REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Khider – BISKRA
Faculté des Sciences Exactes et des Sciences de la Nature et de la Vie
Département d'informatique

N° d'ordre :.................
Série :.........................

Mémoire

Présenté en vue de l'obtention du diplôme de master académique en

Informatique

Option : **Images et vie artificielle**

Design and realization of a voxelization method via shaders

Par :

SERRAYE IBRAHIM

Soutenu le : /09/2020

Devant le jury :

Nom et prénom	Grade	Qualité
Babahenini Djihane	M.C.B	Rapporteur

To my familly...

To my friends...

Acknowledgments

First, I want to thank almighty ALLAH for giving me the will, patience and health to develop this work although the circumstances surrounding us I would like to express my profound gratitude to my supervisors Dr. Babahenini Djihane, for her involvement in this research work
and for the support they have given me, their patience, their availability and the relevance of their advice that have been of invaluable assistance throughout this work. Also, I would like to thank all my professors in the computer science department.

I extend my sincere thanks to the members of the jury, for having accepted to judge this work.

Finally, I thank my sisters, my big family and my friends for encouraging me during this year, and to have always been available when I needed it.

Abstract

In this thesis, we present a voxelization algorithm for surface models, this algorithm uses texture atlas and ray marching algorithm via the GPU to accelerate the execution time, We first load a 3D object, and according to the size of this object, we create a bounding box. Then we create a 3D atlas texture by rendering the object to a 3D texture, finally, we visualize the voxel-based object in the bounding box with a ray marching algorithm using the 3D atlas texture as a voxel grid. The whole algorithm traverses the geometric model only once and is accomplished entirely in GPU (graphics processing unit).
The test results show that the method can generate the 3D voxelization efficiently.

Key-words: GPU, voxelization, textutre atlas, ray marching, 3D texture.

Résumé

Dans ce mémoire, nous présentons un algorithme de voxélisation pour les modèles de surface, cet algorithme utilise un atlas de texture et un algorithme de suivi de rayons via le GPU pour accélérer le temps d'exécution. Nous chargeons d'abord un objet 3D, et en fonction de la taille de cet objet, nous créons une boîte englobante . Ensuite, nous créons une texture d'atlas 3D en rendant l'objet en une texture 3D, enfin nous visualisons l'objet à base de voxel dans la boîte englobante avec un algorithme de suivi de rayons en utilisant la texture d'atlas 3D comme une grille de voxel.L'algorithme entier ne traverse le modèle géométrique qu'une seule fois et est entièrement réalisé en GPU (unité de traitement graphique).
Les résultats des tests montrent que la méthode peut générer efficacement la voxélisation 3D.

Mots clés: GPU, voxélisation, texture d'atlas, suivi de rayons, texture 3D

ملخص

نقدم في في هذا البحث ، خوارزمية التكعيب لنماذج السطح ، وتستخدم هذه الخوارزمية أطلس النسيج وخوارزمية تتبع الأشعة باستخدام وحدة معالجة الرسومات لتسريع وقت التنفيذ. نقوم أولاً بتحميل كائن ثلاثي الأبعاد ، واعتمادًا على ملف حجم هذا الكائن ، نقوم بإنشاء مربع محيط. ثم نصنع أطلس النسيج لعرض الكائن ثلاثي الأبعاد في نسيج ثلاثي الأبعاد ، أخيرًا نظهر الكائن بناءً على خوارزمية التكعيب في ملف المربع المحيط بالكائن وذلك باستخدام خوارزمية تتبع الأشعة لأطلس النسيج ثلاثي الأبعاد.

الخوارزمية بأكملها تعبر فقط النموذج الهندسي مرة واحدة وتتحقق بالكامل في وحدة معالجة الرسومات. تظهر نتائج الاختبار أن الطريقة يمكن أن تولد بشكل فعال نموذج مجسم ثلاثي الأبعاد بالتكعيب.

الكلمات المفتاحية: وحدة معالجة الرسومات ، التكعيب ، أطلس النسيج، تتبع الأشعة ، الأطلس ثلاثي الأبعاد.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Volumetric representation is a substitution of the traditional geometric(surface) representation and plays an important role in computer graphics community since the 1980 s.[Pen04].The representation of Surface and volumetric models frames the two main classes of representing objects in computer graphics. Volume models(voxel based models) are used mainly where volumetric data are available (e.g., medical imaging, CSG, collision detection,analyze 3D data with deep learning). However, surface models are far more efficient to display and store.[The04] Often it is useful to convert surface data to voxels so that methods that work with regularly sampled data can be applied. This process called voxelization and produces a set of values on a regular three dimensional grid and approximates the shape of the model as closely as possible.[The04] This concept was first introduced by Arie Kaufman.[Pen04]

And with the increase of using voxel-based data and voxelizing surface data many researchers have studied voxel representation and used it in computer vision research field.[HWA20]

In the other hand where the geometric complexity in computer scenes is constantly increasing, rendering is becoming more expensive,the number of primitives increases leads to the use of voxel based models and voxelization techniques.[Shi00]

There are many strategies to conduct voxelization. Basically, they can be classified as surface voxelization and solid voxelization methods. Another common classification is binary and multi valued voxelization approaches. [Dec08]

Most previous work focuses on the sampling theory involved in voxelization and rendering. However, due to the rapid development of the modelling and sensor technologies, the size and complexity of the models are even larger. This puts high demands on the performance of the voxelization algorithm, especially for time-critical applications such as virtual medicine,rendering and collision detection.

Most of researchers rely on standard graphics systems for fast voxelization although graphics hardware is designed for surface rendering,it is often possible to utilize it in different tasks. However, to our best knowledge, achieving real-time frame rate for a moderate size volume resolution is still a challenge. Thanks to modern graphics hardware, its powerful flexibility and programmable, and future graphics hardware will probably be able to accelerate even the rendering of voxel models (by combining 3D textures and pixel shaders).[Shi00]

This thesis describes a voxelization algorithm for surface models. We decompose the task into two main stages, creating 3D texture atlas and visualizing with ray marching algorithm, which are accomplished entirely in GPU (graphics processing units).

The resultant volume is represented as 3D textures in the memory which can be reused conveniently. With the use of the GPU we can convert thousand of triangles into a 512^3 voxels in milliseconds.

The main goal of this thesis is to generate a 3D voxelized method using GPU to accelerate the time execution.

The rest of this thesis is organized as follows: chapter 2 gives a brief review of volume modeling. The different voxelization algorithms are outlined in chapter 3. The conception of this work is described in chapter 4. The implementation details and results with experimental results and discussions are presented in chapter 5. Conclusions and future work are addressed in chapter 6.

Chapter 2

Volume Modeling

2.1 Introduction

In few previous years, computer graphics methods are based on trigonometric geometric objects. Nowadays, volume modeling is a sub-field of Volume graphics and is concerned with the rendering of volume and geometric objects. Each object is stored in a discrete 3 dimensional space, where the basic element of this structure is called a voxel.

In this chapter, we will present the volume modeling domain and we will discuss the different volume modeling techniques .

2.2 Volume modeling

In this section, we will define the 3D models to construct a volume modeling and some application domains that use volume modeling.

2.2.1 Definitions

Before defining the volume modeling, we present what is a 3 dimensional model and the 3 dimensional modeling.

1. **3 dimensional model:** A three-dimensional mathematical representation (in X, Y, Z) of any three-dimensional object using a collection of points in 3D space, connected by various geometric entities such as triangles, lines, curved surfaces, and can be displayed visually as a 2D image through a process called 3D rendering.

   Two most common sources of 3D models:

- *3D Scanner:* Scanned directly into a computer from real-world objects using 3D scanners.
- *3D Modeling:* Designed by an artist or engineer using 3D modeling tools.in our work, we interest by this end (3D modeling).

2. **3 dimensional Modeling:** The process of developing a 3D model via specialized software such as Auto-desk Maya, 3Ds Max, Soft-image.

   Almost all $3D$ models can be divided into two categories Solid and Shell-boundary

   - *Solid 3D Model:* Solid 3D model allows to define the volume of the object they represent. This model type is realistic, but more difficult to build and mostly used for non-visual simulations such as medical and engineering simulations, and for specialized visual applications such as ray tracing and constructive solid geometry. The following figure 2.1 shows an example of a solid 3D model:

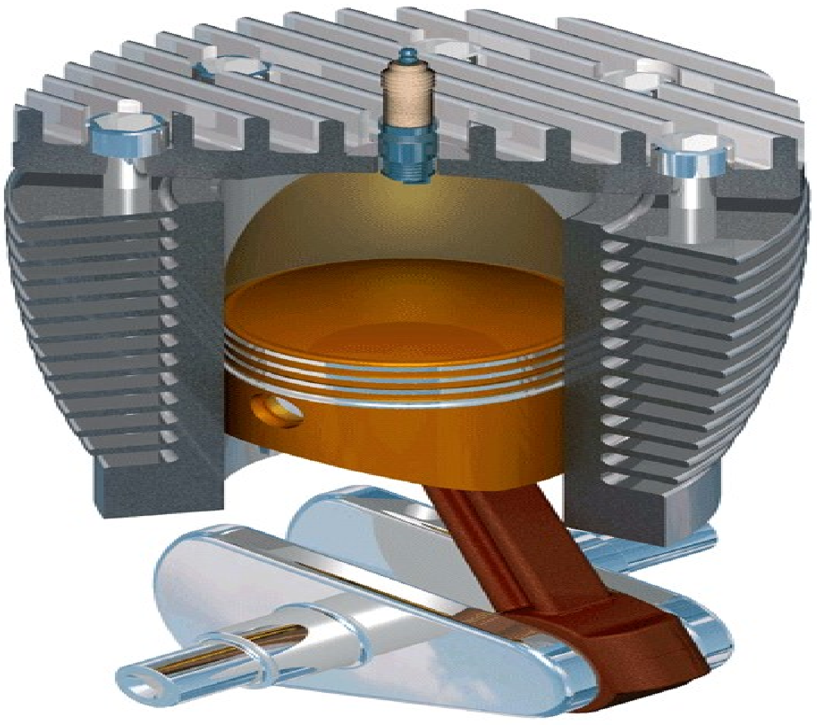

Figure 2.1: Solid 3D Model.

   - *Shell/Boundary 3D Model:* Shell/Boundary 3D Model represents only the surface object, such as the boundary of the object, not its volume (like a thin eggshells), The benefit of this model category, that is easier to work with it than solid models and almost all visual models used in games and film are shell models. As presented in the figure 2.2 below:

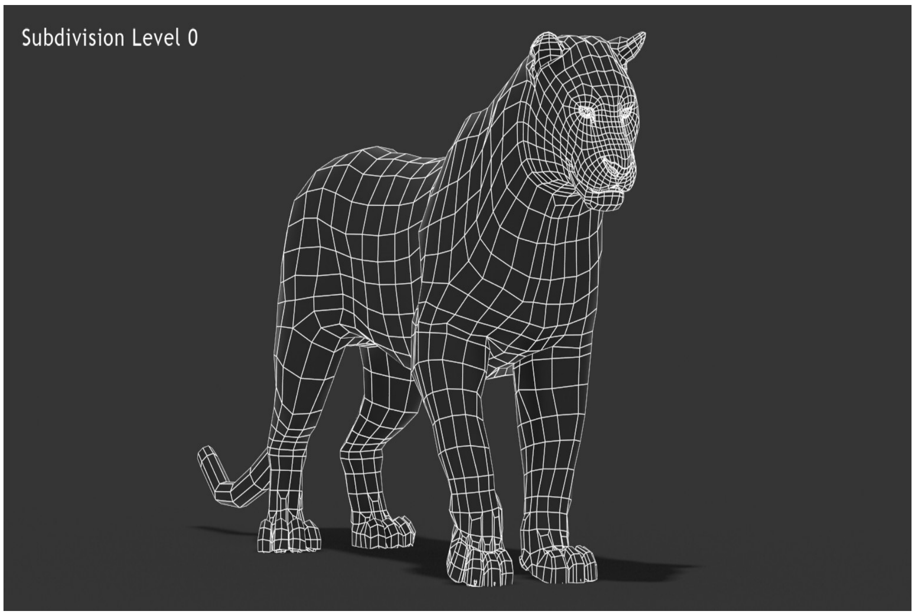

Figure 2.2: Shell Boundary 3D Model.

2.2.2 Volume modeling definition

In this section, we have three definitions of volume modeling

1. **Modeling volume data**

   Volume scanning devices such as MRI,CAT,3D simulation and free-hand ultrasound produce a value of a dependent quantity at various locations in space and each sample of the data consists of a position in space and the measurement or computation of an associated dependent variable. Invoking mathematical means of modelling and representing this type of data is one definition of volume modelling [Nie00]

   The next figure 2.3 represents the results of some scanning device (such as MRI or CAT).

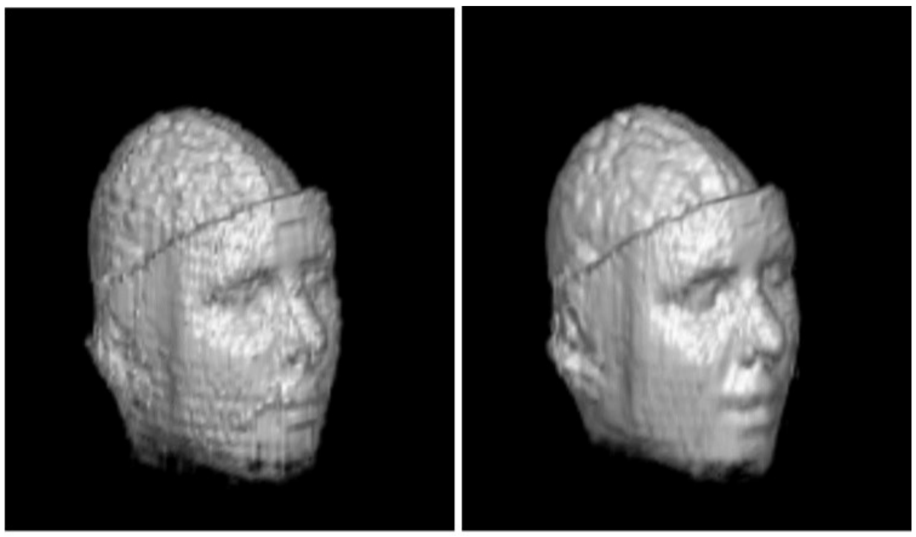

Figure 2.3: Examples of volume model extracted from a MRI and CAT scan.

2. **Analogy to Surface Modelling**

   In Figure 2.4, we see that the flow of information from top to bottom is “surface” to “volume” and left to right is “modelling” to “graphics”. The traditional computer graphics pipeline, which is illustrated in the top half of Figure 2.4, consists of a parametric surface model that is evaluated at a set of parameter values in order to obtain a polygon tessellation or approximation. The polygons are mapped by the viewing transformation to device coordinates and then scan converted. In a similar manner we can envision a “volume graphics” system that takes cells (the 3D analogues of polygons) that have an associated intensity at each vertex and scan converts them to a 3D frame buffer which subsequently is used to produce a volume rendering (either by hardware or software). In the diagram of Figure 2.4, volume modelling is represented by the oval, which is providing the information for the tessellation process. That is, volume modelling from this point of view is whatever is evaluated and used to produce the 3D tessellation with density values at the vertices.[Nie00]

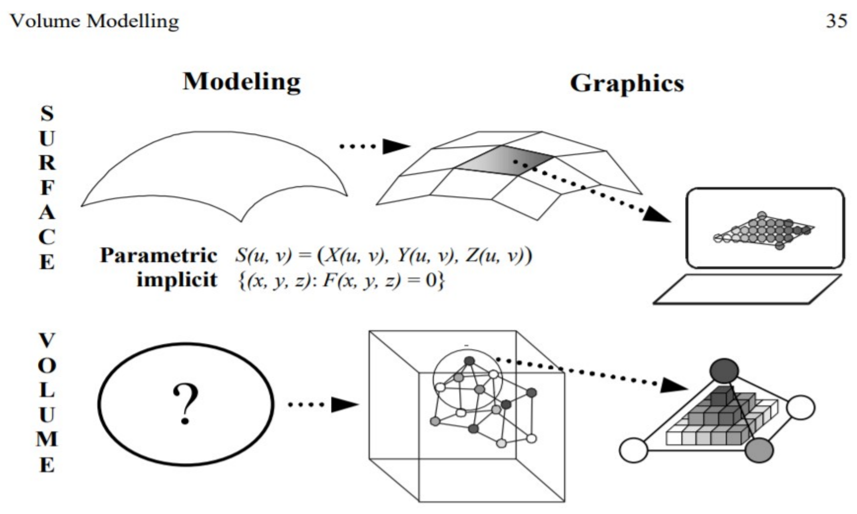
Volume Modelling
35
Modeling
Graphics
SURFACE
Parametric $S(u, v) = (X(u, v), Y(u, v), Z(u, v))$
implicit $\{(x, y, z): F(x, y, z) = 0\}$
VOLUME
?

Figure 2.4: Diagram depicting the analogy between surface modelling and volume modelling.

3. **Input to the Volume Rendering Equation**

   As an example Ray cast volume rendering images are based upon a rendering equation.

   As we can see, all three of these approaches (volume data, volume rendering integral and analogue to surface modelling) lead to the same definition of volume modelling.

   A volume model is a trivariate relationship whose independent argument is a position in 3D space and whose dependent argument is a scalar or tuple of scalars or even a vector. The volume model might also have the aspect of varying over time.[Nie00]

2.2.3 Application domain

The volume modeling is used in lot of domains, such as:

- **Wide Variety of Fields:** Game, Movie, Animation: characters, objects, backgrounds, animations, special effects, etc.

Figure 2.5: 3D Model for Games and Animations.

- **Science:** highly detailed models for calculations, chemical compounds.

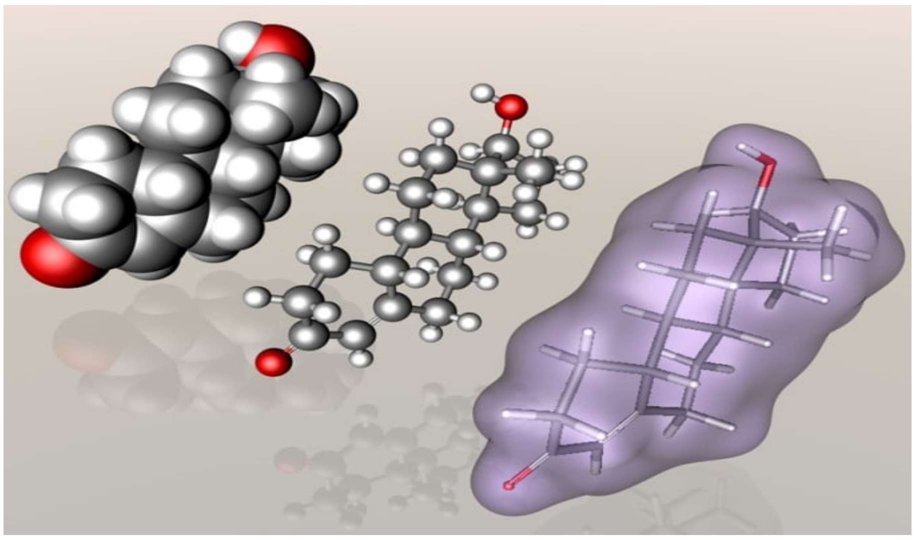

Figure 2.6: 3D Model for science domain.

- **Architecture:** demonstration of proposed buildings and landscapes though software architectural models.

Figure 2.7: 3D model for architecture domain.

- **Engineering:** designs of new devices, dynamics estimation tests for vehicles.

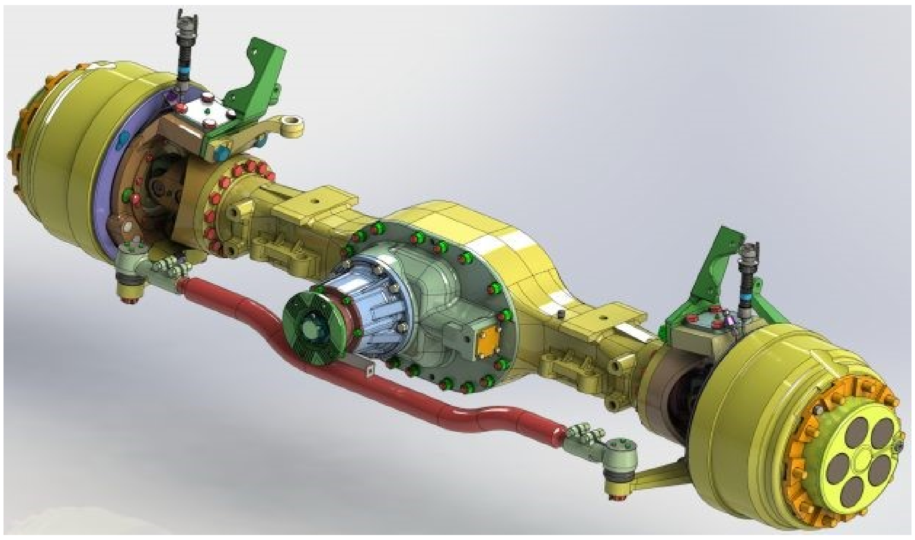

Figure 2.8: 3D Model for engineering domain.

- **Medical:** detailed models of organs.

Figure 2.9: 3D Model for medical domain.

- **Geology:** 3D geological models such as Google Earth.

Figure 2.10: 3D model for geology domain.

2.3 Bounding Volume

The intersection computation being very expensive, it is essential to reduce it by adding to the initial description of the scene a data structure to avoid unnecessary calculations such as the intersection with objects not located on the ray path.[TP06] A Bounding Volume (BV) is a volume that encloses a set of objects. The point of a BV is that it should be a much simpler geometrical shape than contained objects, so that doing tests using BV can be done

much faster than using the objects themselves. Examples of BVs: axis-aligned bounding boxes (AABBs) and oriented bounding boxes (OBBs). [mEHNH08] Bounding volumes are imaginary boxes that wraps around objects that are being checked for collision or computation for being so complicated, like pedestrians on or close to the road, other vehicles and signs. There is a 2D coordinate system and a 3D coordinate system that are both being used. Bounding volumes are simple geometric objects which fit around the objects. This structure is a tree of bounding volumes. In computer graphics, bounding box is used to reduce the amount of ray-object intersections, it is absolutely necessary to use a hierarchical data structure. [Bou14]
The following figure 2.11 presents a sphere into a bounding box, when the figure 2.12 shows the AABB tree of some simple objects:

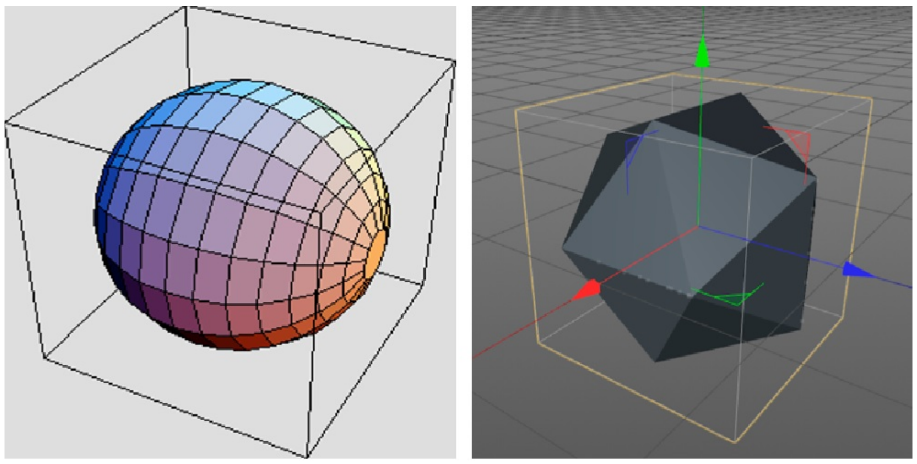

Figure 2.11: Bounding box.

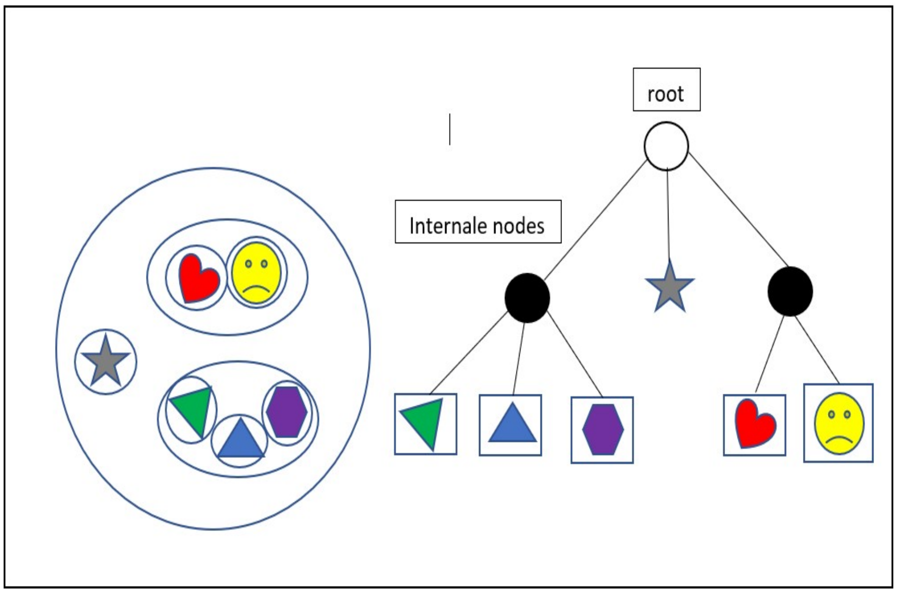

Figure 2.12: Axis Aligned Bounding box (AABB).

2.4 Bounding Volume Hierarchy (BVH)

A popular and effective way to filter out most non-colliding pairs of elements without spending a lot of computation is to use bounding volume hierarchies.

A bounding volume hierarchy covering an object is simply a tree in which each node is associated with a bounding volume, see Figure 2.13 The bounding volume at the root node of the hierarchy covers the whole object, and the bounding volumes at the children of each node together cover the portion of the object that the bounding volume at that node covers. The collection of bounding volumes in each level of a bounding volume hierarchy covers the entire object, and each successive level of the hierarchy gives a tighter and tighter covering of the object.

Bounding volume hierarchies have been used in practice for a long time.

suggested to use bounding volume hierarchies as a way to represent virtual objects in a scene in a hierarchical fashion to render them quickly from any particular view point.

This representation enables efficient clipping algorithm to identify parts of the scene that is not in the viewing window so that no further work is spent on rendering them.

The representation also allows efficient sorting of objects in a scene according to their distance to the view point, enabling fast computation of the visible surface.

Bounding volume hierarchies are also used to compute quickly the point where a light ray first intersects a scene, speeding up rendering in ray tracing. [Ngu06]

BVHs are also excellent for performing various queries. For example, assume that a ray should be intersected with a scene, and the first intersection should be returned. To use a BVH for this, testing starts at the root. If the ray misses its BV, then the ray misses all geometry contained in the BVH. [mEHNH08]

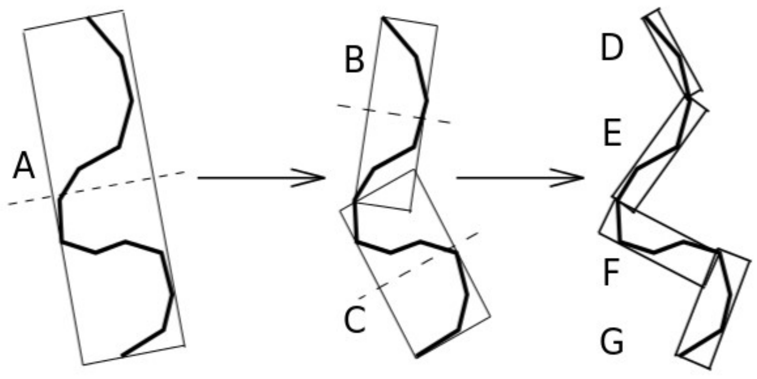

Figure 2.13: A bounding box hierarchy.

2.5 Spatial subdivision algorithms

Spatial subdivision is the process of decomposing the space in which the simulation takes place into subsets, and assigning every object or object primitive to the subset in which it lies.

Spatial subdivision algorithms can be classified based on the type of spatial data structure used and on the method that maps objects to the chosen structure.

We will consider three families of spatial subdivision approaches: Uniform Grid, kd-tree, Octree. [Laz12]

Spatial subdivision is widely used in ray tracing applications to cull the number of objects a ray has to intersect.

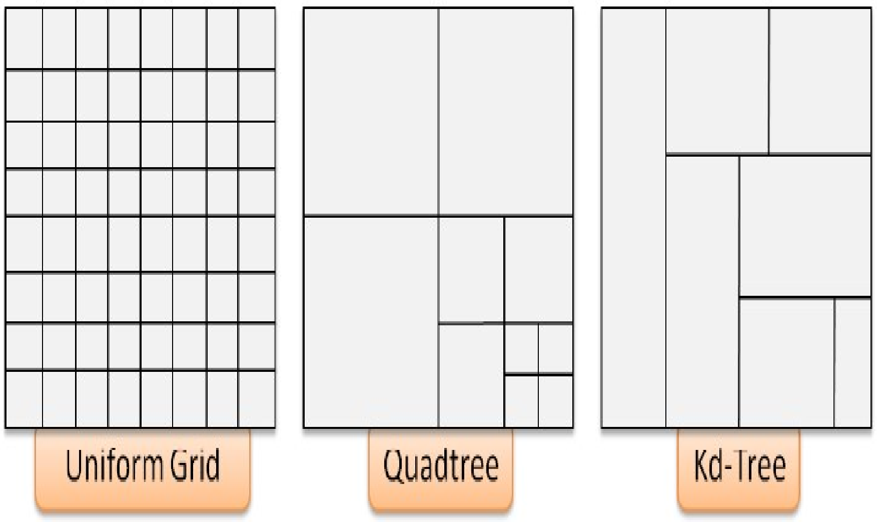

Figure 2.14: The main hierarchical structures used for spatial subdivision.

2.5.1 Uniform grid

The Uniform Grid is a responsive layout control which arranges items in a evenly-spaced set of rows or columns to fill the total available display space. Each cell in the grid, by default, will be the same size.

1. **Algorithm**

   The Uniform Grid has the same resolution in all the blocks throughout the domain, and each processor has exactly one block.

   The rectangular bounding volume of the scene is subdivided into a uniform 3D grid of rectangular cells. [Bou14]

2. **Advantages**

- Easy to construct.
- Easy to traverse.

3. **Disadvantages**

- may be only sparsely filled.
- geometry may still be clumped.

2.5.2 kd-tree

The kd-Tree, abbreviation for ”k-dimensional tree”,is a spatial partitioning of space with k dimensions allowing to structure the data according to their distribution in space.

The kd-Tree is a special case of “Binary Space Partitioning (BSP) trees”. BSP trees subdivide the k-dimensional space by dividing each enclosing volume into two sub-volumes by a plane of space, and recursively reiterating over these two new volumes thus obtained. BSP trees can therefore be represented by a binary tree where the two sub-volumes are the two children of the node corresponding to the bounding volume of higher level.

The cutting planes can be chosen according to the distribution of the data, so that it there are large bounding volumes, where there is not a great concentration of data and Conversely.

In the case of the kd-Tree, these separate planes are always chosen in such a way that their normal is a axes of the space coordinate system (planes always perpendicular to the axes as in figure 2.15). This makes it possible to simplify the construction, but also the path of the tree.[Fle08]

1. **Algorithm**

   Invented in 1970’s by Jon Bentley and the name kD-tree originally meant “3D-trees, 4D-trees, etc.”

   where k was the number of dimensions, Now, people say “kd-tree of dimension d”.

   The bounded 3D world to be ray traced is subdivided into cells of varying size. Each cell contains a list of objects which Pick axis that has “ best ” distribution of objects. and each subs cell contains a list of objects.

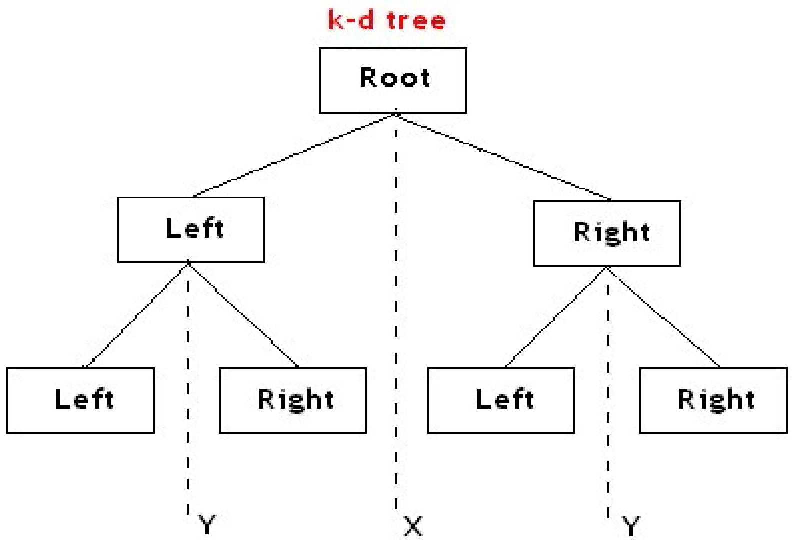

Figure 2.15: the hierarchical structures used for kd-tree.

2. **Advantages**

   - grid complexity matches geometric density.

3. **Disadvantages**

   - more expensive to traverse.

2.5.3 Octree

An octree is constructed by enclosing the entire scene in a box, then the box is split simultaneously along the three axes, and the split point must be the center of the box this create eight new boxes.

1. **Algorithm**

   In three dimensions the square is replaced by a cube and the division into four is replaced by a division into eight sub-cubes – hence octree, since oct = eight.

   An octree division divides each cube into eight sub-cubes. And each node corresponds to a single cube and has exactly eight sub-nodes.

2. **Advantages**

   - The octree branches very rapidly and it doesn't take very many levels to generate lots of nodes.

- Octrees are useful when you have to search a 3D space.

3. **Disadvantages**

- As a data structure isn't difficult.
- Difficult is managing the geometry.
- Dynamic construction often results in an unbalanced tree with areas of space being covered more finely than others.

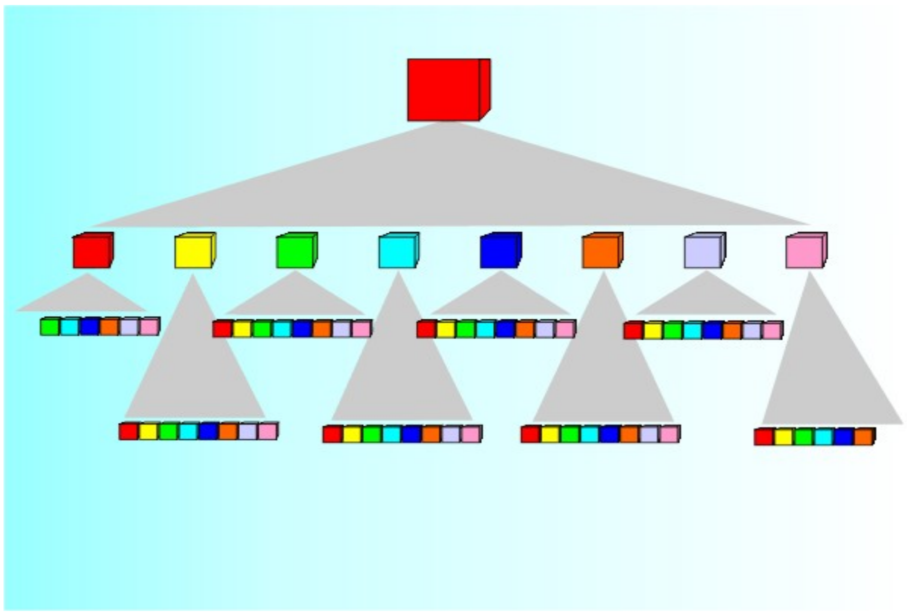

Figure 2.16: Octree algorithm representation.

2.6 Discussion

- For simplifying 3D objects representation in order to minimize the intersection computing time, We use the bounding volume and bounding volume hierarchy (BVH) techniques.

  The bounding volume can be used for any object and it is the unit part for the (BVH), On the other hand (BVH) can be used for more complex and large objects.

- For the optimization and acceleration purpose we use special subdivision algorithms.

  Those algorithms are divided into two main groups: uniform and non-uniform.

  - **Uniform group:** uniform grid.
  - **Non-uniform group:** octree, kd-tree.

- For choosing the best method of representation and the best algorithms that depends on your application your entries and hardware.

In the next table 2.1 we can see some comparison between the three methods.

	Spatial Subdivision Methods		
	Uniform	Non Uniform	
Method	Uniform Grid	Octree	KD tree
Difficulty	Easy	Hard	Very Hard
Resources Allocation	Very Height	Height	Low
Access	Direct (Fast)	Navigation (Slow)	Navigation (Slow)
Search	Very Slow	Fast	Very Fast

Table 2.1: A comparison between the three Spatial Subdivision Methods.

2.7 Conclusion

In this chapter we presented the terminology of volume modeling by his definition and application domain, then we explained the bounding box and the bounding box hierarchy techniques, alongside the algorithms of special subdivision, uniform grid, kd-tree, and octree, finally, we ended up this chapter by a discussion.

In the next chapter, by the name of voxelization we will see then voxelization types and categories followed by some voxel based methods.

Chapter 3

Voxelization

3.1 Introduction

During the last few years, the voxelization has seen a fantastic evolution in several domains, such as the rendering domain, medical imaging, video games....

The main idea of voxelization is to convert a 3D model (scene) into a voxel representation, its goal is to accelerate the intersection tests computation.

Traditional CPU-based volumetric representation make use of hierarchical representation such as octree, kd-tree,... that take several times for generating the data structure. Recently, with the evolution of modern GPU, it is possible to create voxelized scenes with millions of polygons in real-time.

In this chapter we will see the definition of voxel and voxelization, voxelization types according to the nature of data sorting. Moreover, we will focus on some voxel-based methods. In the end we will discuss the differences between those methods.

3.2 Definition

The voxelization is the process of transforming a set of triangles that represent the scene to a set of voxels. In other words, it describes the passage from a continuous representation to a discrete one.

The voxelization is the best approximation to the continuous geometry. It can be used to minimize the ray object intersection when computing the global illumination, [Rap16] therefore, the voxelization method is the process of transforming a 3D model made of polygons into a model made of voxels.

The 3D scene is inserted into an Axis Aligned Bounding Box (AABB). Then, the scene is subdivided into a grid of 3D cells. [Rap16]

Each cell is called voxel, so the voxel (form 'volume element') is the voxel is the 3D conceptual counterpart of the 2D pixel. it is the basic unit of a three-dimensional digital representation of a volume. It is a unit of volume and has a numeric value (or values) associated with it that representing some numerical quantity, such as the color and position [Yag93]

The following figure figure 3.1 demonstrates the main idea of the voxelization process

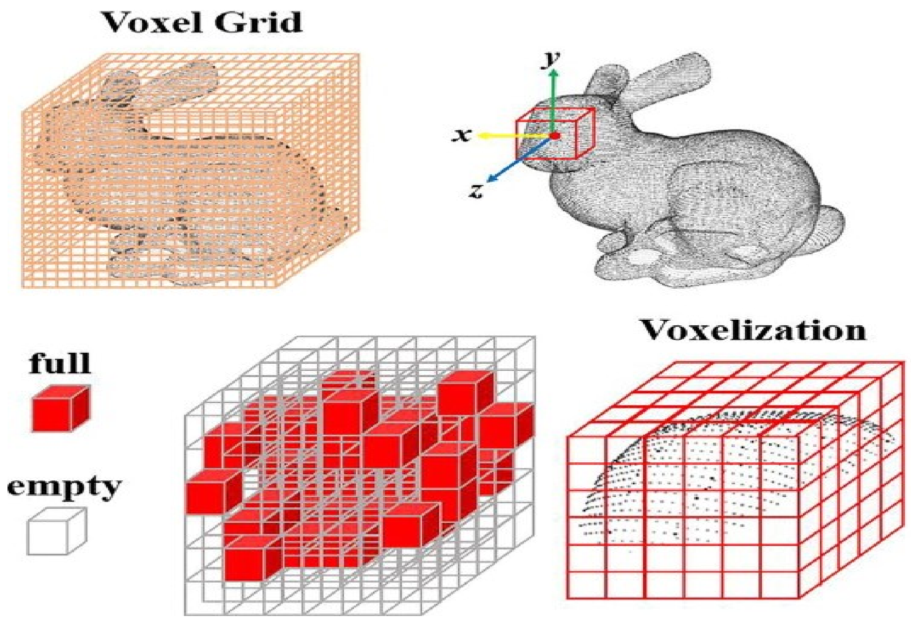

Figure 3.1: The transformation of a 3D mesh into a voxelization and its visualization through a voxel grid.

3.3 Voxelization types (according to the nature of data storing)

The voxel in the three-dimensional regular grid can contain some information about the scene.

The Existing applications of voxelization need to store either one bit (Binary voxelization) or multiple values (Multi-value voxelization) in a voxel.

3.3.1 Binary voxelization

The binary voxel model is stored as a stack of voxel layers represented as 'bit-arrays'. [Pat05].

The particular binary voxelization only uses a Boolean indicating the presence of the geometry or not. [Dec08]

The advantage of the binary voxelization is that has lower memory requirements, since it is sufficient to use a single bit to indicate whether a voxel is active. Especially, when we need to use the voxelization for just accelerate the intersection tests and for computing the visibility

The simplest representation of voxelization is binary encoding that represents if there is a triangle (or part of triangle) in the 3D voxel grid or not. Since each voxel is mapped into the set of (0,1), one bit is used for a voxel. Furthermore, binary voxelization means that the space is either occupied by the object (1) or not (0).

3.3.2 Multi-valued voxelization

In some cases, we need to store in the 3D cell grid, not only the presence of the geometry or not but all the vectors contained in the triangles like the position, normal, and color or texture coordinates, we called this voxelization type, multi-valued voxelization. According to previous researches the binary voxelization method and the multi valued voxelization method depend on how we store or represent the voxel data.

The figure bellow (figure 3.2) shows the difference between binary and multi-valued voxelization

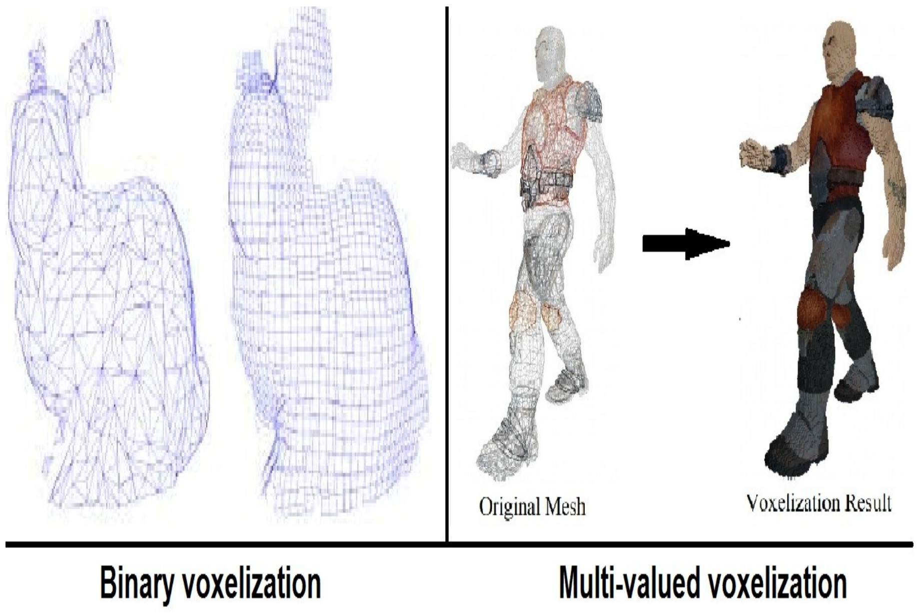

Figure 3.2: Binary and multi-valued voxelization.

3.4 Voxelization categories

3.4.1 Surface voxelization

Surface voxelization concerns transforming a continuous polygonal surface. [Sol19]

That's mean that the process of the surface voxelization algorithm do not care about the inner part of the object (model), it just considers the most front and viewed triangles of the model and change it to a surface of voxels.

In a surface voxelization, all voxels are set that fulfill some overlap or distance criterion with respect to a surface. [Sei10]

3.4.2 Solid voxelization

Solid voxelization represents the process of transforming a polygonal mesh into a voxel representation by associating each polygon of a mesh with the cells in the voxel grid. [Mon09] It capable to detect voxels lying completely inside the model,[Shi00] performing solid voxelization is more involved than surface only, as it requires voxelating the space contained within a model as well.

A straight forward approach simply sets the state of all voxels between two surface boundaries as being 'inside', by flipping bits or flags. [Sol19]

Solid voxelization sets all voxels considered interior to an object. [Sei10]

A solid voxelization where voxels are marked if they lie in the interior of the model. [Dec08]

The final idea of voxelization categories is all about two terms:

- Surface voxelization which consist of representing every surface on the model without considering the interior of it.
- Solid voxelization is the full representation of the model, so both the surface and the interior volume of the model will be transformed into a voxel model.

3.5 Voxel-based methods

3.5.1 Single-pass GPU Solid Voxelization and Applications

3.5.1.1 Main idea

The single-pass GPU solid voxelization and application introduced by Elmar Eisemann and Xavier Decoret as an extension of the original slice map algorithm, and the main idea of this method that it delivers a solid voxelization where voxels are marked if they lie in the interior of the model and how to efficiently convert a watertight model into a high definition binary volume representation with solid interior. [Dec08]

This is important in many contexts like simulations, path finding routines, or visibility computations.

3.5.1.2 The method

- **Input and output**

  As an Input this method take a watertight model and as an output, we get a voxelized model.

- **watertight model:** for a better understanding of this method, we have to explain the watertight model:

  In the world of 3D modeling and design, especially tri and quad poly model meshes (polygon mesh models) there is a chance that models can have gaps or holes. This is why we refer to the term ”watertight”.

- **Single-pass GPU Solid Voxelization**

  To achieve fast solid voxelization a point has to lie in the interior of the object and if for any ray leaving this point and calculate the number of intersections with the object’s surface, and by the number of intersections we determining whether a voxel lies inside the model or not. [Dec08]

  So, the voxel lies inside the model if n (the number of intersections) is odd ($n mod 2 = 1$). Consider for a moment that each voxel contains an integer counter and each fragment increments all voxels situated in front of it. [Dec08]

3.5.1.3 Implementation issues

This definition excludes some models from being usable with this technique and as an example where the definition of an interior is an issue.

- An object (model) with a crack in its hull and, therefore, does not define a proper interior. [Dec08]

- An object (model) contains a supplementary wall that separates the inner volume into two parts. Rays shot from one inner part into the other will intersect the model in a pair amount of intersections, while shooting vertically leads to a single intersection.

  This model is thus not watertight in the above sense. The same holds if the wall coincides with the outer hull. [Dec08]

- An object (model) illustrates a box en globing an inner box. Here, the definition implies that the inner box is an empty region. It is coherent, but not all models are conformed to this. [Dec08]

3.5.2 Real-time Voxelization for Complex Polygonal Models

3.5.2.1 Main idea

Real-time Voxelization is an efficient voxelization algorithm for complex polygonal models and the main idea behind this algorithm achieved by exploiting newest programmable graphics hardware.

First, they convert the model into three separated voxel spaces according to its surface direction. The resultant voxels are encoded as 2D textures and stored in three medium size sheets buffers called directional sheet buffers. These buffers are finally assembled in one worksheet, which records the volumetric representation of the target.

The whole algorithm traverses the geometric model only once and is accomplished entirely in the GPU (graphics processing unit), achieving real-time frame rate for models with up to 2 million triangles. [The04]

3.5.2.2 The method

A triangular mesh model is usually represented as a sequence of vertices with their positions, normals and texture coordinates associated with a list of indices that form each triangle.

Assume a regularly sampled volume P of the size $(2^L \times 2^M \times 2^N)$ voxels with spacing d be defined in the bounding box B of the model.

A voxel p.i.j.k stands for voxelized values including occupancy, density, color and gradient. [The04]

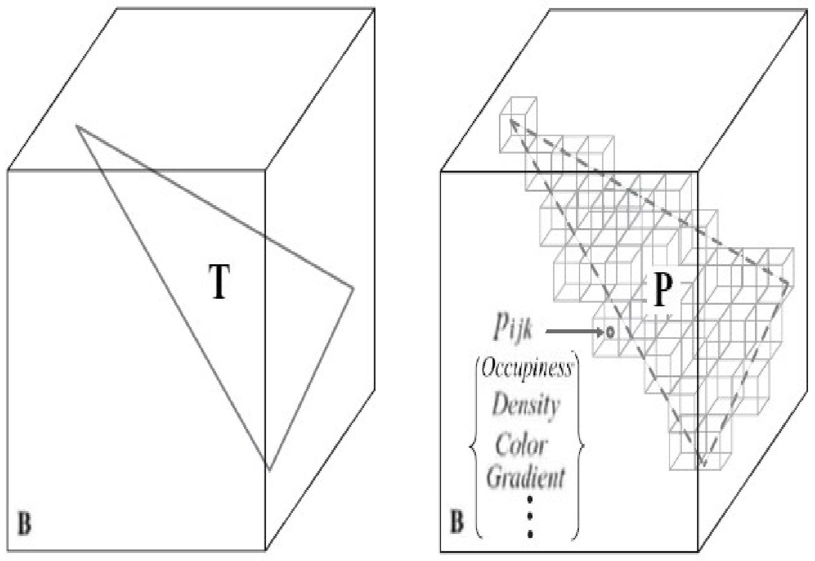

Figure 3.3: The input of voxelization is T and its output is an array of attributed voxels pijk.

In the standard rasterization hardware, triangles are scan and converted into a 2D frame buffer and Only the front most fragments are kept in the frame buffer storing the rasterization results, Whereas, voxelization is a 3D rasterization procedure and hence a separated voxel space is required.

The voxel space consists of an array of voxels that store all voxelized values and it can be represented as 2D or 3D textures in graphics hardware.

Since writing directly to 3D texture is not supported in mainstream graphics card of PC platform, they choose to encode the volume in 2D texture. And they call the texture (worksheet) as it records all voxelization information.

Note that each Texel in the graphics card typically consists of four components for red, green, blue and alpha channels respectively, depending on the bit-depth of each voxel, one Texel can represent one or multiple voxels. For instance, an 8-bit red component can store 8 voxels for binary voxelization.

The conversion from volume space to worksheet invokes an encoding procedure called texelization.

The conversion from the triangles to the discrete(separated) voxels representation can be accomplished in programmable graphics hardware.

The volume is generated slab by slab ,In other words, the worksheet is filled patch by patch. For each slab, only the triangles that intersect the slab are processed. Each chosen triangle is rasterized against an axis direction along which it has the maximum projection area.

The position of each voxel is transformed to its 3D volume coordinates immediately, these coordinates are used to find the correct position in the worksheet.

Note that, the discrete voxel space is only a virtual concept and is not explicitly represented ,In order to add a voxel to the worksheet, a blending operation is carried out at corresponding location, when all triangles are processed, the worksheet encodes the discrete voxel space.

The 2D rasterization in standard graphics hardware involves a 2D linear interpolation process.

If a triangle is parallel to the rasterization direction, the interpolation process results in a line segment in the discrete voxel space. Therefore, a triangle should be rasterized along the axis direction that is most parallel to its orientation. And three directional sheet buffers are used as intermediate space during the rasterization and texelization procedures.

Each sheet buffer represents a part of the discrete voxel space. After these sheet buffers are accomplished, an additional reformulation process is performed to trans code them to the final worksheet, in this stage, each element is first transformed to the discrete voxel space and then encoded to the appropriate texel in the worksheet. Actually, the worksheet reformulates the slabs of the volume along a desired axis direction.

To sum up, the voxelization algorithm consists of three stages as follows:

- Rasterization The triangles are rasterized to the discrete voxel space.
- Texelization Each voxel is encoded and accumulated in some directional sheet buffer.
- Synthesis Three sheet buffers are trans coded to the worksheet representing the final volume.

3.5.2.3 Implementation issues

Typically, there are three deficiencies for graphics hardware-accelerated voxelization algorithms:

1. First, the performance decreases greatly following the increase of the scene complexity and the volume resolution since the model is traversed multiple times.
2. Second, the access of frame buffer demands high bandwidth between main memory and video memory, which is still a heavy bottleneck in modern graphics hardware.
3. Third, the voxelization results are directly stored in color or depth buffer and cost lots of video memory. Therefore, it is difficult to afford interactive frame rate at the volume resolution of $256 \times 256 \times 256$ and above.

3.5.3 Hardware Accelerated Voxelization

3.5.3.1 Main idea

Hardware Accelerated Voxelization introduced by Fang as a hardware accelerated approach to the voxelization of a wide range of 3D objects, including curves/surfaces, solids, and geometric and volumetric CSG (Constructive Solid Geometry) models.

The algorithms generate slices of the object models using a surface graphics processor to form the final volume representations.

Boolean operations in a volumetric CSG model are carried out using frame buffer blending functions. by storing the resulting volume in the 3D texture memory, the algorithms can also volume render the models in real time by 3D texture mapping.

As a result, this approach is able to perform interactive object manipulations and Boolean operations in an intermixed environment of geometric and volumetric objects under a unified volume graphics framework. [Pen04]

3.5.3.2 The method

- **Curve/Surface Voxelization**

  The curve/surface voxelization algorithm generates a volume representation for a scene consisting of an arbitrary number of curve or surface objects that can be accepted by OpenGL[Kau90].

  These include various types of lines and polygons, quadratic curves and surfaces.

  The algorithm is based on the fact that surface graphics displays a curve or a surface by a 2D scan conversion (or rasterization) process. When only a slice of the object is displayed, the result is essentially a slice of the volume from a 3D scan conversion. Since 2D scan conversion is implemented in hardware in modern graphics systems, 3D voxelization must be able to take advantages of it for better performance.

  A bounding box is first defined over the scene as the volume space for voxelization.

  The algorithm proceeds by moving a cutting plane, called Z-plane, parallel to the projection plane, with a constant step size in a front-to-back order. [Pen04]

The thin space between two adjacent Z-planes within the volume space is called a slice (as shown in Figure 3.4).

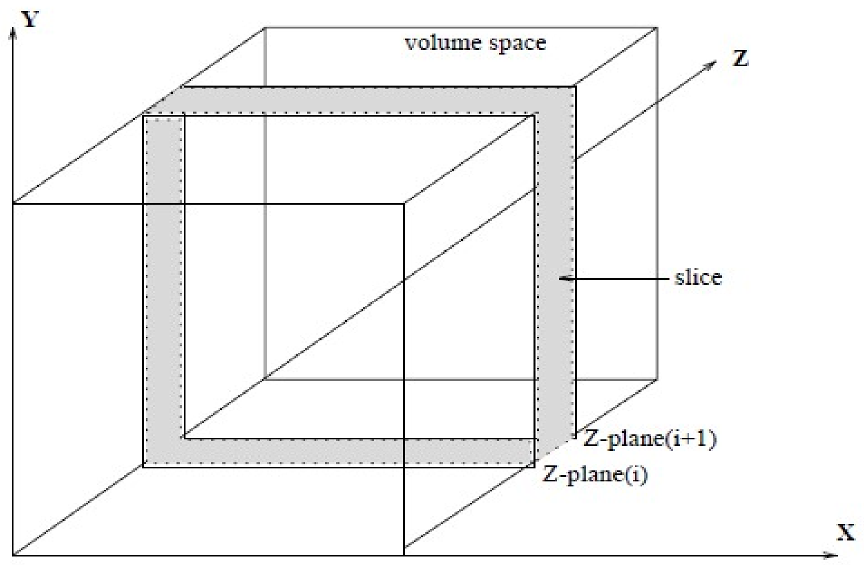

Figure 3.4: Volume Slicing.

For each new Z-plane, the algorithm defines the new slice as the current orthogonal viewing volume, and renders all the curve and surface primitives using standard OpenGL procedures.

The resulting frame buffer image from the display of this slice becomes one slice of the voxelization result of the 3D scene.

The Z-distance between adjacent Z-planes determines the Z-resolution of the volume representation.

The resolutions in the X and Y directions are defined by the size of the display window. [Pen04]

- **Solid Voxelization**

  In the solid voxelization algorithm, multiple solid objects may be considered as one solid representation as long as the boundary surfaces of the different objects do not intersect each other.

  For a given solid representation, the voxelization algorithm aims to generate the set of voxels that are either inside or on the boundary of a solid object.

  The algorithm operates similarly to the surface voxelization algorithm, it proceeds slice-by-slice in a front-to-back order, with the boundary surfaces displayed within each slice.

The frame buffer pixels filled by the display of the current slice constitute the boundary voxels of the solid object within this slice.

The object's interior voxels, which also need to be filled, are however not explicitly scanned by this process.

To generate the interior voxels, the algorithm employs the frame buffer blending function feature in OpenGL with a logical XOR operation to carry the boundary information to the interior of the solid object.

This approach is based on the principle that when shooting a ray from a pixel to the object space, the entering points and the exiting points on the ray against a solid object always appear in pairs, and the voxels between each pair of entering and exiting points are the interior voxels.

A frame buffer blending function is a function that blends the current pixel color values in the frame buffer and the color values of the incoming pixels (to be written to the frame buffer).

The XOR blending function performs a bit-wise XOR operation between the incoming color bits and the existing color bits in the frame buffer.

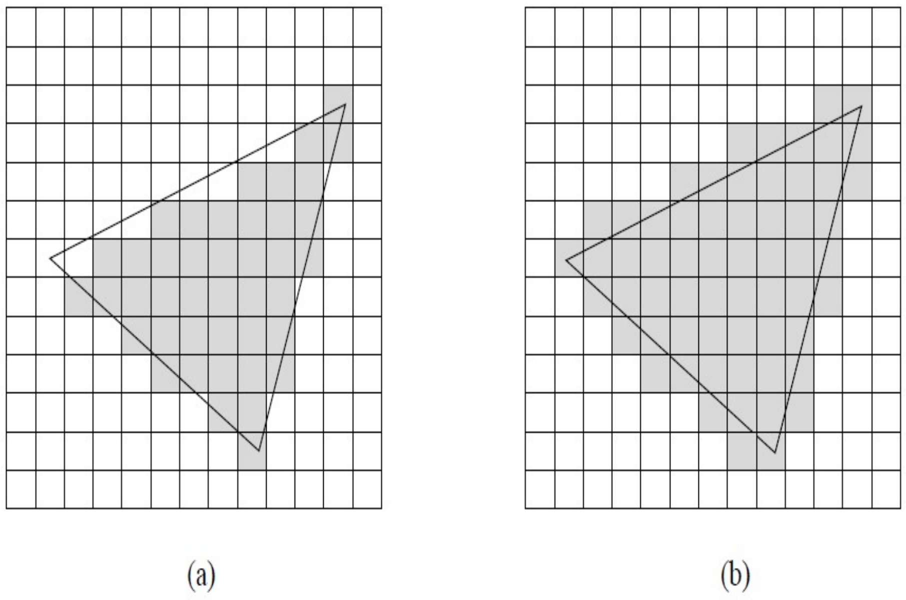

Figure 3.5: A 2D analog of the solid voxelization algorithm; (b) with surfaces super-imposed.

3.5.3.3 Implementation issues

Missing thin regions. When some part of the solid object is very thin (thinner than a slice) in the Z-direction, it could happen that the entering and exiting points for some pixels fall within the same slice.

This algorithm depends on the hardware implementations of the graphics API, so certain properties of the voxelization results are difficult to access. For instance, anti aliasing is used to ensure that the voxelized surfaces are connected and tunnel free. But the exact result may change slightly on different systems due to the different anti aliasing implementations.

The speed of the algorithms also depends mostly on the graphics subsystem rather than the CPU.in other word when the number of objects in the scene is increased, the voxelization process will be proportionally slowed down.

3.5.4 Efficient Hardware Voxelization

3.5.4.1 Main idea

Efficient Hardware Voxelization is an improve on the algorithm proposed by Karabassi and all so that it can be applied to a wider range of objects without sacrificing performance. [Dav97]

This algorithm uses the depth and stencil buffers, available in most popular graphics hardware, to achieve high performance for the voxelization of surface models that have a random topology and it is suitable for both polygonal meshes and parametric surfaces. [Dav97]

3.5.4.2 The method

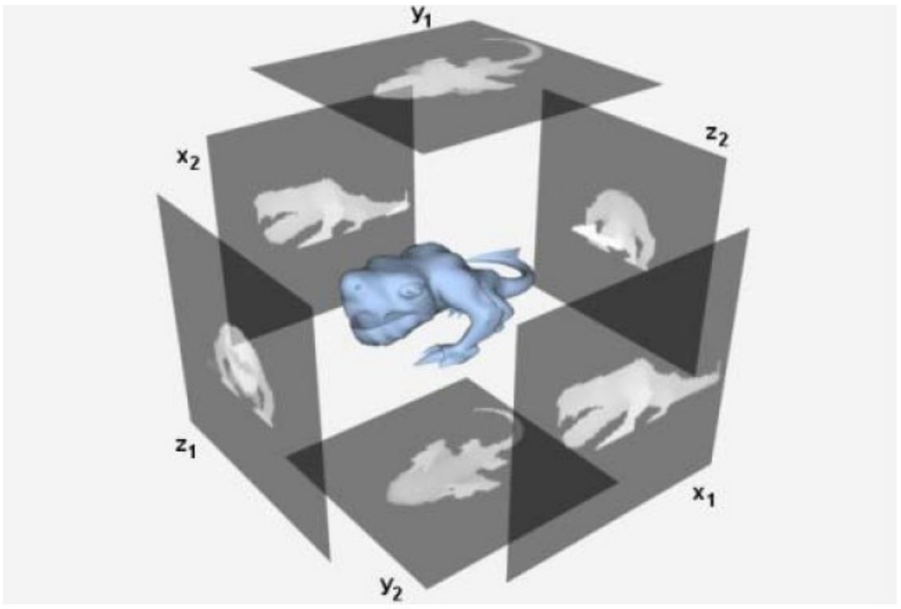

Figure 3.6: Six Z-buffers for a 3D object.

- **Voxelization using six single layer Z-buffers**

  Karabassi algorithm rendered six Z-buffers for each object using OpenGL, two per axis, as shown in (figure 3.6) The camera is placed on each of the six faces of a computed bounding box of the object and the Z-buffers are taken using parallel projection.

  For each pair of opposite directions (e.g., +X, -X). A voxel is inside the object only if it is 'inside' the Z-buffer values for all three pairs. [Dav97]

  assuming that the object surface is closed any ray in any direction from every point inside the object must intersect the object's surface an odd number of times.

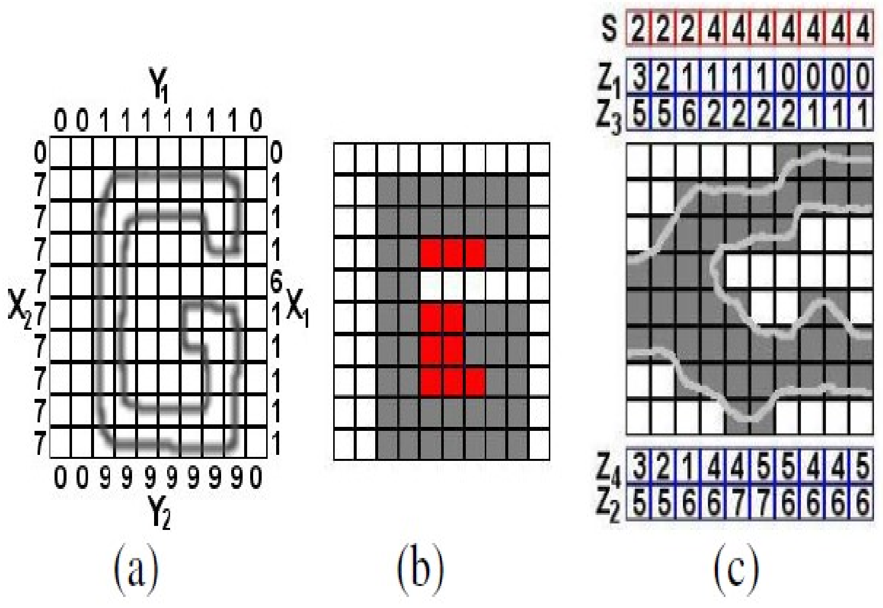

Figure 3.7: Letter 'G' and its Z-buffers (X1, X2, Y1, Y2). (b) Erroneous voxels marked in red. (c) Double Z-buffers and stencil buffer.

- **Voxelization Algorithm**

  Karabassi's algorithm can handle convex and a subset of concave objects.

  The limitation lies in the fact that certain surface details (like concavities) may not be visible from any of the six directions, it is not possible to voxelize such parts of the object correctly. As seen in (Figure 3.7) (a), (b) the letter 'G' was voxelized incorrectly because the concavity is not visible from any direction. Furthermore, it is not easy to tell whether a concave object has been correctly voxelized. [Dav97]

  So, the goal of this Efficient Hardware Voxelization algorithm is a voxelization algorithm that smooth the above problems while using only standard graphics hardware.

And, this algorithm proposes two improvements to Karabassi's algorithm that significantly increase the range of objects that are correctly handled:

1. The first is the use of an arbitrary configuration of more than six Z-buffers.
2. The second is the combined use the Z-buffer and the stencil buffer which can handle up to one concavity per direction.

3.5.4.3 Implementation issues

The cost for this is an extra Z-buffer per direction and a stencil buffer per pair of opposite directions.

This algorithm assumes that the object is strictly closed and has no irregular faces. and also observed that triangles perpendicular(vertical) to the Z-buffer planes may produce certain artifacts, especially in lower voxel resolutions.

This is attributed to the non-linear accuracy of the Z-buffer.

3.5.5 Surface Scanning-based Texture Atlas for Voxelized 3D object

3.5.5.1 Main Idea

Surface Scanning-Based Texture Atlas generator for Voxelized 3D Object (SSTAG) is the first method that attempt to use a surface scanning scheme for a texture atlas because texture atlases are widely used for representing voxel colors and , the texture atlas is typically utilized for encoding voxel colors using image/video encoders , but in this work they represent a 3D object as slices, and texture strips are obtained by scanning voxels along the surface for each slice, the texture strips represent the voxel colors while preserving the connectivity of the voxels.[HWA20]

The framework of the surface scanning-based texture atlases generator (SSTAG) is shown in figure 3.8.

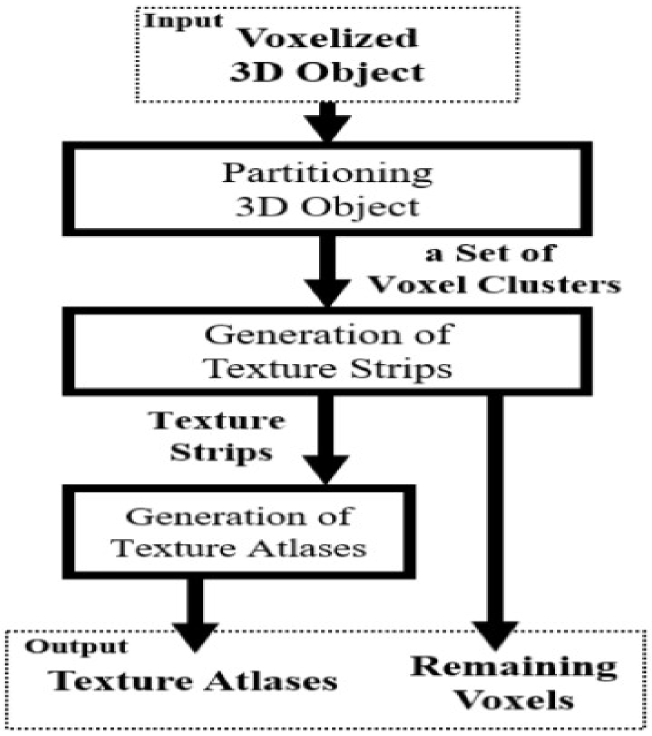

Figure 3.8: Surface scanning-based texture atlases generator (SSTAG).

3.5.5.2 The Method

This method is consists of the following steps:

- **Partitioning 3D object**

  a voxelized 3D object is represented as a set of slices (clusters) using a boundary tracing algorithm.

As shown in figure 3.9, they defined a slice of voxels that have the same X-coordinate (or Y-coordinate or Z-coordinate), and named it X-slice (or Y-slice, or Z-slice).and they define the direction of the slice which is the axis of the slice.and each slice can be considered as a 2D image. [HWA20]

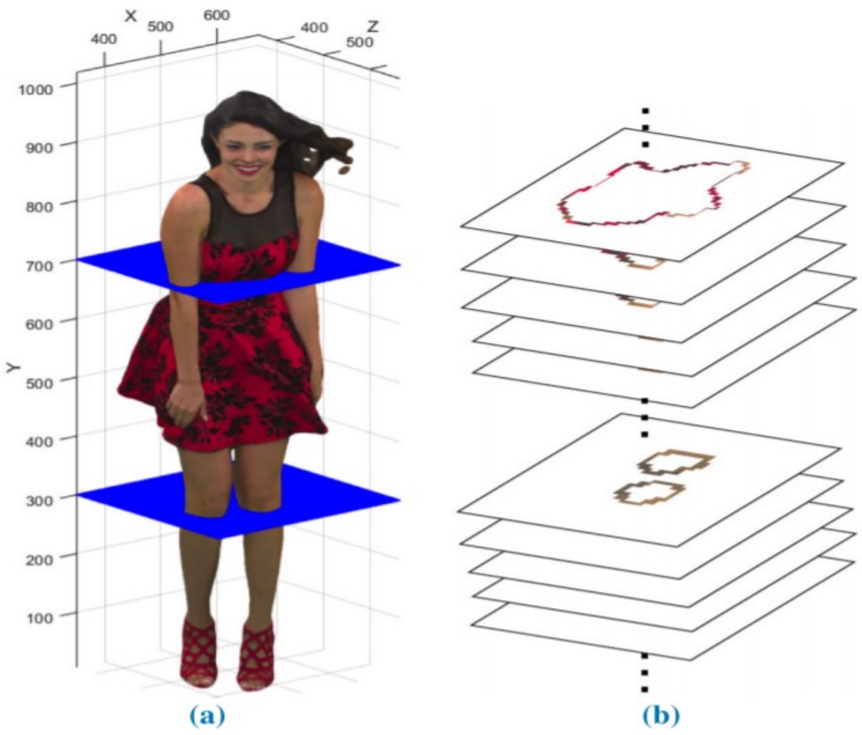

Figure 3.9: Slice representation of voxelized 3D object: (a) Input voxelized 3D object. (b) Set of slices. 2D blobs exist in each slice.

- **Generation of texture strips**

  In this process, voxels are scanned along the boundary voxels using a boundary tracing algorithm for each slice and texture strips are generated by the colors of the scanned voxels.(figure 3.10).

  Through boundary tracing, we generate sequences of voxels considering their connectivity on slices without empty spaces, even if the shape of the 3D object is complex, Here, the slices are regarded as binary images and pixels in the binary images corresponded to the voxels on the slices.

Finally, texture strips are generated by replacing the voxels in the sequences with their corresponding colors.

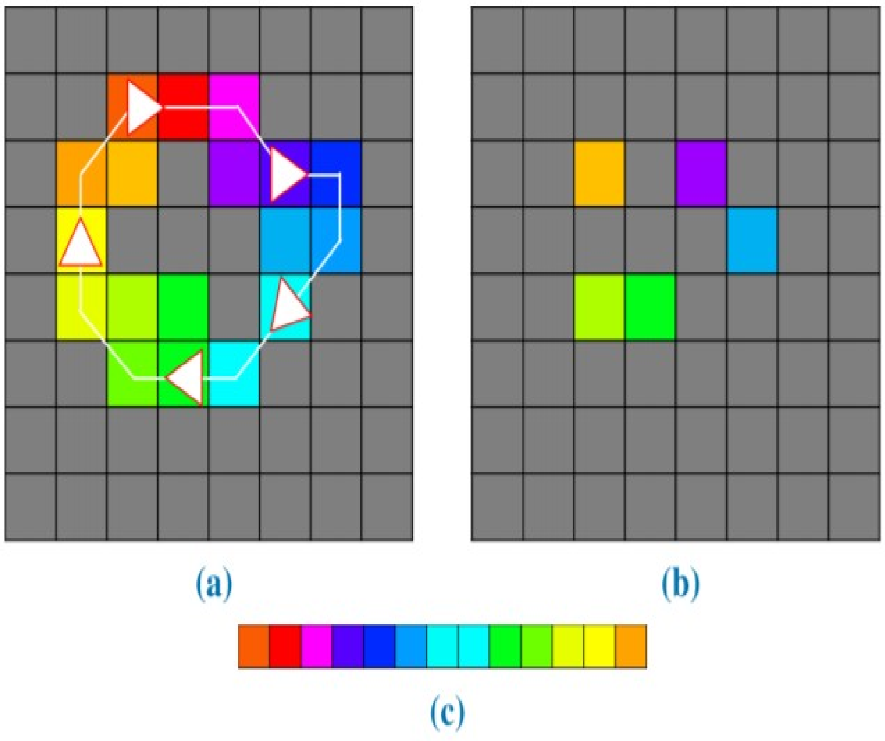

Figure 3.10: Generation of texture strip on slice.

(a) Slice of 3D object with arrows indicating sequence of boundary pixels generated by a boundary tracing algorithm.

(b) RVs that are not scanned.

(c) Texture strip generated from (a).

- **Generation of texture atlases**

  In this process, texture strips combine to generate texture atlases, by aligning the generated texture strips and stacking them along the rows or columns of a 2D plane in the order of slices. as shown in figure : 3.11

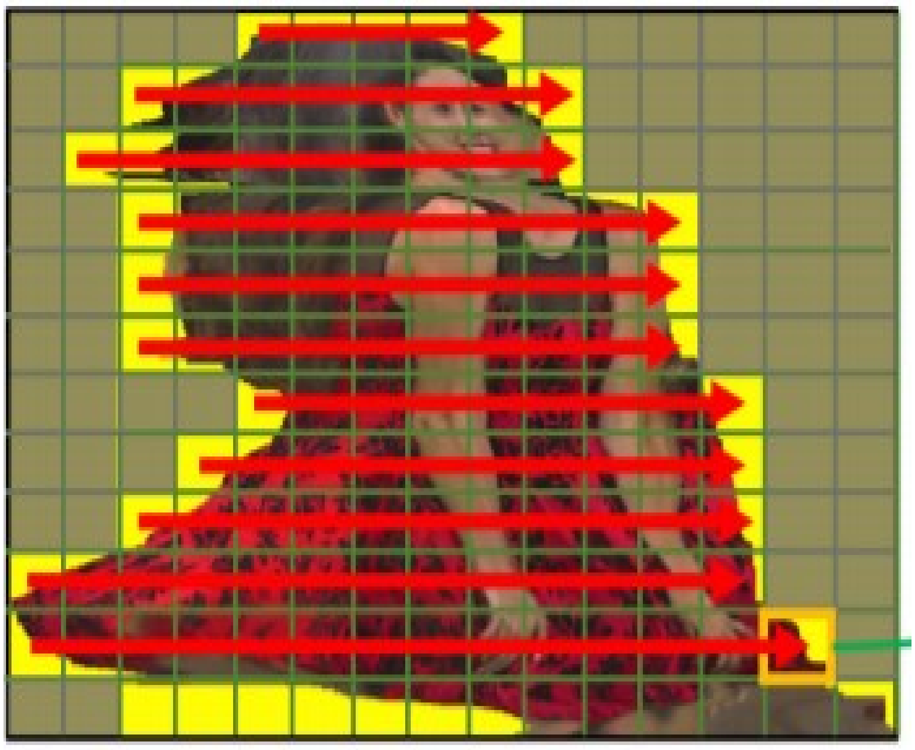

Figure 3.11: Generation of texture atlases.

3.5.5.3 Implementation issues

Even though all voxels are supposed to be scanned in the above processes, the SSTAG only processes the boundary voxels on each slice. Hence, there exist voxels that are not processed by the SSTAG, and they name them as remaining voxels (RVs). To process all voxel colors, the SSTAG is performed repeatedly, until all voxel colors are processed. As shown in figure below (figure 3.12):

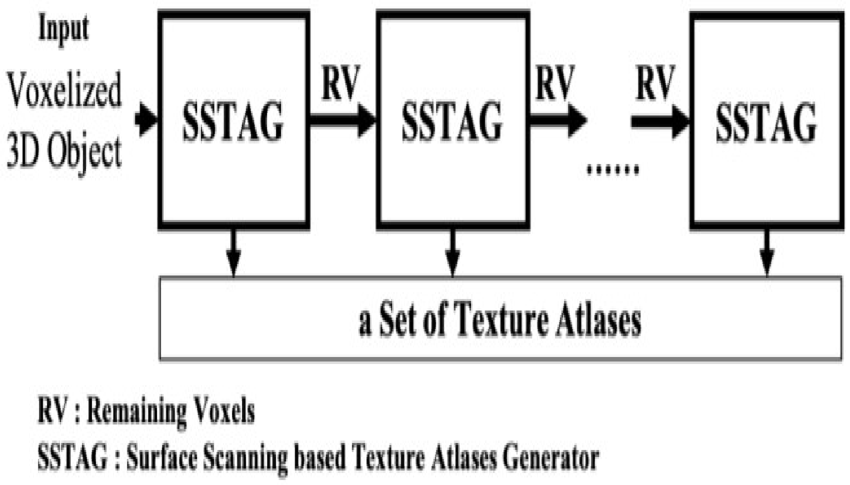

Figure 3.12: Framework of proposed texture atlas generation.

If the input data has a small number of connected voxels, it is difficult to maintain the spatial coherence of voxels in the proposed texture atlases.

When it comes to high standard deviations of the length of the texture strips, the proposed method shows performance degradation.

In the alignment, the connectivity of the voxels is measured approximately.[HWA20]

3.6 Discussion

For the sake of better understanding of the voxelization methods and preparing for our work, a comparison between the different approaches explained above had to be made, considering these variables:

Voxelization types (according to the nature of data storing), Voxelization categories, watertight models.

And represented it in the next table. 3.1

	Voxelization types		Voxelization categories		
	Binary voxelization	Multi-value voxelization	Surface voxelization	Solid voxelization	watertight models
Single-pass GPU Solid Voxelization and Applications	*			*	*
Real-time Voxelization for Complex Polygonal Models		*	*		*
Hardware Accelerated Voxelization	*			*	*
Efficient Hardware Voxelization	*			*	*
Surface Scanning-based Texture Atlas	*		*		*

Table 3.1: Comparison between the different voxelization presented in this chapter.

In the end of this discussion we gain the information that every one of the previous algorithms aim to achieve voxelization representation for watertight models using different approaches and methods.

3.7 Conclusion

We presented in this chapter the definition of a voxelization method as well as the different voxelization types. Five voxelization algorithms and methods were presented in this chapter, alongside the comparison between them after we defined the voxelization terms, its types, and categories.

In the next chapter we will discuss the conception of 3D voxelization method , along side with our goals and the main idea of this work.

Chapter 4

Conception of 3D voxelization method

4.1 Introduction

In this chapter we will determine the goals of our project and the main idea behind it, next we will define our global conception followed by the detailed conception for the CPU code, GPU code and the data passes between them.

4.2 Goals

Our goal is to.

- Voxelize 3d model (that's mean going from triangular mesh representation to voxel-based representation).
- Apply this work on the GPU using shaders to accelerate the execution time.
- Compute the execution time of the different 3D models.

4.3 Main idea and motivation

Data acquired by depth sensors, Lidar, and stereo cameras are commonly converted to voxels because they are more convenient to manipulate than other 3D data representations. Moreover, deep learning, one of the most interesting topics in recent years, utilizes voxels to analyze 3D data. In particular, because 3D data are acquired from depth sensors or stereo cameras, with which it is difficult to collect information from inside an object, many studies in computer vision fields typically use voxelized surface data. With the increased interest in voxelized surface data, many researchers have studied voxel representation as well.

The main idea behind this work is a fast geometry voxelization using shaders. In the first time, we load a 3D object model, and we insert it into AABB structure, then we compute the voxelization using atlas voxelization method, and finally, we visualize the result with ray marching method. (For more details, please go to the section 4.4.2)

4.4 Conception

4.4.1 global conception

In this section, we present our general conception of our application. first, we load our 3D model using the Open Asset Import Library (Assimp), then we create a bonding box for the object , after that we create an atlas 3D texture using frame buffer and an empty 3D texture data , finally we use ray marching algorithm to calculate all the intersections and find the last color for every voxel.

In the next scheme figure 4.1 we present our global conception

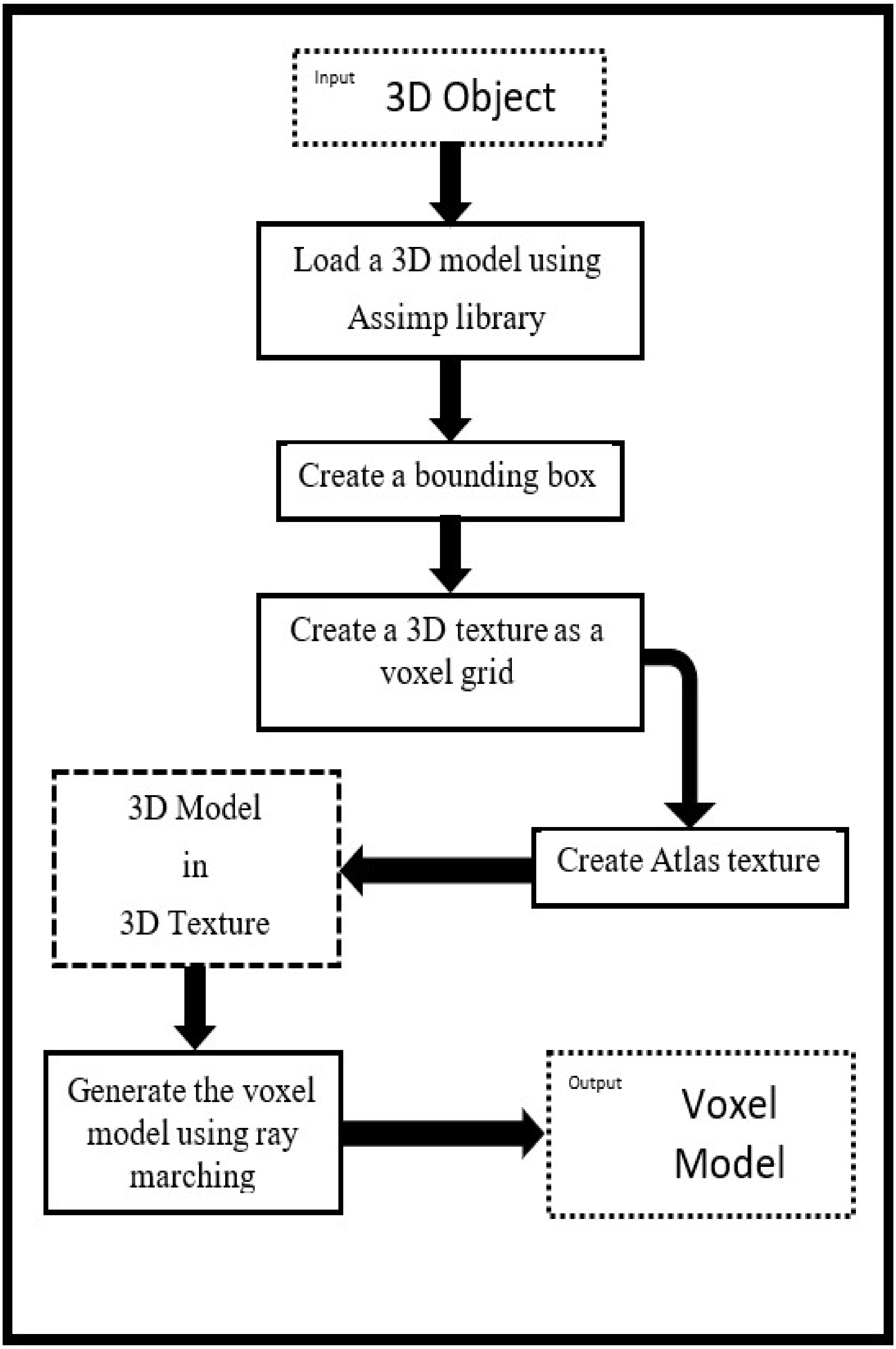
Input
3D Object
Load a 3D model using Assimp library
Create a bounding box
Create a 3D texture as a voxel grid
Create Atlas texture
3D Model in 3D Texture
Generate the voxel model using ray marching
Output
Voxel Model

Figure 4.1: Conception general of the voxelization algorithm.

4.4.2 detailed conception

In this section, we present our detailed conception of our application.starting by explaining each part of the general conception and defining all the methods and algorithms used in these parts.

- **Input:** Our application takes a 3D object as an input and specifically a triangle mesh base object in a raw file like (obj files).

- **Load a 3D model:** This work requires in the first time a 3D object with triangles mesh as an input data, thus for loading these objects , and to ensure the success of loading a wide variety of model types we used the Open Asset Import Library (Assimp) to load our object , Assimp will help us read the raw data from an obj file and turn them to vector data (vertex,normal,UV coordinate),and store theme in the memory when we need them ,and can clear the data when we done.

  The following figure 4.2 shows an example of a 3D triangle mesh base object.

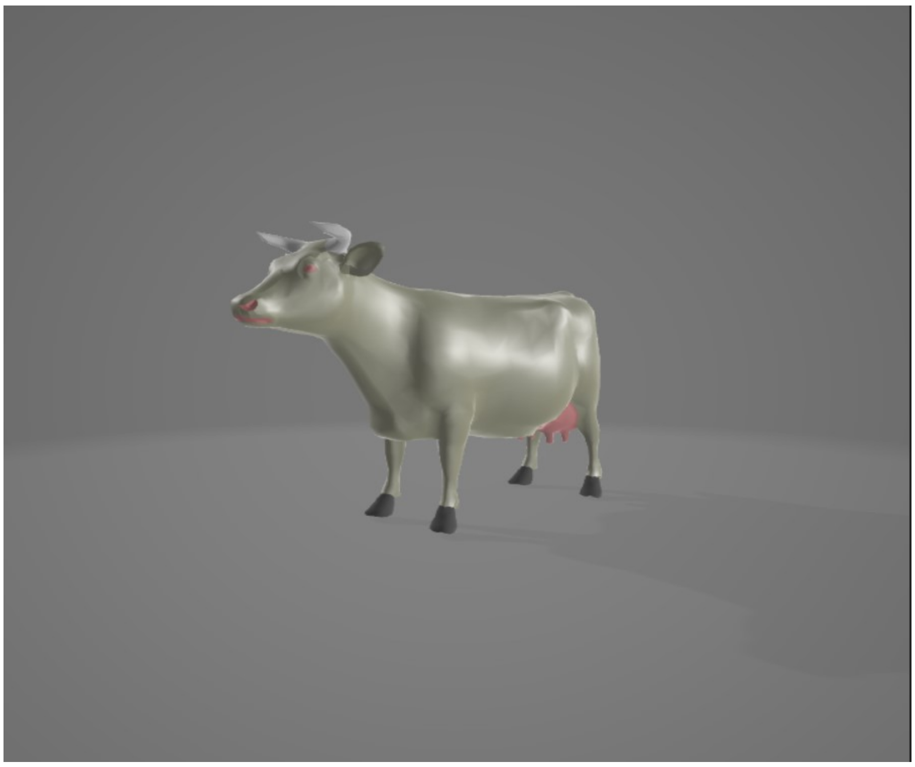

Figure 4.2: Cow 3d object.

- **Create a bounding box:**

  As we have seen in chapter 1, the bounding volume allows us to minimize the intersection ray-object computation. Thus, we insert our 3D object into the bounding volume structure to start the voxelization method.

To create a bounding box we need two points (min point , max point), these points we extract from the coordinates of the 3D object so the min point of the box will take the min(x,y,z) from the object and the max point of the box will take max(x,y,z) of the object.

The next figure 4.3 represent a 3D object in a bounding box.

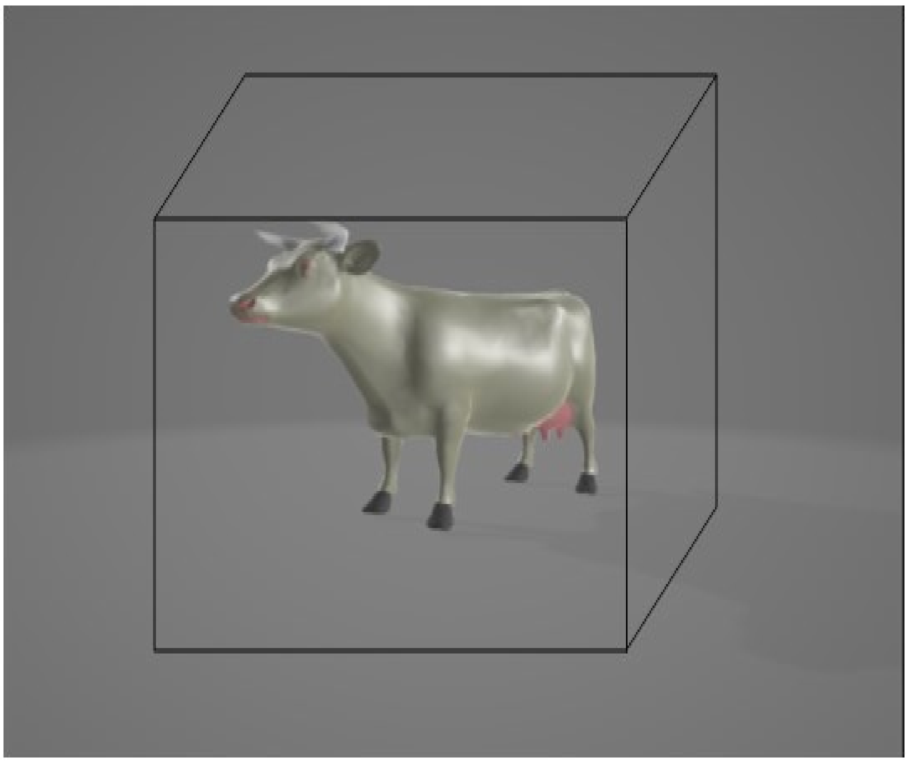

Figure 4.3: Cow 3d object in bounding box.

- **Create a 3D texture as a voxel grid**

  Using OpenGL library to create an empty 3D texture with only one channel (red channel) to store the position of the corresponding vertex from the object as a color in this 3D texture.

  In simplest terms, a 3D textures is a series of ($width \times height \times depth$). You might think of it as a series of two dimension textures where an extra parameter (depth, AKA the R texture coordinate) specifies which 2D texture will be used.

  So, The (width,height)coordinates will be the pixel of a texture and the R coordinate will be the number of slice.

  Alongside these(W,H,R) parameters we need to specify the format of the pixel data. which we use the red, and we need specify the data that the texture will take,which is our object data.

In the end our model will be a stack of 2d textures (3D texture).

- **Create Atlas texture**

  This is the main step in our application, we collect all the previous data and work in this method in order to create Atlas 3D texture.

  To create this texture type, we need to generate an empty 3D texture data (Create a 3D texture as a voxel grid),then we use a frame-buffer object and sweep the 3D model and put every vertex position from the object as a color(using only the red color channel) into the corresponding position in the 3D texture.

  The next figure 4.4 represents a 3D object in an atlas texture.

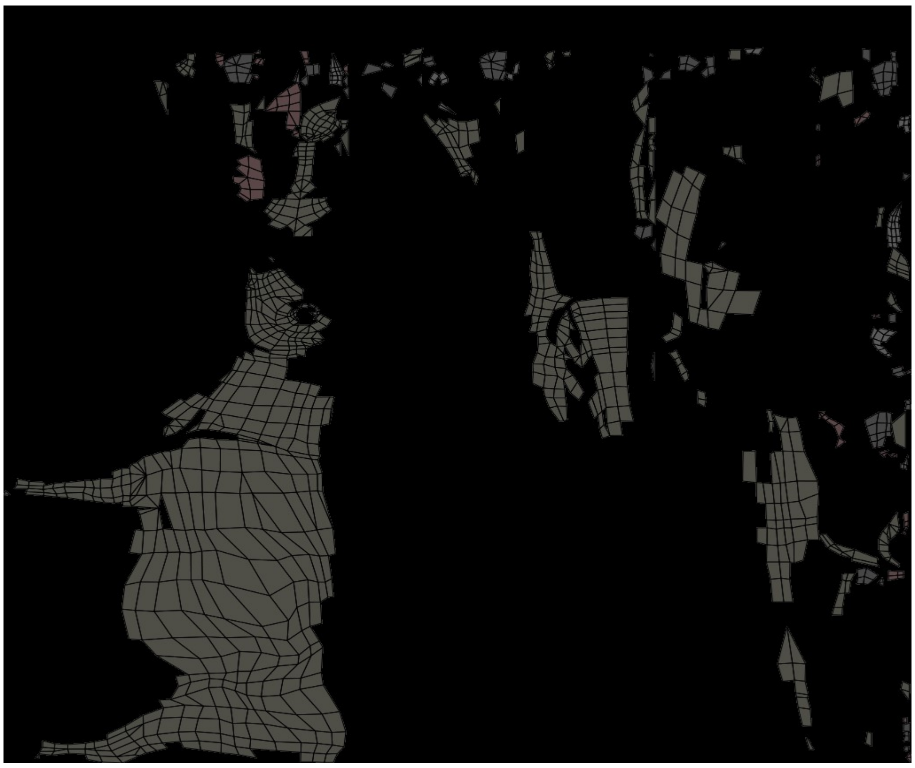

Figure 4.4: Cow 3d object in Atlas texture.

- **Visualize the voxel model using ray marching**

  In order to create the final voxel representation we need the ray marching algorithm to calculate the intersections between the stored data in the 3D atlas texture and the corresponding voxel grid and retrieve the (position stored as a color), and projected in a 2D plane representing the screen.

  The next (figure 4.5), showing a diagram of ray marching algorithm.

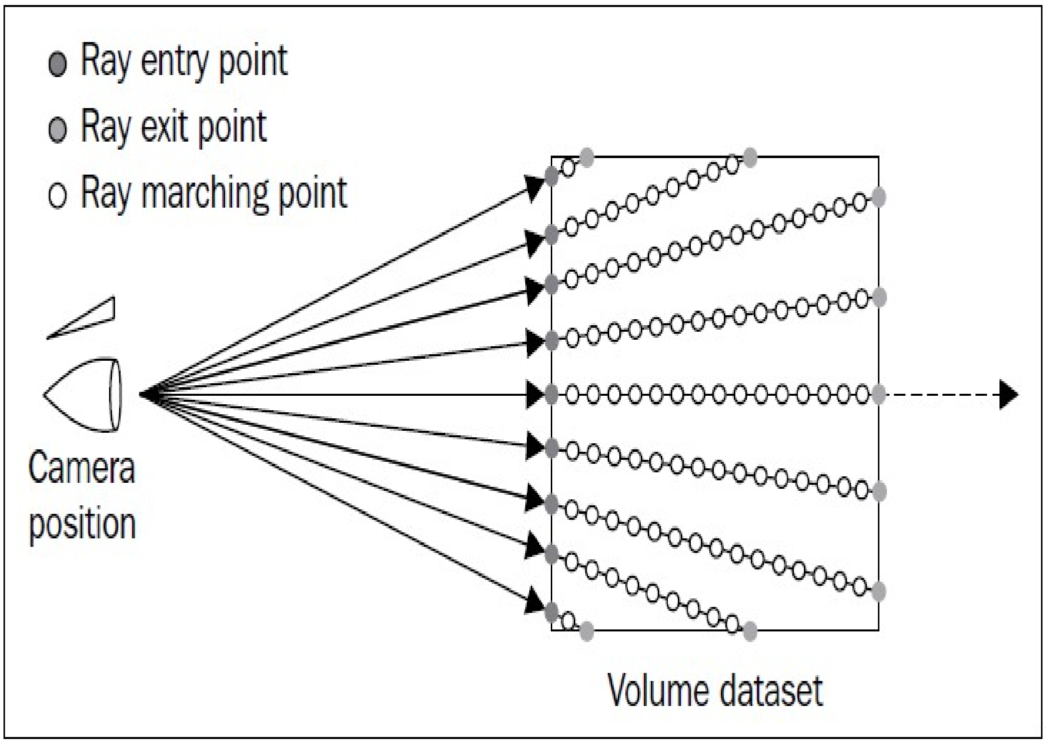

Figure 4.5: Ray marching diagram.

For implementing ray marching algorithm in a single pass GPU with one fragment shader, we followed the next steps.

1. The camera ray direction is calculated by subtracting the vertex positions from the camera position.
2. Compute the intersections using the intersect (ray, box) method.

   Where the ray is the camera ray direction and the box is the pixel entry in the 3D texture volume.
3. Then based on the ray step size which is the voxel size, the initial entry ray position is advanced in the ray direction using a loop, This process is continued forward advancing the current ray position until the ray exits the 3D texture volume calculating all the intersections and retrieving the color of all the voxels in his direction.

   In the next figure 4.6 we presents the ray marching conception.

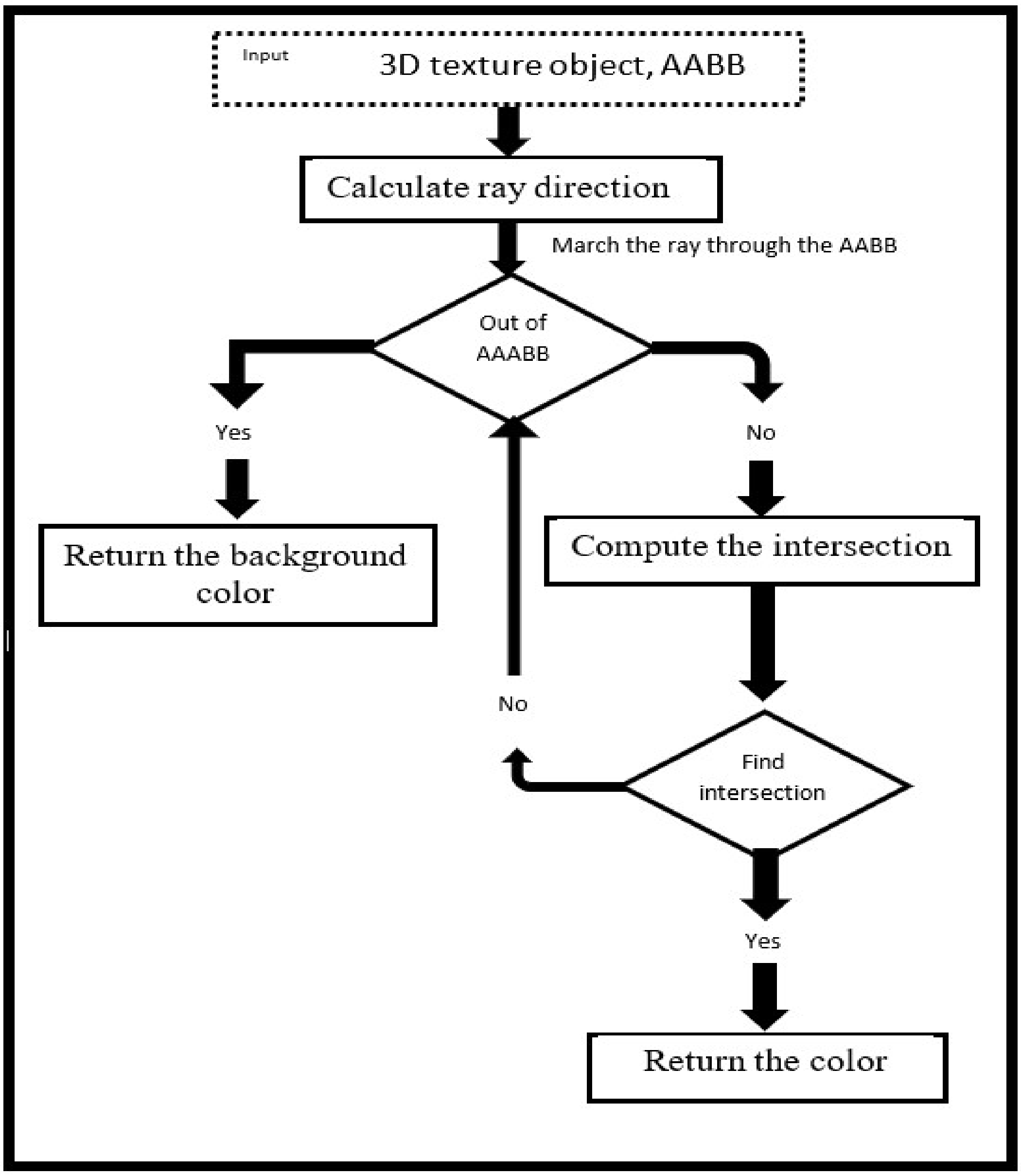

Figure 4.6: Ray marching conception.

- **Voxel model**

  Our output is a 3D voxel model projected on a 2D plane as a screen.

4.5 conclusion

In this chapter by the name conception of 3D voxelization method we start with the goal of our project and the main idea behind it then we define the global conception as a schema followed by a more detailed conception of the method.

In the fourth chapter, we will see the implementation, results and discussion of this work.

Chapter 5

Implementation,results and discussion

5.1 Introduction

In this chapter we will describe the implementation of the different stages of our application. Firstly, we will present the hardware configuration and the environment and library of the machine we used in our project , next we will detail the data structure as well as the algorithms used in our implementation, and finally we will present and discuss our voxelization results.

5.2 Hardware configuration

Our hardware configuration of an ACER laptop includes the following devices:

- **OS :** Windows 10 Enterprise 64 bit.
- **CPU :** Intel(R) Core(TM) i5 7200 U @ 2.50 GHz 4 CPU's.
- **GPU :** NVIDIA GeForce GTX 950 M (2 GB) display memory (4 GB) shared memory.
- **RAM :** 8 GB DDR 4 .
- **Storage :** 256 GB SSD 500 M/s.
- **Screen:** Full HD 1080p image resolution (1,920 x 1,080 pixels) approximately 2 million total pixels.

5.3 Environment and library

5.3.1 Visual Studio 2017

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft, Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms,and it can produce both native code and managed code.

Visual Studio includes a code editor supporting IntelliSense (the code completion component) as well as code re factoring.

The integrated debugger works both as a source-level debugger and a machine-level debugger.

Other built-in tools include a code profiler, designer for building GUI applications, web designer, class designer, and database schema designer.

Visual Studio supports 36 different programming languages like: C,C++,Visual Basic .NET, C sharp,JavaScript, XML, HTML,CSS,Java, and Python.

We are using Visual studio 2017 community version.[mic20]

5.3.2 GLEW

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library.

GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file. [gle20]

We are using GLEW 1.9.0 support for OpenGL 4.3.

5.3.3 SFML

Simple and Fast Multimedia Library (SFML) is a cross-platform software development library designed to provide a simple application programming interface (API) to various multimedia components in computers(system, window, graphics, audio and network), and we are using SFML 2.5.1. [SFM20]

5.3.4 Assimp

Open Asset Import Library (Assimp) is a cross-platform 3D model import library which aims to provide a common application programming interface (API) for different 3D asset file formats. Written in C++, it offers interfaces for both C and C++, and we are using Assimp 3.0.0. [ASS20]

5.3.5 GLM

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that anyone who knows GLSL, can use GLM as well in C++.

GLM isn't limited to GLSL features. An extension system, based on the GLSL extension conventions, provides extended capabilities: matrix transformations, data packing, random numbers, noise, etc...

GLM is a good library for software rendering (ray tracing / rasterisation), image processing, physic simulations and any development context that requires a simple and convenient mathematics library.

GLM is written in C++98 but can take advantage of C++11 when supported by the compiler. [GLM20]

We are using GLM 1.9.0.1.

5.4 Implementation details

5.4.1 3D model loading

After loading our obj model using Assimp library, we need to store the information of the obj file (vertices, normals, colors or UV) in the vectors. For that, we use a GLM vector data:

- **glm::vec3 Vertices:** to store the position of the vertices object from the obj file.
- **glm::**vec3 Normal: to store the object normals from the obj file.
- **glm::vec2 UV:** to store the object texture coordinates from the obj file.

5.4.2 Uniform grid creation

For the uniform grid, we define a structure of the min and the max of AABB.

$$struct \quad AABB\{glm :: vec3 \quad min; \qquad glm :: vec3 \quad max;\}$$

- $AABB.min$: for the min point of the AABB.
- $AABB.max$: for the max point of the AABB.

To calculate the AABB, we simply loop through all the models vertices, and store the x,y,z values if they are less or greater then the min and the max x, y, z values.

```
$BEGIN$
$for(i = 0...vertices\_number)do$
$if(vertices[i] < min)then$
$min = vertices[i];$
$if(vertices[i] > max)then$
$max = vertices[i];$
$END.$
```

5.4.3 Texture atlas

As we mention before in the section 4.4.2, we need to define an empty 3D texture.

For this part we need 2 steps.

- Reserving a memory space with the size of the voxels number for the texture data.

  float Data [voxels number ^{3}]

- Creating the empty texture using $glTexImage3D$ function with the (W, H, R) parameters taking the size of the voxel grid, and the empty data from before.

  glTexImage3D(voxels_number W, voxels_number H , voxels_number R, float data);

Now we need to create the atlas texture(3D object, AABB, and 3D texture), and we do that by following the next steps:

- Setting up the model view projection matrix.
  - Model matrix will take the identity matrix $M[1]$.
  - View matrix will take the look at function.
  - Projection matrix will take the perspective projection.
- Setting up a frame buffer for rendering the model in the 3D texture.
- Rendering the 3D atlas texture using special fragment shader.

  This fragment shader works as follow:

  - **Input**

    3D empty texture.

    voxel grid dimension.

    min and max values of the AABB

  - **calculation**

    Using the model position,AABB values and the voxel grid dimension to calculate the new position inside the 3D texture.

    $$new\_position = \frac{(old\_position - AABB\_min)}{(AABB\_max - AABB\_min)} \times voxel\_grid\_dimension \tag{5.1}$$

    Then we use the GLSL *image_store* function to store a color in the specific position on the 3D texture.

    imageStore(empty 3D texture, new position , a color);

  - **Output**

    A 3D texture filled with object data in the memory.

That's how the 3D texture with the object is created, and it is now in the memory ready for the ray marching algorithm.

5.4.4 Visualization with ray marching

After all the steps from before we can now visualize the 3D voxel model, to do that we need to implement the ray marching algorithm as we indicate in the section 4.4.2.

The next steps representing the implementation of ray marching in this application.

1. **Setting up the scene**

   - The 3D texture is ready in the memory.
   - Setting up the new model view projection matrix for ray marching.
   - Creating a 2D plane representing the screen.

2. **Rendering the model**

   The ray marching fragment shader works as follow.

   - **Input**
     - 3D texture with object model.
     - camera or eye position.
   - **Calculation**
     - Computing the view ray.

       view ray = normalize(screenCoord - cameraPosition).
     - Test for intersection.

       Using (ray, box) intersection function, which is works as follow:

       * Compute the intersection of the ray with all the six box planes.

       * Re-order the intersections to find the smallest and largest intersection on each axis.

       * Return the largest min intersection, and the smallest max intersection in a voxel(box).

       * The next figure (figure 5.1) represents a 2D result of the ray,box intersection function.

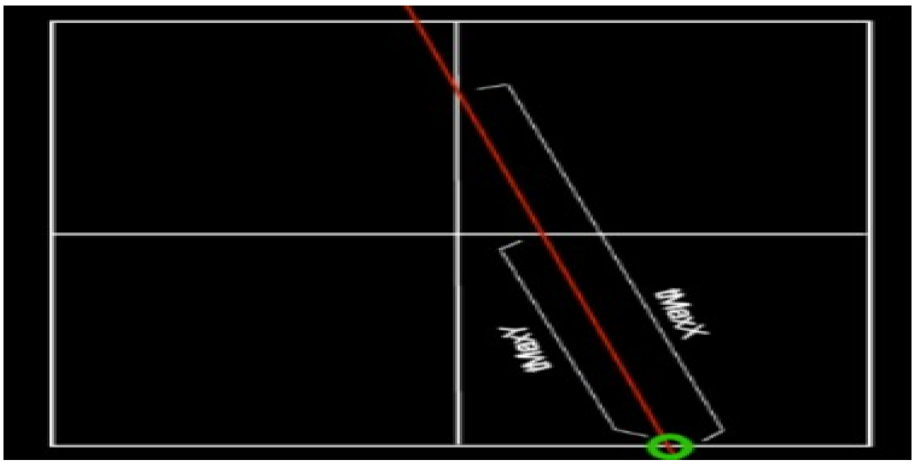

Figure 5.1: 2D representation of tmin and tmax.

The following (figure 5.2) representing a 3D result of the ray,box intersection function.

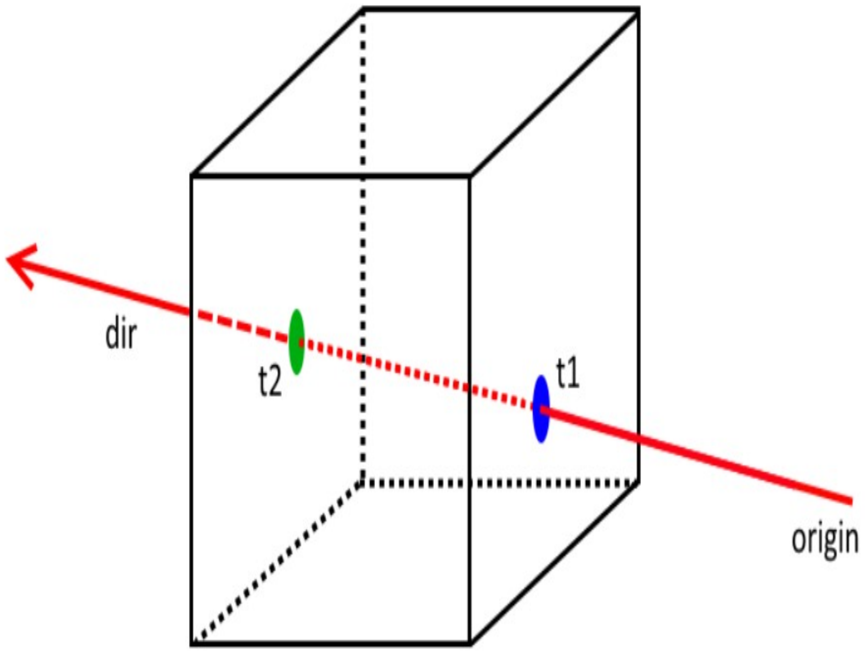

Figure 5.2: 3D representation of tmin and tmax in the AABB.

- If there is no intersection then return the background color.
- If there is an intersection return the color from the 3d texture .
- Loop until the view ray is out of the AABB with traversing the ray using fixed step.

- **Output**

  The output of the ray marching algorithm will be the the projection of the voxelize model on the 2D plane.

5.5 Results and discussion

We present in this section our results of the voxelization method using Atlas texture, and we evaluate and discuss our results.

5.5.1 Results

The next figure 5.3 represents the empty AABB result.

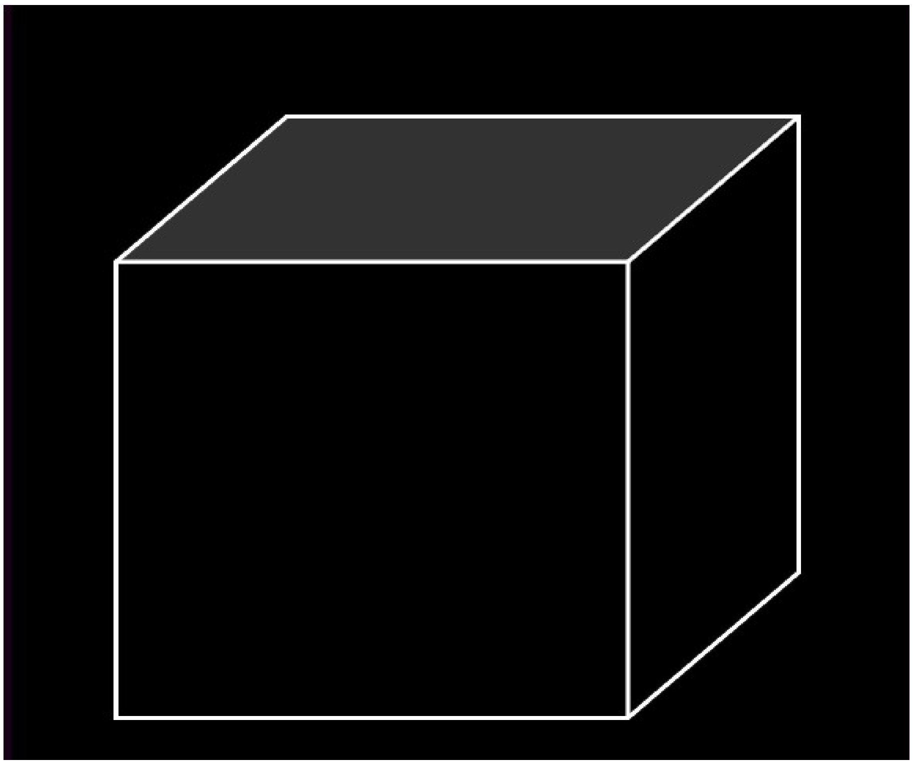

Figure 5.3: Result of the empty AABB.

The next figure 5.4 represents the cow input object.

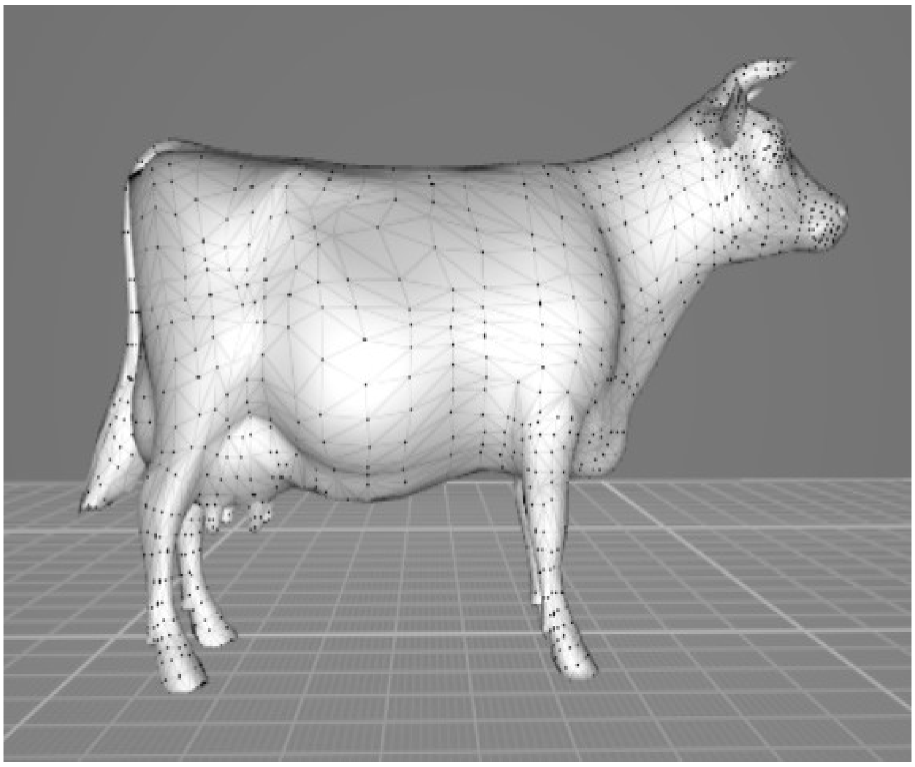

Figure 5.4: Cow Input object.

The next figure 5.5 represents a result for the cow input object with resolution $= 32^3$ voxel.

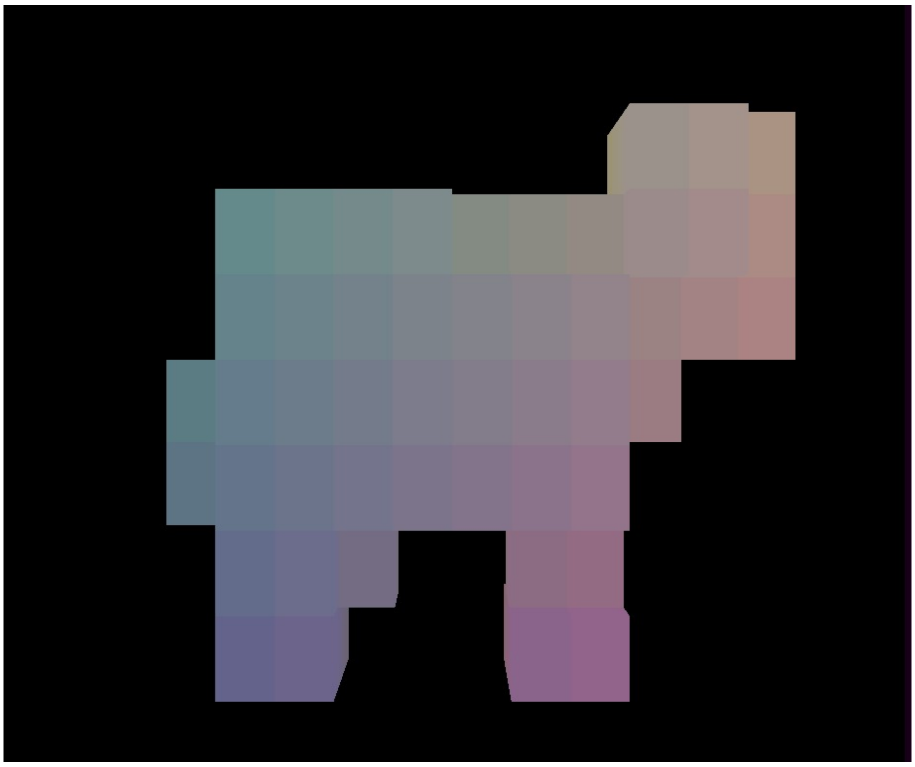

Figure 5.5: Result 1 voxelize cow.

The next figure 5.6 represents a result for the cow input object with resolution = 64^3 voxel.

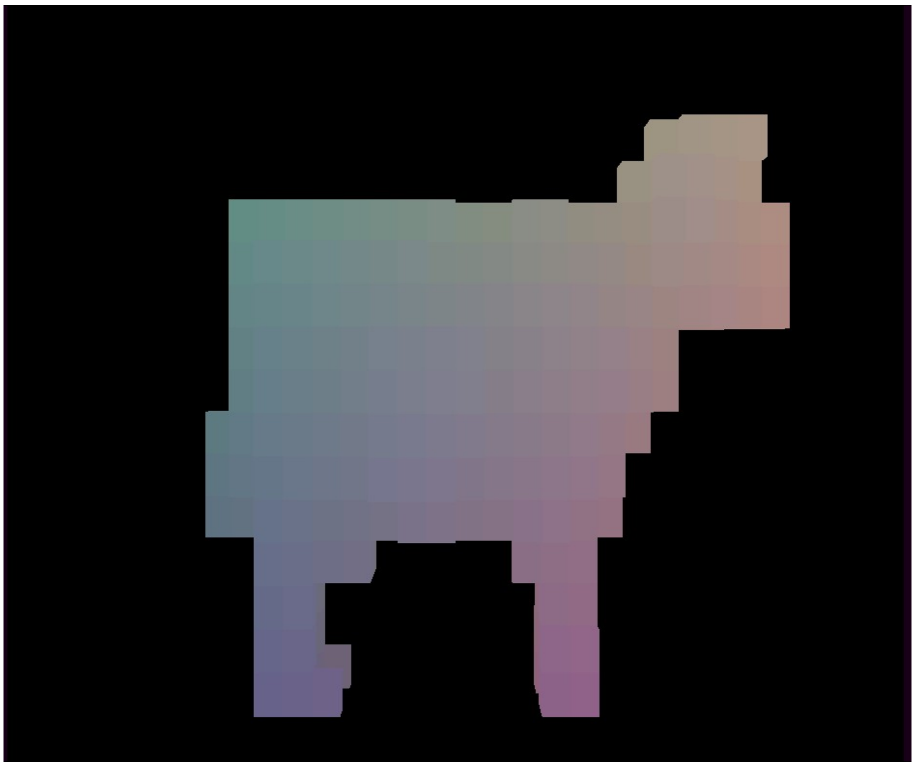

Figure 5.6: Result 2 voxelize cow.

The next figure 5.7 represents a result for the cow input object with resolution = 128^3 voxel.

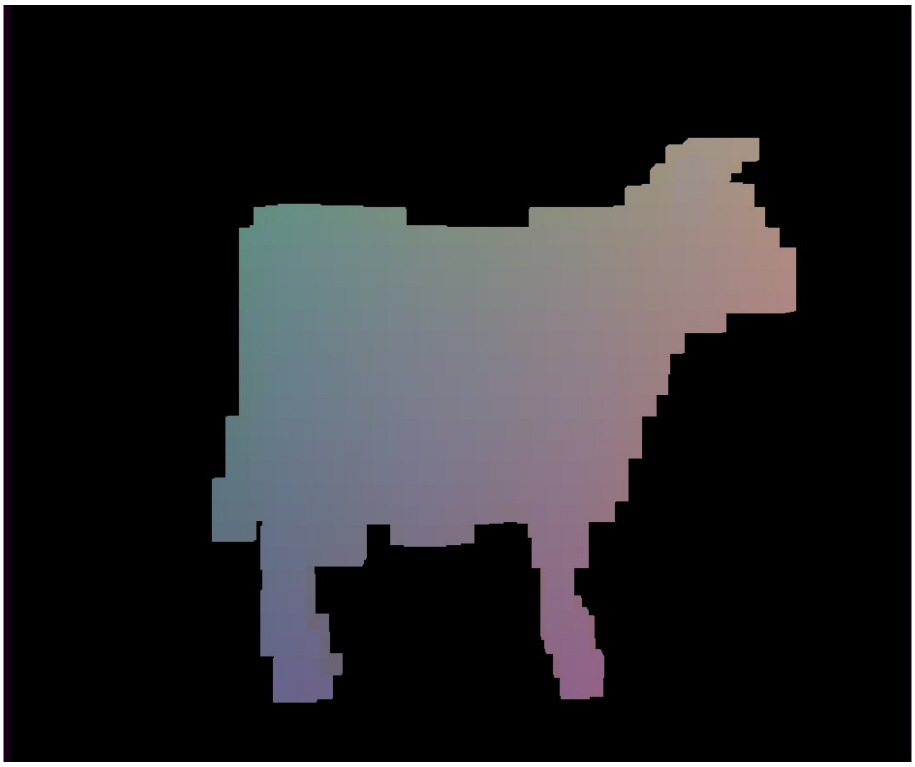

Figure 5.7: Result 3 voxelize cow.

The next figure 5.8 represents a result for the cow input object with resolution = 256^3 voxel.

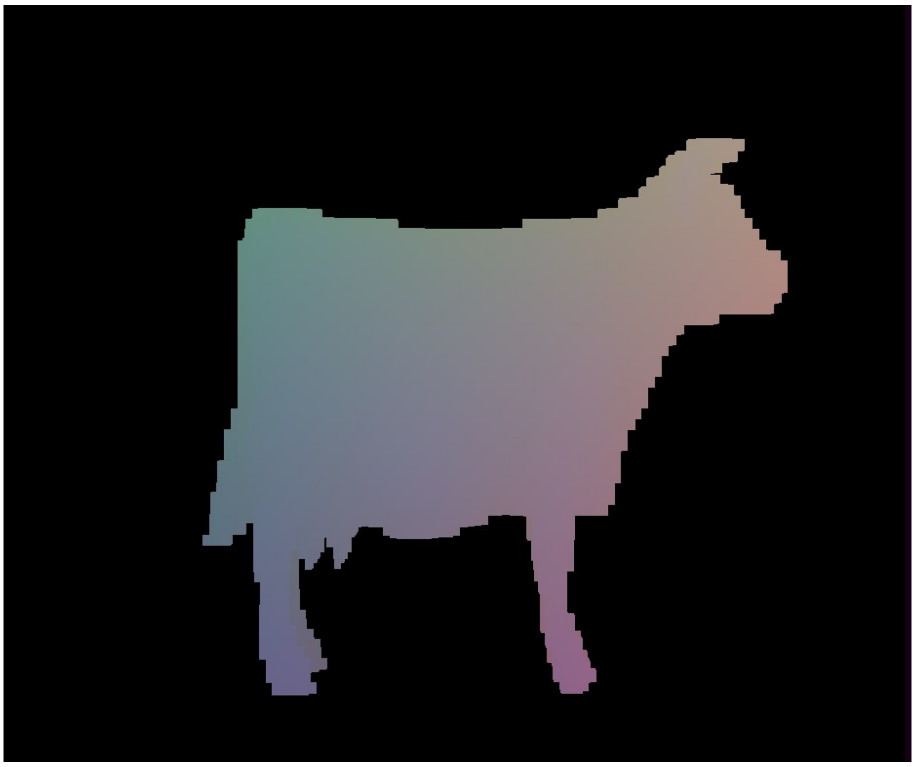

Figure 5.8: Result 4 voxelize cow.

The next figure 5.9 represents the dinosaur input object.

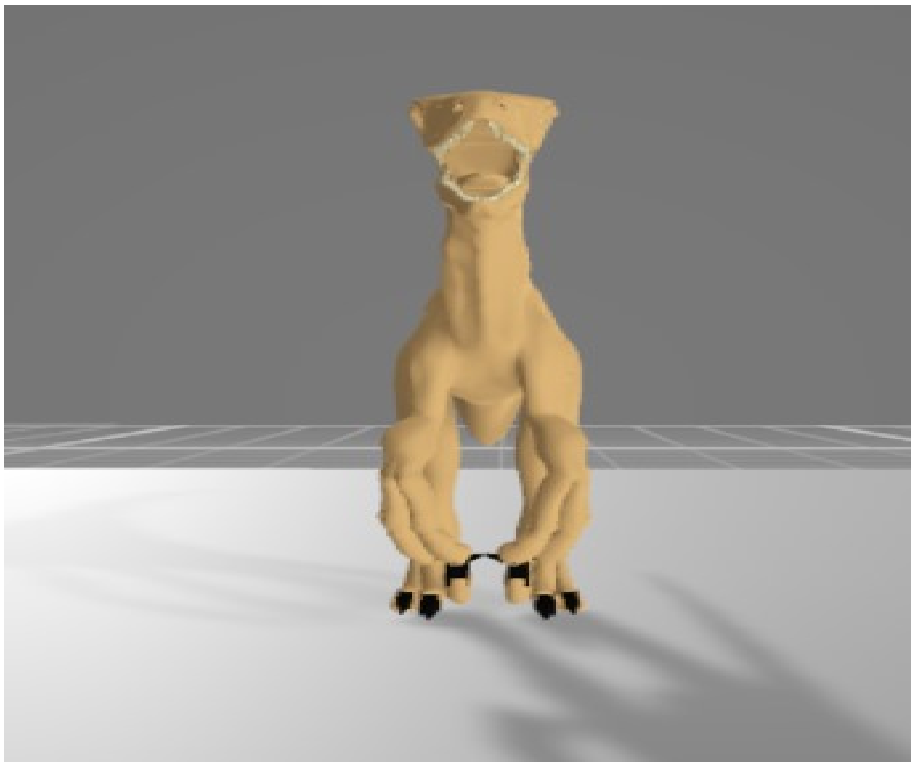

Figure 5.9: Dinosaur Input object.

The next figure 5.10 represents a result for the dinosaur input object with resolution = 128^3 voxel.

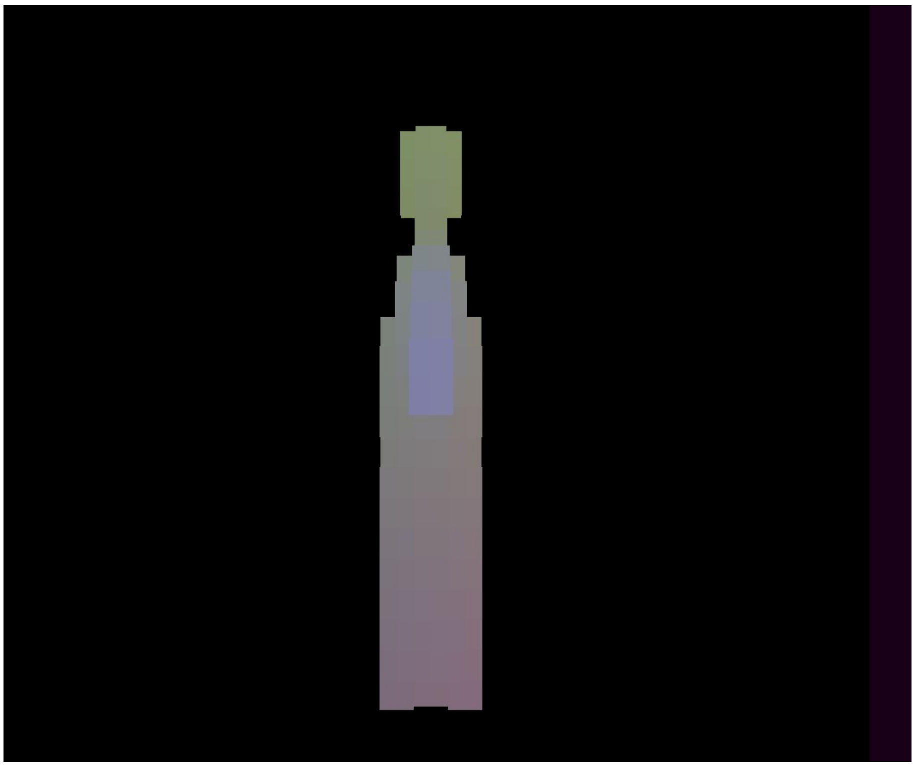

Figure 5.10: Result 1 voxelize Dinosaur.

The next figure 5.11 represents a result for the dinosaur input object with resolution = 256^3 voxel.

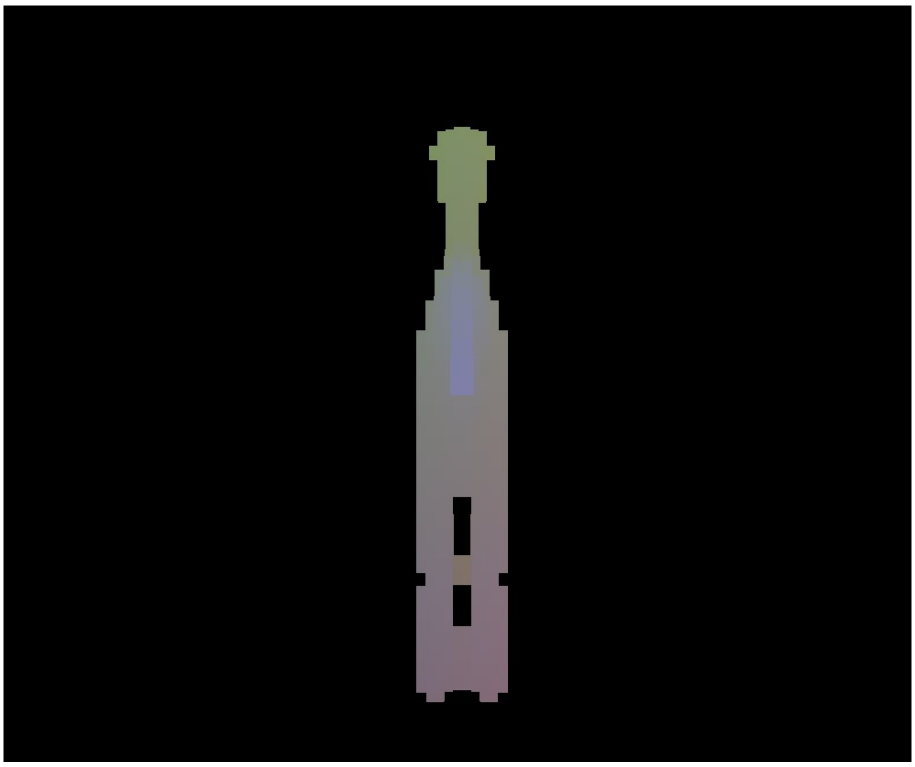

Figure 5.11: Result 2 voxelize Dinosaur.

The next figure 5.12 represents a result for the dinosaur input object with resolution = 512^3 voxel.

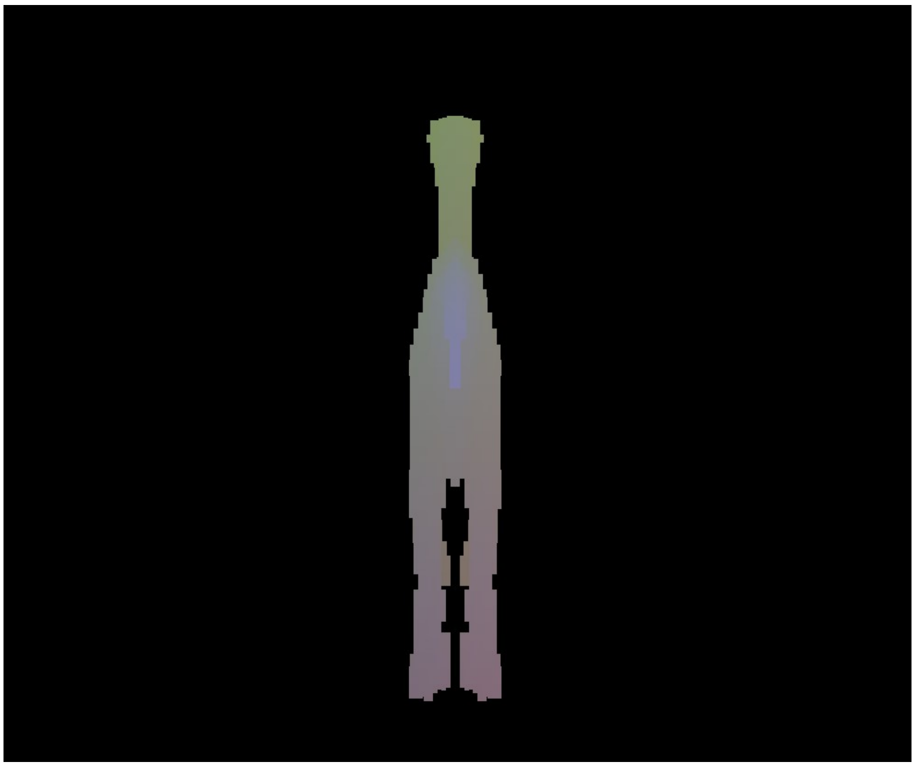

Figure 5.12: Result 3 voxelize Dinosaur.

The next figure 5.13 represents the angel input object.

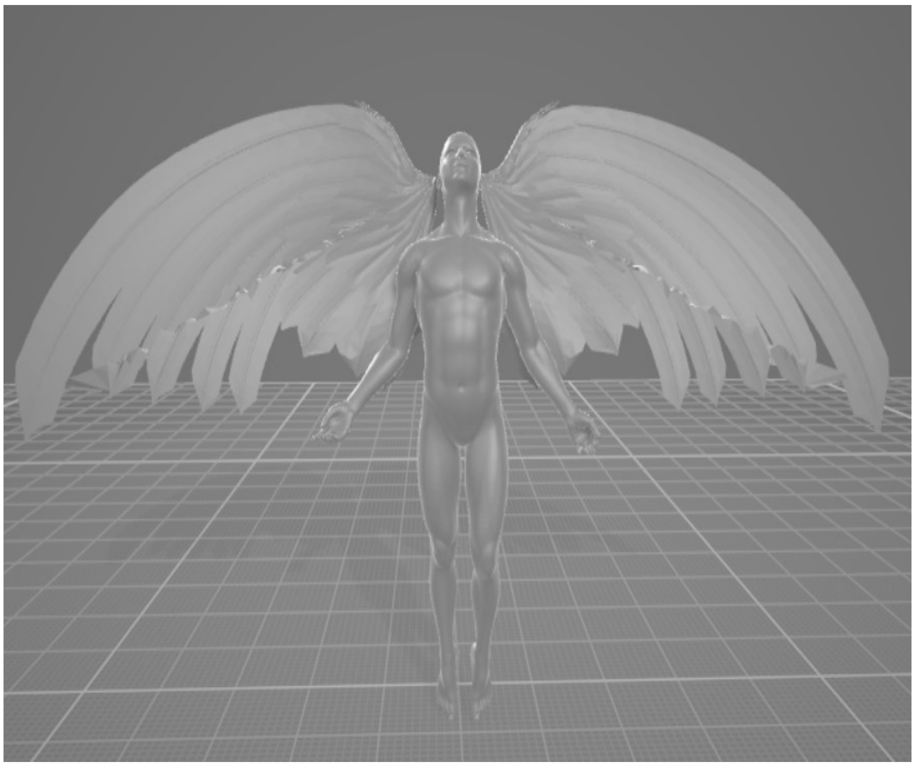

Figure 5.13: Angel Input object.

The next figure 5.14 represents a result for the angel input object with resolution = 64^3

voxel.

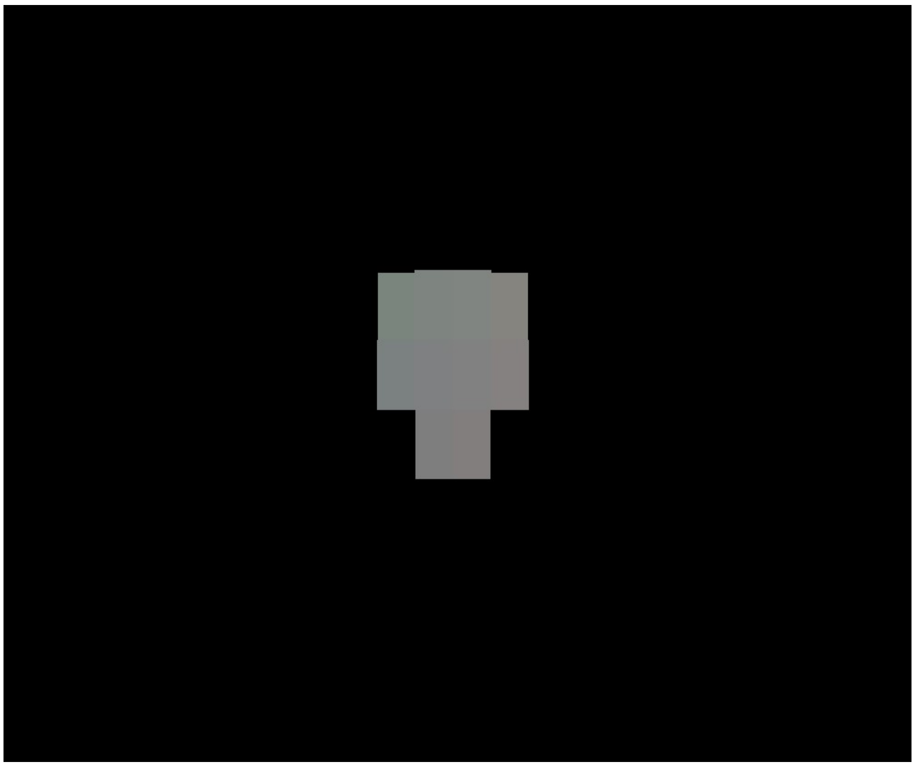

Figure 5.14: Result 1 voxelize angel.

The next figure 5.15 represents a result for the angel input object with resolution = 128^3 voxel.

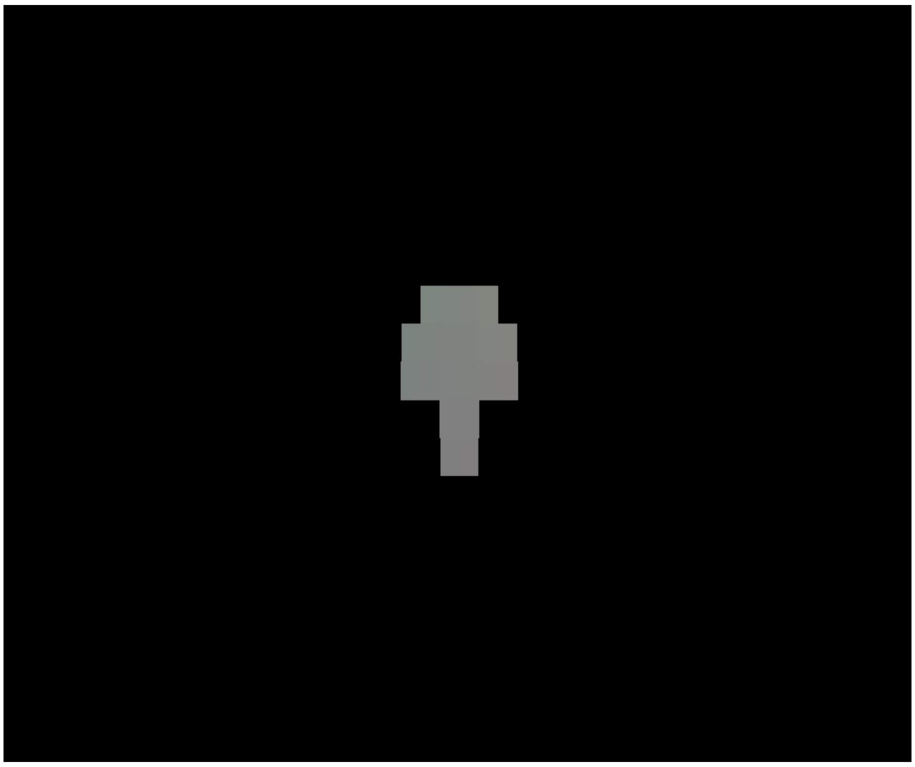

Figure 5.15: Result 2 voxelize angel.

The next figure 5.16 represents a result for the angel input object with resolution $= 256^3$ voxel.

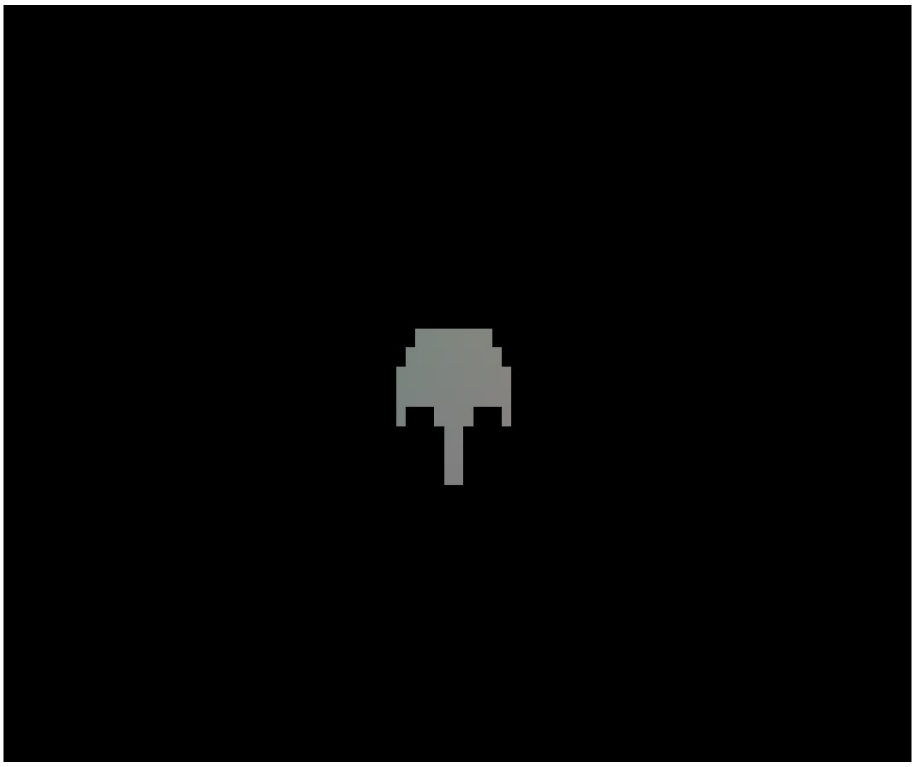

Figure 5.16: Result 3 voxelize angel.

The next figure 5.17 represents a result for the angel input object with resolution = 512^3 voxel.

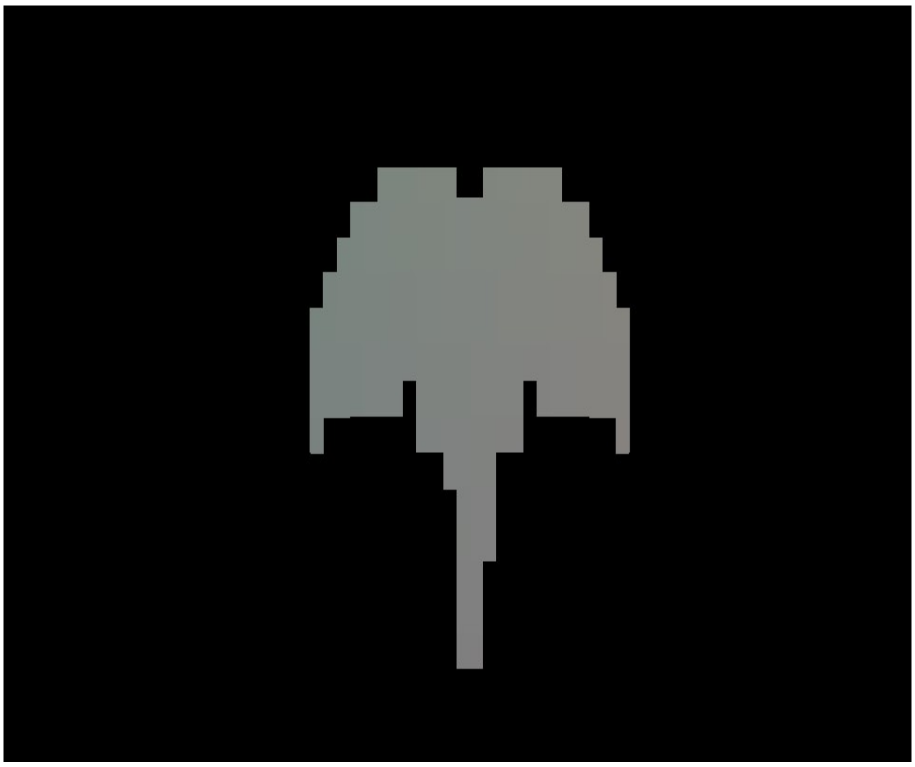

Figure 5.17: Result 4 voxelize angel.

5.5.2 Discussion

The following table 5.1 represents the execution time (millisecond) for different polygon number to five different object.

	Polygon number	execution time for the better result
angel	251012	4889
Dinosaur	47166	1464
cow	5804	816
sofa	1621	806
mill	144	751

Table 5.1: Execution time for atlas texture function for different polygon number.

The diagram below (figure 5.18) represents an execution time graph for different polygon number.

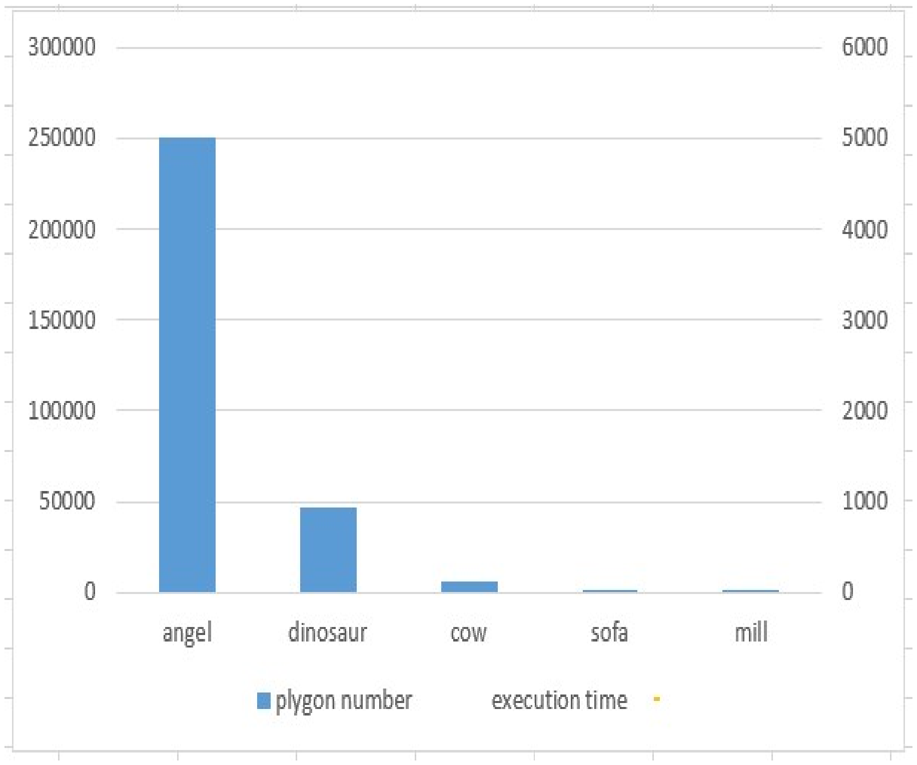

Figure 5.18: Execution time graph for different polygon number.

From the table 5.1 and the diagram (figure 5.18) we learned that the execution time will increase if the polygons number increased but with a slow rate. As we can see the angel model with almost 250000 polygon executed in around 5000 millisecond, and for the dinosaur model with almost 50000 polygon executed in around 1000 millisecond, and for the rest of the models (cow, sofa ,mill) the execution time was so low alongside the number of polygons.

The following table 5.2 represents the execution time (millisecond) for different resolution grid of five different object(angel,dinosaur, cow, sofa, mill), and as we can see the resolution is to the power of 3 and changing from 32 to 512 with each step of the number 2^n.

	execution time (milliseconds) for different resolution grid (R^3)				
	R = 32	R = 64	R = 128	R = 256	R = 512
angel	4602	4615	4669	4690	4889
dinosaur	898	899	902	978	1464
cow	158	186	206	289	816
sofa	135	146	165	236	806
mill	96	101	126	211	751

Table 5.2: Execution time for atlas texture function for different resolution grid.

The next plot (figure 5.19) represents an execution time plot for the different resolution grid.

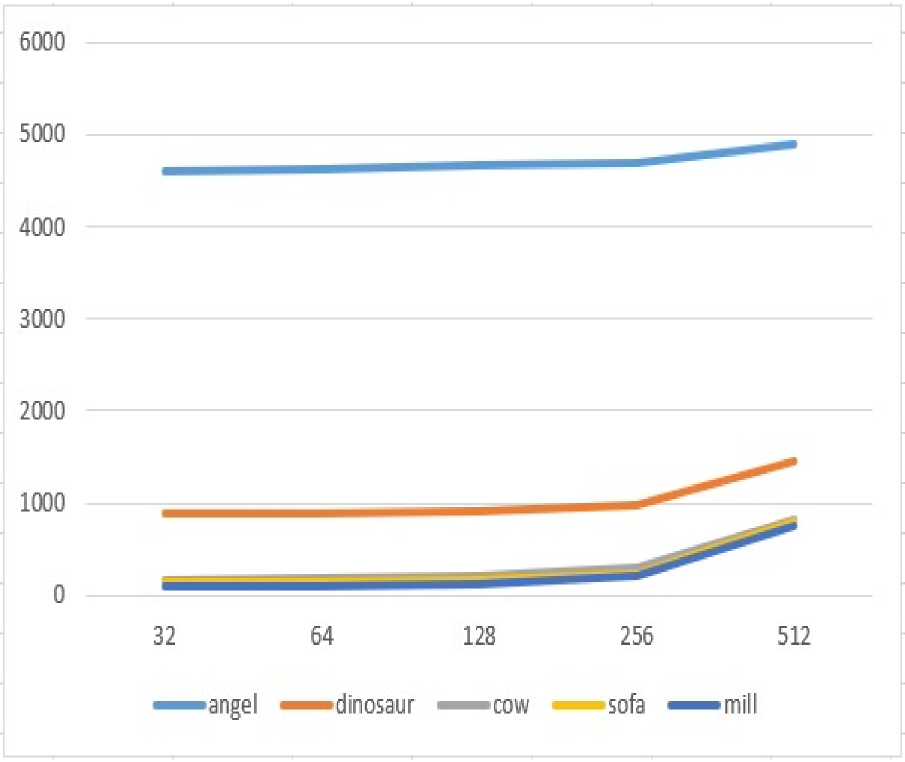

Figure 5.19: Execution time graph for different resolution grid.

As a result of what we have seen in table 5.2 and graph (figure 5.19), the execution time will increase significantly with the increment of resolution ,except between the 256 and 512 resolutions, the execution time increased dramatically for all the models.and as we can observe the angel model executed in around 4500 millisecond for the resolution of 32^3 voxel and the execution time increased slightly till the resolution of 256^3 but from this resolution to 512^3 the execution time increase dramatically and this change apply for the rest of the models.

5.6 Conclusion

We presented in this chapter the hardware configuration of our machine and the environment we use alongside the libraries, then we explained the implementation details of this work followed by some results and a discussion for those results finally we ended this chapter with a conclusion.

Chapter 6

Conclusion

Fast or even real-time voxelization is essential for interactive graphics applications. We have presented a voxelization method via shaders to solve this problem by representing the resultant volume as 3D textures which can efficiently implemented. This method scales very well with increasing voxel and object resolutions.

This thesis presents a voxelization method using texture atlas via shaders in GPU.
We started by presenting the volume modeling domain. Then we described the different voxelization techniques, and we implemented the voxelisation method based texture atlas.
The approach is simple, robust and easy-to-implement.

As a future work is concerned, an important issue is the improvement of the voxelization quality by adding normals to the voxel model and apply simple illumination method, loading more complex object, make the algorithm faster by adding some optimizations techniques.

Bibliography

[ASS20] ASSIMP. The open-asset-importer-lib, 2020.

[Bou14] Kadi Bouatouch. Ray tracing. *IRISA*, 2014.

[Dav97] Mason Woo Jackie Neider Tom Elder Davis. Opengl programming guide. *Addison Wesley Longman Publishing Co*, 1997.

[Dec08] Elmar Eisemann Xavier Decoret. Single pass gpu solid voxelization for real-time applications. *ARTIS INRIA Grenoble University Phoenix Interactive*, 2008.

[Fle08] Cedric Fleury. Le kd tree une methode de subdivision spatiale. *INSA de Rennes*, 17 Janvier 2008.

[gle20] glew. The opengl extension wrangler library, 2020.

[GLM20] GLM. Opengl mathematics, 2020.

[HWA20] TAE YOUNG JANG SEONG DAE KIM SUNG SOO HWANG. Surface scanning-based texture atlas for voxelized 3d object. *IEEE ACCESS*, 2020.

[Kau90] D Cohen Kaufman. Scan conversion algorithms for linear and quadratic objects. *Volume Visualization*, 1990.

[Laz12] Steve Dodier Lazaro. *Dynamic spatial subdivision for n-body collision detection on multi-CPU and multi-GPU architectures.* yENSI of Bourges,INSA Rennes,zUniversity of Rennes IkIRISA, VR4I team, 2012.

[mEHNH08] Tomas Akenine moller Eric Haines Naty Hoffman. Real-time rendering, third edition, 3rd edition. *CRC Press*, 2008.

[mic20] microsoft. visual studio, 2020.

[Mon09] Carlos Tripiana Montes. Gpu voxelization. *LSI Department Polytechnic University of Catalonia*, 2009.

[Ngu06] An Nguyen. *Implicit bounding volumes and bounding volume hierarchies.* Stanford University408 Panama Mall, Suite 217 Stanford CA United States, 2006.

[Nie00] Gregory M Nielson. Volume modelling. *Volume Graphics*, 2000.

[Pat05] Sandeep S Patil. Voxel based solid models representation display and geometric analysis. *Computer Science*, 2005.

[Pen04] Zhao Dong ; Wei Chen ; Hujun Bao ; Hongxin Zhang ; Qunsheng Peng. Real-time voxelization for complex polygonal models. *IEEE*, 2004.

[Rap16] Cosmin Nita Iulian Stroia Lucian Itu Constantin Suciu Viorel Mihalef Manasi Datar Saikiran Rapaka. Gpu accelerated robust method for voxelization of solid objects. *IEEE*, 2016.

[Sei10] Michael Schwarz Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Transactions on Graphics Article No.: 179*, 2010.

[SFM20] SFML. Simple and fast multimedia libraryl, 2020.

[Shi00] HongshengChen ShiaofenFang. Hardware accelerated voxelization. *Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 723 W. Michigan Street, Indianapolis, IN 45202, USA*, 2000.

[Sol19] Grigory Glukhov Aleksandra Soltan. Gpu volume voxelization exploration of the performance characteristics of different gpu-based implementations. *KTH Royal Institute of Technology*, 2019.

[The04] G Passalis Kakadiaris Theoharis. Efficient hardware voxelization. *IEEE*, 2004.

[TP06] Kadi Bouatouch Thierry Priol. Synthèse d'image par lancer de rayon sur un hypercube. *HAL Id: inria-00075800*, 2006.

[Yag93] Arie Kaufman Daniel Cohen Roni Yagel. Volume graphics. *Volume Graphics IEEE Computer Vol 26*, 1993.