Implementation of an Effective Collision Detection Algorithm for View Factor Matrix Calculation

Önay CAN
Ceyhun BAHTIYAR
Muhammet Hüseyin ATES

Int. Master’s Program on Computational Mechanics

Handed in on: January 17, 2007
Abstract

In the project the aim is to calculate the view factor matrix in a more efficient and faster way by implementing an effective collision detection algorithm. In order to obtain view factor matrix, ray tracing method is decided to use as a collision detection method. The working area is assumed consists of triangles. In our method the rays are sent from each triangle to the space. Than every ray is controlled for very early collision(ray-facet intersection) with the other triangles. This controlling step is done with the movement on the ray(ray travelling). Finally, view factor matrix is calculated. During the process of sending ray in order to decrease the expensivity of the algorithm space subdivision methods are considered. Uniform and non-uniform(octree) space division are the two methods that are implemented. And it is observed that octree algorithm is the fastest way.

Figure 1: Non-uniform space sub division(octree voxels)

Keywords: uniform and non-uniform space subdivision, octree, bounding boxes, ray tracing, ray sending, ray travelling, ray-facet intersection, view factor matrix
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>2 Non-Uniform Space Subdivision in Collision detection</td>
<td>3</td>
</tr>
<tr>
<td>2.1 General Information for the objects</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Class point</td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 Class triangle</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Class octan</td>
<td>5</td>
</tr>
<tr>
<td>2.1.4 Class vector</td>
<td>7</td>
</tr>
<tr>
<td>2.1.5 Class surface_grid</td>
<td>8</td>
</tr>
<tr>
<td>2.1.6 Main</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Results</td>
<td>17</td>
</tr>
<tr>
<td>2.3 Possible sources of errors and outlook</td>
<td>19</td>
</tr>
<tr>
<td>3 Non-uniform structured space subdivision</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Idea of bounding boxes</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Octree generation</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Ray sending</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Ray–facet intersection</td>
<td>23</td>
</tr>
<tr>
<td>3.5 Finding exit face and exit point of the voxel</td>
<td>25</td>
</tr>
<tr>
<td>3.6 Finding neighbour voxel</td>
<td>26</td>
</tr>
<tr>
<td>3.7 View factor matrix</td>
<td>26</td>
</tr>
<tr>
<td>4 Uniform and combined model approaches</td>
<td>28</td>
</tr>
<tr>
<td>4.1 General information of the classes</td>
<td>28</td>
</tr>
<tr>
<td>4.2 Reading the input files</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Space subdivision</td>
<td>32</td>
</tr>
<tr>
<td>4.3.1 Creating the voxels</td>
<td>32</td>
</tr>
<tr>
<td>4.3.2 Putting the facets inside the voxels</td>
<td>32</td>
</tr>
<tr>
<td>4.3.3 Allocating the memory for the facet list and filling it</td>
<td>36</td>
</tr>
<tr>
<td>4.3.4 Defining the grid and the corner coordinates of the Voxels</td>
<td>37</td>
</tr>
<tr>
<td>4.3.5 Finding the neighbors of the voxels</td>
<td>37</td>
</tr>
<tr>
<td>4.3.6 Visualization of Grid</td>
<td>39</td>
</tr>
<tr>
<td>4.3.7 Extending the empty voxels</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Ray Sending, Ray traveling and Ray–Facet Intersection</td>
<td>42</td>
</tr>
<tr>
<td>4.4.1 Ray Sending</td>
<td>42</td>
</tr>
</tbody>
</table>
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2 Ray–Facet Intersection</td>
<td>43</td>
</tr>
<tr>
<td>4.4.3 Facet Inside – Outside Checker</td>
<td>45</td>
</tr>
<tr>
<td>4.4.4 Ray Traveling</td>
<td>46</td>
</tr>
<tr>
<td>4.5 Error Analysis</td>
<td>48</td>
</tr>
<tr>
<td>4.6 Results of Uniform and Combined models</td>
<td>49</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>51</td>
</tr>
</tbody>
</table>
# List of Figures

1. Non-uniform space sub division (octree voxels) .......................... 1

2.1 Structure ............................................................................. 3
2.2 Input File ............................................................................ 10
2.3 Bounding Box ..................................................................... 10
2.4 Adressing a Facet with Bounding Box Idea ......................... 10
2.5 Addressing in Octree .............................................................. 12
2.6 Change in Octree Coding ....................................................... 12
2.7 Octree Voxels ..................................................................... 12
2.8 Surface Partitioning .............................................................. 13
2.9 Surface Grid ....................................................................... 13
2.10 First Ray ........................................................................... 14
2.11 All Sent Rays (1000 rays in this case) ................................. 14
2.12 Local to Global Transformation .......................................... 14
2.13 Intersection Point Inside? ................................................... 15
2.14 Voxel Faces and One Ray Through ..................................... 15
2.15 Backface and Front Face Exits ............................................ 16
2.16 Exit from a face ................................................................... 16
2.17 Voxel Exit from a Line and a Corner ................................... 16
2.18 Minus and Plus Exits of the Same Ray Sent from the Same Facet 17
2.19 Depth of Division vs. Time Graph ........................................ 17
2.20 Number of Rays vs. Time Graph .......................................... 18
2.21 Effect of Change in Number of Surface Grid for 1000 Rays .. 18
2.22 Same effect as in previous figure with 5000 Rays .................. 19
2.23 Effect of Change in Number of Facets ................................. 19

3.1 Need of bounding boxes ....................................................... 22
3.2 Idea of creating octree model and storage by pointer method ...... 22
3.3 Real model ........................................................................... 23
3.4 Octree model ...................................................................... 23
3.5 How the ray is sending from a facet ....................................... 23
3.6 Ray facet intersection ............................................................ 24
3.7 Triangle half of ABCD ............................................................ 25
3.8 Intersection point ................................................................. 25
3.9 Corners names ..................................................................... 25
3.10 Faces names ....................................................................... 25
3.11 Finding exit face and exit point ............................................ 25
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12</td>
<td>Finding neighbour voxel</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Population of two voxels</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Two voxels at depth 5</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Voxel’s population using createlisttwo()</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Neighbour finding algorithm</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>Visualization of the grid</td>
<td>40</td>
</tr>
<tr>
<td>4.6</td>
<td>Voxel population reachable in visual environment</td>
<td>40</td>
</tr>
<tr>
<td>4.7</td>
<td>Visual output of the combined model</td>
<td>42</td>
</tr>
<tr>
<td>4.8</td>
<td>Isocurve</td>
<td>43</td>
</tr>
<tr>
<td>4.9</td>
<td>Construction of the plane equation</td>
<td>44</td>
</tr>
<tr>
<td>4.10</td>
<td>Triangle and two candidate points on the plane equation</td>
<td>45</td>
</tr>
<tr>
<td>4.11</td>
<td>Limit of the length</td>
<td>45</td>
</tr>
<tr>
<td>4.12</td>
<td>Combined Model</td>
<td>47</td>
</tr>
<tr>
<td>4.13</td>
<td>Current Voxel</td>
<td>48</td>
</tr>
<tr>
<td>4.14</td>
<td>Calculation of the exact result</td>
<td>49</td>
</tr>
<tr>
<td>4.15</td>
<td>Calculation times and Error vs. Depth</td>
<td>50</td>
</tr>
<tr>
<td>4.16</td>
<td>Depth 7 center and corner points</td>
<td>50</td>
</tr>
<tr>
<td>4.17</td>
<td>Depth 5 center and corner points</td>
<td>50</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Project topic 7 is implementing an effective algorithm for collision detection in order to calculate the view factor matrix in a more efficient and faster way. This project is offered by P+Z Engineering.

P+Z Engineering is a company within the PCL group. Engineering services are given to customers from automotive, aerospace and transportation industries. For this project the problem offered is view factor matrix calculation that is needed for radiation heat transfer. This problem therefore can be encountered, for instance, in an engine, in a car cabin or, as it is in this project, in exhaust of a car.

For view factor matrix calculation, the analytical solution as in equation below:

$$ F_{1-2} = \frac{1}{A_1} \int_{A_1} \int_{A_2} \frac{\cos \theta_1 \cdot \cos \theta_2}{\pi \cdot S_{f-2}} \, dA_2 \, dA_1 $$ \tag{1.1}$$

However, instead of this double integral, ray tracing algorithm is used for the project. In ray tracing thousands of rays are being sent from each triangular finite element to other triangles in domain – space around. Then these rays are traced for any possible collision with other triangles or if no collision is detected, moving with the ray is continued. At the end view factor matrix is calculated.

Since subject is sending thousand of rays for each of the thousands of triangular finite elements in the problem input file, the algorithm seems expensive for implementation purposes. Therefore space subdivision methods are made used to overcome this time consumption of code.

Among space subdivision methods octree algorithm is observed as a fast way to move in the domain with the ray. As a result of this fact, the space subdivision part of the code is based on octree method that is a kind of hierarchical division. There are actually 3 different codes developed for this problem. The methods used and the differences are all explained in detail in the following parts of the report.
Chapter 2

Non-Uniform Space Subdivision in Collision detection

Under non-uniform space subdivision title it is aimed to explain how the “octree” - non uniform – space subdivision is used for calculation of view factor matrix with the help of ray tracing in this code.

Before going into details about the code, it would be better to look at the general structure of it, see Figure 2.1. The code contains a main part namely main.cpp, a header file named my_header.h and five different objects, which will be later used in the calculations and will be explained in detail in preceding pages of the report. How the code is processing thorough the given input file, how it creates the space subdivisions, how rays are sent and how the collisions and also view factor matrix is calculated will be explained while giving detailed information for the main.cpp.

2.1 General Information for the objects

2.1.1 Class point

Class point can be named as the basic tool of the whole code. It is using the coordinate information of the triangular finite elements given in the input file. Therefore it has three private double variables for the coordinate data - namely “double x, y, z” - and one private integer variable - namely “int name” – for the naming of the node / point. The class functions are the basics of the code because without the point data almost nothing is possible and in all other classes point objects are used.

The class functions defined in point class are as follows:
int get_name();
double get_x();
double get_y();
double get_z();
void give_x(double);
void give_y(double);
void give_z(double);

void draw_p();

friend void outfile_inputted_pnts(point*, int);

friend int* input_all (point*, triangle*, int*);

friend double* plane_equation(point, point, point);

friend void* space_lmt_mv_lght(double*, double*, int, point*, int);

friend class triangle;
friend class octan;
friend class vector;
friend class surface_grid;

Class functions used in order to assign values to privates of a point object and also to retrieve the private data of a point object as well.

Outputs private information of a point object.

Any point type array is sent with the number of points in it for check and comparison purposes.

Gets the data from input file exhaust_tria; saves the coordinates to a point type dynamic array, points' names of the facets to triangle type dynamic array and the number of triangles and number of points data to a integer type array.

Using the given three point type data finds the plane equation containing these three points.

Using the transferred data limits of the domain and the minimum side length of the smallest voxel – sub_octan – is found.

In order to be able to use the private values of each class in other classes, they are defined as friends to point class.
CHAPTER 2. NON-UNIFORM SPACE SUBDIVISION IN COLLISION DETECTION

2.1.2 Class triangle

With the point information in hand, coming data in input file is about the finite element facets. These facets are defined with three point data. In code after points one of the mostly used objects is triangle. Three point values and one integer value, namely “int tria_name, point p_zero, point p_one, point p_two” are the privates of the class.

Class functions defined for triangle objects are as follows:

- \textit{point get\_p\_zero()} \hspace{2cm} These triangle class public functions are defined to use the private data of any triangle object.
- \textit{point get\_p\_one()} \hspace{2cm} Outputs private information of a triangle object.
- \textit{point get\_p\_two()} \hspace{2cm} Calculates and gives the center, as a point object, of the triangle by which it is called.
- \textit{void draw\_t();} \hspace{2cm} Sends the first ray from the facet orthogonal to the plane
- \textit{point t\_center();} \hspace{2cm} Sends the first ray from the facet orthogonal to the plane
- \textit{vector create\_rays();} \hspace{2cm} All voxels are checked to find the triangle origin in octree.
- \textit{octan where\_is\_ray(octan*, int);} \hspace{2cm} Able to call it, function explained above is again friend here.
- \textit{friend double* plane\_equation(..);} \hspace{2cm} In order to be able to use the private values of each class in other classes, they are defined as friends to triangle class here as well.

2.1.3 Class octan

Class octan is a crucial class for the space subdivision and building of octree voxels. Six of them being double variables and remaining two being integer variables, octan class has 8 privates in total. The double privates are the corner information for the voxels that are octan objects. One of the integer privates is the name of the octan object. The remaining private is an array of integers that stores the names of the triangles – finite element facets – that are lying in the octan in hand.

Functions defined for octan objects are as follows:
These functions are used again to retrieve private data from any `octan` object created.

Initialization of the first list element to given value \( x \).

Building the corners of an `octan` object as `point` objects.

Function is for building the octree. It is called by the biggest octree `voxel` that is the starting domain of the given input file, with the maximum and minimum coordinate values as its six double private values.

With this function all the triangular finite elements – facets – are covered by cubic `bounding_boxes` that are `octan` objects.

Function is the most important function of the code. For each facet sending the rays, checking other facets against collision, – if no collision found – moving through octree `voxels` and outputting the view factor matrix are done.

Again defining as friend to use in `octan.cpp`
CHAPTER 2. NON-UNIFORM SPACE SUBDIVISION IN COLLISION DETECTION

friend void outfile_sub_octans(int, octan *);  
Outputting the octree *voxels* to a *.msh* file

friend void outfile_bndng_boxes(octan *, int);  
Outputting the *bounding boxes* to a *.msh* file

friend void tria(octan *, int);  
Outputting facets that will be checked on *cmd*.

friend void outfile_next_octans(octan *, triangle, triangle *, point);  
Outfiling the next octree *voxels*, neighboring octans, through which ray will be traced.

friend void trias_in_next_octs(triangle *, point, octan *);  
Function was used to output the triangles in the next *voxel* that will checked.

friend void out_trias_in_oct(octan, triangle *);  
Function was used to output any

friend void output_strt_tria(triangle *, point *, triangle *, int);  
Outputting the intersected facets with respect to the plane equation, not only real intersected ones.

friend void output_rays(int, vector *, triangle *);  
Outputting the rays that are sent from a facet.

friend class triangle;
friend class point;
friend class vector;
friend class surface_grid;

For the sake of the private data retrieval from the other objects again other classes are defined as friends here as it was done before for other classes.

2.1.4 Class vector

For sending the rays the definition of *vector* object is done by this class. Three double private values are defining the magnitudes of the lengths of rays in three coordinate axes, namely the i-component, j-component and k-component of the vector is stored as the privates of that vector.
CHAPTER 2. NON-UNIFORM SPACE SUBDIVISION IN COLLISION DETECTION

Functions defined under this class are:

```cpp
double get_i_comp()
double get_j_comp()
double get_k_comp()
void give_i_comp(double)
void give_j_comp(double)
void give_k_comp(double)

void draw_vec();
```

For retrieving and storing data from / to the private variables of a `vector` object, such functions are used. In order to assign values a double variable is needed as seen.

```cpp
void normalize();
```

Calculating the length of a `vector` object usually before normalization.

```cpp
double length();
```

The length of the `vector` object is changed to unity with this function.

```cpp
friend double operator*(const vector, const vector);
```

Dot product is defined for `vector` objects.

```cpp
friend vector operator^*(const vector, const vector);
```

Cross product is defined for `vectors`.

```cpp
friend double* plane_equation(...);
```

Again as above just to show to class and then to use it...

```cpp
friend class triangle;
friend class point;
friend class octan;
friend class surface_grid;
```

The classes are again defined as friend here to make use of private variables of each other in a less expensive manner.

### 2.1.5 Class surface_grid

`Surface_grid` class is used while dividing the hemisphere around the facet into small curvature partitions. These partitions are obtained using two angles (Namely $\beta$ and $\alpha$). These two angles has two limit values ($\beta_1$ & $\beta_2$ und $\alpha_1$ & $\alpha_2$) so one surface portion has four double private values for ranges of angles. The portion is defined between $\beta_1$ & $\beta_2$ und between $\alpha_1$ & $\alpha_2$. Since these surface portions are used to send the rays with a probability rule in to the hemisphere, each `surface_grid` object has a double private variable for the probability value of itself. Additional to these five a name is given as integer
variable to each partition. So, five double and one integer private values in total define a surface_grid portion.

Functions defined under the surface_grid class:

```c
double phi_a0()
double phi_b0()
double tht_a0()
double tht_b0()
double prob0()
void give_phi_a0(double)
void give_phi_b0(double)
void give_tht_a0(double)
void give_tht_b0(double)
void give_prob0(double)
```

```c
friend int create_surface_grids(int, int,surface_grid*);
```

In order to receive the data assigned to private angle and probability values functions on left line are used. For assigning values to private data of the surface_grid objects, functions on right line are used.

```
friend int create_surface_grids(int, int, surface_grid*);
```

To built the surface partitions on the hemisphere and assigning the angle values to private data for each surface_grid object. Also the probability for each object is calculated and stored to private double variable prob.

```
friend class triangle;
friend class octan;
friend class vector;
friend class point;
```

As it was for other classes, for surface_grid class the remaining classes are defined as friends so that in private data usage no problems would be countered.

### 2.1.6 Main

Obviously, the brain of the code is the main.cpp file. Main reads through parts of code in a sequential manner. These parts are as fallows:

- Input of the point and triangle information from the given input file exhaust_tria.msh.
- Bounding the facets with cubic boxes.
- Calculation of maximum–minimum coordinates for the domain and the smallest side length that will occur after building the octree elements with respect to inputted depth of division.
- Building the octree voxels and addressing each facet in the subdivided space.
- Surface partitions, actually surface_grid objects, built since rays are sent with respect to the probability of each surface_grid into the hemispherical space surrounding the facets.
• Finally, ray–plane manipulations end the code. Rays are sent from facets, traced against collision and continued to be traced through the neighboring voxels in case of no collision.

**Inputting the data**

Since the input file is not in a directly readable manner, some small tricks are used in order not to make any modifications on it and directly sent to code [1]. Till coming to the real data some unnecessary information is passed over with a small *do-while* loop. Then the second *do-while* loop does the storage of real data. But it takes more than expected for this *input_all* function to finish its job because the format of the data is complex for readability. Some tricky checks are used in order to find out whether if the upcoming, next data is really the coordinate data or the power of the already stored coordinate, see Figure 2.2. (Detailed explanations are done on the code as comments.)

\[
\begin{align*}
20031 & \quad -0.02524 \quad -2.6749 \quad -7.334 \quad 4 \\
20032 & \quad -6.823 \quad -2.4231 \quad 0.03912
\end{align*}
\]

**Figure 2.2:** Input File

**Bounding the facets**

The idea of bounding boxes is one of the differences with other methods used in other codes. Each facet has 3 *point* object data. Using these data the maximum–minimum *x*, maximum–minimum *y* and maximum–minimum *z* coordinate values are found and used in building the *bounding_box* of that facet, see Figure 2.3. A *bounding_box* is actually an *octan* object that has the 6 values explained above as private data. *Bounding_box* idea is used in placing the facets in the *octree*. These 6 private coordinate values of each *bounding_box* is then checked in *octree* formation to get the correct address of the facet in space subdivision [2].

For instance the facet on left has some part of it in voxel 2144 however most part of it is in 2148. In order to overcome such trivial situations, checking the 6 private data of the *bounding_box* of the facet is used during addressing a facet in *octree*, see Figure 2.4. On the other hand it has two drawbacks.
CHAPTER 2. NON-UNIFORM SPACE SUBDIVISION IN COLLISION DETECTION

First one is; since the facets will not be addressed to a **voxel** even if one of the 6 coordinate data is not included in the **voxel**, some facets can not be addressed till the end of the given **depth of division**. For the instance given in the Figure 2.4 the facet will be addressed to the root 214, not to any sons of 214 (Neither to 2148 nor to 2144). Second problem may be in collision check. Since the facet is not addressed to 2148 it will not be in the facet **list** of that voxel as well. However if a ray passes through 2148 or 2144, it would be logical that this facet is also checked against collision. But in this case, the ray is waited until it leaves the sons, namely 2148 or 2144 and moves to the root 214.

**Calculation of domain boundaries and the possible smallest side length of the octree voxels**

First of all the **depth_of_division**, till which level the division of space will continue should be inputted by the user. Then together with this integer value, all the **point** data that was inputted at the beginning is also sent into the function **space_lmt_mn{lnght}(...)**. Also two dynamic arrays are also sent into the function in order to retrieve the minimum-maximum coordinates and the minimum side lengths that will occur in the **octree** due to the desired space division level via updating the stored information in the address of these arrays.

**Octree Formation**

For building the **octree** all **bounding_box** information, total number of facets and the desired **depth_of_division** are needed. At the end the formed voxels – i.e. **sub_octans** – are stored in a **octan** type variable. Another output of the function **octan_subs(...)** is the total number of **sub_octans** built by the function. The function is called by the biggest **voxel** that is created by using the max–min coordinates coming from the **space_lmt_mn{lnght}(...)** function. These max–min coordinate data become the privates of the first and the biggest created **octree octan** for the domain.

In the function the first **for loop** is taking each **bounding_box** and putting it in a second **for loop** that has 8 different **if** conditioning in order to address the facet in the subdivided space. This second **for loop** continues till the end value which is the **depth_of_division** – integer value- desired by the user is reached. As mentioned in the **bounding_boxes** section, all the 6 coordinate data of a **bounding_box** around the facet in process is checked with respect to the middle values of the present **sub_octan**.

For instance at the first level using the biggest **octan’s** private data, middle points of the biggest **octan** sides are found. So that the 6 private data of the **bounding_box** that are the maximum and minimum values for the coordinate information of the facet corners can be checked whether if these are all lying in one of the 8 **sub_octans** of the biggest octan in the first level or of the **root octan** in other levels. Since 8 **sub_octans** are possible, in the second **for loop** there are 8 **if** conditions, see Figure 2.5. All these conditions are actually doing the same procedure, only the coding is changing. At each level the **sub_octan** that will be created since the entire private data is found in the same son, is
checked whether if it is already created for any other facet before. If yes, then the facet in process is added to the list of the sub_octan that is created before.

Due to bounding_box idea the coding has different number of digits for each facet. One facet may have only one number that is the first biggest octan, namely “1” or may have as depth_of_divisions digits in coding. If depth_of_division is 6 then coding for facet may be “134285” But 6 digit coding is only possible if all the private data of the bounding_box around the facet are found in the same sub_octans in each level. The coding, i.e. addressing of the bounding_box, i.e. facet is being assigned to the private variable name of the bounding_box.

After the end of for loops explained above, there exists a code part that adds “1” at the beginning of all addressing data. Also it is assigning only “1” to the facets that do not have any address due to bounding_box idea since it is possible to have facet without addresses. It is possible to have the problem of not having all the 6 private coordinate data of the bounding_box in same sub_octan of the first biggest octree voxel. So such facet stays without any coding, i.e. address. In order to overcome such difficulties such a solution is used, see Figure 2.6. Finally the octree obtained for all the facets from input file is as in Figure 2.7.

Surface Partitioning

Surface partitioning is needed to send rays with respect to a probability rule [3]. For the partitioning two angles are subdivided. These two angles are namely α and β (Θ and Φ in Figure 2.8). β is the angle between the ray
that will be sent and the orthogonal axes, i.e. local z coordinate of the facet. The angle \( \alpha \) is the angle in the plane of the facet. These angles have ranges so that the surface partition (\( dA \) in Figure 2.8) can be created. Namely \( \beta_2 - \beta_1 \) and \( \alpha_2 - \alpha_1 \) defines the surface grid. Due to backface culling, \( \beta \) changes between 0–90° and a change between 0–360° \([4]\).

Besides these four double variables, surface partitions, i.e. surface_grid objects in code have one more double private variable called prob that is the probability regarding that surface portion. The probability of a surface portion is calculated with the help of view factor formulation:

\[
F_{12} = \frac{1}{2\pi} \left[ \sin^2(\beta_2) - \sin^2(\beta_1) \right] [\alpha_2 - \alpha_1]
\]

(2.1)

Obviously some reductions, using the relevant assumptions, are made until coming to the probability formulation above. For all surface_grid objects, i.e. for all surface portions calculated probabilities should sum up to “1” in total. (which is satisfied for this code.) For this purpose create_surface_grids(int,int,surface_grid*) function takes the desired divisions for two angles as integer variables and also gets the surface pointer type data to update and store the created surface_grid objects in it. Finally visualization of the obtained structure is as in Figure 2.9.

**Figure 2.8:** Surface Partitioning

**Figure 2.9:** Surface Grid

**Sending Rays, Tracing them through Voxels and Collision Detections**

Last but the least part of the code. Sending all the rays, chasing for collision and moving through the voxels are all done in this part by intersection_check(...) function. The important inputs for the function are the first ray – orthogonal vector sent by create_rays() – for the facet, see Figure 2.10, all the octree voxels, see Figure 2.8, minimum side length possible for the octree, the original octree-voxel of the rays, all facet data and the surface_grid data. It is also worth here to add that sending rays is another difference to other codes due to sending them only to one half of the hemisphere \([4]\).

Before going into the first for loop over all surface_grids, the position vector of the facet center and the local coordinates of the facet are formed as vector objects. The inputted first ray, see Figure 2.10 of the facet that is orthogonal to the facet plane is being the local z coordinate here. Then next is the first and the big for loop processing through number of formed surface portions.
This big for loop contains one more for loop sending the rays with an amount that is obtained by multiplying the total number of rays, see Figure 2.11 to be send and the probability of the surface_grid in hand. In this second loop the rays are formed from the first ray, local z, using β and α value. β (or α) is any random angle between the surface_grid private angles β₂ and β₁ (Same for α as well). It is found using the random function of C++. Actually the random function is giving integer values between 0 and 32767 [5]. So the given value from random is divided by the “RAND_MAX + 1” in order to certainly obtain a value in decimal. As a result, β₁ private data is added to the multiplication of rand() / (RAND_MAX + 1) value with the difference of β₂ and β₁. Then the ray is formed using these two values, i.e. β and α.

Of course the ray is with respect to local coordinates now. Therefore what is done next is converting the ray data to global coordinates, see Figure 2.12 [6]. Then the ray is normalized again to compensate the length changes resulting from coordinate conversion. Sent ray is then chased using the do-while loop. Obviously the facets in the list of origin voxel of the ray have to be controlled against any possible collision. So, this final loop in intersection_check(...) function continues either till a collision is found – that is controlled by variable bingo – or all the facets in the list of the voxel is checked. Collision is checked using the plane equations of the facets in the list. Here comes another difference with other codes written for this project. The difference is related to the collision check with the facets in the list of the voxel. Actually the check is only taking place if the numbering (or name) of the facet that is in the list is smaller than the facet name that is in process. This condition is used since the view factor matrix is symmetric for the elements that have same areas. So, at the end of the day finding an intersection point with the plane equation actually does not mean that a collision is reached. This intersection point on the facet plane is then checked whether if the point is really in the facet borders.

This is controlled by the angle sum. Three vectors are formed between the corners of the facet and the found intersection point, see Figure 2.13 [7].
the angles between these angles are summed. If the sum is found to be $360^\circ$, then the intersection point found is assigned as a real collision point \[7\].

Besides to that, an addition to the related matrix element is done. For instance, if the facet number 3 is in process and if a collision is detected with facet 567 then in the matrix, element on row 3, column 567 is increased one. If not, then the intersection found does not have any interesting effect for the code any more. Here it can be understood that actually the integers will occur in the \textit{view factor matrix} that are greater than one. Actually, all members of the matrix will be divided by a value which stands for the total number of collision detected for the facet in process. So at the end of the day all the members will be certainly lower that “1” and their sum (for one row since the facet stands for a certain row…) If no collision is found, movement to the neighboring, i.e. next \textit{voxel} is taking place. This part is controlled with if conditioning. \textit{Bingo} is the control variable here. When this region is entered due to not able to detect a collision in the original \textit{voxel}, first of all the plane equations of the \textit{voxel} walls are calculated. This calculation is done via \textit{plane equation(…)} function that gets three \textit{point} data as input.

Using the three corners lying in the same wall, all the wall equations are calculated easily \[8\] The wall equations are needed because the goal is to find out the exit of the ray from the \textit{voxel} and then move to the next \textit{voxel} neighboring. Then the ray is checked against an intersection with all these 6 plane equations, see Figure 2.14. Actually of course an intersection will occur with all if the ray is not parallel to the \textit{voxel} wall. But to understand which one is the exit point of ray from the \textit{voxel} the distance of intersection point to the origin of the ray is calculated.

After all 6 distances are calculated the \textit{smallest positive} one is assigned as the \textit{front face} exit and the \textit{biggest negative} one is assigned as \textit{back face} exit, see Figure 2.15 \[9\]. The separation of exit faces as negative or positive exit is done in the preceding \textit{if-else} if conditionings in the code.

The idea of finding the exit of the \textit{voxel} is another difference to other codes used for the project. Since now the exit point is known, the ray can be continued from that point into the next \textit{voxel}. The procedure to find the next \textit{voxel} is starting with finding out whether the ray is leaving the \textit{sub-octan} from a side,
CHAPTER 2. NON-UNIFORM SPACE SUBDIVISION IN COLLISION DETECTION

If the ray is leaving the sub_octan from a face then the minimum side length over two (\(min_{\text{lengths}} / 2\)) is added normal to the face so that it would be certain to reach the next neighboring voxel, see Figure 2.16. The determination of, to which face the addition take place is left to the value of \(equations[\text{minus}_t_{\text{exit}}(\text{or} plus_t_{\text{exit}})][0-1-2]\). For each axis one of the values is called by changing the last integer between 0 and 2.

Therefore, according to the face that the ray is leaving these called values either takes one or zero so that multiplication of this value with \(min_{\text{lengths}} / 2\) will always be in the next voxel.

If the ray is leaving the space from a corner then the \(min_{\text{lengths}} / 2\) is automatically added to all normal vectors of the 3 faces that own the corner. The last chance is to have the exit point on any side of the voxel. For such a case the \(min_{\text{lengths}} / 2\) is then automatically added to the normal vectors of faces that own this side, see Figure 2.17.

These additions are actually taking place over the \(x-y-z\) axis coordinates since they are also the normal vectors for the sides of the voxels of the octree built. When the additions to coordinate values are finished, a new point is obtained in the domain. However the address of this point is not known yet, i.e. in which sub_octan this new point sits is not clear yet. So the point is then checked through all the created sub_octans' private data to see in which one it is lying. This check over all sub_octans can actually be optimized by using a possible neighbor list or a special neighbor finding method [9]. The results
obtained are visualized by using the 30 days free commercial software GID 8, see Figure 2.18 [10].

Figure 2.18: Minus and Plus Exits of the Same Ray Sent from the Same Facet

2.2 Results

About the code performance, there are several trials made. Changing the depth of division, changing the number of surface grids, changing the number of rays sent from a facet and also changing the number of facets are some of the used cases for this code. The results are tabulated and figured. In order to make the trials faster and also to overcome the virtual memory problem of the computer used, generally 100 facets from the given input file are chosen as a small model of the problem. The increase in depth of division is observed to have a positive affect on the code performance, see Figure 2.19. However an optimization would be fine for this variable since from some point on the performance is going worse again. In order to make the trials faster and also to overcome the virtual memory problem of the computer used, generally 100 facets from the given input file are chosen as a small model of the problem.

Figure 2.19: Depth of Division vs. Time Graph

The increase in depth of division is observed to have a positive affect on the code performance, see Figure 2.20. However an optimization would be fine for
this variable since from some point on the performance is going worse again.

Figure 2.20: Number of Rays vs. Time Graph

Changing the total surface grids in the hemisphere of the facet front face has only a distinguishing effect at the beginning. Then after it seems, increase in the number does not really affect the time passed during solution process, see Figure 2.21.

Figure 2.21: Effect of Change in Number of Surface Grid for 1000 Rays

For 5000 rays per facet, changing the total number of surface grids affected the results more relative to previous case. It is something expected because the surface_grid number actually increases the work to do in the code since they are dividing the total number rays to every surface portion, see Figure 2.22.
CHAPTER 2. NON-UNIFORM SPACE SUBDIVISION IN COLLISION DETECTION

Figure 2.22: Same affect as in previous figure with 5000 Rays

Addition to above results all facets are checked with only sending one ray from each facet. The result is again not so suprising, The time is increasing due to increasing number of facets in the input file, see Figure 2.23

Figure 2.23: Effect of Change in Number of Facets

2.3 Possible sources of errors and outlook

After looking at the view factor matrix if some errors are still observable, the possible sources can be explained as well.

One of the possible reasons for the error can be the back face culling method used in code and also explained in previous sections. Since the rays are sent
from one face of the facet, it is still uncertain whether if this face is the front or back face of the triangular finite element. Actually the code provides both of the solutions, i.e. either the ray can be traced from the \textit{back face} of the facet or from the \textit{front face} of the facet. The results were also figured in previous section. However, the orientation of facets information is still a need for the code for further development in future.

Another error source can be the bounding box idea. Because of trying to address and place the facets certainly in \textit{voxels} of the \textit{octree}, they can not be placed to deeper sons of \textit{root voxels}. So if a ray is traced through sons of a root voxel, but not through root voxel itself, then there exits a possibility to miss that facet since it is only in the \textit{facet list} of the root voxel actually. Therefore as a comment for outlook, a \textit{voxel facet} list optimization can be effective for the code.

Although it does not compute a wrong solution, the finding of neighbors seem time consuming since the next point is checked through the \textit{sub-octans} so far built. Therefore an optimization can also be made for the neighbor finding part of the code. It can be helpful for the performance part of the code.

A last subject can be an optimization of the \textit{depth of division} variable. Till now it is inputted by the user manually. However it could also a good idea to optimize this code such that code understands how much division is the best for the given number of facet in the input file. This would not only increase the performance of the code but also would put a plus to the reliability of the code.
Chapter 3

Non-uniform structured space subdivision

In this method non-uniform structured space subdivision is used. Octree model is one of the best models for the non-uniform division.

Firstly, we read the input data from the file. There are two kinds of inputs. One is coordinates of the points which build the geometry. Other is triangles’ point’s numbers.

Secondly, after reading data, to create octree model we find the maximum and minimum points of facets or triangles and create bounding boxes. Instead of center point of a facet we take into account bounding boxes during space subdivision step.

Thirdly, we generate our geometry by using octree method. Octree model is consists of voxels which are in different size which means that non-uniform space subdivision.

Then as a forth step, in the geometry we send rays. The rays are sent from the center of the facets to the whole sphere by controlling two angle values.

After sending rays, fifthly we find the facet that the ray intersects. We search this facet not in the whole geometry but in the voxels. So we start from the voxel where the origin facet lies. This is the starting voxel. We take into account all facets in this voxel and try to find intersection of the ray and a facet. Intersection point is found by calculated plane equation of a facet and ray vector. Because of plane equation always a point is found. So we have to check that if the point is in the facet or not. For that checking we use comparison of triangles’ area.

If there is no intersection than as a sixth step we move into geometry. This means that we find the exit face and neighbor voxel in the direction of the ray. While searching exit face, plane equation of a face can be used but in that method we use known coordinates of the faces. After finding exit face and point we look for the neighbor voxel in the direction of ray and find it. Than we search for any intersection in this neighbor voxel. This process goes in that sequence until finding an intersection or going out the geometry.

Finally, by using total number rays which are sent from one facet and number of hits to another facet, we calculate view factor between two facets and...
view factor matrix is built.

### 3.1 Idea of bounding boxes

Idea of bounding boxes is generated because of the problem during creating geometry when center of facets are used. As you can see in the Figure 3.1, if center of facet is used in generating octree model than some part of the facet can be in the voxel while center of the facet is not. This means that Facet A is not in the member of Voxel A and while searching intersection point, Facet A will not be taken into account. To prevent that phenomenon, bounding boxes are used. Instead of using center of facet, a maximum and minimum point of the facet is calculated and while obtaining octree model both max and min points are used.

![Figure 3.1: Need of bounding boxes](image)

### 3.2 Octree generation

In creating octree, bounding boxes of the facets are used. So not only one point but also two points which are maximum and minimum points are considered. Before starting the process, certain depth or level is decided. In every level if the voxel is non leaf node there become 8 divisions and storage by pointer method is used. The sons are named always in the same direction with numbers from 1 to 8, see Figure 3.2.

![Figure 3.2: Idea of creating octree model and storage by pointer method](image)

In the program every voxel has a list. What is written in that list is the name of the facets which lie in a voxel. While finding intersection point between facets and ray vector in a certain voxel, the facets are called from that list. Every octan or voxel has information of max-min points and name of the triangles.
which are in itself. Below you will see our real model, see Figure 3.3 and octree
model, see Figure 3.4.

![Figure 3.3: Real model](image1)

![Figure 3.4: Octree model](image2)

3.3 Ray sending

In this method, ray scans 360 degrees in the space, see Figure 3.5. $\alpha$ and $\beta$ are
angles. Ray sending is controlled by changing these two angles. $\beta$ is meridian
parameter and $\alpha$ is parallel parameter. In order to obtain correct results in the
view factor matrix, rays must be distributed uniformly. For this purpose we
control increment of $\alpha$ by using $\beta$. For example: $\beta = \beta + 10$ constant increment.
$\alpha = \frac{\beta + 10}{\sin(\beta)}$ increment controlled by $\beta$. As a result, density of ray in all of the
sections in the sphere is same. This means uniform distribution of rays.

![Figure 3.5: How the ray is sending from a facet](image3)

3.4 Ray–facet intersection

In each voxel, see Figure 3.6, by using plane equation of the facets and the ray
vector we find intersection point for all facets:

double* octan::plane_equation(point zero, point one, point two)
{
    double* coefficients = new double[4];
    vector u(one.x-zero.x, one.y-zero.y, one.z-zero.z);
    vector v(two.x-zero.x, two.y-zero.y, two.z-zero.z);
    vector n = u*v;
    n.normalize();
    coefficients[0] = n.i_comp;
    coefficients[1] = n.j_comp;
    coefficients[2] = n.k_comp;
}
zero, one and two are three points of a facet. By using these points we create two vectors on a facet as \( u \) and \( v \). Then we take the cross product of the vectors \((u \times v)\) and get the normal vector. After that, the unity of normal vector is calculated which means normalizing. \( A \cdot x_0 + B \cdot y_0 + C \cdot z_0 + D \) is the plane equation formula. \( A = x \) component of the normal vector \((i_{\text{comp}})\)
\( B = y \) component of the normal vector \((j_{\text{comp}})\)
\( C = z \) component of the normal vector \((k_{\text{comp}})\)
\( D = -(i_{\text{comp}})*(x \text{ comp. of zero}) + (j_{\text{comp}})*(y \text{ comp. of zero}) + (k_{\text{comp}})*(z \text{ comp. of zero})\)

Unit ray vector can be shown like \( \mathbf{R_d} = [i_{\text{comp}} \ j_{\text{comp}} \ k_{\text{comp}}] \). Starting point of the ray vector is center of the facet. \( \mathbf{R_0} = [x_0 \ y_0 \ z_0] \). Intersection point is found:

\[
    tp = -\frac{A \cdot x_0 + B \cdot y_0 + C \cdot z_0 + D}{A \cdot a + B \cdot b + C \cdot c} \quad (3.1)
\]

If \( tp < 0 \) then the ray intersects plane behind origin, i.e. no intersection of interest, else compute intersection point:

\[
\begin{align*}
    \text{int}_\text{px} &= \text{triangless}.\text{center()}.x + \text{tp}\mathbf{R_d}.i_{\text{comp}}; \\
    \text{int}_\text{py} &= \text{triangless}.\text{center()}.y + \text{tp}\mathbf{R_d}.j_{\text{comp}}; \\
    \text{int}_\text{pz} &= \text{triangless}.\text{center()}.z + \text{tp}\mathbf{R_d}.k_{\text{comp}}; \\
    \text{point} \hspace{1pt} \text{inter}(1, \hspace{1pt} \text{int}_\text{px}, \hspace{1pt} \text{int}_\text{py}, \hspace{1pt} \text{int}_\text{pz});
\end{align*}
\]

The next step is checking the intersection point whether it is in the facet or not. For that checking, areas of the triangles are used.

The area of triangle \( \triangle ABC \) is half of \( \triangle ABC \), or \( S = \frac{1}{2}|AB \times AC| \), see Figure 3.7 If the intersection point is inside the facet: \( A(\triangle DAB) + A(\triangle ADC) + A(\triangle BDC) \leq A(\triangle ABC) \) If else then the intersection point is outside, see Figure 3.8 Areas are calculated by using vectors. For example: \( A(\triangle DAB) = \text{Area of triangle } DAB \)

Another important point is finding the closest hitting facet to the origin facet. For this purpose we compare \( tp \) values. As we mentioned before if \( tp \) value is greater than 0, the ray intersects plane in front of origin and this facet can be your target facet but this is not sufficient. Also for target facet \( tp \) value must be smallest positive value.
3.5 Finding exit face and exit point of the voxel

To find the exit face and point, instead of plane equation known coordinate of the faces is used. Firstly, we want to show how we named the corner, see Figure 3.9 and faces, see Figure 3.10.

For example, if the exit face is “1” then exit point’s x coordinate will be definitely x_min of the voxel. Because one coordinates of exit point, ray vector and center of facet are known other two coordinates can be calculated very easily.

```
xp = intersected_octans[g].x_min;
x = xp;
```
subtr = x-x_contr;
div = subtr/R.d.i_comp;
y = y_contr+div*R.d.j_comp;
z = z_contr+div*R.d.k_comp;
exit = 1;

- exit : number of exit face
- xp : known coordinates
- x_contr : center of ray vector
- y_contr : center of ray vector
- z_contr : center of ray vector

For all 6 faces, intersection points are found than these points are checked whether it is in the face or not. If the point is inside the borders of a face, then that face and that point are the results we are searching for, see Figure 3.11.

3.6 Finding neighbour voxel

At this point we know the exit face and exit point. Firstly we find all the voxels which include the exit point that we calculated previously. The reason why we found more than one voxel is the way of our octree generation. Also father of son voxels include that point. Secondly, we search the correct neighbour among these voxels.

During searching neighbour voxel firstly we take into account voxels which are in the same level of the origin voxel as a neighbour voxel. If there is not any voxel in that level than we move onto one upper level. The reason why we are doing like that is the following. Because of bounding box idea, as you can see in the Figure 3.12, father of the voxels must be checked before continuing moving into the geometry. A facet which is in the list of a voxel can appear in the son of that voxel partially.

3.7 View factor matrix

After doing all the steps above at last we calculate view factor matrix. The view factor is obtained as the fraction of total rays leaving from one facet that hits another facet. It is purely geometric relationship, independent of viewpoints or surface attributes.

$$F_{ij} \cdot A_i = F_{ji} \cdot A_j$$  \hspace{1cm} (3.2)
\( F_{ij} \): view factor of \( i^{th} \) and \( j^{th} \) surfaces. \( A_i \): area of the \( i \) th surface. If both surfaces are same than view factor matrix is symmetric. In our example areas of facets are approximately equal each other. So \( F_{ij} \) and \( F_{ji} \) must be same or very close to each others. If we look at the results, for the facets near to the origin facet they are in acceptable range, in other words results are very close to exact results. But for the facets which are far away from the origin facet we obtain serious deviation between numerical results and exact results. Another factor that affects the view factor matrix is total number of rays that are sent from a facet. If you decrease that number then you obtain totally wrong results. You must increase density of the rays up to a certain limit in order to obtain acceptable results.
Chapter 4

Uniform and combined model approaches

4.1 General information of the classes

In this part of the report two different working modules will be explained. The first module is the code finding the formfactor matrix by dividing the control volume into equal-sized 3D grids. In the other code the full voxels are unstructured, but the empty voxels have hierarchic form. The reasoning of the followed strategies will be clarified in the forthcoming parts of the report. In these two codes, the one month free software GiD® and its postprocess library functions were used to prove the validity of different substeps of process and also final result. It will be started to take a look to the class structure of the codes. The important friends and member functions are going to be explained later.

**class Point:**

The simplest class of the code. It stores the id of the point and the three coordinates of it. Only one friend function is important, which is used to send the correct connectivity of the current element to an object “Facet”. The other friend functions are used to arrive the information of some points in order to construct the unstructured three (friend void extendtheemptyvoxelsone(...)), and draw the three and prove that the home-facet relations are correct or not (friend void gidfacetinsidetest(...),friend void gidvisibility(...)) . These friend functions will be explained later.

**class Vector:**

Class vector is an internal tool whose functions are used to make the implementation of the code easier. It has two constructers. One of them is creating a vector by taking the three directional vectors, and the other one creating a vector by using two specified points ( Point oneend, Point otherend). Its functions are used to find the normals and the plane equations of the “Facet”
objects. In addition to that, rarely but not never, it helped by some ray sending and intersections cases. The functions are able to find the magnitude of a vector, normalize a vector, cross and dot products of two vectors, find the angle between two vectors, divide a vector to a double and multiply it with a double, and sum two vectors. It is set to be friend with the other classes, so that there won’t be a “private part access” problem to call the “Facet” plane equation coefficients.

**class Facet:**

Class vector is one of the two most memory required class of the code. It private the name of the Facet, the three corner point of the Facet, the normal vector of the Facet, the plane equation constants \((A,B,C,D)\) of the Facet, the grid coordinates of the home voxel of the facet center, and a vector of size 4 which stores the integer names of the home voxels. The number of home voxels can be one to four. A facet is inserted to a voxel list if one of the corner points or center is covered by that voxel. Theoretically it can be maximum four. Beside of that in the functions which are constructing the grid, the multiple locating of a facet into the same voxel is prevented. That means even though more than one corner or the center is bordered by the same voxel, this facet is sent to this voxel list only once. The most important functions of class “Facet” is listed below. More necessary explanation will be done later.

*void createlist(*);

This function locates the facets into their position in octree. In other words it stands for the first step of creation of the octree. It assumes that a facet lies inside of a voxel if the center is inside of it. The octree generation is done by simple conditionals.

*void createlisttwo(*);

This function is the safer and faster mode of the previous one. It is safer because it considers that the facet belongs to the list of a voxel if either one of the corner points or the center of the facet lies inside of this voxel. The construction procedure is totally different then the first one. It finds the home voxel position by using some double division and integer conversion techniques.

*void raythrough(*, *);

This function constructs the rays and sends it to formfactor calculator. The ray generation is totally uniform in space. The validity of this ray sending procedure is both verified by using simple geometries and the final result. Some of the other tasks of this function is to count number of rays send, and prevent the critical ray directions against some division by zero dangers. More about this function will be written later on.

*void Formfactor(*, *);

This function takes the current voxel and checks for the intersections of the current ray. After finding an intersection point with one facet plane equation,
it calls `char intersection(_:);` which returns with a “y” if the intersection point lies inside of this facet borders or not. The efficiency and correctness of this function is quite critical due to the frequent call of it. Please remember that every nonparallel plane-ray couple has an intersection points. That means all for those couples we need an inside-outside test. After this intersection point verification, the function Formfactor saves the nearest succesfull shot in the correct place of it in the visibility vector of the origin. If there are no successful shots in the current voxel, then it calls the following function.

`int raypropogate(_,_,_,_,_,_)`;
This function finds the next voxel in a efficient way. For the unstructured code, it calls another function called `friend int findtheneighbourofemptyvoxel(..)` to find the next voxel.

**class Octan:**

Class *Octan* is the calls of voxels. It privates the name of the voxel, the depth of the voxel (not unique for unstructured one), the number of facets inside of it, the names of the facets inside of it, a counter counting the current facet id in the filling process, the grid coordinates of the voxel, the cubic dimensions of the voxel, the positions of the corner planes (double corners[6]), and the integer names of the neighbors. In order to save space for the increasing depth, the names of the facets are preserved by just integers in a dynamic array rather preserving the facets themselves (’int* facetsname;’ but not ‘Facet* facetsinside;’). The same logic is also applied for the neighboring voxels. Just their integer name is saved as private but not themselves (’int neighbourname[6];’ but not ‘Octan neighbors[6]’). In such a way one can call the necessary object from facet name list and neighbor name by using the main arrays. Without these considerations, just because of the memory hashing, one can not work by using an average computer with this code. In addition to that, as you will see in the following pages, in these codes, there is a possibility to reach all of the information about a voxel by just using its name. The most important functions of class *Octan* is as follows;

`void openthespaceforfacets();`;
After calling `void createlisttwo(..);` or `void createlist(..);` the dynamic array for ’int* facetsname;’ is allocated by using this function.

`void putthefacetsinvoxel(_);`;
This function is called by a member function of class *Facet* named as, `void putitinsidevoxel (_._);`: This function fills ’int* facetsname;’ if the name of the corresponding facet fills the necessary requirements, such as having at least one corner or center point inside of this voxel.

`void voxelgridcoordinates(_._);`;
This function is another step of the octree. It takes the integer names of the dummy voxels and the limits of the space, and creates their dimensions, grid
coordinates and their corner coordinates. It consist the data about the facet population. The missing information about the voxels is just remained as the name of the neighbors and the extension of empty voxels.

```c
void neighbours();
```

finds the neighbors by filling the array 'int neighbourname[6]'. It considers the nonexistency of the neighbors if the voxel is located at the borders or corners. The members of this private array will be used to traverse the array in the octree. The advantage of this array is diminishing for the unstructured empty voxels, whose neighbors are also changing by decreasing their depth. For the empty voxels the ray traverse algorithm has done with; friend int findtheneighbouremptyvoxel().

```c
friend void extendtheemptyvoxelsone();
```

This friend function is also worth to speak about it here a little. This finishes the octree generation by extending the empty voxels as big as we can. That means we have now an unstructured octree after this step. The extension of the borders is applied only to the empty voxels, due to the quite homogenous distribution of the facet population into the voxels. So, it was not necessary to build a hierarchic structure for full voxels.

It would be better if it we take a look to the functions in detail in the sequential order

### 4.2 Reading the input files

The given single input file is divided into two pieces. One of them includes the nodal information 'ifstream infile2("node.txt");'. The other one stores the element id's and their connectivities 'ifstream infile1("facet.txt");'. Basic `<fstream>` library functions and operators are used to read the data (<<, >>, putback(), open(), close()). Because of the strange significance figure output of the nastran ASCII file, some dummy variables assigned and some conditions are introduced. The probable user in the future should keep in mind that, if some other kind of input file is used, there can be a problem because of those conditionals.

The input files are read twice. The purpose of the first reading process is just to learn the size of the Facet* array and Point* array. After that the memory allocation is done, the files are closed and opened again, and in the second reading attempt, the available private parts of those arrays are stored in those arrays by using the constructors. Another thing which has done at the and of these reading lines is reading the maximum and minimum coordinates of the control volume, and storing it in an array called; `double borders[6]`. This array will be used in the forthcoming functions for several different construction, correction and safety purposes.
4.3 Space subdivision

4.3.1 Creating the voxels

This part is done with the constructor and with the = operator. The names of the voxels should be chosen such that a direct correlation between the names, grid coordinates and the coordinates of the corners can be constructed. Please remember here that the number of voxels is only dependent on the depth of the tree and its size is given as:

```cpp
int voxelnumber = pow(8, depth);
```

The size of the voxels array should not be more than that in order to preserve the space which become very critical for increasing depths. In order to speak a little about the constructor, it is defining just the dimensions and the name of the voxels as follows:

```cpp
int power = pow(2, depth); // number of voxels at on edge of the root box...
for(int j = 0; j<3; j++)
    dims[j] = (borders[2*j] - borders[2*j+1])/power;
```

The number of facets inside of this voxel, their names, the grid coordinates of the voxel, and the coordinates of the edge points are initialized as zero here.

4.3.2 Putting the facets inside the voxels

For this part there are two different functions were written. One of them is considering that the facet is inside of the voxel, if and only if the facet center is inside of it. For smaller depths this assumption is giving good results. But after running and observing the visibility values qualitatively in the visual environment, we concluded that for the optimum depth, this assumption fails. However we still agreed that, for some geometries, one can obtain accurate results by using this listing technique. This algorithm is provided in one of the member functions of class “Facet”

```cpp
void Facet::createlist(double borders[], Octan voxels[])
{
    for(j = 0; j < 6; j++)
        minborders[j] = borders[j];
    do
        {
            xmid = ((borders[0] - borders[1])/2.f) + borders[1];
            ymid = ((borders[2] - borders[3])/2.f) + borders[3];
            zmid = ((borders[4] - borders[5])/2.f) + borders[5];

            if(center.x <= xmid && center.y <= ymid && center.z <= zmid)
            {
                borders[0] = xmid;
            }
            .
            .
            q = q+1;
        } while(q < depth);
```
The logic behind it is, dividing the root box borders for each time into two (dividing the current candidate box into eight), until we reach the depth given. So at the end we reach a small voxel, in which our facet’s center point is sitting. After this step the gridcoordinates of our home voxel is found just finding the difference of modified border values then the root box border values, and dividing them to the corresponding dimension of the box;

\[
\text{for}(j = 0; \ j < 3; \ j++) \ \ \ \ \ \ \ \text{gridc}[j] = (\text{borders}[2*j+1] - \text{minborders}[2*j+1])/\text{dimens}[j];
\]

The trick of the relation between the integer names and the gridcoordinates of the voxels enters here. The idea is the gridcoordinates xyz, represents the integer name (decimal system), if xyz is considered as a number in the numbering system pow(2,depth). In the code it appears like that;

```c
int power = pow(2,depth);
// gridc[0] = x coordinate of the voxel in the grid.
```

In order to give an example, the voxel having grid coordinates x=2,y=1,z=0 In depth 3 has the decimal integer name of;

\[
\text{name*} = 2 \cdot (2^2)^2 + 1 \cdot (2^3)^1 + 0 \cdot (2^3)^0 = 136
\] (4.1)

The conversion in vice versa is also easy if it is necessary to know the gridcoordinates of a voxel if its name is known. If we return back to the listing algorithm, after finding the grid coordinates, now we know the integer name of our home voxel. In this algorithm, the facet are not send to its home voxel, but to a counter, (voxels[voxelname].findthecapacity();) which calculates the number of facets lying inside of that voxel in order to open a dynamic space for all of the voxels after finishing this step. As a result, the population of two voxels is looking like in Figure 4.1.

At smaller depths the population is looking healthy with the center point assumption. But if we go to bigger depths, we observe that with the increasing facet size voxel size ratio, some data is lost. For example two voxels at depth 5 are looking like the following Figure 4.2 if we use the function “createlist()”.

As one can observe some information is lost for the upper voxel. That means some facets are not included in that voxel just because their center point is not inside of it. This is creating an accuracy problem if we move further in depth. And as one can imagine increasing the depth is critically necessary to fasten the formfactor algorithm, being our main task for this course.

In order to overcome this problem, we developed another function which lists the facets inside of the voxels with a different logic. It assumes that a facet is sitting inside of a voxel, if either one of its corner points or its center it inside of it. This consumes a little more space then the previous assumption due to the fact that a facet can be in more then one list. However this procedure deserves to be proceeded because of its obvious accuracy with respect to the previous one. This method is done in the function “createlisttwo()” as follows;
CHAPTER 4. UNIFORM AND COMBINED MODEL APPROACHES

Figure 4.1: Population of two voxels

Figure 4.2: Two voxels at depth 5

void Facet::createlisttwo(double borders[], Octan voxels[])
{
    double dgridcords[3];
    double power = pow(2.f, depth);

    for(int j = 0; j < 3; j++)
    {
        dimens[j] = (borders[2*j] - borders[2*j+1]) / pow(2.f, depth);
        dgridcords[j] = 0;
    }

    dgridcords[0] = (c.x-borders[1])/dimens[0];
    dgridcords[1] = (c.y-borders[3])/dimens[1];
    dgridcords[2] = (c.z-borders[5])/dimens[2];

    for(j=0;j<3;j++)
    {
        gridc[j] = int(dgridcords[j] + pow(10.f,-10.f));
    }

    correction = gridc[2]*gridc[1]*power+gridc[0]*power*power;
    voxelname[3] = int(correction+pow(10.f,-10.f));
Here, just the process done for the corner “c” is shown. The same procedure is applied for the other two corners and the center of the facet. After that, for each successful facet, the counter function is called in order to open the dynamic array for the facet lists later on. The procedure is simple algebra. Taking the coordinate of the corner point or the center point, making a double division and then an integer conversion. The result is giving the grid coordinates of the home voxels. One facet can belong to maximum four different, minimum one voxel lists. The names of the home voxels are found again with numbering system algebra as explained before. Later on the corresponding home voxels will be stored in int voxelname[4]. Here only the counting process is done.

```cpp
voxels[voxelname[0]].findthecapacity();
if(voxelname[1] != voxelname[0] && voxelname[1]!=0)
    voxels[voxelname[1]].findthecapacity();
    voxels[voxelname[2]].findthecapacity();
    voxels[voxelname[3]].findthecapacity();
```

The conditionals are standing for preventing to count the same facet to the same voxel again and again. Otherwise the function would lose its effectiveness. There is another conditional in the code to prevent a double to integer conversion error, which is worthless to be explained here. As a result, the same voxel’s population shown in Figure 4.2 are looking like now, see Figure 4.3.

![Figure 4.3: Voxel's population using createlisttwo()](image)

The problem of data loss seems to be solved for greater depths. Otherwise the observed consequences would be just passing the ray through and finding an intersection point with a completely another facet or sending the ray into the empty space although it has to find a target. It is observed and verified qualitatively that this is a very critical point which has to be done. Because it should be “slow but true” rather than “fast and wrong”.

4.3.3 Allocating the memory for the facet list and filling it

So far we knew the capacity of every voxel. There were some ideas about allocating the memory for the facet lists. However what has to be done here is not to allocate the memory more than necessary. This was the reason why we were using the counter function in the previous step. The memory allocation is done with:

```c
for(n = 0; n < voxelnumber; n++)
voxels[n].openthespaceforfacets();

void openthespaceforfacets()
{
    if(facetsinvno != 0)
        facetsname = new int[facetsinvno];
}
```

Here it is prevented to try to open the space for empty voxels, due to the impossibility to open an array of size 0 in c++. After this step the names of the facets should send to their corresponding homes. This done as with three different functions follows:

```c
for(k = 0; k < i; k++)
f[k].putitinsidevoxel(voxels,borders);

void Facet::putitinsidevoxel(Octan voxels[], double borders[])
{
    voxels[voxelname[0]].putthefacetsinvoxel(facetname);

    if(voxelname[1]!=voxelname[0] && voxelname[1]!=0)
        voxels[voxelname[1]].putthefacetsinvoxel(facetname);

    if(voxelname[2]!=voxelname[1] && voxelname[2]!=voxelname[0] && voxelname[2]!=0)
        voxels[voxelname[2]].putthefacetsinvoxel(facetname);

        voxels[voxelname[3]].putthefacetsinvoxel(facetname);
}

void putthefacetsinvoxel(int currentname){facetsname[counter]=currentname; counter++;}
```

The integer variable “counter” was set to zero in the beginning for all of the voxels. This variable controls the position of the next facet in the array of int facetsname[facetsinvno];. In the function “putinsidevoxel”, the conditionals are standing for repetitive loading of the same facet to the same voxel. Because it is highly probable that all of the corners and also the center of the facet sit in the same voxel. One important point which has to be mentioned here is, the rays are send from the center point of the facet. That is a consequence of one of our big assumptions that the origin facet has negligible area and to be considered as a single point. Therefore the first candidate targets are considered to be in the same voxel where our origin facet’s center is sitting but not the corner homes.
4.3.4 Defining the grid and the corner coordinates of the Voxels

The good thing about our models is that the corner and the grid coordinates of the voxels are explicitly given with the depth, the borders of the root and the name of the voxel. With this advantage, we used some simple division and remainder operations. Those operations are the inverse of the ones which are explained before.

```
for(k = 0; k < voxelnumber; k++)
    voxels[k].voxelgridcoordinates(k,borders);

void Octan::voxelgridcoordinates(int k, double borders[])
{
    int power=pow(2,depth);
    for(int l=2;l>=0;l--)
    {
        gridcoords[l]=k%power;
        k=k/power;
    }
    for(int i=0;i<5;i=i+2)
    {
        corners[i]=(gridcoords[i/2]+1)*dims[i/2]+borders[i+1];
        corners[i+1]=gridcoords[i/2]*dims[i/2]+borders[i+1];
    }
}
```

For example the gridcoordinates of the voxel having the name 136 at depth 3 are;

```
gridcoords[2] = 136 % pow(2,3)=0;
136/ pow(2,3) = 17;
gridcoords[1] = 17 % pow(2,3)=1;
17/ pow(2,3) = 2; //integerdivision
gridcoords[0] = 2 % pow(2,3)=2;
```

And the coordinates of the corners are found just multiplying these gridcoordinates with the corresponding dimensions of voxels and adding the initial values (the minimum borders of the root box). These border values limiting the box size are necessary in order to trace the ray in the tree.

4.3.5 Finding the neighbors of the voxels

The neighbors of the voxels should be specified to send the ray through the tree. This step is also not differing between the combined model and the structured one. The nice thing of our numbering system method is that the neighbor names are explicitly given with the name of the voxel. The neighbors are stored in class “Octan” in an array of size 6, just with their integer names to preserve space. The even numbered array members are the positive side neighbors, whereas the odd numbered ones are the negative sided ones.

```
int neighbourname[6];
// neighbourname[0] = east(+x) neighbor
// neighbourname[1] = west(-x) neighbor
// neighbourname[2] = +y neighbor
// neighbourname[3] = -y neighbor
// neighbourname[4] = north(+z) neighbor
// neighbourname[5] = south(-z) neighbour
```
The neighbour finding algorithm is quite easy to implement. The idea is again taken from the numbering system conversion. As we explained before, the x-grid coordinate of a voxel is controlled by a coefficient of \(\text{pow}(\text{pow}(2, \text{depth}), 2)\). That means if a voxel is neighbour to another one in the positive x direction then its name is bigger then the other one with an amount of \(\text{pow}(2, 2 \times \text{depth})\). Because we know that those two share the same y and z grid coordinates. In order to give an example, see Figure 4.4, the neighbours of facet 136 are in depth 3.

![Neighbour finding algorithm](image)

Figure 4.4: Neighbour finding algorithm

The algorithm done is as follows;

```cpp
for(n = 0; n < \text{pow}(2, 3 \times \text{depth}); n++)
    \text{voxels}[n].\text{neighbours}(outfile);

\text{void} \text{Octan::neighbours(ofstream}\text{& outfile)}
{
    \text{for(int} i=0; i<6; i++)
        \text{neighbourname}[i]=-1;
    \text{int} power=\text{pow}(2, \text{depth});
    \text{if(gridcoords[0]!=power-1)}
        \text{neighbourname}[0]=name+power*power;
    \text{if(gridcoords[0]!=0)}
        \text{neighbourname}[1]=name-power*power;
    \text{if(gridcoords[1]!=power-1)}
        \text{neighbourname}[2]=name+power;
    \text{if(gridcoords[1]!=0)}
        \text{neighbourname}[3]=name-power;
    \text{if(gridcoords[2]!=power-1)}
        \text{neighbourname}[4]=name+1;
    \text{if(gridcoords[2]!=0)}
        \text{neighbourname}[5]=name-1;
    \text{//for}(i=0; i<6; i++)
    \text{//cout<<neighbourname[i]<<endl;}
}
```

The conditions above are staying to prevent not to find an unexisted neighbour. If the voxel is positioned either in the edges or corners then minimum one,
maximum three neighbours are missing. The names of those missing neighbours are thus prevented to be renamed again and have an initial value of -1. This number will be used to check if there is no neighbour for that ray anymore to travel, in other words understanding that the ray reached until the end of the space.

Another comment which has to be done is that the neighbours which are lying south east, south west or vice versa are neglected here. That means instead of taking 27 neighbours into the process, just 6 biggest candidates were selected. Actually we have another algorithm which stores 9 neighbours at each direction. However, this process is not bringing any extra accuracy, but costing time in ray travel. So we sacrificed a small amount of accuracy for the speed of the process.

4.3.6 Visualization of Grid

After creating the lists for the voxels and defining their grid coordinates and their corner coordinates we can visualize the geometry and observe if the process is done correctly or not. For this purpose the free GiD postprocess library functions are used. The visualization of the grid is done with this function:

```c
int voxelnumber=pow(8,depth);
void gidoutput(Octan voxels[],int voxelnumber) {
...
}
```

The functions which are used are listed sequentially beneath;

```c
//Open the post mesh file...
GiD_OpenPostMeshFile( "test.post.msh", GiD_PostAscii);

//Open the post result file
GiD_OpenPostResultFile( "test.post.res", GiD_PostAscii);

//Begin the mesh (the voxels are hexahedra)
GiD_BeginMesh("TestMsh", GiD_3D, GiD_Hexahedra,8);

//Write the point labels and coordinates
GiD_BeginCoordinates();
//End the coordinates
GiD_EndCoordinates();

//Write the element label and connectivities (there are 8 connectivities
//for each voxel)
GiD_BeginElements();
//Finish writing element list
GiD_EndElements();
//Finish the mesh
GiD_EndMesh();

//Define a central gauss point for writing the population of voxels
GiD_BeginGaussPoint("gausses", GiD_Hexahedra, NULL, 1, 0, 0);
//Finish writing the gauss points
GiD_EndGaussPoint();

//Define the result group
GiD_BeginResult("facets_in_the_voxels","voxel_information",1.0,GiD_Scalar,
GiD_OnGaussPoints,"gausses",NULL,0,NULL);

//Write the number of facets inside of the voxel to the postprocess file
GiD_WriteScalar(voxels[i].name, voxels[i].facetsinvno);
```
//Finish the process end close the files
GiD_EndResult();
GiD_ClosePostMeshFile();
GiD_ClosePostResultFile();

The visualization of the grid (the empty ones are not shown) is like shown for depth five in Figure 4.5.

![Visualization of the grid](image1)

**Figure 4.5:** Visualization of the grid

And the population in a voxel is reachable in the visual environment, see Figure 4.6.

![Voxel population reachable in visual environment](image2)

**Figure 4.6:** Voxel population reachable in visual environment

### 4.3.7 Extending the empty voxels

This part of the process was not obligatory but deserved to see and compare with the grid. One function is provided to extend the empty coordinates as
much as possible. One reason why we preferred to extend just the empty voxels
but not all is, as one can see from the Figure 4.6 in the working depth, the
population is distributed to the voxels uniformly. This is due to the fact that
the mesh size was kept constant and thus we have unisized facets in our problem.
Therefore we expect a homogenous distribution of facets for the working depth.
The visual grid above is also proving this by showing the color blue as dominant
marker.

The second reason of choosing the combined model (hierarchic empty voxels
and structured full ones) is obviously the ray travel in the grid improvement.
But after making the calculations, we observed that the combined model is not
faster but slower then the structured one for the working depths. The reason
of this is shortly loosing the neighboring information found in structured model
in the combined one. This problem and the impossibilities to overcome it will
be explained in the section of ray traveling.

If we need to explain how the extension of empty voxels is done, we can say
shortly that this procedure is constructing an octree over the grid. The space
is starting to be divided into eight parts. After that it is checked either there
is a full voxel standing in this area or not. If we find out that all of the voxels
inside of the subspace are empty, then we are extending the small voxel corner
coordinates to the current subspace. This procedure stops if we reach one level
up to our previous depth.

It is done with the following subroutine; (friend function)

```c
void extendtheemptyvoxelsone(double borders[], Octan voxels[], int voxelnumber, int depth)
{
    int power;
    int k;
    int currentgridcoords[3];
    double currentcorners[6];
    int numberoffullvoxels;
    double dims[3];

    for(int d=1; d<depth-1; d++) // turn one less than depth
    {
        power=pow(2,d); // modify the number of voxels on one edge at each time
        cout<<"extending for depth"<<'t'<<d<<endl;
        for(int j=0; j<3; j++)
            dims[j]=(borders[2*j]-borders[2*j+1])/power;
        for(int i=0; i<pow(power,3); i++)
        {
            numberoffullvoxels=0; // initialize the number of full voxels inside of division zero
            k=1;
            for(int l=2; l>=0; l--)
            {
                currentgridcoords[l]=k%power;
                k=k/power;
            }
            for(int j=0; j<5; j=j+2)
            {
                currentcorners[j]=(currentgridcoords[j/2]+1)*dims[j/2]+borders[j+1];
                currentcorners[j+1]=currentgridcoords[j/2]*dims[j/2]+borders[j+1];
            }
            for(j=0; j<voxelnumber; j++)
                if(voxels[j].facetsinvno!=0)
```
if(all of the corners of the j'th small voxel is inside of the subdivision)
   numberoffullvoxels++; // count the full voxels

if(numberoffullvoxels==0) //if here is no full voxel
{
   for(j=0; j<voxelnumber; j++)
      if(all of the corners of the j'th small voxel is inside of the subdivision)
         voxels[j].cornerchange(currentcorners,d); //change its depth and corner coordinates
}
}

The visual output of the combined model is looking like at depth 5, see Figure 4.7.

\[ \text{Figure 4.7: Visual output of the combined model} \]

### 4.4 Ray Sending, Ray traveling and Ray–Facet Intersection

#### 4.4.1 Ray Sending

Ray sending is one of the most critical discussions of this work. Forgetting about the speed and accuracy of the process, just finding out a way to send the rays into the space uniformly is a fundamental work. Without knowing a uniform ray sending strategy, we can not speak about a speed of the process. For the structured and for the combined model spherical parametric system is used to control the direction of the rays. We used one meridian and one parallel curve parameters and a tricky increment control in order to keep number of rays per unit area constant.

Consider the two isocurves given in the Figure 4.8. In order to keep the ray density constant, we should control the increment of alfa by using $\beta$. For
example consider we want to send 180 rays on the equator curve. If our aim is uniform distribution, then we should send $180 \cdot \sin(\beta_2)$ number of rays on the $\beta_2$ parallel curve. This is only possible if we set the increment of $\alpha$ to $1 + \frac{1}{\sin(\beta_2)}$ on that iso curve. Thus, the ray sending loop reduces to:

```c
for(beta=0.0000001;beta<179.00001;beta=beta+1)
{
    for(alfa = 0.0000000001; alfa < 359.000001; alfa = alfa+1/sin(betad))
    {
        ray.i = sin(betad)*cos(alfad);
        ray.j = sin(betad)*sin(alfad);
        ray.k = cos(betad);
    }
}
```

4.4.2 Ray–Facet Intersection

After sending the ray, first the candidate facets in the home voxel have to be checked for an intersection. The intersection algorithm has two main steps. One of them being the ray plane equation solver, the other one is an intersection point inside/outside test. The ray plane equation solver is something well known, but has to be mentioned here.

The main idea is, if any vector starting from origin and lasting at any point of our plane dot producted with the normal vector, it will give us a constant value. And this constant value represents the shortest distance of the plane to the origin. Thus the term $Ax+By+Cz$ standing for the dot product of the normal vector with the vector connecting the origin and the candidate point.

$$A \cdot x + B \cdot y + C \cdot z + D = 0 \quad (4.2)$$

Let’s make us clear what are those coefficients and how the plane equation is constructed, see Figure 4.9.

Consider we have three parameters here. First the normal vector of the plane as known. Second a point in that plane (the center or the corners are also known) and a point which we also know, and has to be proved if it lies on the plane or not.

The main idea is, if any vector starting from origin and lasting at any point of our plane dot producted with the normal vector, it will give us a constant value. And this constant value represents the shortest distance of the plane to the origin. Thus the term $Ax+By+Cz$ standing for the dot product of the normal vector with the vector connecting the origin and the candidate point.
CHAPTER 4. UNIFORM AND COMBINED MODEL APPROACHES

Figure 4.9: Construction of the plane equation

\[ A = \text{normal}.i; \]
\[ B = \text{normal}.j; \]
\[ C = \text{normal}.k; \]

And obviously D should be that constant mentioned above, standing for the shortest distance of our plane to origin. D can be easily found by inserting to the equation a point known on the plane (for example one corner point)

\[ D = -1 \cdot (A \cdot (c.x) + B \cdot (c.y) + C \cdot (c.z)); //c \text{ here is one of the corner points.} \]

Now, consider we have a ray originated from the center point, and travels on the direction of unit vector “Vector dir” which is one of the rays produced by the previous section. Then the end point coordinates can be calculated if we say that the ray travels with an amount of unknown parameter “t” as follows;

\[ \text{Intsc}_x = \text{center}.x + t \cdot \text{dir}.i; \]
\[ \text{Intsc}_y = \text{center}.y + t \cdot \text{dir}.j; \]
\[ \text{Intsc}_z = \text{center}.z + t \cdot \text{dir}.k; \]

And of this point are inserted to the equation it has to be satisfied.

\[ A(\text{Intsc}_x) + B(\text{Intsc}_y) + C(\text{Intsc}_z) + D = 0; \]
\[ A(\text{center}.x + t \cdot \text{dir}.i) + B(\text{center}.y + t \cdot \text{dir}.j) + C(\text{center}.z + t \cdot \text{dir}.k) + D; \]

Please remember here that we want to find a value for “t”. This value will exist always if our ray is not parallel to our plane. That shows the importance of the facet inside-outside test. This function is visited frequently out of the cases that the ray is parallel to our facet, being seldom. Back to the current problem, if we want to find a “t” value;

\[ t = \frac{-A \cdot \text{center}.x - B \cdot \text{center}.y - C \cdot \text{center}.z - D}{A \cdot \text{dir}.i + B \cdot \text{dir}.j + C \cdot \text{dir}.k} \] (4.3)

This plane-ray solver is located in the code in the function \texttt{void Facet:: Formfactor(....)}. In the code the numerator and denominator are defined separately. The denominator is calculated first and used as a check parameter in order to make sure that the ray is not parallel to the plane. Otherwise the debugger would give a “division by zero” warning. Before introducing the Formfactor function completely, it would make the sequence much more clear if we explain the facet inside-outside function.
4.4.3 Facet Inside – Outside Checker

As we explained before the “t” value standing for the distance between the center of the origin facet and the intersection point on the plane may exist for most of the rays send from that origin. Therefore another intersection check is necessary to make sure that the intersection point lies inside of our polygon. There are several methods for that purpose being valid for more general polygons. Here the one that we applied will be explained.

Consider a triangle and two candidate points on the plane equation of that facet but one of them being inside and one of them outside as shown in the Figure 4.10.

The idea is as follows:

1. Two vectors are created starting from one edge and directing to the other two edges of our target triangle.

2. Any point on the plane can be reached by using those two normalized vectors.

3. If one of the coefficients of those position determiners is zero or if the length of them is not between the necessary limits, then the point can be considered as outside.

The upper limits of the lengths for those two directions can be described as in Figure 4.11.

---

**Figure 4.10:** Triangle and two candidate points on the plane equation

**Figure 4.11:** Limit of the length
And finally the code for the outside inside check process is:

```cpp
char Facet::intersection(Point p) {
    Vector dirone(b,c); //create a vector from b to c
    Vector dirtwo(b,a); // create a vector from b to a
    double biga = Magnitude(dirone); // find the length of the edges
    double bigb = Magnitude(dirtwo);  //normalize the edge vectors
    dirone=Normalize(dirone);         //normalize the edge vectors
    dirtwo=Normalize(dirtwo);
    double one = (p.y-b.y-((p.x-b.x)*dirtwo.j/dirtwo.i)) /
                 (dirone.j-(dirone.i*dirtwo.j/dirtwo.i));
    double two = (p.x-b.x-one*dirone.i)/dirtwo.i;
    if(one>=0 && two>=0 && one<=biga && two<=bigb*(biga-one)/biga) //check borders
        return 'y';
    else return 'n';
}
```

As you can see this subroutine is kept as short as possible. According to wish, the edge vectors and also the lengths of the edges can be calculated before and stored. This will for sure increase the speed. The reason why it has not done here is just the memory problem. And the main loop of function Formfactor and some important add-it’s are given below;

```cpp
for(k=0;k<currentvoxel.facetsinvno;k++) //turn as the potential of the current voxel
{
    denominator=......; //calculate the denominator as explained above
    if(denominator!=0 && facetname!=currentvoxel.facetsname[k])
        //cut the loop if the ray is parallel to the plane equation, or the target is the origin
        {
            numerator=......; //calculate the numerator...
            t=numerator/denominator; //Find the distance
            intsc=sump(center,(t*dir)); //find the intersection point
            if(f[currentvoxel.facetsname[k]-1].intersection(intsc)=='y' && t>0 && t<tcompareplus) //change the succesfull target according to the magnitude of t, if the point
                //is inside of the target
                {
                    tcompareplus=t;
                    succesfullname=currentvoxel.facetsname[k]-1;
                }
        } //end of the outer if
}
if(succesfullname!=-1) //if the shot is in this voxel then succesfullname is modified.
    visibility[succesfullname]++;
```

### 4.4.4 Ray Traveling

Two conditions have to be checked whether the ray should travel in the geometry or not. The first necessary and sufficient condition is that the ray finds a target in the current voxel. For this case it is no use to elongate the ray further more. The other condition is to check that the ray reaches to the end of our geometry. Those conditions are checked for each step of the do-while loop of the Fromfactor function. The condition can be combined as; while(currentvoxelname!=-1 &&
    succesfullname===-1). As you can remember from the previous section, the succesfullname is initialized as -1 and modified as long as we found a successful target intersection point. The first condition benefits from the initialization of neighbor names. Note here that the unexisted neighbor name is allocated as -1, if the voxel locates at the end limits of the geometry. For example the voxels
located at the far east part of the geometry has east neighbors but initialized as -1.

The ray traveling has done with the function int Facet::raypropogate(...). This function supposed to find the next intersection point on the voxel walls first. This is done like as follows;

```c
double k[6];
double kmin=50000000000000
for(i=0;i<6;i++)
{
    k[i]=(currentvoxel.corners[i]-inter[i/2])/rayd[i/2];
    if(k[i]>0 && k[i]<kmin)
    {
        kmin=k[i];
        imin=i;
        check=1;
    }
}
```

Here in this part the k[] array is standing for the distances of the ray to the corresponding walls of the structure. In this section, the implementation is kept as short as possible by using the advantages of our wall name (double corners[6];) definitions and neighbourname definitions (int neighbourname[6];). As a result at the and “imin” gives us at which wall the nearest intersection is found. Actually this data gives us for the uniform grid the next neighbor name directly. For example, imin equals to 4 means that the next neighbor is the north one. In the uniform case, we are just returning in this function with the integer name of the next voxel like; “ret=currentvoxel.neighbourname[imin]”. However if we are in the combined model the situation becomes a little complicated and therefore slower. The difficulty which has to be expressed here that, without making any floating point operations, the smallest next voxels can not be found.

Consider this case here in Figure 4.12. The ray has 5 candidates among the east neighbours, each of them can be the next one. As opposite to the uniform model the light orange next neighbour can not be found without making any calculations. This is the main reason of inefficiency of combined model with respect to the uniform model for the depth values less then 6.

This problem has no practical solution. No of the existed models for non-uniform models guaranty an explicit neighboring algorithm. So, the calculation of the next neighbor for a respectively bigger home voxel is most probably inescapable. This is tried to be done in an efficient way as follows;

```
First the intersection point on the home voxel wall is found;

int a = imin/2;
int b = (imin/2+1)%3;
int c = (imin/2+2)%3;
inter[a] = currentvoxel.corners[imin];
inter[b] = inter[b]+k[imin]*rayd[b];
inter[c] = inter[c]+k[imin]*rayd[c];
```

**Figure 4.12: Combined Model**
After this point it is checked if the “currentvoxel” is one of the smallest voxels having the larger depth in the geometry or not. If it is one of the smallest voxels then the next neighbor is explicitly known and the algorithm does not differ from the uniform model, see Figure 4.13.

![Current Voxel](image)

**Figure 4.13:** Current Voxel

```c
if(currentvoxel.depth == depth)
ret = currentvoxel.neighbourname[imin];
```

But if the case is not this, the procedure of finding the next voxel has three steps. In the first step we are elongating the ray with a very small amount.

```c
for(i = 0; i < 3; i++)
inter[i] = inter[i]+rayd[i]*0.00000000001;
```

And we are calling a function which tells us where this elongated point is located in the geometry. The procedure is similar of creating the uniform grid and listing algorithms. We are taking the point, making a double division and converting it to an integer values. For each of three processes we obtain one grid coordinate of the voxel.

```c
for(int j=0;j<3;j++)
{
    dimensions[j] = (copyborders[2*j]-copyborders[2*j+1])/pow(2.f,depth);
    dgridcords[j] = (inter[j]-copyborders[j*2+1])/dimensions[j];
    igridcords[j] = int(dgridcords[j]+pow(10.f,-10.f));
}
correction = igridcords[2]+igridcords[1]*power+igridcords[0]*power*power;
returnvalue = int(correction+pow(10.f,-10.f));
```

The reason of double to integer conversions has two reasons. One of them is obviously to obtain an integer array index for the int neigbourname[6]. The other one is to prevent a very popular survival of programmers. (i.e $3.9999 = 3$).

And after here the necessary functions are visited again and again as explained before.

### 4.5 Error Analysis

The error analysis for formfactor matrix elements has a known difficulty. This is coming from the fact that the impossibility to calculate the global error. Although some analytical solution exists for all of the origin-target couples, just due to the complex shadow relation between the facets, the analytical solution is not calculatable for those couples if one another facet is lying between them. For this reason only the analytical solutions for the facet couples which are not blocked were applicable. We used those results and obviously called as “local error.”
First take a look of the calculation of the exact result in the code. The sphere drawn around the origin target radius represents the visibility potential of our origin center. For the analytical solution first the area of the target has to be projected on that sphere, and the area of this projection has to be calculated. This projection divided to the whole sphere surface area will give us the exact solution, see Figure 4.14. In the code:

```cpp
void exactvisibility(Facet origin, Facet target)
{
    double areaoftarget;
    Vector edgeone(target.a, target.b);
    Vector edgetwo(target.a, target.c);
    Vector r(origin.center, target.center);
    double distancebetween = Magnitude(r);
    double coefficient = cos(Angle(r, origin.normal) * asin(1) / 90) * cos(Angle(r, target.normal) * asin(1) / 90);
    areaoftarget = Magnitude(edgeone ^ edgetwo) / 2;
    double exactvisibility = areaoftarget * coefficient / 4 / 3.14 / distancebetween / distancebetween;
    cout << "exactvisibility:" << exactvisibility << endl;
}
```

**Figure 4.14:** Calculation of the exact result

### 4.6 Results of Uniform and Combined models

The optimum result for this process is found as 45 minutes with the uniform model. It is not represented on the diagram, see Figure 4.15, due to the more then 20 percent error values at depth 6. In this solution A facet was considered to be inside, if its center is inside of the voxel. However this diagram is done with 10333 rays from each facet and a facet is considered to be inside of a voxel if either one of the corners or the center sit inside of this voxel, thus giving more accurate results.

As one can see from the Figure 4.17 for depth 5 the assumption of center and corner points is giving good results. However if we further move to depth 7, we can observe that the loss of data is huge and thus according to the local
Figure 4.15: Calculation times and Error vs. Depth

Error analysis around 50 percent, see Figure 4.16. We can also emphasize that after depth 7, the combined model started to show its influence on calculation time. And according to that, we can fortune as an outlook about combined model that, if another geometry is solved (having less element size/volume dimensions ratio) for higher depths the combined model would give much more better results then the uniform one.
Bibliography


