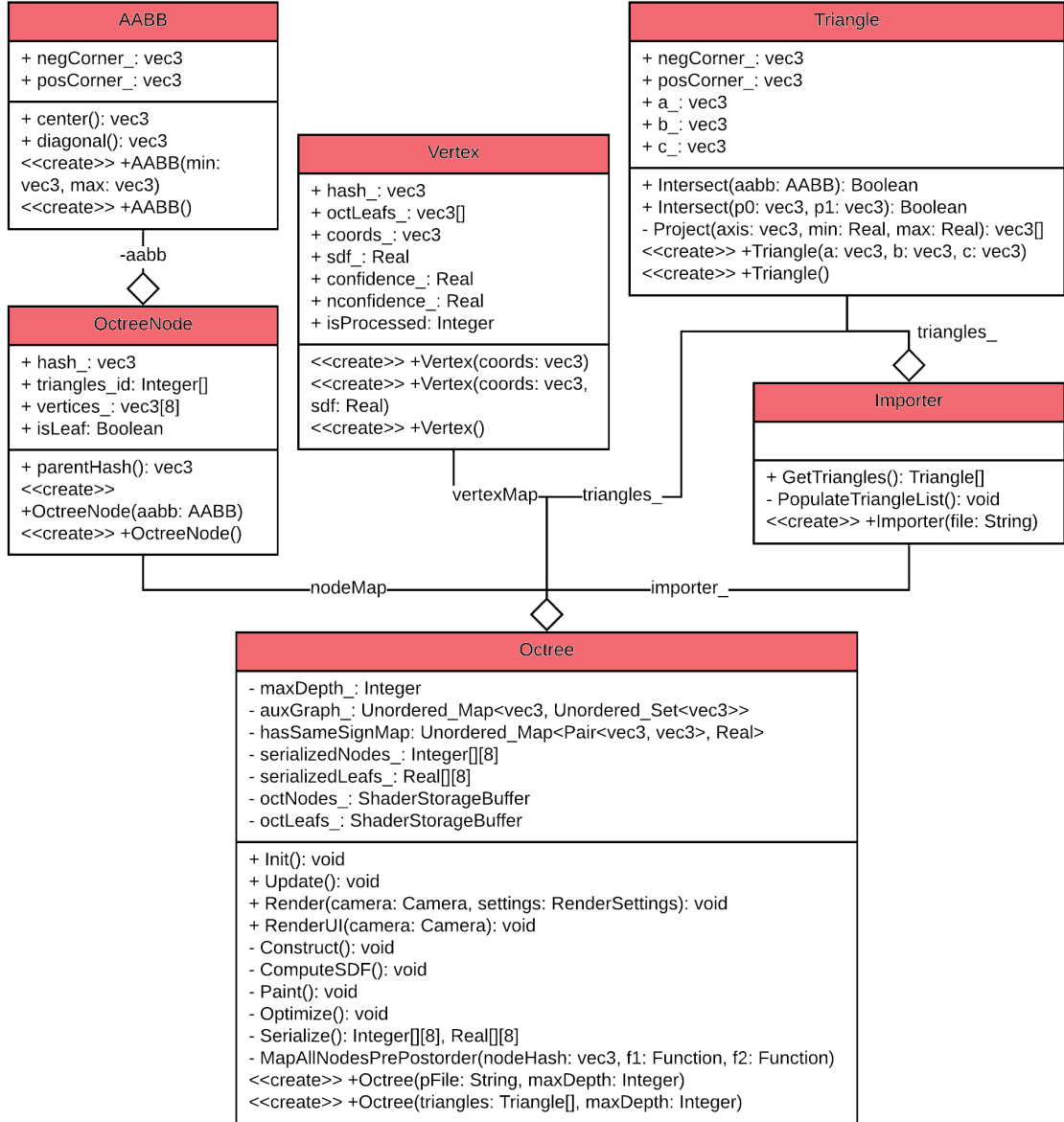


Figure 5.2: Class diagram of the components of the software solution

in a primitive or a group of primitives, building a cube from those extreme points. It is useful for many computer graphics applications, since it provides a unified shape for every primitive which contains basic positioning information. From the minimum and maximum points, it is possible to calculate the center and diagonal of such cube.

The center point and diagonal vector of the **Aabb** are not stored as attributes but rather are defined as functions, as can be seen in Code 5.1. This is done to avoid inconsistencies when updating the values of **negCorner_** or **posCorner_**. By implementing them as functions, the center and diagonal values are recalculated every time they are called. However, since they constitute of simple calculations

helped by the `inline` directive, they do not require excessive processing power. Furthermore, this implementation saves memory as a natural consequence of not storing two extra `vec3` attributes in the class.

```
1 class AABB
2 {
3 public:
4     AABB();
5     AABB(const glm::vec3& min, const glm::vec3& max);
6     ~AABB();
7
8     glm::vec3 negCorner_;
9     glm::vec3 posCorner_;
10    inline glm::vec3 center() const;
11    inline glm::vec3 diagonal() const;
12};
```

Code 5.1: AABB class prototype

5.2.2 Triangle

The `Triangle` class relates to the primitive of same name. It stores the three vertices of a triangle primitive (`a`, `b`, `c`) and has additional attributes `negCorner_`, `posCorner_` to be able to compare its instances to the ones of `AABB` class.

There are two `Intersect` methods in the `Triangle` class due to the *function overloading* feature of *C++*. Both functions test the intersection of the triangle with elements described by their parameters.

The first method uses the *Separating Axes Theorem (SAT)* [9] to determine whether a given `AABB` cube intersects the triangle. In this algorithm, it is necessary to calculate the projection of points in relation to an axis. Such subtask is described in the SAT algorithm and tackled in the private function `Project` of the `Triangle` class.

The second `Intersect` method receives two points `p0`, `p1` which represent the line segment $l = p1 - p0$. Then, it uses a line segment-triangle intersection test described in *Badouel* [10] to determine whether the triangle intersects `l`. The prototype of class `Triangle` is shown in Code 5.2.

```

1 class Triangle
2 {
3 public:
4     Triangle();
5     Triangle(const glm::vec3& a, const glm::vec3& b, const glm::vec3&
6             c);
7     ~Triangle();
8
9     bool Intersect(const AABB& aabb) const;
10    bool Intersect(const glm::vec3& p0, const glm::vec3& p1) const;
11
12    glm::vec3 negCorner_;
13    glm::vec3 posCorner_;
14    glm::vec3 a_;
15    glm::vec3 b_;
16    glm::vec3 c_;
17 private:
18    void Project(const std::vector<glm::vec3>& vec, const glm::vec3
19                axis, float& min, float& max) const;
20 };

```

Code 5.2: Triangle class prototype

5.2.3 Importer

The `Importer` class is responsible for parsing a triangle mesh input file and creating a list of objects of class `Triangle` (Code 5.2) from the information of vertices, edges, and faces of the input file. It uses the *Open Asset Import Library* (*Assimp*) [11] to parse files from a wide variety of 3D model formats and store the triangle information into a list of triangle instances later used by the `Octree` class.

`Importer` class inherits from a utility class `SFile` available from the renderer that tackles file opening, converting file content into a string and file closing for the software solution.

There are two methods available in the prototype of Code 5.3. The first one is `GetTriangles`, which returns the triangles stored in the list of triangles (`triangles_`), attribute of the `Importer` class. The second one (`PopulateTriangleList`) uses the *Assimp* library to populate the local list of trian-

gles. The latter is a private function called whenever the local list of triangles needs to be updated.

The structure of class **Importer** was set up to have independent methods for retrieving the triangles from a local attribute (**GetTriangles**) and for populating the local variable from the input file (**PopulateTriangleList**) to allow future improvements with the detection of file modifications. A new feature could be developed which would allow the **Importer** class to adapt to an input file being modified. It would update the local list of triangles and trigger the construction, optimization, and serialization of a new octree structure automatically whenever a file is modified.

```

1 class Importer : public SFile
2 {
3 public:
4     Importer(const std::string& file);
5     std::vector< Triangle >& GetTriangles();
6 private:
7     void PopulateTriangleList();
8
9     std::vector< Triangle > triangles_;
10 };

```

Code 5.3: Importer class prototype

5.2.4 OctreeNode

The **OctreeNode** class describes the attributes and methods for instances of octree nodes. It contains an **AABB** instance for storing its position in the coordinate system (**aabb_**), an array of references to intersecting triangles (**triangles_id_**), and an array of references to correspondent vertices in the auxiliary graph (**vertices_**). Moreover, it stores a unique hash which identifies the octree node (**hash_**), a boolean variable to differentiate leaves and non-leaf nodes (**isLeaf**), and a function to calculate the hash of its parent (**parentHash**). The prototype of the **OctreeNode** class is displayed in Code 5.4.

It is important to notice that, in *C++*, *structs* are equivalent to classes, with the difference being the default scope of variables and methods. Therefore, the prototype in 5.4 can be considered a class with all elements having *public* visibility. Moreover,

the reason for the `parentHash` function to be chosen over a new attribute to store the equivalent value is the same as in Code 5.1: inconsistency avoidance.

Types `hash_octNode` and `hash_ptVertex`, introduced in this class, are equivalent to a vector of three unsigned integer elements of 64 bits each (`glm::u64vec3`). The renaming is done through the directive `using` to differentiate the hashes which identify octree nodes and vertices (Section 5.2.5). Even though their underlying type is the same, the renaming facilitates the readability and maintainability of the code. These hashes are unique identifiers of octree nodes and vertices and will be further discussed in Section 5.3.1.

```

1 struct OctreeNode
2 {
3     OctreeNode();
4     OctreeNode(AABB& aabb);
5     ~OctreeNode();
6     hash_octNode hash_;
7     std::vector<unsigned> triangles_id_;
8     AABB aabb_;
9     std::array<hash_ptVertex, 8> vertices_;
10    bool isLeaf;
11    inline hash_octNode parentHash() const;
12 };

```

Code 5.4: OctreeNode class prototype

5.2.5 Vertex

The `Vertex` class provides the model for vertices of the auxiliary graph (Section 4.3). These vertices represent points in the 3D coordinate system where corners of octree nodes are located. They were created as a separate class to avoid repetition of vertices shared by more than one octree node, thus making the SDF computation algorithm (Section 4.4) faster and more efficient.

The class prototype (Code 5.5) shows that each vertex has a unique identifier (`hash_`), and an array of references of octree nodes that share such vertex (`oct Leafs_`). It also stores its 3D coordinates (`coords_`), its SDF value (`sdf_`), its confidence and temporary confidence values (`confidence_` , `nconfidence_`),

and the state of execution of the vertex in the SDF computation algorithm (`isProcessed`).

There are two aspects of the `Vertex` class being modified to facilitate the implementation of the SDF computation algorithm (Section 5.3.4). Firstly, the states of execution stored in each vertex differs from the ones explained in Section 4.4.2. For the implementation, there are three states instead of two for efficiency reasons. Due to this fact, states are stored as an integer instead of a binary boolean value. Secondly, the `nconfidence_` attribute is added as an extra field to avoid a new auxiliary structure for storing temporary confidence values during the algorithm.

```
1 struct Vertex
2 {
3     Vertex();
4     Vertex(const glm::vec3& coords);
5     Vertex(const glm::vec3& coords, const float sdf);
6     ~Vertex();
7
8     hash_ptVertex hash_;
9     std::vector<hash_octNode> octLeafs_;
10    glm::vec3 coords_;
11    float sdf_;
12    double confidence_;
13    double nconfidence_;
14    unsigned isProcessed;
15};
```

Code 5.5: `Vertex` class prototype

5.3 Octree

This section explains the main class of the thesis: the `Octree` class. It is responsible for discussing the implementation of the data structures in the `Octree` class (5.3.1), the traverse of the octree (5.3.2), its construction (5.3.3), the SDF computation of its vertices in the auxiliary graph (5.3.4), its optimization (5.3.5) and serialization (5.3.6), as well as the lookup of the serialized octree (5.3.7). All subsec-

tions except 5.3.7 implement code for the CPU, in *C++*. For the serialized octree lookup, since it is executed in the GPU, *GLSL* was chosen for its implementation.

5.3.1 Data Structure

The octree data structure can be stored in many ways. The most common approach is to have octree node instances connected by pointers to children nodes. An octree starts at a node called `root`. It also has regular nodes which point to other (children) nodes, and leaf nodes being octree nodes pointing to non-valid elements (NULL pointers, in *C++*). This way of representation allows fast addition and removal of elements in the octree while having $\log(h)$ complexity for the lookup of a node, h being the height of the octree.

Even though $\log(h)$ is considered a low complexity in algorithms, our software solution aimed at achieving an even faster lookup in an octree. This could be done by assigning hash codes to each octree node and storing all nodes in an unordered map. The unordered map structure provides constant lookup time for any element, making such operation scalable for an octree of arbitrary height. Since octree nodes point to their children, it is still possible to add and remove elements as fast as in the most common implementation approach. The trade-off for this optimization is the extra memory allocated for storing the hashes in the unordered map. The prototype for the octree data structure is shown in Code 5.6.

```
1 | std::unordered_map< hash_octNode, OctreeNode > nodeMap;
```

Code 5.6: Octree Data Structure Prototype

The hash of each octree node (`hash_octNode`) is a unique identifier of such structure. As seen in Section 5.2.4, the octree node hash type is equivalent to a vector of three unsigned integers of 64 bits each (`glm::u64vec3`). Since the octree can be interpreted as a three-dimensional binary tree, three binary decisions (1 or 0) have to be made for traversing every level of an octree. The aforementioned decisions are stored in the three fields of the octree node hash. The hashing of an octree node describes the path (sequence of decisions of 1 or 0) starting from the `root` until a given node. This technique is an extension of the *Huffman coding* [12] for three

dimensions. The `root` node of an octree, for our implementation, has hash `h = {1, 1, 1}`.

In addition to the octree data structure, the auxiliary graph is also part of the octree class. Differently than the octree, the auxiliary graph is divided into two structures: a list of vertices and a list of edges. It was represented in this fashion to show interchangeable ways of representing a graph: by storing it into two different structures for vertices and edges (auxiliary graph), or storing edges as attributes of vertices and having a single list of vertices (octree).

Following the idea implemented in the storage of the octree nodes, the list of vertices in the auxiliary graph is composed by an unordered map of unique hashes which link to their correspondent vertex. The list of edges is also an unordered map connecting vertex hashes to sets of hashes of their neighboring vertices in the graph. The set structure was chosen to avoid duplicates generated by the auxiliary graph construction algorithm (3) in an efficient way. Both lists of vertices and edges (Code 5.7) have constant lookup time because of their underlying unordered map structure.

```

1 // List of vertices
2 std::unordered_map< hash_ptVertex, Vertex > vertexMap;
3 // List of edges
4 std::unordered_map< hash_ptVertex, std::unordered_set<
    hash_ptVertex > > auxGraph;
```

Code 5.7: Auxiliary Graph Prototype

Despite being implemented with the same structure (`glm::u64vec3`), the hash for vertices in the auxiliary graph (`hash_ptVertex`) differs in its construction strategy from the aforementioned hash for octree nodes. This occurs due to the fact that many vertices are shared between octree nodes, thus not being possible to use the path to a single node as a unique identifier for a vertex. However, it is possible to calculate the vertex hash from the hash of any octree node which contains it.

For every octree node hash, three unsigned integer elements are stored. To calculate the hash for the eight vertices of such octree node, it is sufficient to firstly add either 0 or 1 (indicating positioning according to an axis) to every element (add corner), which constitutes eight possibilities. Then, eliminate the zeros to the right of every element (simplify), making it unique for vertices, since the last zeros in

the previously defined sequence only indicate height information on the octree and refer to the same vertex. A one-dimension visualization of the process that generates hashes of vertices can be seen in Figure 5.3.

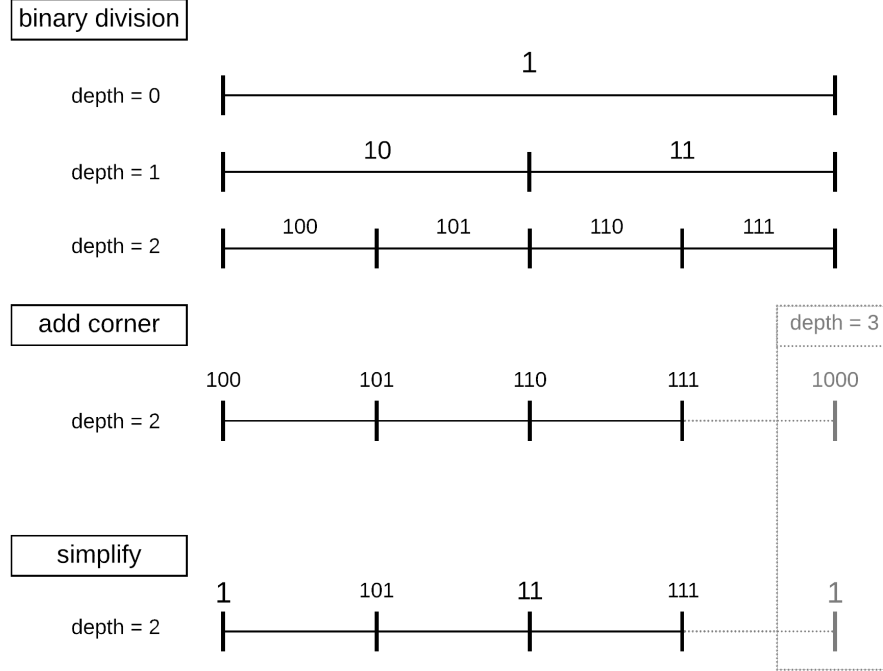


Figure 5.3: One-dimension visualization of the process that generates unique hashes for octree vertices. From binary divisions representing the octree depth, a corner is added. Afterwards, zeros to the right are discarded in the simplification process.

The process of generating unique hashes (Figure 5.3) starts with the subdivisions of binary numbers in a one-dimensional space, each division representing a depth level of the octree structure. When calculating unique references to corners, the last reference hash of a level (right-most position) represents the same corner as the first one (left-most position). The difference being the depth of the octree of which the hash has been calculated. Such equivalence is highlighted during the simplification process. Zeros to the right in a binary number are discarded. The resulting references, after simplification, are unique.

Such approach is scaled for three dimensions, resulting in the used methodology of the proposed software solution for generating unique hashes for octree vertices in the \mathbb{R}^3 .

5.3.2 Traversal

The implementation of Algorithm 1 uses a private template function of the `Octree` class named `TreeTrav`. This procedure takes the same arguments as described in the algorithm: an octree node (Section 5.2.4), and two functions for the possibility of applying routines in both pre-order and post-order. Since it is a simple generic function and does not contain any subproblem, it can be implemented exactly as described in the algorithm. A template method was chosen to allow parameter functions `f1` and `f2` of any type in a single, efficient, and concise implementation.

5.3.3 Construction

The construction of the octree follows the steps of Algorithm 2, constituting its core structure. Its implementation relies on the *Tree Traversal Algorithm* (whose implementation is described in Section 5.3.2) to recursively apply its core routine, and create every node of the octree. It is implemented as a private member function of name `Construct`, located in the `Octree` class.

Differently than the traversal procedure, Algorithm 2 requires the resolution of multiple subproblems. These tasks usually have multiple solutions, constituting implementation choices.

The first subproblem is presented in line 1 of the *Octree Construction Algorithm* (2), in which it is necessary to query the depth of a node given as parameter. In our implementation, the depth of a node is calculated by its hash, which is assigned during its creation. It is trivial to calculate the depth through the hash of a node, since it reflects the path from the `root` to such node in the octree. Thus, the length of the aforementioned path (0s and 1s) of a component of the hash will constitute the depth of a node.

The other subproblem present in the construction algorithm, in line 9, is the triangle-cube intersection test. It is needed for deciding which triangles of the scene have their references stored in a given node for computing the SDF in a later stage. The method chosen by this implementation was the *Separating Axes Theorem* [9], which has been previously discussed in the implementation of the `Triangle` class (Section 5.2.2).

5.3.4 SDF Computation

The SDF computation is divided in two steps: the distance values computation, and the inside/outside partition, as seen in section 4.4. Both steps are implemented as private member functions of the `Octree` class, named `ComputeSDF` and `Paint`, respectively. The former is described by Algorithm 5, and the latter by Algorithms 6, 7, and 8.

The implementation of Algorithm 5 includes two subproblems: the SDF of a triangle, needed in line 6, and the *closest triangles subproblem* (Section 4.3.3), explicitly called in line 3. The triangle SDF was extracted from an online article [13] which aggregates signed distance functions for various primitives. The *closest triangles subproblem* is solved by Algorithm 4. However, such procedure presents the task of finding a leaf node containing a given point $p \in \mathbb{R}^3$, in line 9. This task is resolved by traversing the octree starting from its `root`, and, in every iteration, choosing the child node that includes `p`. The loop stops once a leaf is found, and such leaf node is returned as a result.

Algorithm 6 is the main algorithm for assigning confidence levels for all vertices in the auxiliary graph. The `confidence_` of a vertex determines the sign of its previously calculated distance value. The only subproblem of this procedure is determining the initial vertex, which needs to be outside of the mesh structure (line 2). For the proposed software solution, the size of the `AABB` of the `root` node is increased by a small factor, guaranteeing that all of its vertices will be outside of the triangle mesh. Once this operation is performed, it is sufficient to assign any of such vertices as the initial `v0` of the algorithm.

For the implementation of Algorithm 7, there are some optimizations that can be done to make the procedure more efficient. Firstly, when retrieving the vertex with maximum confidence (line 1), it would be very costly to perform a maximum search every time the function is called. To address this performance issue, vertices in set `V` are stored as a *priority queue*, in which the vertex with highest confidence always stays in the front of the structure. This way, the retrieval of a vertex with maximum confidence has complexity $O(1)$.

Secondly, in line 9, it needs to be checked whether a neighboring vertex `n` is a member of set `V`. This task could be solved by performing a search on set `V` every

time it was executed. A more efficient approach would involve sacrificing memory by storing an extra attribute in every object of type `Vertex` to determine if it was in such collection. For our implementation, a new state `inqueue` is assigned to the vertex together with states `processed` and `unprocessed`. This way, it is not necessary to search set V , and no additional memory is used for this operation.

Moreover, a simplification (which needs to be adapted in the implementation) is done on Algorithms 6 and 7, when confidence levels of neighboring vertices are updated. Every time a vertex v is processed, all its neighboring nodes have their confidence values updated to reflect the new change in v . In practice, once a vertex has been processed, its confidence should not be changed if one of its neighbors is processed. Therefore, it is necessary to store the confidence levels of non-processed vertices in a temporary structure and only record the confidence levels once a vertex is processed. For the software of this thesis, the implementation for the aforementioned temporary structure is substituted for a single extra attribute for objects of type `Vertex` named `nconfidence_` (see Section 5.2.5).

Algorithm 8 uses the relation $\|f(va) - (-f(vb))\| > \|va - vb\|_2$ and a segment check to calculate the confidence of two vertices having the same or opposite signs. The significant subproblem of this procedure is the intersection of a triangle and a line segment (line 9), crucial for the segment check. This task is solved by a well-known method described in Badouel [10], and implemented as part of the `Triangle` class (Section 5.2.2).

5.3.5 Optimization

The octree optimization has Algorithms 9 and 10 as its core structure. It can be applied to the entirety of the octree by using Algorithm 9 as post-order function argument of the `Tree Trav` generic function (Section 5.3.2). The optimization process is represented as a private member function (`Optimize()`) of the `Octree` class.

The implementation of Algorithm 9 asserts the height h of the input node (line 2) by checking whether such node is a leaf ($h = 0$), or if any of its children nodes are non-leaf ($h > 1$), and rejecting if it either is true. Furthermore, in line 6, the algorithm does not state the error thresholds for deleting the subtree starting in the input node. After fine-tuning with multiple mesh experiments, our implementation

uses 1% of the diagonal of the smallest node in the octree as the absolute error threshold. For the equivalent relative error measurement, the mean of the relative errors have to be less than 0.1 (indicating a 1 to 10 ratio of the *absolute error/SDF value* for each vertex).

For the *Error Estimation Algorithm* (10), neither the type of interpolation nor how it can be calculated are provided in line 6, given that it depends on the application. For the software of this thesis, linear interpolation is used, since all vertices of a node have the same importance for estimating a new point. This type of interpolation can be calculated by estimating the SDF value of a new point $p \in \mathbb{R}^3$ by a weighted average of the SDF values of the known vertices, in which the weight of a known vertex can be described as its distance to the new point ($w_i = \|p - v_i\|$).

In the implementation of this thesis, an efficient approach is taken towards linear interpolation. Given that the number of known vertices to be used in the estimation is even (eight vertices for every node), they are divided equally into two vectors. Then, both structures are interpolated with the `glm::mix` function [14], which accepts both units and vectors. The result of the interpolation is a new vector of same size as one of the inputs (half of the original amount). This is repeated until a single value is returned as a result of the interpolation. Our approach is more efficient than performing a regular weighted average due to optimizations on vector operations.

5.3.6 Serialization

The process of serializing an octree was explained in Section 4.6. Even though a valid, more concise alternative was presented by using recursion (line 8 of Algorithm 11), the implementation of the serialization chooses to use Algorithm 1 to map core functions to all nodes in the octree instead of relying on recursion. This is done with the intent of having a unified way of traversing the octree, which increases the modularity and maintainability of the software. The octree serialization is represented as a private member function of the `Octree` class, denominated `Serialize`.

The implementation of this adaptation to use the *Tree Traversal Algorithm* (1) relies on the simulation of recursion by using the *stack* data structure. It is possible to record the positions of all nodes in a temporary stack, in a pre-order traversal.

This allows each parent node to access the information of its children nodes in a post-order traversal, since they have been already processed in pre-order.

By following such execution rule, every parent node can decide if a child node was leaf or non-leaf (task of line 8 of Algorithm 11), and record its previously calculated position accordingly in the array of nodes (`Integer[] [8]`), or array of leaves (`Real[] [8]`), returned by the algorithm.

Additionally, it is necessary to determine to which array (node or leaves) such position refers to. In the implementation of this thesis, a flag is set to differentiate positions in both vectors. This flag is the last bit of the 32-bit integer number responsible for storing the position of a node in one of the aforementioned vectors. If set to 1, the flag indicates that the number refers to a position in the array of leaves. If 0, it points to a spot in the array of nodes.

The last bit of the integer was chosen as a flag for simplicity and efficiency of storing two information (the position, and to which array it refers to) in a single 32-bit value. This reduces the number of valid indices (positions) by half to fit in a 31-bit integer value. Though, in practice, it still constitutes of a very high number of possibilities (1,073,741,824).

Both the arrays of serialized nodes and serialized leaves are stored in the `Octree` class as attributes named `serializedNodes_` and `serializedLeafs_`. These are later moved to the `octNodes_` and `octLeafs_ ShaderStorageBuffer` structures, which are a representation of the buffers used in the *GLSL* shader program to allow real-time visualization using GPU resources.

5.3.7 Lookup

The lookup of a serialized octree is the only operation which is not a member of the `Octree` class. This occurs due to the fact that it is implemented in *GLSL*, being detached from the rest of the octree building operations (Sections 5.3.2 to 5.3.6). Its implementation follows the steps of Algorithm 12, being tailored for execution in the GPU.

There are many abstract steps in Algorithm 12, dependent on each the implementation of the serialized octree. Firstly, in line 1, the position of the `root` node in the `nodesArray` can vary, depending on the serialization algorithm. For our imple-

mentation, since the nodes are processed in pre-order, the `root` node corresponds to the first element (of index 0) in `nodesArray`.

Moreover, line 4 presents a task of finding the child node which contains or is closest to point $p \in \mathbb{R}^3$. It can be solved by comparing the center point of the AABB of the current node with point p in relation to the three axes X, Y, and Z. The result of the three comparisons will result on which child node is closest p , since these constitute eight possibilities.

An example of this solution, in two dimensions, is shown in Figure 5.4, in which a point p is compared to the center of a quadtree (two-dimensional equivalent of octree). In such figure, it is possible to visualize four regions described by the comparison between p and the center c . Point p is located on the northwest quadrant, since its horizontal component is smaller than $c.x$, and its vertical component is bigger than $c.y$.

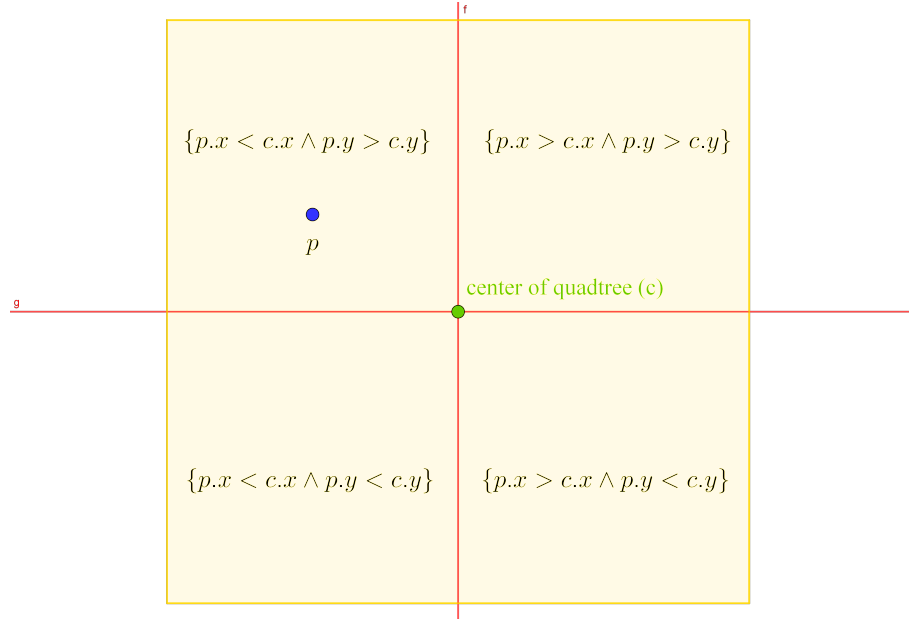


Figure 5.4: Finding a child node that contains p in two-dimensions.

Algorithm 11 also does not mention how an `index` of a leaf node can be distinguished from one that refers to a non-leaf node (line 5). The implementation of this distinction can be easily performed due to the flag incorporated into the last bit of such integer, as seen in Section 5.3.6. It is sufficient to compare the last bit of a given `index` to decide whether it points to a leaf node.

Lastly, in line 9, the procedure returns an interpolation of point p in relation to the coordinates of the leaf node that contains it. Following the implementation choice of the octree optimization (Section 5.3.5), a linear interpolation is performed and the estimated SDF for p is returned.

5.4 Testing

The testing of the software is divided into two subsections: user interface tests (5.4.1), focused on the user experience, and the automated tests (5.4.2), providing an overall testing for the program as a closed package.

5.4.1 User interface tests

The user interface tests target any user-level, with or without prior knowledge or familiarity with the subject or the medium (computer) of this thesis. They involve any input mean operated by the user and the response of the program to any action triggered by such interactions.

One of the most important aspects of the program is to contain the basic instructions of how the user should interact with it. With that in mind, the *Instructions* section of the *Mesh Selection* tab (Section 2.2.1) was developed to provide a brief description of all the components of the interface.

All the interactions executed by the use of the mouse and keyboard to the *Main Window*, presented in Section 2.2.2, are widely used by the gaming and simulation industry. This was implemented to provide a familiar interface to more experienced users, while still maintaining its simplicity for others. The aforementioned interactions were included as part of the *Sphere Tracing* renderer, prior to the start of the software subject of this thesis. After the integration of our solution into the renderer, the interactions with the *Main Window* were tested extensively inside and outside octree objects.

Another major aspect for the quality of user experience is the responsiveness of the interface, even under heavy load. The octree build and update, available in both *Mesh Selection* and *Node List* tabs, were developed in an asynchronous manner. This feature makes the user interface responsive even when the octree is being built, while

providing feedback through the progress bars, which are synchronized and display the same data across multiple tabs. Synchronization tests with different input mesh files were performed to guarantee the robustness of the program.

Moreover, all the basic and advanced interactions with the *Auxiliary Window*, such as expanding and hiding sections of all tabs, and modifying values in the *Node List* tab through the sliders and checkboxes, were tested to a high extent both individually and in conjunction with others.

Stress tests were performed on buttons and all other triggers (eg. hide/expand sections) to avoid deadlocks and other unwanted reactions, when facing adverse user interactions. Furthermore, input items such as buttons are made unavailable by being hidden to avoid the aforementioned unwanted use.

Lastly, for every user interaction, the program provides a feedback. When prompted to input a path to a file in the *Mesh Selection* tab, for instance, the user is prompted with a progress bar and receives a real-time updates on the octree build procedure. When feeding incorrect file paths, an instructive message appears explaining the reason for the error. And when a major task is completed, such as the *automated tests* routine, the progress bar is hidden and a prompt is provided to inform the user of the end of such task. Figure 5.5 shows two of the aforementioned feedback states for the user: the incorrect file prompt, and the finished *automated tests* sequence.



Figure 5.5: Examples of feedback from the program to distinct user interactions. The interaction on the left shows a message rejecting an incorrect file input. The one on the right shows a directive informing the end of the *automated test* routine.

5.4.2 Automated tests

Automated tests were incorporated into the program to test the solution by experimenting with known mesh files (which have been proved to be correct). This provides a safe test of differently-shaped input meshes that stresses specific aspects of the octree build process. In addition, basic input tests were implemented to increase robustness under incorrect inputs, such as non-existent or non-supported files.

As seen on the right interaction of Figure 5.5, there are two sections that constitute the automated testing structure: the Importer and the Octree. The *Importer* section aggregates three simple tests for testing file handling. It launches three instances of class *Importer* (Section 5.2.3) to test an empty input file, an incorrect file path, and a valid file. Such instances are executed in a sequential way, since they do not require extensive computing power, being almost instantaneously tested.

The second section, the Octree, refers to tests that involve the use of meshes that have been previously tested. By testing instances of Octrees that process four distinct well-defined meshes, the developer can test new additions to the project solution, and the user can verify the correctness of the software.

Furthermore, each of the selected meshes used in this section have different characteristics which focuses on a specific stage of the octree build. The *tetrahedron.obj* mesh is made of only four triangles, being the equivalent of a triangle for closed meshes. However, for the software, the process of estimating the tetrahedron is similar to that of any other object. All stages of the octree building are processed, even for such simple mesh. As a result, it can generate many unnecessary nodes during its construction, depending on the maximum depth of the octree.

Therefore, the key process to be tested when analyzing the estimation of the tetrahedron is the octree optimization (Section 4.5). With a well-defined optimization, with proper error thresholds, the number of nodes in the final estimation is reduced drastically. This way, the parameter for passing the test case of the *tetrahedron.obj* mesh is the percentage of reduction on the number of nodes during the optimization process.

Meshes contained in *Bunny.obj* and *Suzanne.obj* are considered standard when testing ray tracing (or *sphere tracing*, in our case) applications. They are made of a considerable amount of triangles, 5002 and 968, respectively. Both test all the stages

of the octree build process in an even manner. The application should be able to render the aforementioned meshes as a prerequisite to process more complex shapes.

The most complex mesh of the ones present on the test cases is the *Skull.obj*. It consists of 80016 triangles that describe an interesting shape. It tests the capacity of the software solution to process large and dense sets of triangles. The performance is less important than the capacity of correctly estimate such mesh. Thus, it is put as a final test case to certify that the solution is scalable.

5.5 Generated SDF Estimations

This section displays important SDF estimations generated by the solution proposed by this thesis. Furthermore, it uses the consequences of describing an object by SDFs rather than regular rasterization, such as the possibility of real-time computing of the offset operation, to advocate for the use of this software.

A complex estimation of the *Skull.obj* mesh (80016 triangles) is shown in Figure 5.6. It depicts a human skull that is an example of a highly complex mesh. Its octree has a maximum depth of 8 and 768,000 nodes. Without a programming approach, such as ours, that uses the octree data structure, it would not be viable to estimate a *signed distance function* that would describe such mesh.

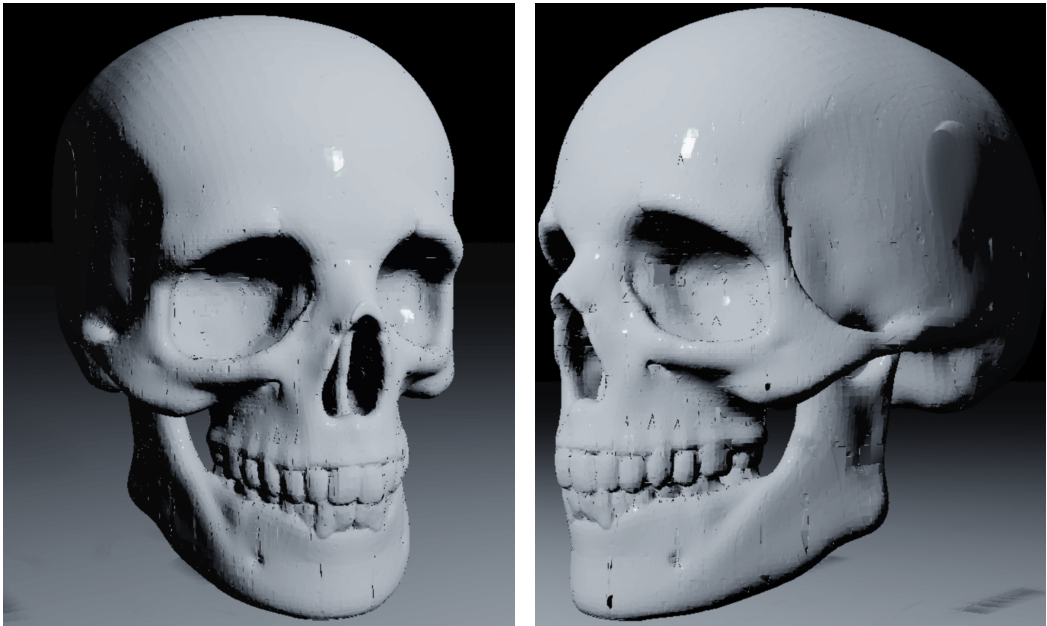


Figure 5.6: SDF estimation of the *Skull.obj* mesh from two different angles.

The real-time offset operation can be performed on any object described by SDFs, when using sphere tracing. Since our solution creates an SDF that represents any triangle-based mesh, it is possible to extend the aforementioned universality to all objects described by triangles. In Figures 5.7 and 5.8, it is possible to see representations of meshes *Bunny.obj* and *Suzanne.obj* as well as their negative and positive offset estimations.

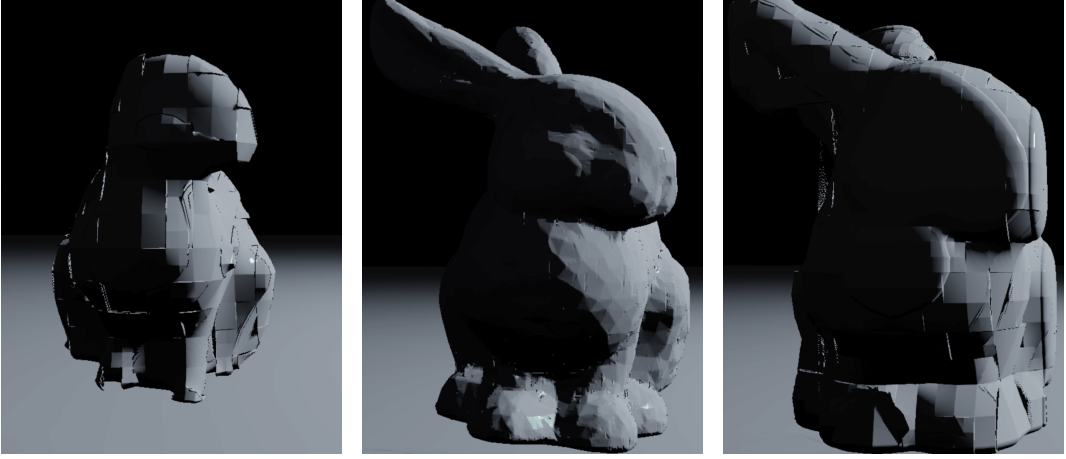


Figure 5.7: *Bunny.obj* with negative offset (left), zero offset (center), and positive offset (right).

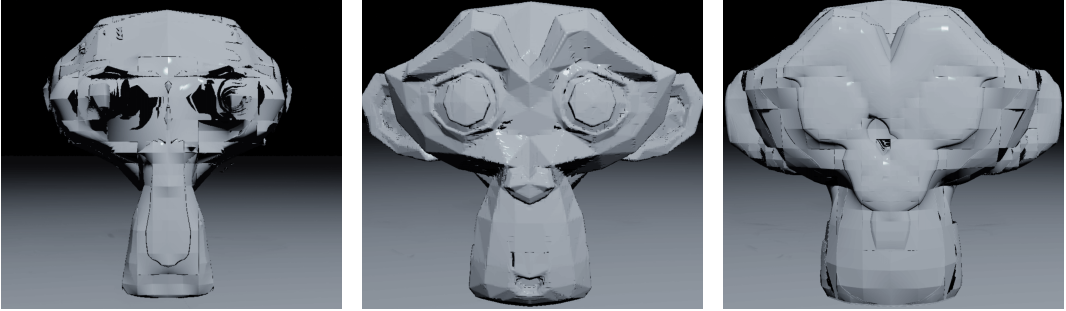


Figure 5.8: *Suzanne.obj* with negative offset (left), zero offset (center), and positive offset (right).

Another estimation made from a more heterogeneous mesh, in which there are small and concentrated portions of the mesh, can be seen in the representation of the *Cybertruck.obj* mesh (Figure 5.9). It is possible to visualize that areas such as the tires of the truck demand more SDF computing than its ceiling, due to the higher level of detail in such sections. Meshes with similar characteristics reinforce

the need to have an adjustable maximum depth parameter of the octree to be able to portray details in deeper levels.

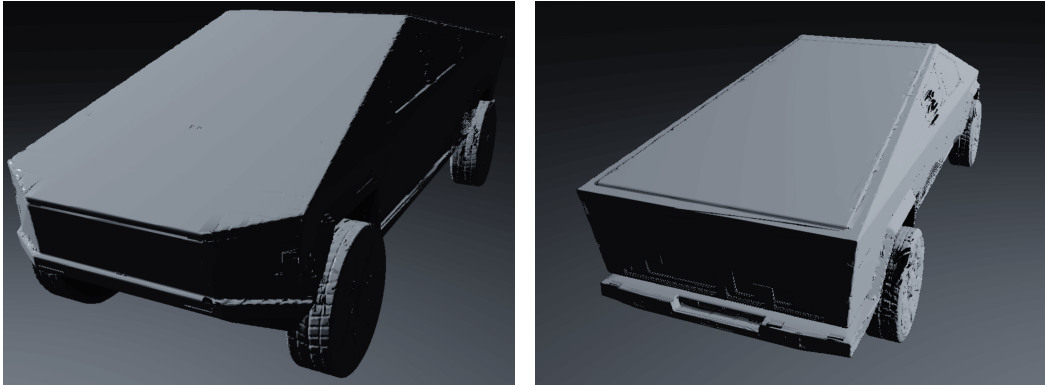


Figure 5.9: Estimation of the *Cybertruck.obj* mesh in two different angles.

Having shown important SDF estimations of triangle-based surfaces via our software, the implementation section is finished. The conclusion of the thesis is presented in Chapter 6.

Chapter 6

Conclusion

In this thesis, we presented a solution for the real-time visualization of surfaces described by *signed distance fields*. Such approach was motivated by the poor performance of SDFs when rendering triangle-based surfaces and it was inspired by related work by Bærentzen [3].

A detailed user guide, with numerous illustrations and examples was divided into two sections, focusing on different user experiences with the intention of including all user types. Following this stress on the ease of the user and reader, chapters were distributed in a way to maximize the time spent investigating a topic of interest.

We discussed various algorithms in a generic fashion for detachment from technologies and increased focus on problem-solving. Moreover, implementation analyses with detailed explanations were provided to explain the steps taken to build the software solution, and to instigate the reader to reflect on the challenges presented and fully understand the reasons for every implementation decision.

The octree, main data structure of the solution, was analysed in an extensive manner, including its theoretical background (Section 3.2), the discussion of many related algorithms (Chapter 4), and its implementation choices (Section 5.3), focused on performance and efficiency.

SDF estimations generated by our solution (Section 5.5) showed accurate estimations for highly complex surfaces. This fact can be seen when comparing the estimations with the triangle-based representations by using the debugging feature *render triangle mesh*, available in the *Node List Tab* (Section 2.3.1). Such approximations would not be possible without the use of an acceleration structure such as

the octree.

Our proposed solution that aimed at a real-time visualization of triangle meshes defined by SDFs achieved not only its goal, but also provided a simple user interface for interacting with such rendering. This allowed the real-time use of operations (eg. offset) that take advantage of SDFs.

Further improvements can be made in the software to allow the estimation of open meshes natively. This would require a new method for defining the inside and outside of a surface.

Moreover, as a future work, parallel-focused optimizations can be performed on the most resource-demanding steps of the octree build process. Multiple threads could be implemented to handle independent parts of the octree construction, SDF computation, and inside/outside partition, taking advantage of the multiple cores present in most current CPUs.

Finally, this thesis presented a new approach for solving the inefficiency of *signed distance functions* in dealing with surfaces defined by triangle meshes. It used an octree data structure and showed relevant SDF estimations for complex meshes. As a result, SDFs can be used to describe typical triangle-based objects in many applications, such as modelling software and rendering engines.

Bibliography

- [1] C. Bálint, G. Valasek, and L. Gergó, “Operations on Signed Distance Functions”, *Acta Cybernetica*, pp. 1–12, 2018.
- [2] J. Hart, “Sphere tracing: a geometric method for the anti aliased ray tracing of implicit surfaces”, *The Visual Computer*, no. 12, pp. 527–545, 1996, ISSN: 0178-2789.
- [3] A. Bærentzen and H. Aanæs, “Generating Signed Distance Fields From Triangle Meshes”, Jan. 2002.
- [4] C.-Y. Chen, K.-Y. Cheng, and H. M. Liao, “A Sharpness Dependent Approach to 3D Polygon Mesh Hole Filling”, in *EG Short Presentations*, J. Dingliana and F. Ganovelli, Eds., The Eurographics Association, 2005. DOI: 10.2312/egs.20051012.
- [5] *Blender Project Homepage*, <http://www.blender.org>, Accessed: 2019-11-25.
- [6] G. M. Adel’son-Vel’skii and E. M. Landis, “An algorithm for the organization of information”, in *Soviet Mathematics Doklady*, 3rd ed., 1962, pp. 1259–1263.
- [7] S. Frisken, R. Perry, A. Rockwood, and T. Jones, “Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics”, *SIGGRAPH ’00 Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 249–254, 2000.
- [8] J. D. Foley, “Constructive Solid Geometry”, in *Computer Graphics: Principles and Practice*, 1996, pp. 557–558.
- [9] S. Gottschalk, “Separating axes theorem”, *Technical Report TR96-024*, Department of Computer Science, UNC Chapel Hill, 1996.
- [10] D. Badouel, “An Efficient Ray-Polygon Intersection”, in *Graphics Gems*, 1990.

- [11] *Assimp Library Homepage*, <http://www.assimp.org>, Accessed: 2019-11-11.
- [12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Huffman Coding”, in *Introduction to Algorithms, Second Edition*, 2nd, The MIT Press, 2001, pp. 385–392.
- [13] *Distance functions*, <http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>, Accessed: 2019-11-20.
- [14] *GLM Common Functions*, <https://glm.g-truc.net/0.9.4/api/a00129.html>, Accessed: 2019-11-20.

List of Figures

2.1	Extracted software solution.	8
2.2	Starting screen of the program.	9
2.3	Expanded view of the Instructions section of the Auxiliary Window. .	10
2.4	Expanded view of the Mesh Loading section of the Auxiliary Window.	11
2.5	Expanded view of the Statistics section of the Auxiliary Window. . .	12
2.6	Different angles for rendering of the mesh included in <i>Suzanne.obj</i> . . .	13
2.7	Expanded view of the Node List tab.	14
2.8	Outside (left) and inside (right) octree nodes of mesh <i>tetrahedron.obj</i> .	15
2.9	Visualization options applied to mesh <i>computer.obj</i>	17
2.10	Expanded view of the <i>Node List</i> tab with selected (dark blue) and hovered (light blue) nodes.	18
2.11	Expanded view of the <i>Automated Test</i> tab.	18
2.12	View of the <i>Automated Test</i> tab during the test routine.	19
3.1	Illustration of a ray-surface intersection.	23
3.2	Illustration of an octree data structure.	24
4.1	Workflow of octree algorithms following the natural order of exe- cution. The diagram partitions the algorithms into processing units (CPU or GPU) intended for implementation.	25
4.2	Visualization of octree describing the mesh contained in <i>Skull.obj</i> . . .	27
4.3	Octree of mesh described in <i>Skull.obj</i> with inside (left) and outside (right) nodes highlighted.	33
4.4	Octree of mesh described in <i>Tetrahedron.obj</i> before (left) and after (right) its optimization process. It features a reduction of 50% on the number of octree nodes, in this case.	37

5.1	Constructive Solid Geometry of plane, fractal, and octree. The addition of the plane to the intermediate images was implied, for simplicity. The fractal is a primitive featured by the previously existent <i>Sphere Tracing</i> renderer. The octree representation of mesh included in <i>Suzanne.obj</i> is a contribution of our software.	44
5.2	Class diagram of the components of the software solution	46
5.3	One-dimension visualization of the process that generates unique hashes for octree vertices. From binary divisions representing the octree depth, a corner is added. Afterwards, zeros to the right are discarded in the simplification process.	54
5.4	Finding a child node that contains p in two-dimensions.	60
5.5	Examples of feedback from the program to distinct user interactions. The interaction on the left shows a message rejecting an incorrect file input. The one on the right shows a directive informing the end of the <i>automated test</i> routine.	62
5.6	SDF estimation of the <i>Skull.obj</i> mesh from two different angles. . . .	64
5.7	<i>Bunny.obj</i> with negative offset (left), zero offset (center), and positive offset (right).	65
5.8	<i>Suzanne.obj</i> with negative offset (left), zero offset (center), and positive offset (right).	65
5.9	Estimation of the <i>Cybertruck.obj</i> mesh in two different angles. . . .	66

List of Codes

5.1	AABB class prototype	47
5.2	Triangle class prototype	48
5.3	Importer class prototype	49
5.4	OctreeNode class prototype	50
5.5	Vertex class prototype	51
5.6	Octree Data Structure Prototype	52
5.7	Auxiliary Graph Prototype	53