JAGIELLONIAN UNIVERSITY
IN KRAKÓW

FACULTY OF PHYSICS, ASTRONOMY AND APPLIED COMPUTER SCIENCE

Sławomir Konrad Tadeja
album number: 1019735

Application of 3D computer graphics techniques used in video games for Monte Carlo calculations of multi-particle transport code

Magister's Thesis
Major: Applied Computer Science (general)

Supervised by
Prof. dr hab. Paweł Moskal
Marian Smoluchowski Institute of Physics
Nuclear Physics Division

Kraków, 2016

version corrected on 18.07.2016 and 26.02.2018

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia   Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia   Podpis kierującego pracą

Podziękowania

Chciałbym wyrazić najgłębsze podziękowania wobec mojego promotora prof. dr-a hab. Pawła Moskala, który otworzył przede mną drzwi do świata nauki. Dziekuje mu również za jego cierpliwośc oraz bycie moim przewodnikiem w trakcie pisania tej pracy. Jestem bardzo wdzięcznym moim kolegom: dr. Michałowi Silarskiemu, mgr Dominice Hunik oraz mgr. Michałowi Smolisowi, z którymi miałem przyjemność pracować przy projekcie SABAT - ta praca nie mogłaby powstać bez Was. Niektóre części tej pracy zostały napisane podczas mojego stażu w Europejskiej Agencji Kosmicznej (ESA). Chciałbym podziękowac mojemu mentorowi panu Quirienowi Wijnandsowi oraz panu Robertowi Blommestijnowi, za to, że dzięki nim mogłem pracowac w tak stymulującym środowisku. Chciałbym również podziękować paniom mgr Annie Czeluśniak i Ewie Łanoszce z Sekretariatu Dydaktycznego oraz kierownikowi studiów, panu dr. hab. Pawłowi Górze, za wszelką pomoc okazaną mi na przestrzeni lat. Na końcu, chciałbym podziękować moim rodzicom, którzy nieustatnie mnie wspierali w trakcie mojej edukacyjnej przygody. Zawsze okazywali mi wsparcie, nieważne jak błędne podejmowałem decyzje.

Acknowledgements

I would like to express my deepest gratitude toward my supervisor, Prof. Dr Hab. Paweł Moskal for introducing me to the world of science at the academic level. I would also like to thank him for his patients and guidance. I am grateful to my colleagues: Dr Michał Silarski, Mgr Dominika Hunik and Mgr Michał Smolis, with whom I had the pleasure to work in the SABAT Collaboration - this work could not have been accomplished without all of you. Some parts of this work were developed during my traineeship period at the European Space Agency (ESA). I would like to thank my mentor Mr. Quirien Wijnands for his tutorship and Mr. Robert Blommestijn for allowing me to work in such stimulating environment. I would also like to thank Mgr Anna Czeluśniak and Ewa Łanoszka from the Didactic Secretariat and to the course coordinator, dr hab. Paweł Góra, for all the help and enthusiasm that I received from them over the years. Last but not least, I would like to thank my parents, who encourage me throughout my educational endeavours. No matter how wrong I was in my decisions, they always supported me.

"Thank goodness we don't have only serious problems, but ridiculous ones as well."
— E. Dijkstra, *A Letter to My Old Friend Jonathan*

Streszczenie

Materiały niebezpieczne, takie jak: narkotyki, broń biologiczna czy materiały wybuchowe, posiadają bardzo charakterystyczny skład chemiczny. Są zbudowane głównie z węgla, azotu, tlenu i wodoru. Ta specyficzna budowa czyni jest bardzo podatnymi na wykrycie przy użyciu nowej metody detekcji zwanej atometrią, która pozwala na nieinwazyjną analizę chemiczną dowolnej substancji w czasie rzeczywistym. Technologia ta opiera się na badaniu stechiometrycznym danego obiektu przy użyciu wiązek neutronów.

Grupa eksperymentu SABAT (**S**toichiometry **A**nalysis **b**y **A**ctivation **T**echniques), złożona z doświadczonych naukowców i studentów, prowadzi na Uniwersytecie Jagiellońskim badania nad tymi nowatorskimi, bazującymi na atometrii, systemami detekcji zagrożeń chemicznych. Aby wspomóc tę pracę został opracowany i zaimplementowany pakiet symulacyjny, który umożliwia bardzo dokładne modelowanie emisji neutronów, stanowiących podstawę opisanej powyżej metody. Z powodu skomplikowania i znacznej złożoności obliczeniowej rzeczonej symulacji wiele technik optymalizujących zużycie zasobów komputerowych musi zostać wykorzystane. Jedną z możliwości jest zastosowanie technologii zaczerpniętych z dziedziny grafiki komputerowej i gier video m.in *śledzenia promieni* (ang. *ray tracing*) oraz implementacji specjalistycznych struktur danych, takich jak *drzewa k-wymiarowe* (ang. *k-d trees*). W poniższej pracy zostanie omówiona motywacja stojąca za wyborem w/w algorytmów i technologii, jak również ich adaptacja na potrzeby rzeczonego pakietu symulacyjnego.

Abstract

Hazardous materials such us drugs, biological weapons and explosives posses a very distinctive chemical composition. They consist mainly from carbon, nitrogen, oxygen and hydrogen elements. This unique characteristics makes them an ideal target for the novel detection method called atometry, which allows non-invasive, real-time, chemical analysis of any kind of substance. The basis of this technology is stoichiometry analysis of a given object conducted with the neutron beams.

The **S**toichiometry **A**nalysis **b**y **A**ctivation **T**echniques (SABAT) Collaboration is a team of experienced scientists and students from the Jagiellonian University, working on these novel, atometry-based, systems for chemical threat detection. To support this research, a sophisticated Monte Carlo simulation package was developed to model in detail the neutron emissions. Due to the computational complexity of this simulation, plenty of optimization techniques needs to be used. One of them is an application of the technologies borrowed from the field of computer graphics and video games, such as the *ray tracing* or the specialized *acceleration data structures* such as the *k-d trees*. This thesis will describe the motivation behind the selection of this particular algorithms and data structures, and also the direct application of them in the simulation package.

Contents

Chapter 1

Introduction

1.1 Principle of Atometry

Hazardous materials such us drugs (e.g. cocaine: $C_{17}H_{21}NO_4$), biological weapons and explosives (e.g. TNT: $C_7H_5N_3O_6$) posses a very distinctive chemical composition. They consist mainly from carbon, hydrogen, nitrogen and oxygen elements.

substance	**stoichiometric formula**	**ratio C:H:N:O**
Trinitrotoluene (TNT)	$C_7H_5N_3O_6$	1.2 : 0.8 : 0.5 : 1
Hexogen (RDX)	$C_3H_6N_6O_6$	0.5 : 1 : 1 : 1
Nitroglycerine	$C_3H_5N_3O_9$	0.33 : 0.56 : 0.33 : 1
Cocaine	$C_{17}H_{21}NO_4$	4.25 : 5.25 : 0.25 : 1
Heroine	$C_{21}H_{23}NO_5$	4.2 : 4.6 : 0.2 : 1
Amphetamine	$C_9H_{21}NO_4$	4.25 : 5.25 : 0.25 : 1

Table 1.1: Molecular content of few selected hazardous substances (table adopted from [5]).

This unique characteristics makes them an ideal target for the novel detection method called atometry, which allows non-invasive, real-time, chemical analysis of any kind of substance. The basis of this technology is stoichiometry analysis of a given object conducted with the neutron beams. Neutrons penetrating tested substance causes excitation of the atomic nuclei to the higher energy states. These nuclei emit gamma quanta when they return to the ground state. The energy of each quantum is characteristic for the type of the element, thus it can be used to detect relative content of the different elements' atoms in the irradiated substance [3, 6, 9].

1.2 Stoichiometry Analysis by Activation Techniques

The **S**toichiometry **A**nalysis **b**y **A**ctivation **T**echniques (SABAT) Collaboration is a team of experienced scientists and students from the Jagiellonian University, working on these novel, atometry-based, systems for chemical threat detection. To support this research, a sophisticated Monte Carlo simulation package was developed to model in detail the neutron emissions. Modern neutron generators are able to emit about 0.6×10^{11} neutrons per second [8]. During the simulation all of these particles have

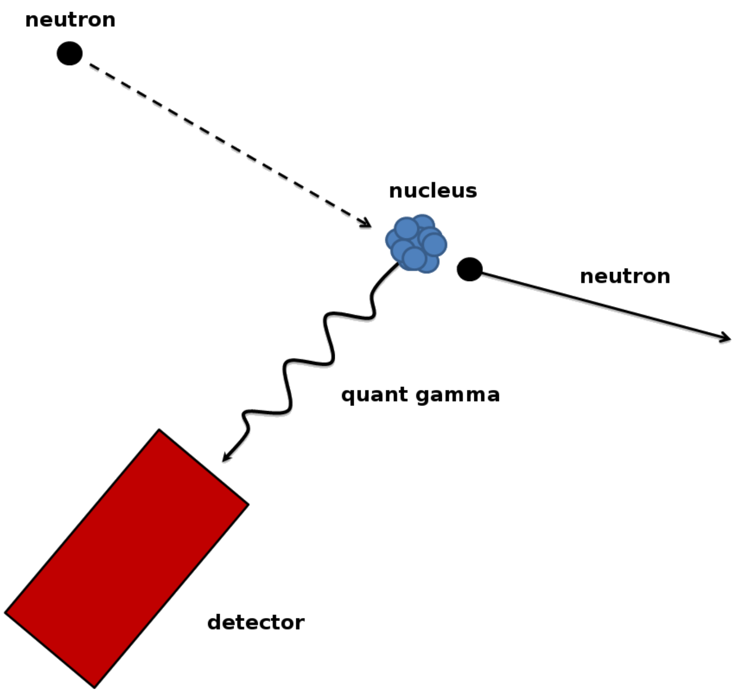


Figure 1.1: Scheme of the neutron interaction with nucleus of a given substance (diagram adopted from [2]).

to be tracked separately. Moreover, neutrons can interact with the matter's nuclei in several processes which may include neutron capture or inelastic and elastic scattering, et cetera [5]. This makes the entire simulation very resource- & time consuming. Furthermore, all the secondary particles such as gamma quanta are also being tracked. Due to computational complexity, plenty of optimization techniques needs to be used. There are mainly two approaches used for the purpose of speeding up the simulation execution time. Both of them have certain trade-offs.

First one relies on the hardware environment in which the simulation is running. Improving it may require additional resources such as the new memory chips or new, more powerful central processing unit (CPU), or even exchanging of the whole hardware system, what can be very costly. The second optimization method can be done at the software level. It can be achieved either with the help of multi-threading technologies or by the minimalization in the number of the needed computations. Multi-threading is currently supported by almost all computer platforms, however it still requires third-party libraries, which makes it platform-dependent. Another software-level solution is an application of the technologies borrowed from the field of computer graphics and video games, such as the *ray tracing* or the specialized *acceleration data structures* such as the *k-d trees*. This thesis describes the motivation behind the selection of this particular algorithms and data structures, and also the direct application of them in the SABAT simulation package.

1.3 Contents of the Chapters

1.3.1 Main Content

The second chapter will briefly describe three most widely used particle transport code packages, including FLUKA, MARS and Geant4. This information is provided to present the difference in terms of the geometry/simulation scene description used by all of them and the one supported by the SABAT package.

The third chapter will very briefly describe basic mathematical concepts, tools and numerical methods, that are needed for understanding of the atometry Monte Carlo simulation. These *building blocks* are later used to develop the simulation 3D scene. Furthermore, an introduction to the algorithms and technologies from the field of computer graphics and video games such as the *polygon meshes* will be given. Third chapter contains also description of the specialized *acceleration data structures*, such as the *k-d trees* and the *bounding boxes*.

The fourth chapter describes the *particle tracking algorithm* and discusses advantages and disadvantages of the two approaches: the *non-flexible solution* and the more sophisticated one, based on the *ray tracing* technique. Moreover, the *Möller–Trumbore intersection algorithm* will be presented as an example of the *ray tracing* implementation.

The fifth chapter will describe the SABAT simulation package including adapted coding style conventions for the purpose to maintain the source code readability. Furthermore, this chapter will include installation instruction of the SABAT framework in the Ubuntu 12.04.5 LTS (Precise Pangolin) operating system.

The sixth chapter will present and briefly discuss some of the results obtained with the SABAT package. An example of the simulation performed for the purpose of underwater threat detection system will be given.

The seventh and final chapter will describe achieved results, findings and conclusions with regards to application of the 3D computer graphics techniques used in the video games for the Monte Carlo calculations of multi-particle transport code. It will also present ideas for further research, that could not be covered due to the scope of this thesis and the time constraints.

1.3.2 Appendix A

Additionally, appendix A contains description of the SQLite database engine used to store the data crucial for the simulation. Furthermore, it outlines the reasons behind the selection of these particular *Relational Database Management System* (RDBMS). Appendix A provides insides to the C++ wrapper that was written for the purpose of speeding up the execution of the database queries, and to ease up dealing with the retrieved results.

Chapter 2

Existing Transport Codes

Quite a few particle transport code packages exists. They were developed throughout the years, mostly within the nuclear physics scientific community. In this chapter a brief description of the three most widely used such packages will be given. Only aspects relevant for this thesis are mentioned.

2.1 Geant4

Geant4 is one of the most widely used, multi-particle transport code packages. Its application spans over fields such as space exploration, high-energy, nuclear and medical physics. Geant4 is able to handle large, complex geometries defined with the help of boolean operations, Constructed Solid Geometry (CSG) and tessellated solids. The package is also capable to deal with "moving objects" [12, 13].

2.2 FLUKA

FLUKA is a Monte Carlo simulation package for calculations of particle trajectories and interactions with matter. FLUKA is able to simulate around 60 different particles. Combinatorial Geometry (CG) is defined by two fundamental concepts: bodies, and regions which are Boolean combination of bodies. Each region has to have the same material compositions. FLUKA Fortran source code is written and maintained by the team of scientists from the European Organization for Nuclear Research (CERN) and from the Istituto Nazionale di Fisica Nucleare (INFN) [10, 11].

2.3 MARS

MARS Code System is another, Monte Carlo based, particle transport code package. Simulated particles are tracked throughout user defined simple or complex geometry. MARS models scene use so called zones which are contiguous array of volumes, that can have any shape. Additionally, MARS recognize geometry description generated by FLUKA. More detail reading can be found in the official manual [14].

Chapter 3

Scene Representation

3.1 Basic 3D Geometry Concepts and Implementation

This chapter will very briefly describe basic mathematical concepts and tools, that are needed for understanding of the SABAT Monte Carlo simulation. These *building blocks* will be later use to develop a model of the objects in physical universe and for modelling of the interactions between them using numerical methods implemented with the help of C++ programming language. More mathematical details can be found in a number of publications on analytical geometry, such as [17]. Some operations such as the calculation of a line-plane intersection or finding of the distance between a point and a plane were implemented using formulas found in references [17, 18, 19].

3.1.1 Point in $\mathbb{R}^3$

Cartesian coordinate system specifies each point uniquely in the $\mathbb{R}^3$ by an ordered triplet of signed numerical coordinates $P = (x, y, z)$. This components are the positions of the perpendicular projections of the point onto the three mutually perpendicular planes.

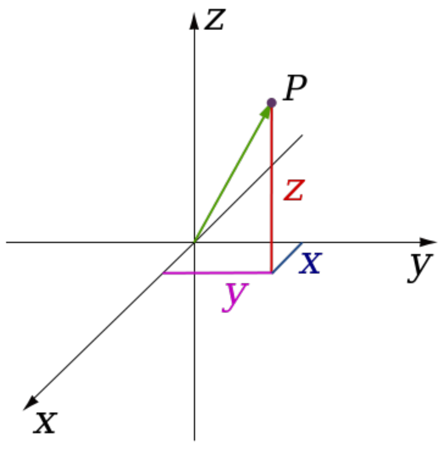


Figure 3.1: Point in $\mathbb{R}^3$ (diagram adapted from [15]).

Given two points $P_1 = (x_1, y_1, z_1)$ and $P_2 = (x_2, y_2, z_2)$, the distance between them can be calculated using Pythagorean Theorem [17]:

$$|P_1P_2| = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{3.1}$$

All the basics operations with regards to the points in $\mathbb{R}^3$ where implemented in the `class Point3D` with an additional operation `Point3D::Translate()` (List. 3.1), which *adds* two points summing up their coordinates accordingly. This method is later on used by *the kd-tree* to find the middle point of all triangles based on theirs centroids. The C++ source code of this class is kept in two files: the `/Geometry3D/geo_point_3d.h` which contains the class declaration and the `/Geometry3D/geo_point_3d.cpp` with its definition.

Listing 3.1: Declaration of the `Point3D::Translate()` method.

```
void Point3D::Translate(Point3D& point) {
  this->x_ += point.x_;
  this->y_ += point.y_;
  this->z_ += point.z_;
}
```

3.1.2 Vector in $\mathbb{R}^3$

An Euclidean vector in $\mathbb{R}^3$, is an ordered triplet of numbers, denoted as a letter with an arrow $\overrightarrow{v} = \langle x, y, z\rangle$ and has direction and length (magnitude). Vector can be thought of as an displacement of point $P_1 = (x_1, y_1, z_1)$ to point $P_2 = (x_2, y_2, z_2)$ and marked as $\overrightarrow{P_1P_2} = \langle (x_2 - x_1), (y_2 - y_1), (z_2 - z_1)\rangle$ where P_1 is an *initial point* and P_2 is a *terminal point*. Vectors can be added and subtracted according to the vector algebra. Length of the vector can be obtained using formula: $\|\overrightarrow{v}\| = \sqrt{x^2 + y^2 + z^2}$. If s is a scalar and $\overrightarrow{v} = \langle x, y, z\rangle$, then $\overrightarrow{v}s = \langle sx, sy, sz\rangle$ and $\frac{\overrightarrow{v}}{s} = \langle \frac{x}{s}, \frac{y}{s}, \frac{z}{s}\rangle$ [17, 19].

The standard basis for $\mathbb{R}^3$ is a set of three specific, so called *unit vectors* - $\overrightarrow{i} = \langle 1, 0, 0\rangle$, $\overrightarrow{j} = \langle 0, 1, 0\rangle$, $\overrightarrow{k} = \langle 0, 0, 1\rangle$ - pointing in the direction of the axes of the Cartesian coordinate system. Any vector in $\mathbb{R}^3$ can be expressed as a linear combination of these three vectors [17].

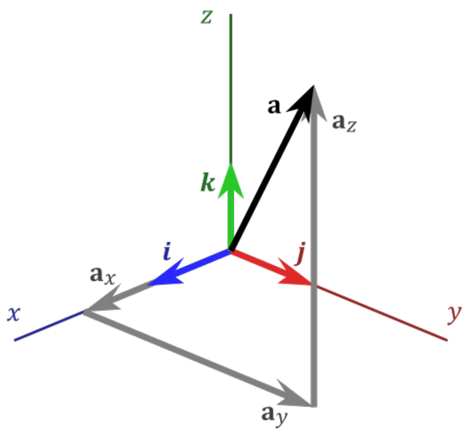


Figure 3.2: The Standard Basis for $\mathbb{R}^3$ (diagram adapted from [16]) under the GFDL).

Additionally we can define two operations on vectors: the *dot product* and the *cross product*. Dot product of two vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ is defined as Eq. 3.2:

$$\overrightarrow{v_1} \cdot \overrightarrow{v_2} = \|\overrightarrow{v_1}\|\|\overrightarrow{v_2}\| \cos\phi = x_1x_2 + y_1y_2 + z_1z_2 \tag{3.2}$$

where ϕ is an angle between the vectors [17]. Cross product of two (linearly independent) vectors $\overrightarrow{v_1} = x_1\overrightarrow{i} + y_1\overrightarrow{j} + z_1\overrightarrow{k}$ and $\overrightarrow{v_2} = x_2\overrightarrow{i} + y_2\overrightarrow{j} + z_2\overrightarrow{k}$ is a vector perpendicular to both of them defined as [17]:

$$\begin{aligned}\overrightarrow{v_1} \times \overrightarrow{v_2} = \|\overrightarrow{v_1}\|\|\overrightarrow{v_2}\| \sin(\phi)\,\overrightarrow{n} &= (y_1z_2 - z_1y_2)\overrightarrow{i} + (x_1z_2 - z_1x_2)\overrightarrow{j} + (x_1y_2 - y_1x_2)\overrightarrow{k} \\ &= \langle (y_1z_2 - z_1y_2), (x_1z_2 - z_1x_2), (x_1y_2 - y_1x_2) \rangle\end{aligned} \tag{3.3}$$

where ϕ is an angle between the vectors and $\overrightarrow{n}$ is a unit vector perpendicular to the plane spanned over the $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ vectors with the direction provided by the right-hand rule. If $\overrightarrow{v_1} \times \overrightarrow{v_2} = 0$, then $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are parallel to each other. All the basics operations with regards to the vectors in $\mathbb{R}^3$ where implemented in the `class Vector3D`. The C++ source code of this class is kept in two files: the `/Geometry3D/geo_vector_3d.h` which contains the class declaration and the `/Geometry3D/geo_vector_3d.cpp` with its definition.

3.1.3 Line in $\mathbb{R}^3$

The line in the 3D space can be described with the help of its direction vector and a point lying on the line. The line parallel to the vector $\overrightarrow{v} =< x_v, y_v, z_v >$ that is passing trough the point $P = (x_0, y_0, z_0)$ has a parametric equation [17]:

$$\begin{aligned} x &= x_0 + tx_v \\ y &= y_0 + ty_v \\ z &= z_0 + tz_v \\ -\infty &< t < +\infty \end{aligned} \tag{3.4}$$

$$t = \frac{x - x_o}{x_v} = \frac{y - y_o}{y_v} = \frac{z - z_o}{z_v} \tag{3.5}$$

where the last equation Eq. 3.5 is the symmetric form of Eq. 3.4 and $t\overrightarrow{v}$ describes the distance from the $P = (x_0, y_0, z_0)$ [17]. In case of the line passing trough two points $P_0 = (x_0, y_0, z_0)$ and $P_1 = (x_1, y_1, z_1)$ the parametric equations looks as follows [19]:

$$\begin{aligned} x &= x_0 + t(x_1 - x_0) \\ y &= y_0 + t(y_1 - y_0) \\ z &= z_0 + t(z_1 - z_0) \\ -\infty &< t < +\infty \end{aligned} \tag{3.6}$$

$$\overrightarrow{v} =< v_x, v_y, v_z >=< x_1 - x_0, y_1 - y_0, z_1 - z_0 > \tag{3.7}$$

The above equations were implemented in the class `Line3D`. The C++ source code of this class is kept in two files: the `/Geometry3D/geo_line_3d.h` which contains the class declaration and the `/Geometry3D/geo_line_3d.cpp` with contains its definition.

3.1.4 Plane in $\mathbb{R}^3$

The plane in $\mathbb{R}^3$ with its normal vector $\overrightarrow{n} =< A, B, C >$ has a following general form:

$$Ax + By + Cz + D = 0. \tag{3.8}$$

The plane can be also described with the help of the three non-collinear points $P_0 = (x_0, y_0, z_0)$, $P_1 = (x_1, y_1, z_1)$, $P_2 = (x_2, y_2, z_2)$ lying on it [17]:

$$\begin{aligned} \overrightarrow{r} &= \overrightarrow{P1P0} =< x_1 - x_0, y_1 - y_0, z_1 - z_0 > \\ \overrightarrow{s} &= \overrightarrow{P2P0} =< x_2 - x_0, y_2 - y_0, z_2 - z_0 > \\ \overrightarrow{n} &= \overrightarrow{r} \times \overrightarrow{s} =< x_n, y_n, z_n >=< A, B, C > \\ D &= -(Ax_0 + By_0 + Cz_0) \end{aligned} \tag{3.9}$$

or in a similar way with the help of three vectors $\overrightarrow{v_0} =< x_0, y_0, z_0 >$, $\overrightarrow{v_1} =< x_1, y_1, z_1 >$, $\overrightarrow{v_2} =< x_2, y_2, z_2 >$ using a following equations [17]:

$$\begin{aligned} \overrightarrow{r} &= \overrightarrow{v_1} - \overrightarrow{v_0} =< x_1 - x_0, y_1 - y_0, z_1 - z_0 > \\ \overrightarrow{s} &= \overrightarrow{v_2} - \overrightarrow{v_0} =< x_2 - x_0, y_2 - y_0, z_2 - z_0 > \\ \overrightarrow{n} &= \overrightarrow{r} \times \overrightarrow{s} =< x_n, y_n, z_n >=< A, B, C > \\ D &= -(Ax_0 + By_0 + Cz_0) \end{aligned} \tag{3.10}$$

To make the values of the $\overrightarrow{n}$ components smaller the vector can be normalized using the Eq. 3.11:

$$\frac{\overrightarrow{n}}{\|\overrightarrow{n}\|} = \frac{< x_n, y_n, z_n >}{\sqrt{x_n^2 + y_n^2 + z_n^2}} = \frac{< A, B, C >}{\sqrt{A^2 + B^2 + C^2}} \tag{3.11}$$

The distance l between any given point $P = (x, y, z)$ and the plane described as $Ax + By + Cz + D = 0$ can be calculated with the help of Eq. 3.12 [18]. The point lies on the plane $\Leftrightarrow l = 0$.

$$l = \frac{|Ax + By + Cz + D|}{\sqrt{A^2 + B^2 + C^2}} \tag{3.12}$$

The point $P_i = (x_i, y_i, z_i)$ of intersection between a line with the direction vector $\overrightarrow{v} =< x_v, y_v, z_v >$ and passing trough point $P_0 = (x_0, y_0, z_0)$ and a plane with the normal vector $\overrightarrow{n} =< x_n, y_n, z_n >$ can be calculated as follows [18, 19]:

$$t = \frac{D - Ax_0 - By_0 - Cz_0}{\overrightarrow{n} \cdot \overrightarrow{v}} = \frac{D - Ax_0 - By_0 - Cz_0}{Ax_v + By_v + Cz_v} \tag{3.13}$$

$$\begin{aligned} x_i &= x_0 + tx_v \\ y_i &= y_0 + ty_v \\ z_i &= z_0 + tz_v \end{aligned} \tag{3.14}$$

The correctness of the calculations can be verified by checking if the distance l between the point $P_i = (x_i, y_i, z_i)$ and the plane (Eq. 3.12) is equal to zero $l = 0$ or by checking if the dot product of the line direction vector and the plane normal is different than zero $\overrightarrow{v} \cdot \overrightarrow{n} \neq 0$ [19].

The above equations were implemented in the class `Plane3D`. The C++ source code of this class is kept in two files: the `/Geometry3D/geo_plane_3d.h` which contains the class declaration and in the file `/Geometry3D/geo_plane_3d.cpp` with contains its definition.

3.2 Polygon Meshes

3.2.1 Convex Polygon

A convex polygon is a two-dimensional, not self-intersecting figure in which all of the interior angles are $\leqslant 180^\circ$. Every such polygon can be triangulated, which means it can be divided into a set of non-overlapping triangles [25]. In the case of the SABAT package the triangulation is done automatically when the description of the 3D objects from which the simulation scene is constructed of are read from the XML file.

3.2.2 Triangle Mesh

One of the simplest example of the 3D polygon meshes is the mesh that uses triangles to describe the faces (i.e. triangle mesh).

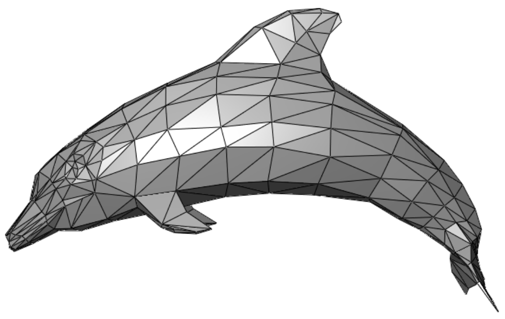


Figure 3.3: An example of a triangle mesh (adapted from [22]).

Due to the small number of possible 3D objects that can be used in the SABAT package, there can be at most three triangles meeting in a single vertex. Also, no special data structure for the mesh storage was implemented. The informations are kept on the different description layers. For instance, the 3D object (e.g. `Cuboid3D`) can be build from a six rectangular faces (`Rectangle3D`). Each of these rectangles is constructed from two non-overlapping triangles $\triangle_{ABC}$ and $\triangle_{CDA}$ (`Triangle3D`). Fig. 3.4 presents the typical elements from which the mesh can be build from. Other example can be found in Fig. 4.5. Furthermore, the 3D plots of the generated meshes used during the simulations can be found in Fig. 3.9 and in Fig. 3.10.

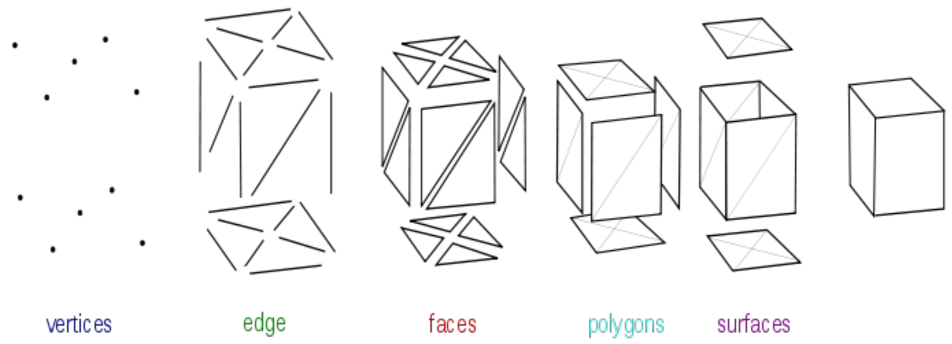


Figure 3.4: Mesh modelling overview (adapted from [21] under the WP:CC BY-SA).

3.3 Constructive Solid Geometry

In the CSG a more complex object or a surface is build with the help of a set of basic 3D objects and shapes such as, for instance, cubids, spheres, tubes or cons that are joined together using Boolean logical operators for sets: intersection, union, difference [25]. This method is used in the FLUKA software [10].

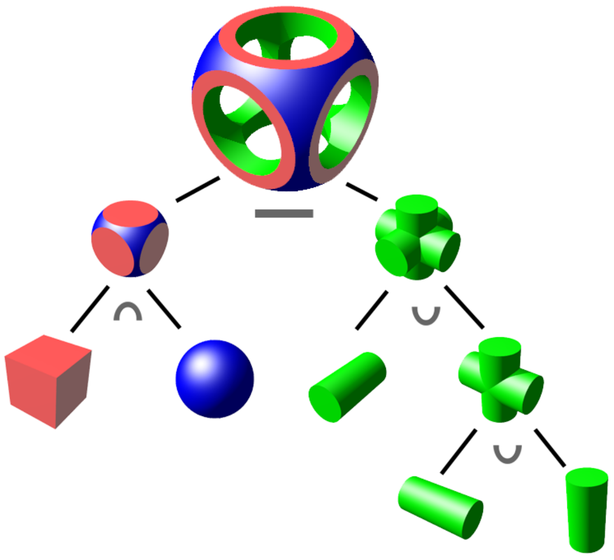


Figure 3.5: Binary tree of a CSG object constructed with the help of two union $\cup$, single intersection $\cap$ and a single difference $-$ operations (adapted from [20] under the WP:CC BY-SA).

3.4 Space Partitioning Data Structures

There are two main data structures used in the scene partitioning algorithms: *kd-trees* (short for the *k-dimensional tree*) and the *bounding volumes*. In this chapter, a short description of both of them will be given. If the space partitioning data structure is correctly balanced, the time complexity can be on average logarithmic $O(log_2(n))$ and $O(\sqrt[3]{n})$ in the worst case scenario, where n is the number of rays [25]. The SABAT package uses a mixture of both structures listed above.

3.4.1 Bounding Volumes

A *bounding volume* for a collection of 3D objects is another object containing the entire collection completely within itself. If the ray cannot intersect with the *bounding volume* it is also not possible for the ray to interact with any of the items surrounded by it [25]. The *bounding volume* can take any shape such as, for example, a pyramid or cone. However, due to the computational cost associated with the computing of it and the necessary update calculations in case the object surrounded by the *bounding volume* changes or moves, the most widely used are shapes of axis-aligned bounding box (cuboid) and sphere [25]. Fig. 3.6 shows a *bounding box* containing a single triangle with its centroid C given by Eq. 3.15:

$$C = (\frac{\sum_{i=1}^{3} x_i}{3}, \frac{\sum_{i=1}^{3} y_i}{3}, \frac{\sum_{i=1}^{3} z_i}{3}) \tag{3.15}$$

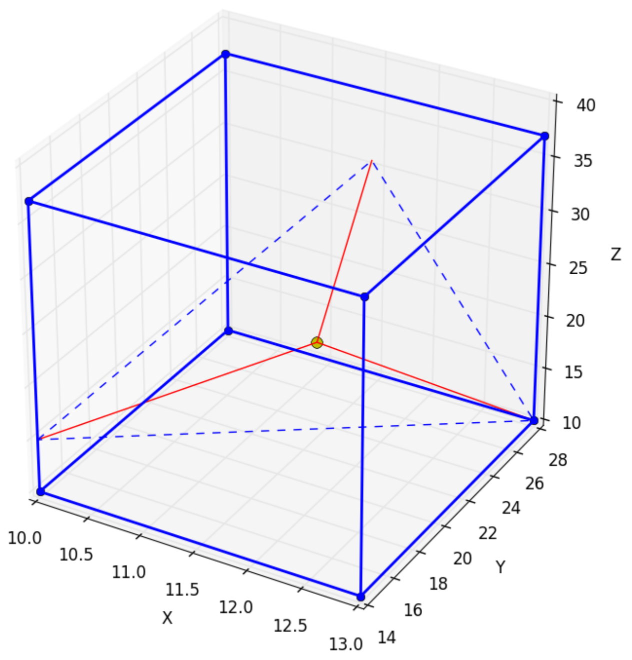


Figure 3.6: Rectangular cuboid (hexahedron) as a bounding box with all facets parallel to the origin axes surrounding a triangle with its centroid.

3.4.2 K-D Tree

A *k-d tree*, or in case of the SABAT package *3-d tree* since the simulation take place in the $\mathbb{R}^3$, is a binary search tree of which every node cuts space into two subspaces using a plane parallel to one of the origin axes [25]. The computational cost of generating the tree is high; insert operation can have linear time complexity for the worst case scenario $O(n)$ with an average $O(log\ n)$. This yields $O(n\ log(n))$ on average for the construction of the tree. However, the scene used by the SABAT software is static, thus the tree has to be build only once at the beginning of the simulation. There is a few ways for selection of the splitting position. Implemented method is based on the *spatial median splitting*. The subdivision of the space stops when certain threshold is reached by the number of triangles in the single node [25]. The implementation used in the SABAT is largely based on the code from [26] and it was developed during the author's traineeship period with the ESA.

The particle tracking with the *k-d tree* will not be efficient for the scene with many objects contained completely within other object. For instance, in the case of the `scene/A2.xml` geometry shown in Fig. 3.9 all the detector parts and the irradiated substance are immersed in the biggest object containing sea water (the black hexahedron). The tree nodes contain many overlapping triangles, and since we need to calculate points *in* & *out* (i.e. the points in which the particle would enter and leave the object if it would go straight trough it without any physical interaction), intersections with all of the remaining triangles that belong to a single figure and can be stored in a different leafs have to be checked either way. Thus the use of the *k-d tree* is not beneficial, especially since the building of the tree requires some initial computation in the first place. Tables Tab. 3.1 and Tab. 3.3 shows overall times of the package execution as a function of the number of initially emitted neutrons for the geometries `scene/G2.xml` and `scene/H2.xml`, respectively. Assuming that the measured time follows the Poisson probability distribution the average time needed to simulate a single neutron can be found in Tab. 3.2 and Tab. 3.4 for the scenes `scene/G2.xml` and `scene/H2.xml`, respectively. The constant time required to build the data structure has no direct impact on the comparison since the tree was build in all of the cases, but in some of them it was not used in the tracking.

sample size	**100**	**1000**	**10000**
kd-tree	46.66 $[s]$	454.29 $[s]$	4283.98 $[s]$
without kd-tree	47.35 $[s]$	444.27 $[s]$	4364.31 $[s]$

Table 3.1: Overall execution times with and without the *k-d tree* used for tracking (`scene/G2.xml`).

sample size	**100**	**1000**	**10000**
kd-tree	0.47 ± 0.07 $[s]$	0.45 ± 0.02 $[s]$	0.43 ± 0.01 $[s]$
without kd-tree	0.47 ± 0.07 $[s]$	0.44 ± 0.02 $[s]$	0.44 ± 0.01 $[s]$

Table 3.2: Average time of a single neutron simulation with and without the *k-d tree* (`scene/G2.xml`).

sample size	**100**	**1000**	**10000**
kd-tree	1.58 $[s]$	14.96 $[s]$	136.08 $[s]$
without kd-tree	5.02 $[s]$	33.03 $[s]$	320.48 $[s]$

Table 3.3: Overall execution times with and without the *k-d tree* used for tracking (`scene/H2.xml`).

sample size	**100**	**1000**	**10000**
kd-tree	0.02 ± 0.01 $[s]$	0.15 ± 0.01 $[s]$	0.14 ± 0.01 $[s]$
without kd-tree	0.05 ± 0.02 $[s]$	0.33 ± 0.01 $[s]$	0.32 ± 0.01 $[s]$

Table 3.4: Average time of a single neutron simulation with and without the *k-d tree* (`scene/H2.xml`).

If the constant time needed to build the tree for a given scene is neglected, than the average time needed to simulate a single neutron is almost the same for tracking algorithm with and without the *k-d tree* implemented. Fig. 3.7 shows the geometry of the `scene/G2.xml` scene and Fig. 3.8 shows the `scene/H2.xml` scene.

Tables 3.3 and 3.4 contains data generated with the scene `scene/H2.xml` which is similar to the `scene/G2.xml`. The only difference is that the black cuboid containing sea water was removed thus the vacuum space was created and the remaining objects are no longer contained within another object. This lead to a different depth and split of triangles in the *k-d tree* (i.e. removal of triangles belonging to the object with whom most of the particle's trajectories would intersect with) which in turn cut the average execution time roughly by half. The statistical uncertainty is very high for the small sample size ($n = 100$), therefore there are high fluctuations in the first columns of all the tables. Unfortunately, this type of the scene is not used in the simulations performed for the SABAT experiment where the background due to the neutron interactions with water has to be carefully studied. However, the *k-d tree* can be used for the purpose of geometry testing and in the experiments where the background can be neglected.

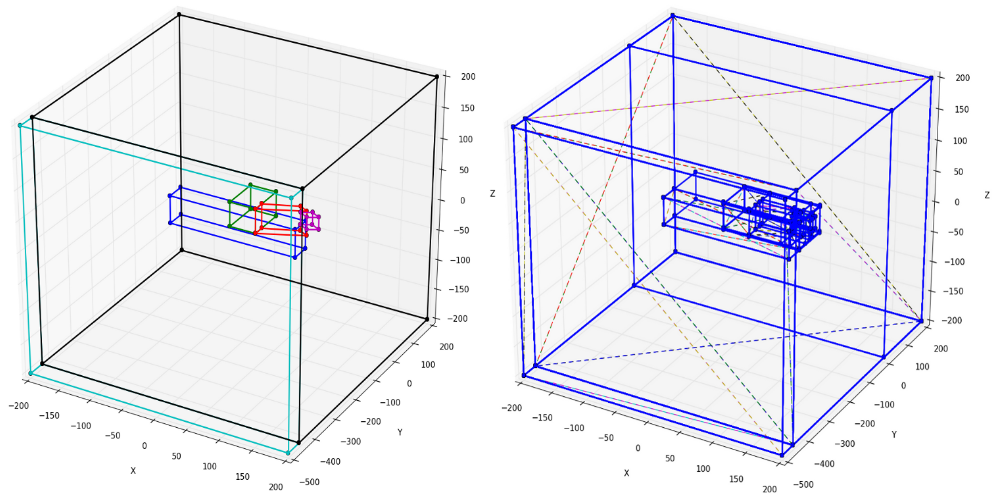


Figure 3.7: The left hand figure shows the `scene/G2.xml` geometry in which the detector contour is magenta, the neutron guide is green, the γ guide is red, the hazardous substance (sulphur mustard) is blue, the sea bottom is cyan and the whole setup is immersed in the sea water marked with the black contour. The figure of the right presents the *k-d tree* build on top of this scene's triangle mesh with the blue *bounding boxes* (i.e. subspaces). Both of the figures were generated with the `geo_check.py` script presented in the Sec. 3.6.

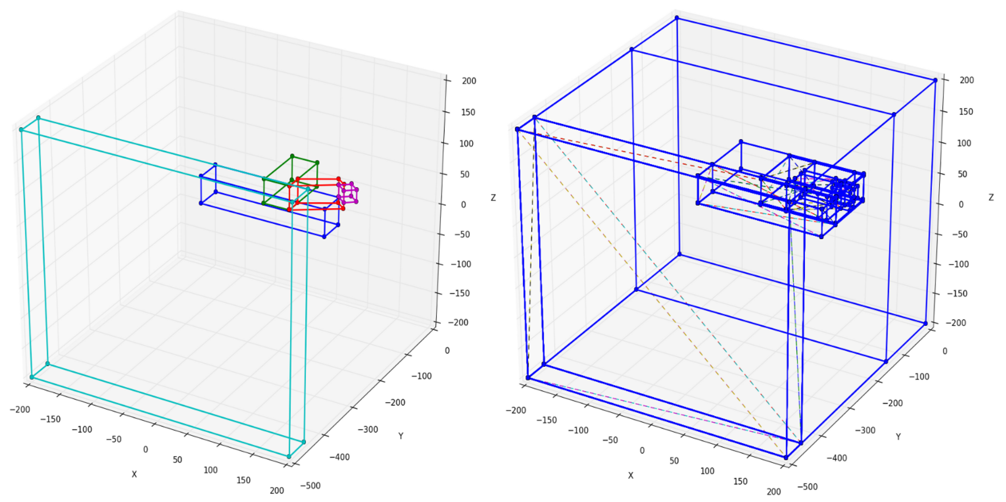


Figure 3.8: The left hand figure shows the `scene/H2.xml` geometry in which the detector contour is magenta, the neutron guide is green, the γ guide is red, the hazardous substance (sulphur mustard) is blue and the sea bottom is cyan. The only difference between this scene and the `scene/G2.xml` is the sea water removal. The figure of the right presents the *k-d tree* build on top of this scene's triangle mesh with the blue *bounding boxes* (i.e. subspaces). Both of the figures were generated with the `geo_check.py` script presented in the Sec. 3.6.

3.5 Geometry Input File

In the earliest versions of the SABAT package, the geometry description has to be hard-coded into the C++ source code. In the current software version (June 2016), the scene has to be defined in an input file. The XML structure (i.e. tags) of this file and the portion of the code responsible for the geometry readout and parsing was written by Michał Smolis. The details of the scene description XML will be discussed to give a holistic view of the whole geometry handling within the SABAT software. List. 3.2 show partial description of the possible scene kept in the `scene/A2.xml` file. Some parts were removed for clarity, however, every object within the scene is declared in exactly the same way, thus all the other 3D shapes can be defined accordingly.

Listing 3.2: Part of the geometry description XML file: `scene/A2.xml`.

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <PARAMETERS>
4    <NEUTRON_BUNCH_COUNT>1</NEUTRON_BUNCH_COUNT>
5    <NEUTRON_BUNCH_SIZE>100</NEUTRON_BUNCH_SIZE>
6    <SCENE> <!-- scene A2 -->
7      <!-- sulfur mustard -->
8      <OBJECT detector="0">
9        <SHAPE sorted="0">
10        -97, -150,  22.5,  97, -150,  22.5,
11         97, -100,  22.5, -97, -100,  22.5,
12        -97, -150, -22.5,  97, -150, -22.5,
13         97, -100, -22.5, -97, -100, -22.5
14       </SHAPE>
15       <SUBSTANCE density="1.2" type="MOST_ABUNDANT">
16         <COMPOUND formula="(ClCH2CH2)2S" fraction="1"/>
17       </SUBSTANCE>
18     </OBJECT>
19
20     <!-- edited -->
21
22     <!-- detector air tube -->
23     <OBJECT detector="0">
24       <SHAPE sorted="1">
25         20, -100, -20,  20, -100, 20,
26         83,  -49, -20,  83,  -49, 20,
27         20,  -70, -20,  20,  -70, 20,
28         69,  -35, -20,  69,  -35, 20
29       </SHAPE>
30       <SUBSTANCE density="0.0012" type="MOST_ABUNDANT">
31         <COMPOUND formula="N2" fraction="0.7547"/>
32         <COMPOUND formula="O2" fraction="0.232"/>
33       </SUBSTANCE>
34     </OBJECT>
35
36     <!-- edited -->
37
38   </SCENE>
39 </PARAMETERS>
```

The description of the XML input file will be given with regards to the above listing (List. 3.2). First, all of the scene description has to be contained between the overall `<PARAMETERS></PARAMETERS>` tags. The tags from the lines 4 and 5 specifies the number of neutrons emitted from the source and they have to be provided as an integer numbers. If these values are not given the simulation run is going to fail with an error message. They can be placed everywhere within the parameters, but it is worth to note that the XML structure is strictly hierarchical, so, for instance, they cannot be kept within the `<SCENE></SCENE>` tags which are describing the scene's geometry (i.e. all the 3D objects within the scene). This hierarchy is clearly visible on the XML example.

Each object has to be specified within the `<OBJECT detector="0"></OBJECT>` tags. Attribute `detector` if set to 1 marks the object as a detector, thus it will be treated differently in the simulation. There has to be at least one object specified on the scene or the simulation run is going to fail with an error message. Every object is defined by the two things, its chemical composition and 3D shape/volume. The object's chemical compositions has to be provided between the `<SUBSTANCE></SUBSTANCE>` tags and it is given by the substance density, type and the list of compounds from which the substance is composed of with their overall fraction. The substance has to have at least one compound specified with the `<COMPOUND></COMPOUND>` tags. The `<SHAPE sorted="1"><SHAPE>` tags contains the list of eight vertices of the only available 3D shape (e.g. Fig. 4.5). The software is able to automatically sort the vertices using the *selection sort algorithm* ($O(n^2)$) if they are constructing cuboid that has all of its faces axis-aligned. If the objects has a different shape, the attribute `sorted` should be marked as 1 and the vertices should be listed in the order given below. Each of the four points should be a face (quadrilateral) of the object. Fig. 4.3 and Fig. 4.5 give the correct vertices number labelling.

1	3	7	5
0	2	6	4
0	1	5	4
3	2	6	7
5	7	6	4
1	3	2	0

Once the changes were made to the XML file, the SABAT source code has to be re-compiled and re-run on one or multiple processes. Details of how to do it can be found at Subsec. 5.2.2.

3.6 Geometry Visual Validation Tool

The simulation scene used by the SABAT software can be build from many 3D objects such as cubes (Fig. 4.3) or truncated pyramids (Fig. 4.5). Furthermore, the package does not have any GUI so each of the objects has to be specified independently with all its vertices listed in the XML input file. The scene is also static i.e. it cannot be altered during the simulation execution. The latter allows to visually inspect the scene description prior to the complex, time consuming software run. Thus, the visual validation can potentially save a lot of time and computing power spent on the simulation that would most likely produce wrong results. For these reasons a Python script was written. This script uses Matplotlib library [27] to generate an interactive 3D environment in which the 3D object can be rotated with the help of the computer mouse. Moreover, the plot can be exported to a number of formats such as PNG, JPEG or SVG. An example of the output obtained with this script can be found in Fig. 3.9. The first version of this software was implemented during author's stay at the ESA.

The script invoked without parameters will plot the generated mesh, hence the simulation has to be executed first to generate the script input file `mesh.txt` with the triangle mesh description.

Listing 3.3: Triangle mesh plotting.

```
$ python geo_check.py
```

To visualize the scene geometry, the script has to be executed with the additional command line arguments: a flag `--geo` and a path to the XML file with the geometry description.

Listing 3.4: Geometry visualization.

```
$ python geo_check.py --geo scenes/SCENE_FILE.xml
```

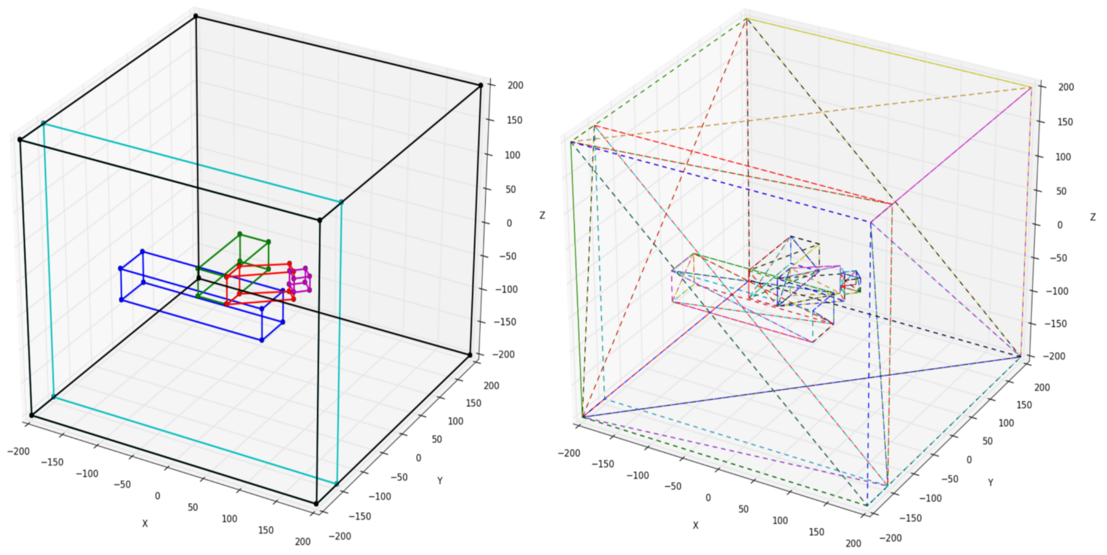


Figure 3.9: The figure on the left side shows the scene geometry extracted directly from the XML file `scenes/A2.xml`. On the right hand side is the triangle mesh generated at the beginning of the simulation. Both of the figures were generated with the `geo_check.py` script.

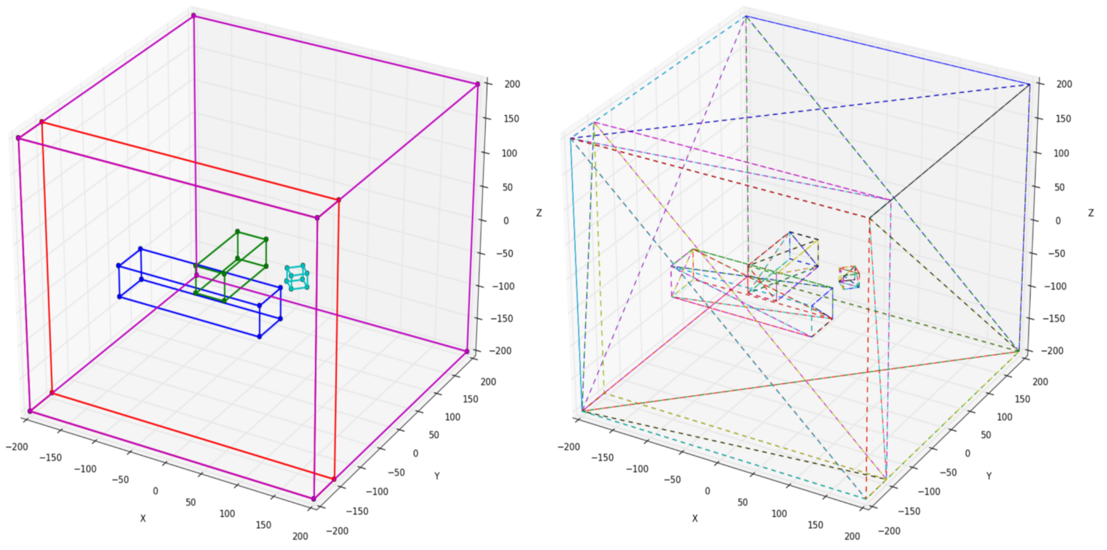


Figure 3.10: The figure on the left side shows the scene geometry extracted directly from the XML file `scenes/C2.xml`, which is the version of the geometry shown on the previous Fig. 3.9 without the γ quanta guides. On the right hand side is the triangle mesh generated at the beginning of the simulation. Both of the figures were generated with the `geo_check.py` script.

The output data generated with the SABAT executable (`./Atometry`) is initially stored in two temporary files Subsec. 5.2.3. This data includes entrances with the position (e.g. point in $\mathbb{R}^3$) in which the particle was generated (via the emitter in case of neutron or in the reaction in case of γ quanta) or place where the given particle changed its trajectory due to the certain process (i.e. in elastic or inelastic scattering). Since the particles are distinguishable by their ID, once the data is aggregated (Subsec. 5.2.3) the entire particle flying path can be tracked and plotted in the form of the sharply curve. Next listing List. 3.5 shows the arguments that has to be passed to the script in order to plot the total flying path of the selected particle within the given geometry.

Listing 3.5: Visualization of the particle flying path.

```
$ python geo_check.py --geo scenes/SCENE_FILE.xml --track PARTICLE_ID
```

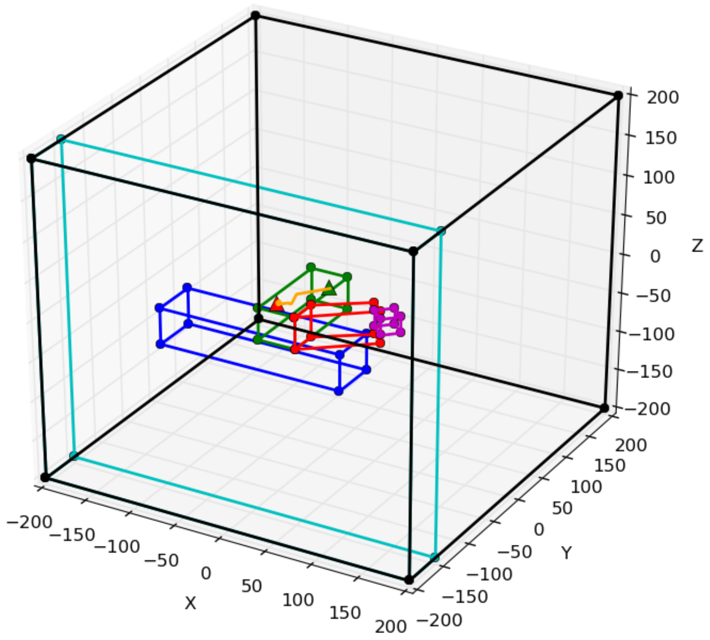


Figure 3.11: The thick orange curved line is a trajectory of the neutron passing through the `scene/A2.xml` scene. As it was expected, neutron path was quickly ended in the sea water. Every change in direction marks the place of the particle interaction with matter. The small green triangle marks the place of neutron emission and the red one the point of its disappearance (e.g. if the particle's energy falls down below certain threshold) or where neutron was captured by some nucleus (Fig. 4.1).

To print in terminal the list of all the possible execution arguments, the script has to be executed with the option `--help` or in short version `-h`.

Listing 3.6: Visualization script output.

```
$ python geo_check.py --help
Script plots the geometry directly from the XML file,
or the mesh from the mesh.txt file generated during the
Atometry  execution, or plots trajectory of a particle
within the given geometry (XML file).
Script requires:
  Python 2.7.x, (2, 7, x)
  Matplotlib, version 1.1.1rc
  NumPy, version 1.8.0

Usage: python geo_check.py [option] [long option]
```

```
Option:
       -h
       -u
       -m
Long option:
       --help
       --usage
       --man
       --geo [path/SCENE.xml]
       --geo [path/SCENE.xml] --track [particle_id]
       --list
```

The script also allows to find the particles which took part in the highest/lowest number of reactions by using an additional command `--list`.

Listing 3.7: Visualization script output.

```
$ python geo_check.py --list
gamma particles:
        [          ID] [   entrances]
maximum [        1783] [          22]
minimum [        5035] [           1]
neutron particles:
        [          ID] [   entrances]
maximum [        3111] [          53]
minimum [        3077] [           2]
```

Chapter 4

Particle Tracking

3D particle tracking is the calculation of the trajectory of a single particle within certain 3D scene. This scene is build from a solid objects and a space between them. Every object and the space possess certain volume, shape and chemical composition (or contains vacuum). In this description, subatomic particle is essentially, a mathematical point in 3D Cartesian space (x, y, z) that have a certain characteristic i.e. type (e.g. photon, electron), position, mass, energy and momentum.

To find the point within the object where the reaction occurred (e.g. the neutron interaction with matter) or if at all, the three key elements are required:

1. The particle's trajectory within the given object.
   (a) The point in which the particle would enter the object (point *in*).
   (b) The point in which the particle would leave the object if it would go straight trough it without any physical interaction (point *out*).
   (c) The line determined by the two points *in* & *out* used in the calculations of the possible reaction point.
2. The physical characteristic of the particle (e.g. type, position, mass, energy, momentum etc.) with its current position within the simulation scene (x, y, z).
3. The chemical composition (compound) of which the object was made from with its chemical characteristic used in the calculations of the possible reaction point.

All the steps presented above are written with assumption that it was determined in the earlier simulation stage, that the particle will hit the given object. The description of the two latter points is not within the scope of this thesis, thus it is not going to cover more that it is absolutely necessarily. Their details can be found in the publications [6] and in Fig. 4.1 and Fig. 4.2 on the next page.

4.1 Non-Flexible Solution

Every object in the scene is modelled as a 3D primitive (solid) object. No two objects can occupy the same space. The 3D scene of the simulation is static, which means that *the geometry* and *the beam center point* has to be defined before the simulations starts. Also the information regarding compound from which the object is build has to be predefined and available in the database. In the current version of the software (June 2016) the object and scene description has to be provided in the form of the XML file.

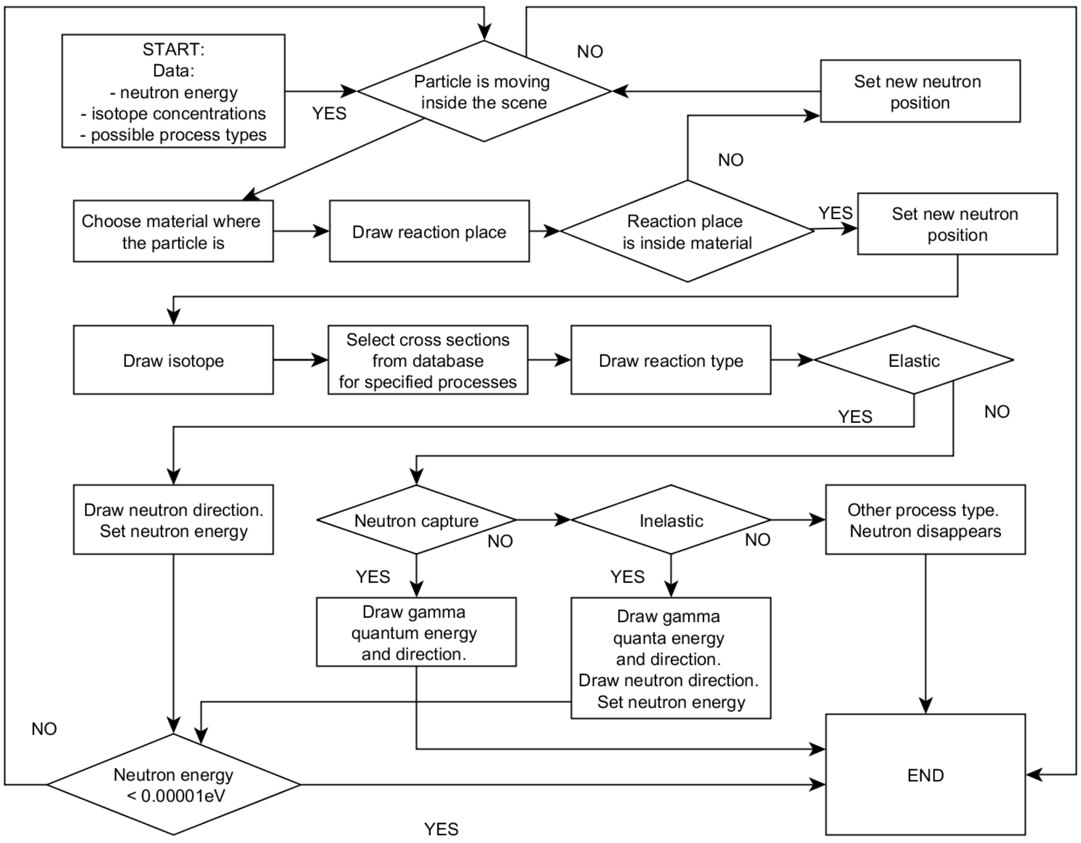

START:
Data:
- neutron energy
- isotope concentrations
- possible process types
YES
Particle is moving inside the scene
NO
Set new neutron position
NO
Choose material where the particle is
Draw reaction place
Reaction place is inside material
YES
Set new neutron position
Draw isotope
Select cross sections from database for specified processes
Draw reaction type
Elastic
YES
NO
Draw neutron direction. Set neutron energy
Neutron capture
NO
Inelastic
NO
Other process type. Neutron disappears
YES
Draw gamma quantum energy and direction.
YES
Draw gamma quanta energy and direction. Draw neutron direction. Set neutron energy
NO
Neutron energy < 0.00001eV
END
YES

Figure 4.1: Neutron tracking algorithm used (diagram adapted from [6]).

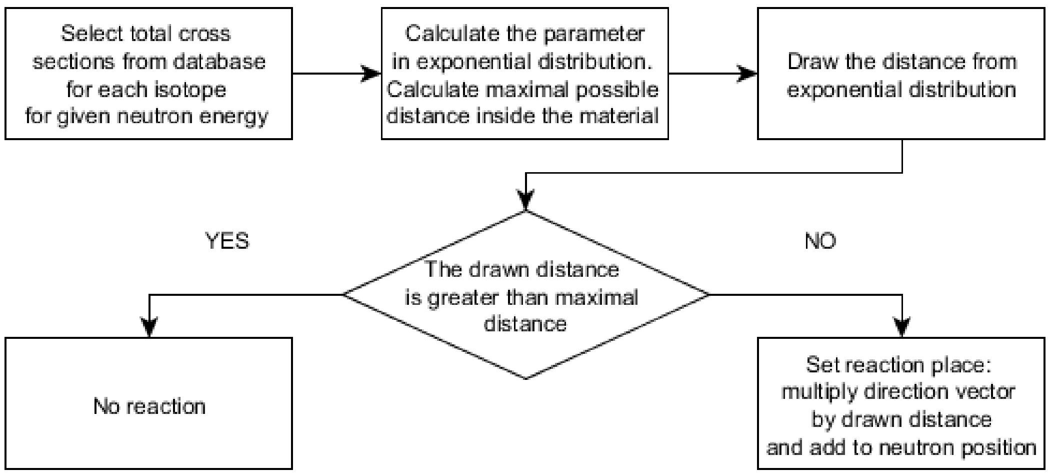

Select total cross sections from database for each isotope for given neutron energy
Calculate the parameter in exponential distribution. Calculate maximal possible distance inside the material
Draw the distance from exponential distribution
YES
The drawn distance is greater than maximal distance
NO
No reaction
Set reaction place: multiply direction vector by drawn distance and add to neutron position

Figure 4.2: Neutron reaction place (diagram adapted from [6]).

In the first implementation of the SABAT package only one, single type of the primitive was available: the cuboid (hexahedron) implemented in the class `Cuboid3D`. The C++ source code of this class is kept in two files: the `/Geometry3D/geo_cuboid_3d.h` which contains the class declaration and the `/Geometry3D/geo_cuboid_3d.cpp` with its definition. All the faces of the cuboid had to be parallel to the coordinate axes. Each of such an objects is uniquely determined by its vertices (3D points), so only the information about them is stored in the computer memory. These vertices should be given in correct order, so the program can determine which ones belongs to the correct faces. Fig. 4.3 shows the correct ordering in which the vertices should be listed in the XML config file. These vertices are also used to determine the equations of the planes on which the cuboid sides lies upon.

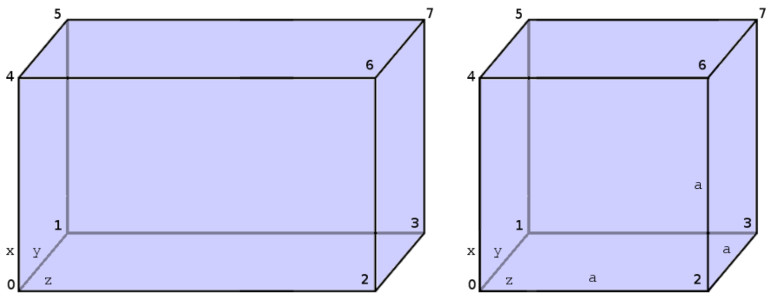


Figure 4.3: An example of the cuboid (left side) and the cube (right side) with the required vertices ordering listed (adapted from [23]).

The points of intersection between the given line (ray) and the cuboid was calculated in the two simple steps:

1. Check if the line is intersecting with any of the six planes given by the faces of the cuboid.
   (a) Iterate over all the planes.
   (b) If the line intersects with the plane calculate the point of intersection.
2. Check if the point lies inside of the rectangle given by the subset of coordinates of its vertices.

In the second step, since all the rectangle's edges are parallel to the origin axes, the check if the point is inside 2D figure becomes trivial. The older method used in the earlier versions of the SABAT package shown in the List. 4.1 is based on a simple coordinate range checks.

Listing 4.1: Method of determination if the point is inside of the rectangle: range check.

```
bool Rectangle3D::IsPointInside(const Point3D& P, const int i) const {
  // i argument is a number of the planes on which the cuboid faces lay upon.
  switch (i) {
    case 0:
    case 1:
      // plane XY
      if(fcmp(P.x_,A_.x_,EPSILON) != -1 && fcmp(P.x_,C_.x_,EPSILON) != 1) {
        if(fcmp(P.y_,A_.y_,EPSILON) != -1 && fcmp(P.y_,C_.y_,EPSILON) != 1) {
          return true;
        } else return false;
      } else return false;
    case 2:
    case 3:
      // plane YZ
      if(fcmp(P.y_,A_.y_,EPSILON) != -1 && fcmp(P.y_,C_.y_,EPSILON) != 1) {
        if(fcmp(P.z_,A_.z_,EPSILON) != -1 && fcmp(P.z_,C_.z_,EPSILON) != 1) {
          return true;
        } else return false;
      } else return false;
    case 4:
    case 5:
      // plane XZ
      if(fcmp(P.x_,A_.x_,EPSILON) != -1 && fcmp(P.x_,C_.x_,EPSILON) != 1) {
        if(fcmp(P.z_,A_.z_,EPSILON) != -1 && fcmp(P.z_,C_.z_,EPSILON) != 1) {
          return true;
        } else return false;
      } else return false;
  }
}
```

It is possible that the method used to determine the points of ray-cuboid intersection described above will return more than just two points of intersection (hits) between the line and the cuboid. If the intersection occurs in the cuboid vertex, there would be a three identical points of intersection calculated since there is at most three planes adjacent in the vertex. If the intersection take place on the cuboid edge, there would be a two identical points of intersection determined. There maybe be more points of intersection found if two vertices or edges would be hit. Some of these special cases may arise due to the calculation precision. There is an eight possibilities in total presented in Tab. 4.1.

#	*hits description*
0	no intersections occurred
1	single intersection with one side of the cuboid (1 point)
2	exactly two intersection points: *in* & *out*
3	a vertex of the cuboid was hit (3 identical points), or an edge was hit (2 identical points) and a one, single plane (1 point)
4	two edges were hit (2 sets of a 2 identical points), or a vertex was hit (3 identical points) and a one, single plane (1 point)
5	a vertex was hit (3 identical points) and a one, single edge was hit (2 identical points)
6	two vertices were hit (2 sets of a 3 identical points)

Table 4.1: The eight possibilities of the line-plane intersection calculation outcome.

In the newest version of the SABAT package distributed with this thesis, the cuboid differs from the original one due to the application of *the ray-tracing* algorithm for the line-object intersection check. The details of this new approach are shown in the next chapter. A similar kind of an object is also used to generate *the bounding boxes* in the Sec. 3.4.1.

4.2 Ray Tracing Algorithm

Ray tracing algorithm is a rendering technique used in 3D computer graphics for simulation of the optical effects. Algorithm casts rays onto the scene through the image pixels and tracks the reflected light [25]. The method is described by the Fig. 4.4. Ray tracing algorithm on itself is very computationally demanding since all the rays have to be traced independently. However, the amount of calculations can be vastly minimalize by combination with other techniques such as the acceleration data structures like the *kd-trees*. Moreover, use of *the parallel and distributed processing* can be greatly beneficial for improving the speed of the calculations [9].

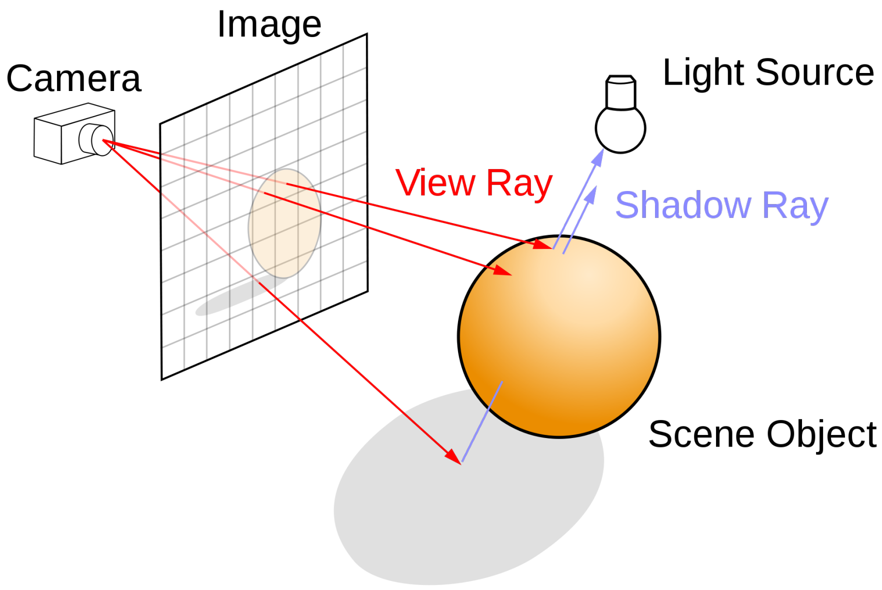


Figure 4.4: The ray tracing algorithm extends rays from the light source into a scene (diagram adapted from [33] under the GFDL).

4.2.1 Möller–Trumbore intersection algorithm

The main goal of using this method are correctness, speed of calculations in the terms of complexity and the flexibility in the 3D object shape. Thus, the parts of the code responsible for the calculation of the ray reflection angle were omitted, since they are not used in later simulation's steps. The original C code of this method can be found here [34]. The implementation shown in the List. 4.2 is heavily based on the code from [35] under the WP:CC BY-SA license. Since in the further calculations, the coordinates of the intersection point are needed, the method seems to be lacking of its most important advantage which was the relatively low complexity of the original method described by Möller and Trumbore [34]. However, the 3D simulation scene remains static, hence the calculations of the plane equation on which the triangle lies are made only once at the beginning of the simulation. One additional step of finding the point in which the ray is crossing the plane is needed (List. 4.2:26). Also, the centroid (Eq. 3.15) of each triangle has to be calculated only once.

Listing 4.2: Method `Triangle3D::IsIntersecting()`.

```
1  bool Triangle3D::IsIntersecting(const Line3D& line,
2                                  Point3D& intersec) const {
3
4    Vector3D P = (line.direction_vector_).CrossProduct(vecCB_);
5    ATDouble det = P.DotProduct(vecAB_);
6
7    if (fcmp(det, 0.0, EPSILON) == 0) return false;
8    ATDouble invDet = 1 / det;
9
10   Vector3D T(line.P_, B_);
11   ATDouble u = T.DotProduct(P) * invDet;
12   // The intersection lies outside of the triangle.
13   if (fcmp(u, 0.0, EPSILON) == -1 ||
14      fcmp(u, 1.0, EPSILON) == 1) return false;
15
```

```
  // Prepare to test the v parameter.
  Vector3D Q = T.CrossProduct(vecAB_);
  ATDouble v = Q.DotProduct(line.direction_vector_) * invDet;
  // The intersection lies outside of the triangle.
  if (fcmp(v, 0.0, EPSILON) == -1 ||
     fcmp(u + v, 1.0, EPSILON) == 1) return false;

  ATDouble t = vecCB_.DotProduct(Q) * invDet;
  if (fcmp(t, 0.0, EPSILON) != 0) {
    return plane_.IsIntersecting(line, intersec, t);
  }

  return false;
}
```

4.3 Non-Flexible and Ray Tracing Solutions Comparison

The cuboid is made from eight vertices and the six rectangular faces. By the definition all of the adjacent faces should remain perpendicular to each other. The non-flexible solution requires four `fcmp()` function calls and a calculation of the line-plane intersection for each of the six cuboid faces. The ray-tracing solution requires six `fcmp()` function calls, a calculation of the line-plane intersection and three `DotProduct()` and two `CrossProduct()` operations for two triangles (i.e. $\triangle_{ABC}$ and $\triangle_{CDA}$) for each of the six cuboid faces. It is clearly visible that the latter needs much more calculations than the former. However, the main strength given by the *the Möller–Trumbore intersection algorithm* is the possibility to use of a new type of a 3D objects. For instance, the simulation scene could be described with the help of the cuboid rotated to any orientation in the 3D space, the rhombohedron or any other 3D figure that has eight vertices and six faces. An example of such figure, the truncated pyramid, with the correct vertices listing order, can be found in Fig. 4.5. This approach was easiest to implement and test, since only the code of the class `Rectangle3D` had to be alter. All the other layers of code on top of it (e.g. parts responsible for the input handling and the scene description, etc.) remained unchanged. In the future there will be pure virtual base class `Shape3D` added, after which all the other shapes will be inheriting. A skeleton of this class is already included in the SABAT package.

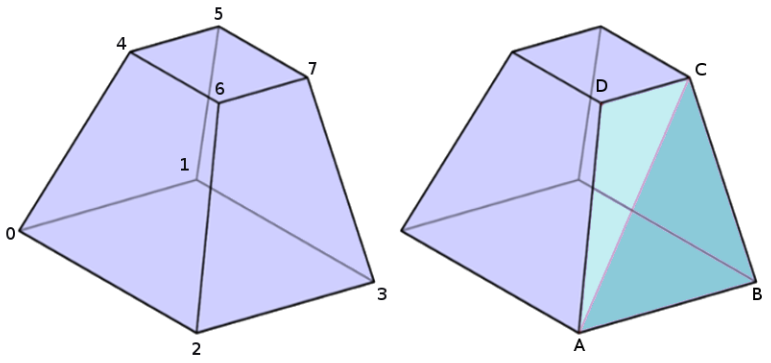


Figure 4.5: Truncated square pyramid (adapted from [24] under the GFDL). The figure on the left have the vertices listed in the order of an input required for the package. The figure on the right shows the triangles $\triangle_{ABC}$ and $\triangle_{CDA}$ that will be extracted from one of the faces (quadrangular figure) of the pyramid. The strict and correct order of the 3D figure vertices is needed so the automatically generated mesh triangles will not overlap with each other.

Chapter 5

SABAT Simulation Package

The SABAT package is a simulation software developed by one of the teams working within *the Prof. Paweł Moskal Research Group* [1] at the Jagiellonian University. The very first version of this software was designed and implemented by Michał Smolis and by the author of this thesis, during the Summer School at the M. Smoluchowski Insitute of Physics in July 2012. The School was organized and supervised by prof. dr hab. Paweł Moskal. The current version of the software (June 2016) is somewhat limited, and can be mostly use to study the neutrons interactions with matter. The reason behind this, is the package's main purpose: the Monte Carlo simulation of the novel neutron-based method for chemical threat detection. The software is under ongoing development and improvements in both of the key area: information technology and physics. The latter includes, for instance, the new types of possible reactions. The first one, consists of the implementation and testing of a new approaches to the particle tracking (e.g. *ray-tracing* and *kd-trees*), data storage (SQL query optimization), GUI and the parallel & distributed computing [9].

5.1 Coding Style

5.1.1 Comparison of the Floating-Point Numbers

Across the entire SABAT source code, the $fcmp()$ function has been used to compare two floating-point values. The fcmp package implements the Kunth's idea for more faultless floating-point comparison. The latest release of this code provided by T. C. Belding can be found here [30]. The $fcmp()$ function takes exactly three arguments: first two are the values to be compared (x_1 & x_2 respectively) and the third one is to specify comparison precision (*epislon*). As typical for C-like code, the functions returns:

$$fcmp() = \begin{cases} -1 & \text{if } x_1 < x_2 \\ 0 & \text{if } x_1 = x_2 \\ 1 & \text{if } x_1 > x_2 \end{cases} \tag{5.1}$$

Listing 5.1: The `fcmp()` function header.

```
int fcmp(double x1, double x2, double epsilon);
```

5.1.2 Used Constants and Calculation Precision

Throughout the code some constant values such as `EPSILON`, `EPSILON_5`, `DOUBLE_PI` and `PI`, were used. They were put into a single header file `geo_consts_3d.h`. Additionally, this file contains also the definition of the type `ATDobule` and the value

AT_INFINITY which marks the infinity. All the real numbers in the SABAT source code are declared to be of the ATDobule data type.

Listing 5.2: Global constant values used in the SABAT simulation package.

```
#ifndef GEO_CONSTS_3D_H
#define GEO_CONSTS_3D_H

#include <limits>

namespace {
// A floating point range for all the calculations.
typedef long double ATDouble;

const ATDouble EPSILON   = 0.0000000001; // 10^-10
const ATDouble EPSILON_5 = 0.00001; // 10^-5

// Each of the below values has a 25 digits after the decimal point.
const ATDouble PI        = 3.1415926535897932384626433;
const ATDouble DOUBLE_PI = 6.2831853071795864769252867;
const ATDouble HALF_PI   = 1.5707963267948966192313216;

// Marks the "infinity" value in the system.
const ATDouble AT_INFINITY = std::numeric_limits<long double>::max();
}

#endif // GEO_CONSTS_3D_H
```

5.1.3 Coding Style Guideline and the Coding Conventions

Source code of the SABAT package currently (June 2016) counts over 16000 lines of code and consists from over 100 files (CPP and H).

Listing 5.3: List of all the SABAT package files with their lengths.

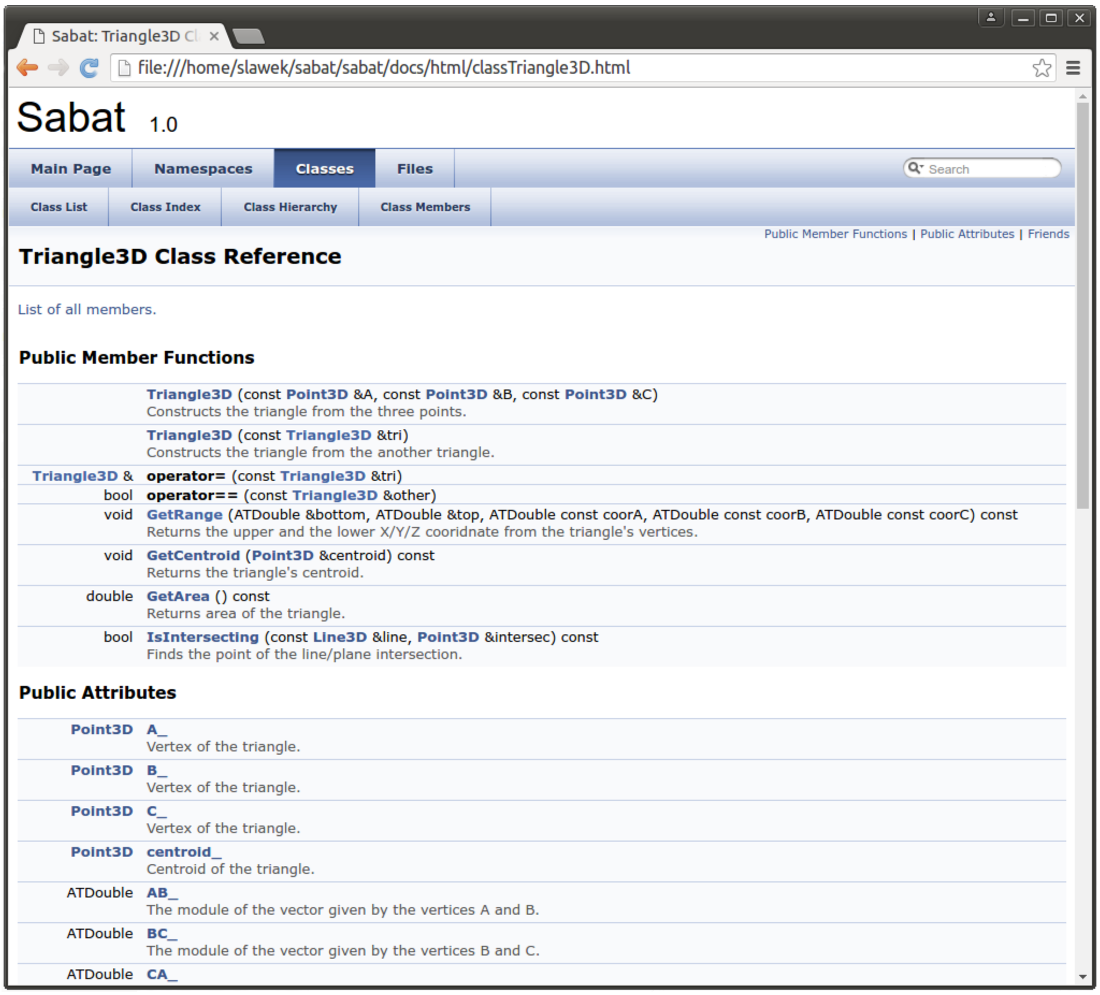
```
$ find . -name '*.cpp' -o -name '*.h' | xargs wc -l
    44 ./ParticleGenerator3D/particle_generator_3d.h
    61 ./ParticleGenerator3D/particle_generator_3d.cpp
   151 ./sabatfunkcje5_1.cpp
    33 ./ConfigurationStorage/configuration_storage.h
   124 ./ConfigurationStorage/configuration_storage.cpp
    27 ./Utilities/timer.h
    47 ./Utilities/timer.cpp
    26 ./DataBase/sql_no_insert.cpp
   138 ./DataBase/sqlite3_wrapper.cpp
    96 ./DataBase/sqlite3_wrapper.h
    52 ./DataBase/sqlite3_row.cpp
    73 ./DataBase/sqlite3_row.h
   108 ./DataBase/data_base_test.h
    17 ./Geometry3D/geo_shape_3d.cpp
    33 ./Geometry3D/geo_triplet_3d.cpp
    97 ./Geometry3D/geo_point_3d.cpp
    40 ./Geometry3D/geo_consts_3d.h
   355 ./Geometry3D/geo_bounding_box_3d.cpp
    46 ./Geometry3D/geo_shape_3d.h
    83 ./Geometry3D/geo_rectangle_3d.h
   250 ./Geometry3D/geo_cuboid_3d.cpp
   163 ./Geometry3D/geo_bounding_box_3d.h
   240 ./Geometry3D/geo_vector_3d.cpp
    46 ./Geometry3D/geo_forward_dec_3d.h
   101 ./Geometry3D/geo_triangle_3d.h
   194 ./Geometry3D/geo_plane_3d.cpp
   106 ./Geometry3D/geo_plane_3d.h
    53 ./Geometry3D/geo_line_3d.cpp
   119 ./Geometry3D/geo_kdtree_3d.h
    55 ./Geometry3D/geo_fcmp_3d.h
    74 ./Geometry3D/geo_triplet_3d.h
    61 ./Geometry3D/geo_line_3d.h
   284 ./Geometry3D/geo_kdtree_3d.cpp
    87 ./Geometry3D/geo_rectangle_3d.cpp
   137 ./Geometry3D/geo_cuboid_3d.h
```

```
     62 ./Geometry3D/geo_point_3d.h
    128 ./Geometry3D/geo_vector_3d.h
    166 ./Geometry3D/geo_triangle_3d.cpp
    288 ./simulation.cpp
     60 ./Random/exp_generator.h
     86 ./Random/random.h
     65 ./Random/exp_generator.cpp
     71 ./Random/random.cpp
     83 ./debug.h
     56 ./Physics3D/test_physics_3d.h
    786 ./Physics3D/phy_substance.cpp
     83 ./Physics3D/phy_particle_3d.h
     74 ./Physics3D/phy_consts_3d.h
     23 ./Physics3D/phy_object_3d.h
     20 ./Physics3D/phy_object_3d.cpp
    234 ./Physics3D/phy_substance.h
     20 ./Physics3D/phy_detector_3d.h
     61 ./Physics3D/phy_consts_3d.cpp
     17 ./Physics3D/phy_detector_3d.cpp
    186 ./Physics3D/phy_particle_3d.cpp
     56 ./profiling/flat_profile_2_CSV.cpp
     97 ./simulation.h
    237 ./ConfigurationFile/configurationfile.cpp
    120 ./ConfigurationFile/test_configuration_file.h
   2112 ./ConfigurationFile/tinyxml2/tinyxml2.h
   2444 ./ConfigurationFile/tinyxml2/tinyxml2.cpp
     62 ./ConfigurationFile/configurationfile.h
    312 ./main.cpp
     25 ./Factories/sceneafactory.h
     21 ./Factories/scenedfactory.h
    181 ./Factories/sceneafactory.cpp
     18 ./Factories/scenegammatestfactory.h
     20 ./Factories/sceneffactory.h
     19 ./Factories/scene02factory.h
    119 ./Factories/scenedfactory.cpp
    148 ./Factories/scenebfactory.cpp
     23 ./Factories/scenecfactory.h
     22 ./Factories/scenebfactory.h
     20 ./Factories/scene03factory.h
     86 ./Factories/sceneffactory.cpp
     21 ./Factories/sceneefactory.h
     31 ./Factories/scenegammatestfactory.cpp
     87 ./Factories/scene03factory.cpp
    141 ./Factories/scenecfactory.cpp
     87 ./Factories/scene02factory.cpp
     14 ./Factories/simple_scene_factory.h
    108 ./Factories/sceneefactory.cpp
      0 ./Factories/simple_scene_factory.cpp
     14 ./master_test.h
     82 ./DataManagement/data_file.cpp
     31 ./DataManagement/data_files_manager.h
     71 ./DataManagement/neutron_data_file.h
     20 ./DataManagement/data_files_manager.cpp
     38 ./DataManagement/data_file.h
     48 ./DataManagement/neutron_data_file.cpp
     50 ./DataManagement/gamma_data_file.cpp
     74 ./DataManagement/gamma_data_file.h
    215 ./CMakeFiles/CompilerIdCXX/CMakeCXXCompilerId.cpp
     28 ./SourceGenerator/source_generator.h
     31 ./SourceGenerator/source_generator.cpp
   1377 ./AuxiliaryFunctions/tracker_functions.cpp
    107 ./AuxiliaryFunctions/data_management_functions.h
    376 ./AuxiliaryFunctions/tracker_functions.h
    508 ./AuxiliaryFunctions/data_management_functions.cpp
   1100 ./AuxiliaryFunctions/test_aux_functions.h
  16991 total
```

To mange such a complex project and maintain the source code readability, it was decided to adapt the coding style proposed in the online resource: *Google C++ Style Guide* [31].

5.1.4 Commenting Style: Doxygen

The Doxygen [32] is a user-friendly tool for automatic generation of documentation from the annotated C++ source code. It can generate the documentation among others, in the HTML, PDF or LaTeX format. To make a full use of the Doxygen, specific comment style has to be used in the entire source code. For clarity of the examples, the Doxygen-style comments were omitted or slightly changed in the listings presented in this thesis. Furthermore, the documentation shows only the comments made in the H files (e.g. mostly class definitions). The comments in the CPP files are not used to generate the documentation, however there is a copy of the methods' comments kept before the methods' declarations. A copy of the SABAT package documentation generated with the help of Doxygen will be provided as a DVD appendix to this thesis.


Sabat: Triangle3D Cl
file:///home/slawek/sabat/sabat/docs/html/classTriangle3D.html
Sabat 1.0
Main Page
Namespaces
Classes
Files
Search
Class List
Class Index
Class Hierarchy
Class Members
Public Member Functions | Public Attributes | Friends

Triangle3D Class Reference

List of all members.

Public Member Functions

	Triangle3D (const **Point3D** &A, const **Point3D** &B, const **Point3D** &C) Constructs the triangle from the three points.
	Triangle3D (const **Triangle3D** &tri) Constructs the triangle from the another triangle.
Triangle3D &	**operator=** (const **Triangle3D** &tri)
bool	**operator==** (const **Triangle3D** &other)
void	**GetRange** (ATDouble &bottom, ATDouble &top, ATDouble const coorA, ATDouble const coorB, ATDouble const coorC) const Returns the upper and the lower X/Y/Z cooridnate from the triangle's vertices.
void	**GetCentroid** (**Point3D** &centroid) const Returns the triangle's centroid.
double	**GetArea** () const Returns area of the triangle.
bool	**IsIntersecting** (const **Line3D** &line, **Point3D** &intersec) const Finds the point of the line/plane intersection.

Public Attributes

Point3D	**A_** Vertex of the triangle.
Point3D	**B_** Vertex of the triangle.
Point3D	**C_** Vertex of the triangle.
Point3D	**centroid_** Centroid of the triangle.
ATDouble	**AB_** The module of the vector given by the vertices A and B.
ATDouble	**BC_** The module of the vector given by the vertices B and C.
ATDouble	**CA_**

Figure 5.1: The local browsable HTML version of the SABAT package documentation generated with the help of Doxygen.

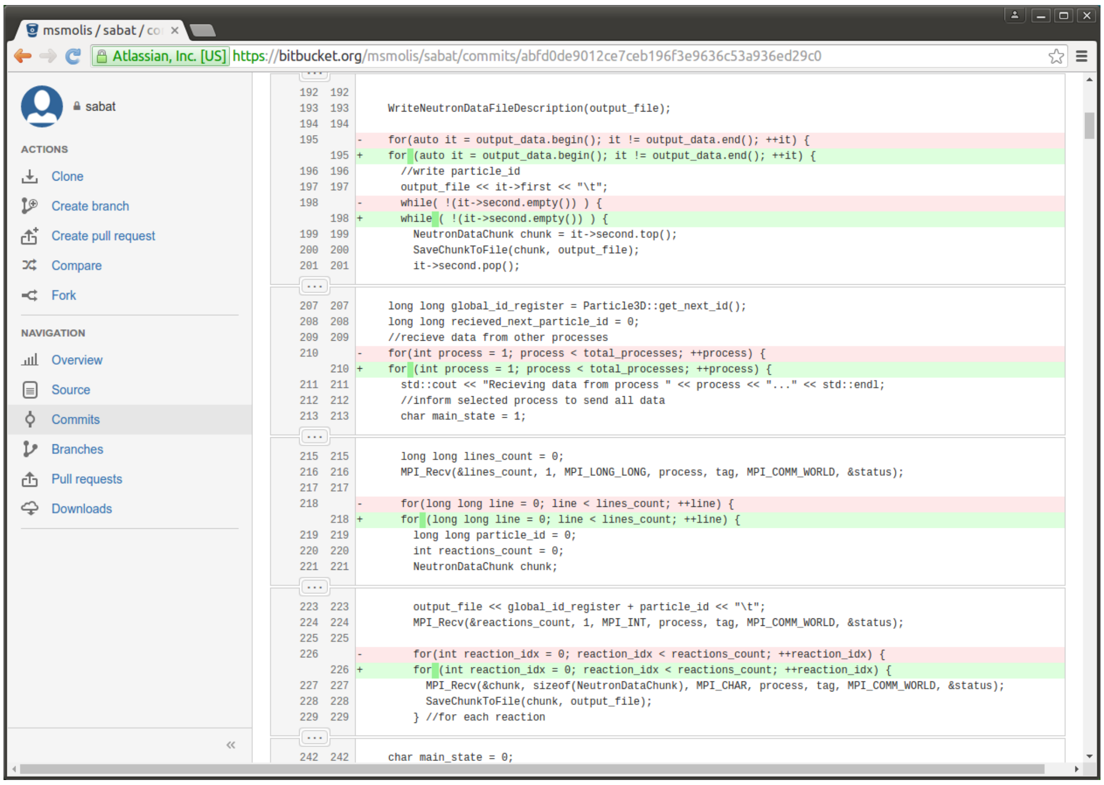
5.2 Obtaining and Installation

5.2.1 Obtaining the SABAT package

In the current development state, the SABAT package has to be downloaded from the BitBucket.org website, which is a Git [44] remote repository. Git is one of the most widely used software version control system. For now, the access to the source code is restricted only to the package developers and available upon request. A copy of the newest SABAT package version will be provided as a DVD appendix to this thesis.

Listing 5.4: Cloning of the SABAT repository (USER: user's account name).

```
$ mkdir sabat
$ cd sabat
$ git clone https://USER@bitbucket.org/msmolis/sabat.git
$ git checkout master
```



```

    WriteNeutronDataFileDescription(output_file);

-   for(auto it = output_data.begin(); it != output_data.end(); ++it) {
+   for (auto it = output_data.begin(); it != output_data.end(); ++it) {
      //write particle_id
      output_file << it->first << "\t";
-     while( !(it->second.empty()) ) {
+     while ( !(it->second.empty()) ) {
        NeutronDataChunk chunk = it->second.top();
        SaveChunkToFile(chunk, output_file);
        it->second.pop();
...
    long long global_id_register = Particle3D::get_next_id();
    long long recieved_next_particle_id = 0;
    //recieve data from other processes
-   for(int process = 1; process < total_processes; ++process) {
+   for (int process = 1; process < total_processes; ++process) {
      std::cout << "Recieving data from process " << process << "..." << std::endl;
      //inform selected process to send all data
      char main_state = 1;
...
      long long lines_count = 0;
      MPI_Recv(&lines_count, 1, MPI_LONG_LONG, process, tag, MPI_COMM_WORLD, &status);

-     for(long long line = 0; line < lines_count; ++line) {
+     for (long long line = 0; line < lines_count; ++line) {
        long long particle_id = 0;
        int reactions_count = 0;
        NeutronDataChunk chunk;
...
        output_file << global_id_register + particle_id << "\t";
        MPI_Recv(&reactions_count, 1, MPI_INT, process, tag, MPI_COMM_WORLD, &status);

-       for(int reaction_idx = 0; reaction_idx < reactions_count; ++reaction_idx) {
+       for (int reaction_idx = 0; reaction_idx < reactions_count; ++reaction_idx) {
          MPI_Recv(&chunk, sizeof(NeutronDataChunk), MPI_CHAR, process, tag, MPI_COMM_WORLD, &status);
          SaveChunkToFile(chunk, output_file);
        } //for each reaction
...
    char main_state = 0;
```

Figure 5.2: The Bitbucket web-based GUI with some of the SABAT package commits.

5.2.2 Installation of the SABAT Package

The installation instructions presented in this paragraph are written with the assumption that the reader has an access to the remote repository of the SABAT package with at least the read privileges. Instructions shown below were executed and tested in the Ubuntu 12.04.5 LTS (Precise Pangolin) operating system.

Required Third-Party Libraries

First, the required third-party libraries and dependences have to be installed into the user's system. This includes: SQLite [37], OpenMPI [45] and CMake [46]. Some of them are standard on almost all of the Linux platforms, others are not.

Listing 5.5: Installation of the libraries required by the SABAT package.

```
$ sudo apt-get update && sudo apt-get dist-upgrade
$ sudo apt-get install openmpi-bin openmpi-common openssh-client \
openssh-server libopenmpi1.3 libopenmpi-dbg libopenmpi-dev
$ sudo apt-get install libsqlite3-dev
$ sudo apt-get install cmake
```

Source Code Compilation

To build the solution, CMake tool is required. The `CMakeLists.txt` file should be located in the the main project directory.

Listing 5.6: Cloning the SABAT repository (USER: user's account name).

```
$ cd sabat
$ make
$ cmake CMakeLists.txt
```

Program Execution

To run the program, user has to specify the `SCENE_FILE` which is an XML file containing the selected scene description. An example of the geometry input file `scenes/A2.xml` with its description can be found in the Subsec. 3.5. The visualization of the same scene in the two forms, read directly from the XML files and as the generated triangle mesh respectively, can be found in Fig. 3.9. Moreover, Fig. 3.10 shows the geometry described in the `scenes/C2.xml` file.

Listing 5.7: Source code compilation and execution on a single process.

```
$ ./Atometry scenes/SCENE_FILE.xml
```

In case of running on multiple processes user has to also specify the total number of them:

Listing 5.8: Source code compilation and execution on multiple processes.

```
$ make
$ mpirun -np NO_OF_PROCESSES ./Atometry scenes/SCENE_FILE.xml
```

5.2.3 Output Data Format

All the data gathered during the simulations is saved into text files in the subdirectory `temp_output_files` under the main simulation folder. This includes, amongst others, particle direction, reactions, the number and the type of the particles created in the reactions, the place of the reaction and material in which the reaction occurred. To aggregate the data, the user has to execute the simulation with the additional parameter `--join-data`. The simulation will be skipped, and the `output_files` directory containing the final data will be created.

Listing 5.9: Aggregation of the data.

```
$ ./Atometry --join-data
```

5.3 Source Code Testing

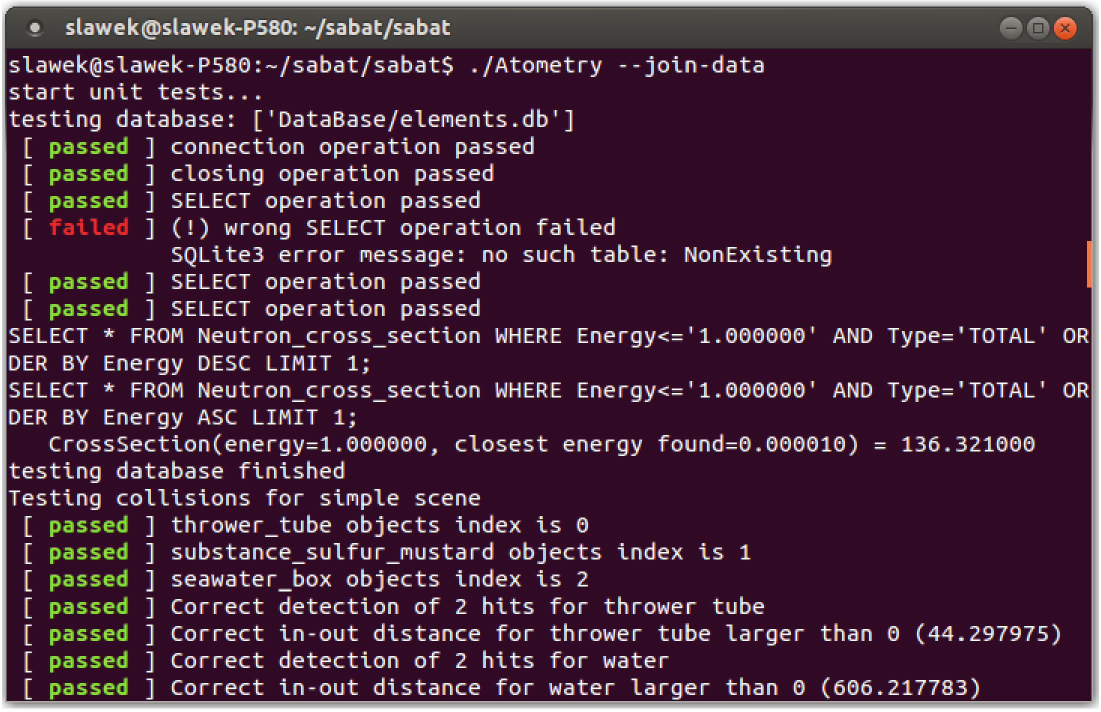
To test some essential parts of the source code (e.g. database connection, particle tracking, etc.), unit test were implemented. These tests are used to find out if the errors occurred in, for example, the particle tracking or the 3D scene geometry description. The vast majority of them are executed before the simulation starts and in case of failure the simulations is going to be terminated, since the obtained results will more likely be faulty. The tests are printed in the terminal with the help of macros presented in the List. 5.10.

Listing 5.10: Format of the tests output in terminal.

```
#ifndef MASTER_TEST_H
#define MASTER_TEST_H

#define CPASS std::cout << " [ "<< "\033[1;32m"<<"passed"<<"\033[0;m"<< " ] "
#define CFAIL std::cout << " [ "<< "\033[1;31m"<<"failed"<<"\033[0;m"<< " ] "
#define OFFSET "            "

#endif // MASTER_TEST_H
```



```
slawek@slawek-P580: ~/sabat/sabat
slawek@slawek-P580:~/sabat/sabat$ ./Atometry --join-data
start unit tests...
testing database: ['DataBase/elements.db']
 [ passed ] connection operation passed
 [ passed ] closing operation passed
 [ passed ] SELECT operation passed
 [ failed ] (!) wrong SELECT operation failed
            SQLite3 error message: no such table: NonExisting
 [ passed ] SELECT operation passed
 [ passed ] SELECT operation passed
SELECT * FROM Neutron_cross_section WHERE Energy<='1.000000' AND Type='TOTAL' OR
DER BY Energy DESC LIMIT 1;
SELECT * FROM Neutron_cross_section WHERE Energy<='1.000000' AND Type='TOTAL' OR
DER BY Energy ASC LIMIT 1;
   CrossSection(energy=1.000000, closest energy found=0.000010) = 136.321000
testing database finished
Testing collisions for simple scene
 [ passed ] thrower_tube objects index is 0
 [ passed ] substance_sulfur_mustard objects index is 1
 [ passed ] seawater_box objects index is 2
 [ passed ] Correct detection of 2 hits for thrower tube
 [ passed ] Correct in-out distance for thrower tube larger than 0 (44.297975)
 [ passed ] Correct detection of 2 hits for water
 [ passed ] Correct in-out distance for water larger than 0 (606.217783)
```

Figure 5.3: An example of the output generated by the unit tests.

Moreover, it is possible to spot the errors in the calculations when the data is plotted in the form of the histograms or power spectra, however in this particular case, the knowledge from the field of particle physics is required. Hence, the validation of the data produced in the simulation was initially done by Dr. Michał Silarski and Dominika Hunik. The substantial errors in the geometry and particle tracking methods can be spotted on the large distributions of gamma quanta interaction points in any of the $x-y$, $x-z$, $z-y$ views. An example of such an error is shown in Fig. 5.4. In this particular case, the error occurred due to the use of the old method of the solid object geometry description which did not allowed the use of the *skewed* cuboids.

Section 4.1 contains the details regarding the initially available 3D objects (e.g. cuboids) that could be used to create the simulation scene. Moreover, the Sec. 4.3 points out the main difference between the new and old way of the geometry description and explains why the new approach gives much more flexibility to the user.

5.4 Software Profiling

A software profiler is a tool that tracks running times of all the functions and methods in the executed program. This information can be very helpful in identifying possible

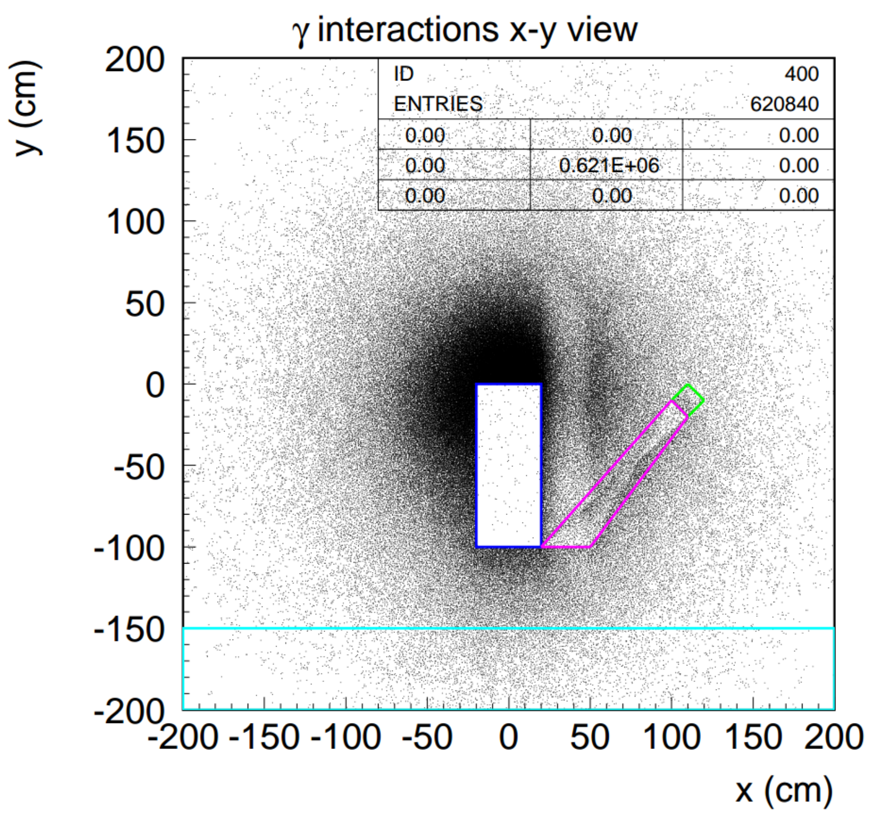


Figure 5.4: An example of the error in the geometry description clearly visible on the $x - y$ view plot of the γ quanta interactions.

bugs and bottlenecks in the source code. To profile the SABAT package the GNU gprof [36] profiler was used. The version of the source code which was profiled had only the *ray-tracing* implemented (i.e. without the *kd-trees*).

To enable profiling the program has to be complied with additional `-pg` flag which has to be added into the `CMakeLists.txt` file located in the main project directory. After the files was edited, the software has to be re-complied normally with the `make` command.

Listing 5.11: Compilation flags in the `CMakeLists.txt` file.

```
set (CMAKE_CXX_FLAGS "-std=c++0x -lsqlite3 -Wall -O3 -g3 -pg")
```

After the compilation the package has to be executed as usual to produce the profiling data that will be saved into the `gmon.out` file. Next this file has to be pass to the gprof to analyse and generate the profiling outcome.

Listing 5.12: GNU gprof execution.

```
$ gprof Atometry gmon.out > profile.txt
```

The profile will be a text file containing the execution times of all the called functions and methods. For the purpose of identifying possible bottlenecks the most interesting seems to be the columns *% time*, *self seconds*, *calls* and *self ms/called* from the flat profile. First one describes the total running time of the function/method. The second contains the total execution time of the function/method. The third one shows the number of the function/method calls. And finally, the last one presents the total execution time of the function/method on average per call. An example profile file may look as the one on the List. 5.13.

Listing 5.13: A part of the analysis file made by the GNU gprof (edited for clarity).

```
  %    self              self
 time  seconds calls   ms/call name
 12.56 0.49   3807287     0.00 Vector3D::CrossProduct(Vector3D const&) const
  4.62 0.18   6500364     0.00 Point3D::Point3D(Point3D const&)
  4.36 0.17   7557740     0.00 Vector3D::DotProduct(Vector3D const&) const
```

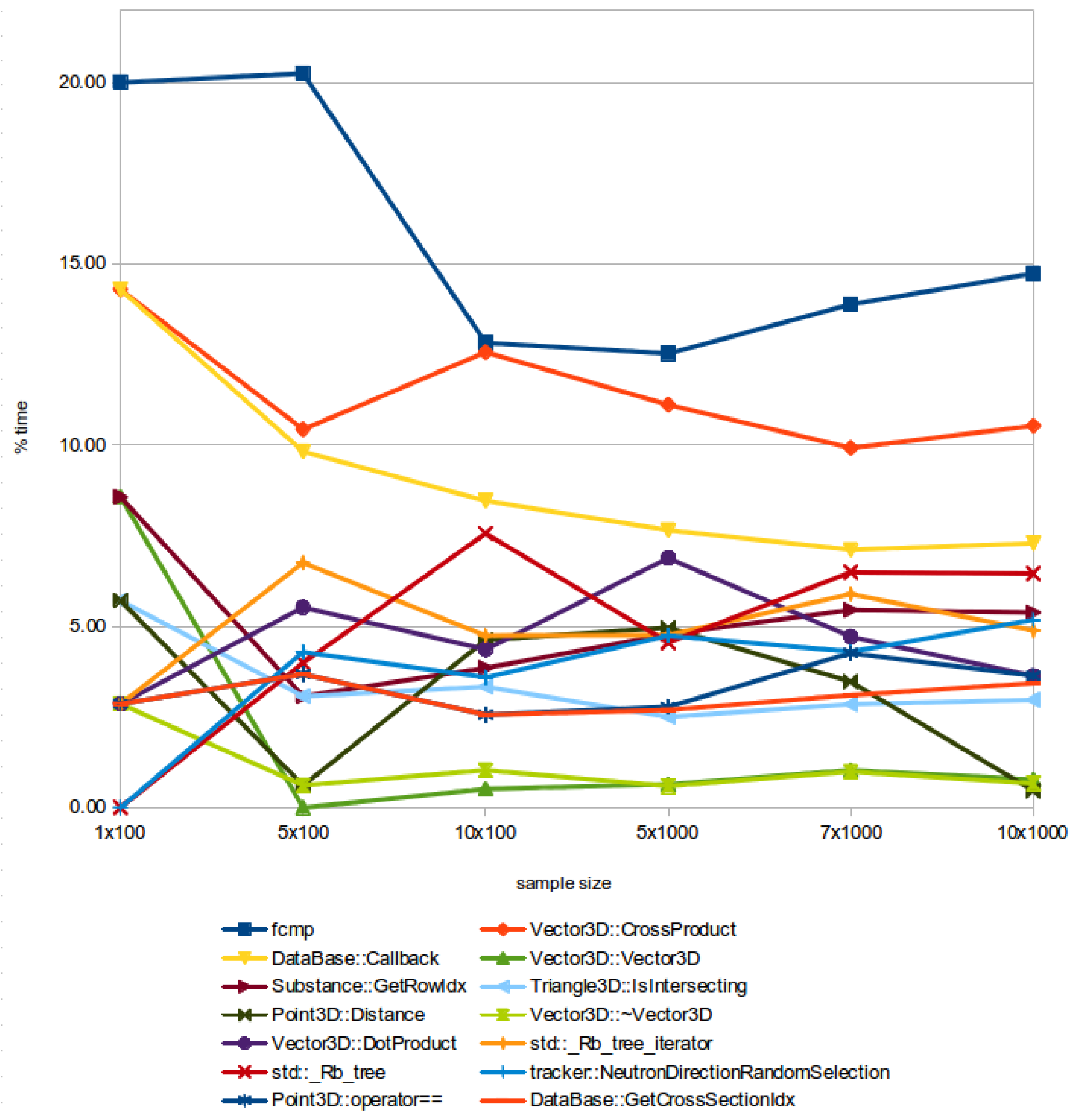


Figure 5.5: The plots shows the total percentage [%] of time used by the given function/method for the six different simulation samples of increasing size. The full names of the functions/methods were edited (e.g. removed parameters and arguments) for the graph clarity. There are differences between the profile samples in both, the order of the items with regards to the total time, and in the listed top entries. The detail information can be found in the profile files `profileSAMPLESIZEsceneA2.txt`. The floating-number comparison and the vector cross product operation are taking most of the simulation time. Furthermore, the operations related to the retrieving data from the database (including both `std::Rb_tree` and `std::Rb_tree_iterator` of the STL's `std::map` container used by the `Row` class) are accounted for roughly 33% of total simulation time. Hence the way of the data handling maybe a good point for the future optimization. The simulations were conducted in the single process mode on the Intel(R) Core(TM) i3 CPU M 330 @ 2.13GHz (2 cores, 4 threads), under the Ubuntu 12.04.5 LTS (Precise Pangolin) operating system.

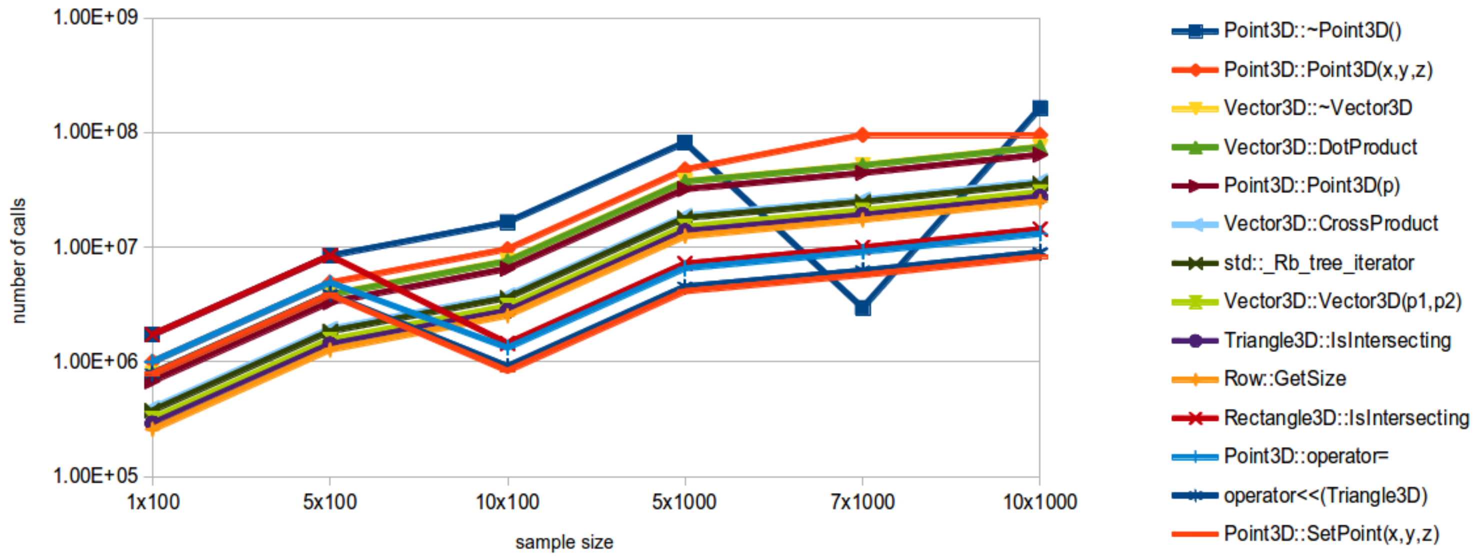


Figure 5.6: The plots shows the total number of calls (log scale) of the given function/method for the six different simulation samples of increasing size. The full names of the functions/methods were edited (e.g. removed parameters and arguments) for the graph clarity. There are differences between the profile samples in both, the order of the items with regards to the total number of calls, and in the listed top entries. The detail information can be found in the profile files `profileSAMPLESIZEsceneA2.txt`. The number of call for the `fcmp` and `DataBase::Callback` shown in Fig .5.5 were not calculated by the gprof tool, most likely due to the fact that they are written in C language and not C++ like the rest of the code. The simulations were conducted in the single process mode on the Intel(R) Core(TM) i3 CPU M 330 @ 2.13GHz (2 cores, 4 threads), under the Ubuntu 12.04.5 LTS (Precise Pangolin) operating system.

Chapter 6

Example of the SABAT Package Applications

Hazardous materials such as the drugs or explosives posses a very distinctive chemical composition (Tab. 1.1). One of the very promising, non-invasive detection method of such substances is the Neutron Activation Analysis [7]. More detail description of this technique can be found in section Sec. 1.1.

6.1 Underwater Threat Detection System

After the Second World War most of the German chemical weapons, including for instance, rusted tanks with mustard gas, were sunk in the Baltic Sea shortly after the War was over. Some research estimates that if so much as 1/6 of the chemical agents would be spread at the Sea, the Baltic marina life could be destroyed for the next 100 years [7]. The SABAT packages is being used to study of the novel, atometry-based, underwater threat detection system that can help to find the exact placement of these dangerous chemicals [6, 7], so they can be excavated and safely utilized. The Monte Carlo study is conducted to simulate a different versions of the underwater detection device with the emphasis put on the special guides (tubes) for gamma quanta and neutrons beams [7]. Fig. 6.1 shows how the prototype of such detector could look like.

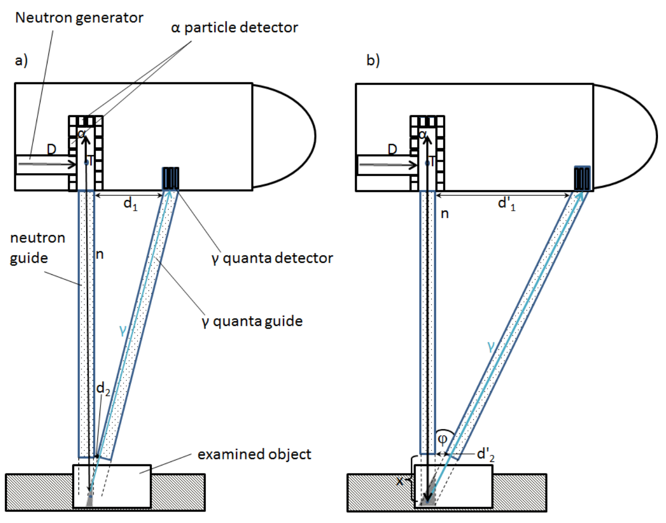


Figure 6.1: Schematics of the underwater detector (adapted from [6]).

6.2 Background Study

After so many years from the end of the Second World War the sunken munition shells and tanks have to be covered with the sea mud layer. Also the neutron rays have to travel through the sea water. Neutron particles that react with the water and mud will create an intense environmental background noise [6]. To overcome this problems, the idea of the neutron guides was introduced [4, 6]. The details of the particle interactions are not part of this thesis and can be found in any of the [4, 6, 7] papers. The Monte Carlo simulation is used to determine most effective and optimal design of the detector geometry which includes its shape (e.g. guides, detector), types of the neutron emitter & gamma detector and the device construction materials.

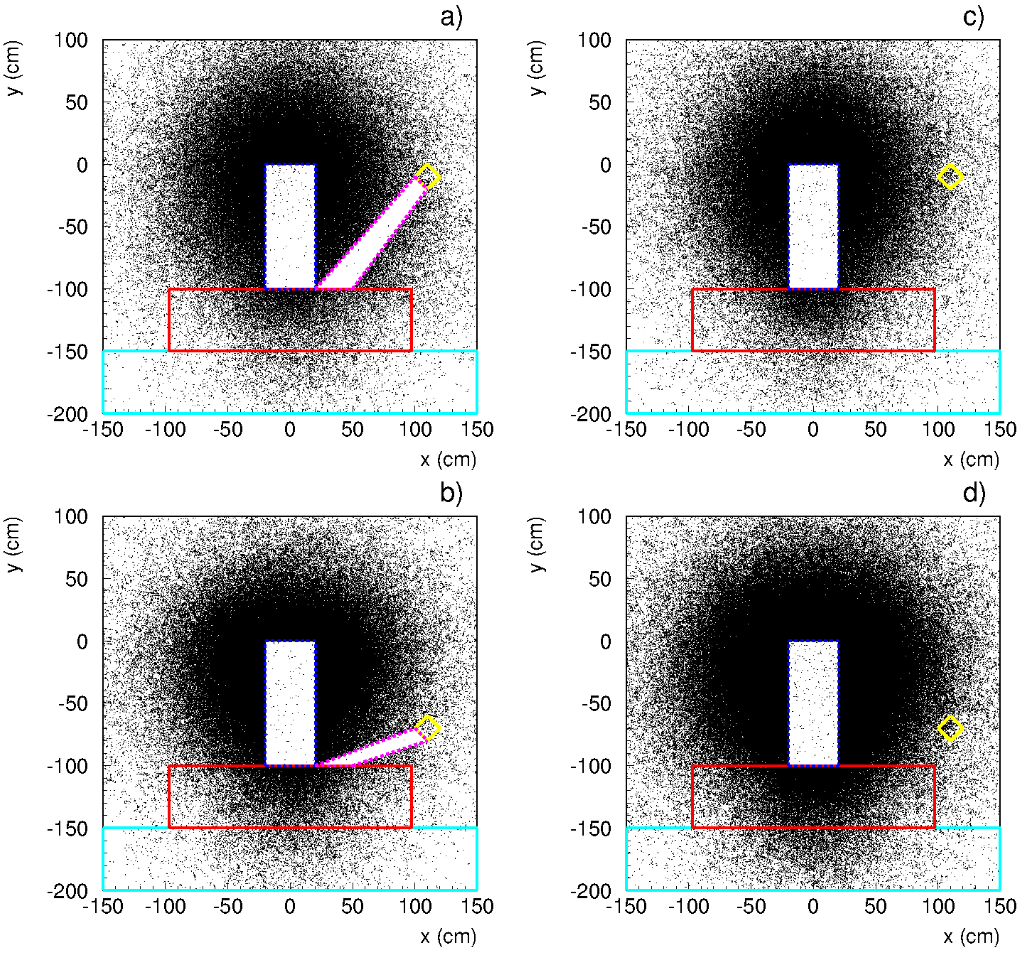


Figure 6.2: An example of the Monte Carlo simulation of the γ quanta interactions presented in the $x - y$ views. On all the plots, the detector contour is yellow, the neutron guides are blue, the γ guides are magenta, the hazardous substance (sulphur mustard) is red and the sea bottom is cyan. The a) and b) shows two different setups with the gamma quanta guides of a different shape, size and angle with respect to the sea bottom. Also the detector occupies a different position. The same setup but without the γ quanta guides is shown on the c) and d) respectively. For both of setups, the neutron emitter is a point-like source placed on the top of the neutron guides (adapted from [7]).

In conclusion, based on the example shown above, presented method and the SABAT software can successfully be used to design the device for the underwater threat detection. The Monte Carlo study can be applied to optimize the tentative 3D spatial setup of the apparatus elements, which includes the neutron emitter and the γ quanta detector positions with respect to each other and to the irradiated substance (e.g. chemical munition shells). Moreover, the most favourable shapes and sizes of the guides and the other elements of the device can be determined with the reasonable high accuracy. Furthermore, the package can be helpful with the selection and testing of the possible construction materials which can be very expensive. Thus, it maybe not possible or feasible to test them all in the laboratory due to their cost, needed equipment and the safety regulations with regards to the neutron irradiation of the given material, and later transportation and utilization of it.

Chapter 7

Summary and Perspectives

7.1 Findings and Conclusions

The main goal of this thesis was an implementation and application of 3D computer graphics technologies used in video games for Monte Carlo calculations of multi-particle transport code. A brief description of the geometrical primitives such as the point, vector, line, plane and cuboidal objects in the 3D space was given. Furthermore, optimization techniques borrowed from the field of computer graphics such as the acceleration data structures (e.g. the *kd-trees* and the *bounding volumes*) and ray-tracing algorithm for the purpose of the particle-object detection were presented. Based on these studies, a particle tracking algorithm for the Monte Carlo simulation of the neutron interaction with matter was designed and implemented in the C++ programming language as the SABAT software package.

The Monte Carlo research conducted with the help of the SABAT package lead to three publications [5, 6, 7] that will be a basis for construction of the novel atometry-based underwater threat detection system described in more detail in the previous Chap. 6.

7.2 Ideas for Further Research

Due to the computational complexity of the SABAT simulation, plenty of optimization techniques needs to be used. This thesis presented a few of them, like the usage of the *ray tracing* or the *kd-tree* algorithms.

As it was mentioned in the Subsec. 1.3.2 and proven by the source code profiling (Fig. 5.6), communication with the database has a substantial impact on the overall simulation execution time. One idea that could possibly eliminate this bottleneck is to copy the whole database in the RAM memory when the simulation starts. It could be reasonable solutions, especially since currently the `elemnts.db` is roughly of the size of 49 MB.

The raw results obtained during the simulation must be properly analysed. This task require the help of additional tools. One of them is the ROOT package [47], which is one of the most widely used analysis frameworks in the field of particle physics. It can deal and manage even massive data sets spread over multiple files. Output files generated during the SABAT simulation could be adjusted to implement of the so-called trees structures that can be immediately read and process by the ROOT.

The SABAT package is also missing user-friendly GUI. Even though, the geometry description has to be provided in a simple-structured XML file (Subsec. 3.5), thus this method seems to not be efficient enough, especially for the user who is not accustomed to the software. Such interface should not only be portable between the different environments, but also provide a possibility to interactive modelling of the scene with the additional automatic error checking ability. Such GUI would inevitably widen the pool of possible users. Since the geometry can be very complicated and the Monte Carlo simulation may require a very large sample, it maybe feasible to design an online interface in one of the available most popular technologies like HTML/CSS/JavaScript, Python/CGI or JavaFX that would allow to model the input files and run the simulation on a specialized computer farms on many CPUs at once.

Another idea of improvement can be made in the scene geometry modelling area. Since the 3D objects (e.g. cuboids) are essentially described with the help of the triangle meshes, it maybe feasible to use of a computer-aided design (CAD) tools such as the AutoCAD [28], Blender [29] or any other software that can model 3D simulation scene and export it into the required format. If needed, a custom made plugins for the SABAT package can be made to adjust the output of the CAD applications. To the best of the author's knowledge, all of the listed programs are able to triangulate of the polygons faces and are widely used, in both, the industrial and in the academic environment. Implementation of this idea could make the package more flexible for the users, and may lead to the spread of the application in other areas than the nuclear physics research.

With regards to accelerating data structures like the *kd-trees* it maybe feasible to test a different split function such as the *surface area heuristic* which may yield a better results. Furthermore, other partitioning data structures such as the *bounding interval hierarchy* or the *binary space partitioning* can give a better results.

Further ideas, such as *the parallel and distributed processing* or *advanced random number generators*, were discussed in the theses [9] of the other SABAT Collaboration members.

Appendix A

SABAT Database

A.1 Description of the Selected Data Storage

Development of the physical phenomena simulation requires three main components: a geometrical description of the simulation scene, a mathematical description of the laws of physics and a lots of initial information. Depending on the simulation type and scope, this data includes experimentally measured or carefully calculated values such as particles' cross sections, elements' atomic mass and number, physical constants and the chemical composition of the objects from which the simulation scene is build. Thus, volume of these data set can decrease or grow. Most likely, during the simulation execution all this data cannot be stored in the process operation memory. Also, the decision has to be made whether or not to allow the users to made changes in this database in order to enrich user's experience and to extend the application's Graphical User Interface (GUI).

Taking all the points enlisted above under consideration, the decision was made, to use SQLite [37] database engine. This allows to distribute SABAT package with all the necessary data in a form of a single, additional file. Due to the elasticity provided by the SQL, it was possible to adjust the database schema throughout the simulation development process whenever support of the new physical effects was needed.

A.1.1 SQLite

Nearly all of the modern, most widely used database management systems, implements the Relational Database Management System (RDBMS), that was designed by E. F. Codd in 1970 and it is based on the mathematical concept of relations [38]. Data stored in the RDBMS is manage through the queries written in the Structured Query Language (SQL). SQLite is an *open source* software library that implements SQL database engine [37]. SQLite was chosen due to certain advantages over all the others, commercially or freely available database engines such as MySQL [39] or MariaDB [40]. First and foremost the source code for SQLite is in the public domain, which means that can be used for any purpose without any charge. Also, a full, complete SQLite database has a form of a single, cross-platform file, that can be copied between different 32/64-bit systems. Additionally SQLite provides very elastic, clear, yet powerful C/C++ API and a detail documentation available online.

Source code and precompiled binaries for Linux, Windows, Max OS X (x86), Windows Runtime, Windows Phone 8, .NET, can be downloaded from the SQLite official website [37]. Various management tools and packages for variety of Unix-like systems, such as RPMs & DEBs, exists. SQLite official homepage [37] claims that *SQLite is likely used more than all other database engines combined*. Since from the very beginning we planned to distribute SABAT package as a cross-platform, free software, these unique features, was above all, the reason for selection of the SQLite database engine.

A.1.2 Database Schema

Initial database scheme named `elemnts.db` was designed and implemented in SQL by the author of these thesis, and populated with the data by Dominika Hunik. Later adjustments and changes in the scheme (e.g. addition of the database triggers) were done by Michał Smolis and Dominika Hunik. All the data regarding particle energy spectra, photon cross sections, etc. were obtained by Dominika Hunik and by Dr. Michał Silarski from [41] and with the help of the web program XCOM [42].

The extended entity–relationship (EER) model of the database was generated in two steps. First, the SQL code was extracted from the database using the `dump` command. Secondly, the schema diagram was generated automatically from the dump file with the help of MySQL Workbench [43]. To show the relations between the entities, the diagram is using the Crow's foot notation explained in the Fig. A.1:

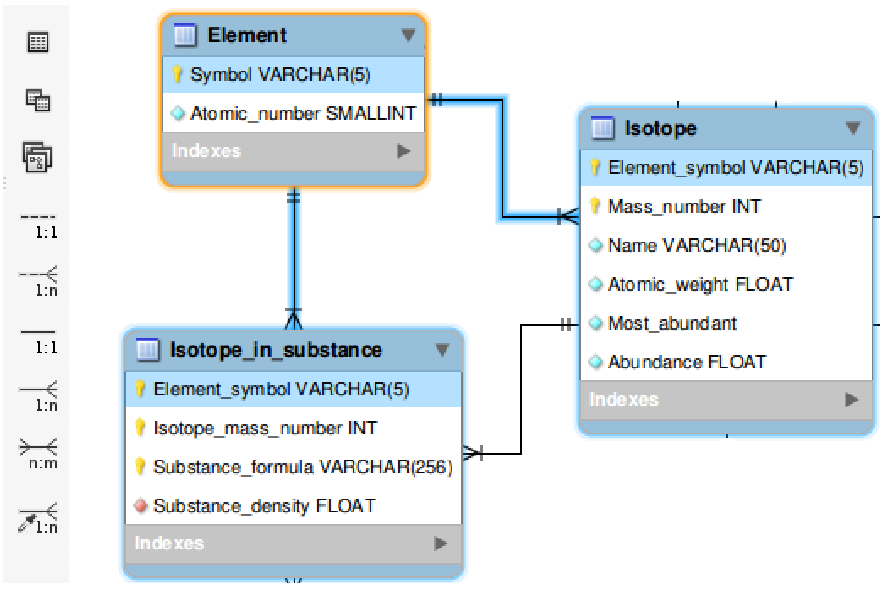


Figure A.1: Crow's foot notation with an example.

On Fig. A.1 one can see that the `Symbol` from the entity `Elemnt` is referenced by the `Isotope(Elemnt_symbol)` and by the `Isotope_in_substance(Elemnt_symbol)`. These relationships are one-to-many ($1 : n$), which means that each row (tuple) is referenced by zero, one or more rows (tuples) in the relating tables. This allows to store in the single place the informations frequently used by the other entities. Furthermore, this approach helps to preserve the consistency of the data kept in the database.

Dumped SQL schema is also used as a backup in case the database would become corrupted. This way, restoration of the SQLite database becomes trivial and can be completed within minutes.

Listing A.1: Generating the database dump and restoring the database from it.

```
$ sqlite3 elements.db .dump > dump.sql
$ sqlite3 elements.db < dump.sql
```

A.2 SQLite Database C++ Wrapper

In every step of the simulation a variety of different information is being retrieved from the database. Thus, database queries are one of the simulation's bottlenecks. Trade-offs has to be made between the code readability and the query execution time. Also, the extracted data (table rows, so-called tuples, in case of RDBMS) has to be provided in the form that is easy to manipulate. For these purposes, a C++ wrapper for the SQLite API was written. To ease up the usage of the database query results, a set of preprocessor macros doing automatic type casting was prepared.

Two different types of the queries were recognized. Both are taking as an argument SQL statement in the form of the `std::string` and a reference to the `Row` collection which will be populated with the retrieved results. Normally, query can result in an empty set if no row was found in the database matching exactly specified parameters. *Cross-section query* is a special case of the `SELECT` operation. If there is no entry in the database, the row with the energy closest in the terms of absolute value is going to be retrieved. Additionally, returned value marks the state of the query execution.

Listing A.2: Queries types and possible results.

```
enum OperationResult {
  CONNECT_FAIL, // connection - failed
  CONNECT_OK,   // connection - succeeded
  CLOSING_OK,   // closing connection - succeeded
  CLOSING_FAIL, // closing connection - failed
  SELECT_OK,    // SELECT operation succeeded
  SELECT_FAIL   // SELECT operation failed
};

OperationResult SELECT(std::string query, Row& rows);
OperationResult SELECTCrossSect(std::string query, Row& rows);
```

The `Row` type is essentially a collection of rows returned by the database query. Since before the query execution the number and the size of the retrieved entries is not known, the data structure should be elastic enough to support dynamic, run-time applied changes. To achieve this goal, `class Row` was designed. This class consist of a main dataset - `std::vector` - in which every item is a row retrieved from the database. Moreover, every `RowItem*` itself is a pointer to a dynamic collection of the type `std::map`. Every pair `<key, mapped_value>` in the `RowItem` element consist of two `std::string`. First one, the `key` is the column's name, and the `mapped_value` is the associated value. C++11 standard and the STL's container `std::map` ensures that complexity of both insert and access operations are logarithmic in size, respectively $O(log_2(n+1))$ and $O(log_2(n))$.

Listing A.3: Data structures for database readout.

```
typedef std::map<std::string, std::string> RowItem;
typedef std::map<std::string, std::string>::const_iterator RowItemIter;
```

Element		
column name	*data type*	*constraint*
Symbol	VARCHAR(5)	NOT NULL
Atomic number	SMALLINT	UNIQUE NOT NULL
PRIMARY KEY (Symbol)		

Table A.1: Entity `Element`.

Isotope		
column name	*data type*	*constraint*
Element symbol	VARCHAR(5)	NOT NULL
Mass number	INT	NOT NULL
Name	VARCHAR(50)	NOT NULL
Atomic weight	FLOAT	NOT NULL
Most abundant	BOOL	NOT NULL DEFAULT FALSE
Abundance	FLOAT	NOT NULL DEFAULT 0
PRIMARY KEY (Symbol)		
FOREIGN KEY (Element symbol) REFERENCES Element (Symbol)		

Table A.2: Entity `Isotope`.

Process type		
column name	*data type*	*constraint*
Type	VARCHAR(50)	NOT NULL
PRIMARY KEY (Type)		

Table A.3: Entity `Process_type`.

Substance		
column name	*data type*	*constraint*
Formula	VARCHAR(256)	NOT NULL
Density	FLOAT	NOT NULL
Name	VARCHAR(256)	—
PRIMARY KEY (Formula, Density)		

Table A.4: Entity `Substance`.

Compound		
column name	*data type*	*constraint*
Name	VARCHAR(100)	NOT NULL
PRIMARY KEY (Name)		

Table A.5: Entity `Compound`.

Isotope in substance		
column name	*data type*	*constraint*
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
Substance_formula	VARCHAR(256)	NOT NULL
Substance_density	FLOAT	NOT NULL
PRIMARY KEY(Element_symbol, Isotope_mass_number, Substance_formula)		
FOREIGN KEY (Element_symbol) REFERENCES Element (Symbol)		
FOREIGN KEY (Isotope_mass_number) REFERENCES Isotope (Mass_number)		
FOREIGN KEY (Substance_formula, Substance_density) REFERENCES Substance (Formula, Density)		

Table A.6: Entity `Isotope_in_substance`.

Substance in compound		
column name	*data type*	*constraint*
Compound_name	VARCHAR(100)	NOT NULL
Substance_formula	VARCHAR(256)	NOT NULL
Substance_density	FLOAT	NOT NULL
Percentage	FLOAT	NOT NULL
PRIMARY KEY (Compound_name, Substance_formula, Substance_density)		
FOREIGN KEY (Compound_name) REFERENCES Compound (Name)		
FOREIGN KEY (Isotope_mass_number) REFERENCES Isotope (Mass_number)		
FOREIGN KEY (Substance_formula, Substance_density) REFERENCES Substance (Formula, Density)		

Table A.7: Entity `Substance_in_compound`.

Gamma process type		
column name	*data type*	*constraint*
Type	VARCHAR(50)	NOT NULL
PRIMARY KEY (Type)		

Table A.8: Entity `Gamma_process_type`.

Neutron cross section		
column name	*data type*	*constraint*
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
Type	VARCHAR(50)	NOT NULL
Energy	FLOAT	NOT NULL
Value	FLOAT	NOT NULL
Is_interpolated	BOOL	NOT NULL DEFAULT FALSE
Origin	VARCHAR(50)	—
PRIMARY KEY (Element_symbol, Isotope_mass_number, Energy, Type)		
FOREIGN KEY (Type) REFERENCES Process_type (Type)		
FOREIGN KEY (Element_symbol, Isotope_mass_number) REFERENCES Isotope (Element_symbol, Mass_number)		

Table A.9: Entity `Neutron_cross_section`.

Neutron angular distribution		
column name	*data type*	*constraint*
Type	VARCHAR(50)	NOT NULL
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
Energy	FLOAT	NOT NULL
Number	INT	NOT NULL
Coefficient	INT	NOT NULL
PRIMARY KEY (Type, Element_symbol, Isotope_mass_number, Energy, Number)		
FOREIGN KEY (Element_symbol, Isotope_mass_number)) REFERENCES Isotope (Element_symbol, Mass_number)		
FOREIGN KEY (Type) REFERENCES Process_type (Type)		

Table A.10: Entity `Neutron_angular_distribution`.

Max probability neutron angular distribution		
column name	*data type*	*constraint*
Type	VARCHAR(50)	NOT NULL
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
Energy	FLOAT	NOT NULL
Value	INT	NOT NULL
PRIMARY KEY (Type, Element_symbol, Isotope_mass_number, Energy, Number)		
FOREIGN KEY (Element_symbol, Isotope_mass_number)) REFERENCES Isotope (Element_symbol, Mass_number)		
FOREIGN KEY (Type) REFERENCES Process_type (Type)		

Table A.11: Entity `Max_probability_neutron_angular_distribution`.

Energy level		
column name	*data type*	*constraint*
Number	INT	NOT NULL
Energy	FLOAT	NOT NULL
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
PRIMARY KEY (Number, Element_symbol, Isotope_mass_number)		
FOREIGN KEY (Element_symbol, Isotope_mass_number)) REFERENCES Isotope (Element_symbol, Mass_number)		

Table A.12: Entity `Energy_level`.

Inelastic gamma quantum production multiplicity		
column name	*data type*	*constraint*
Number	INT	NOT NULL
Energy	FLOAT	NOT NULL
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
PRIMARY KEY (Number, Element_symbol, Isotope_mass_number)		
FOREIGN KEY (Element_symbol, Isotope_mass_number)) REFERENCES Isotope (Element_symbol, Mass_number)		

Table A.13: Entity `Inelastic_gamma_quantum_production_multiplicity`.

Neutron capture gamma quantum production multiplicity		
column name	*data type*	*constraint*
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
Energy	FLOAT	NOT NULL
Value	FLOAT	NOT NULL
PRIMARY KEY (Element_symbol, Isotope_mass_number, Energy)		
FOREIGN KEY (Element_symbol, Isotope_mass_number)) REFERENCES Isotope (Element_symbol, Mass_number)		

Table A.14: Entity `Neutron_capture_gamma_quantum_production_multiplicity`.

Generator reaction		
column name	*data type*	*constraint*
Type	VARCHAR(50)	NOT NULL
Projectile_element_symbol	VARCHAR(5)	NOT NULL
Projectile_mass_number	INT	NOT NULL
Target_element_symbol	VARCHAR(5)	NOT NULL
Target_mass_number	INT	NOT NULL
Secondary_product_element_symbol	VARCHAR(5)	NOT NULL
Secondary_product_mass_number	INT	NOT NULL
Energy	FLOAT	NOT NULL
PRIMARY KEY (Type)		
FOREIGN KEY (Projectile_element_symbol, Projectile_mass_number, Target_element_symbol, Target_mass_number, Secondary_product_element_symbol, Secondary_product_mass_number) REFERENCES Isotope (Element_symbol, Atomic_mass_number, Element_symbol, Atomic_mass_number, Element_symbol, Atomic_mass_number)		

Table A.15: Entity `Generator_reaction`.

Gamma cross section		
column name	*data type*	*constraint*
Element_symbol	VARCHAR(5)	NOT NULL
Isotope_mass_number	INT	NOT NULL
Type	VARCHAR(50)	NOT NULL
Energy	FLOAT	NOT NULL
Value	FLOAT	NOT NULL
Origin	VARCHAR(50)	—
PRIMARY KEY (Element_symbol, Energy, Type)		
FOREIGN KEY (Type) REFERENCES Gamma_Process_type (Type)		
FOREIGN KEY (Element_symbol) REFERENCES Isotope (Element_symbol)		

Table A.16: Entity `Gamma_cross_section`.

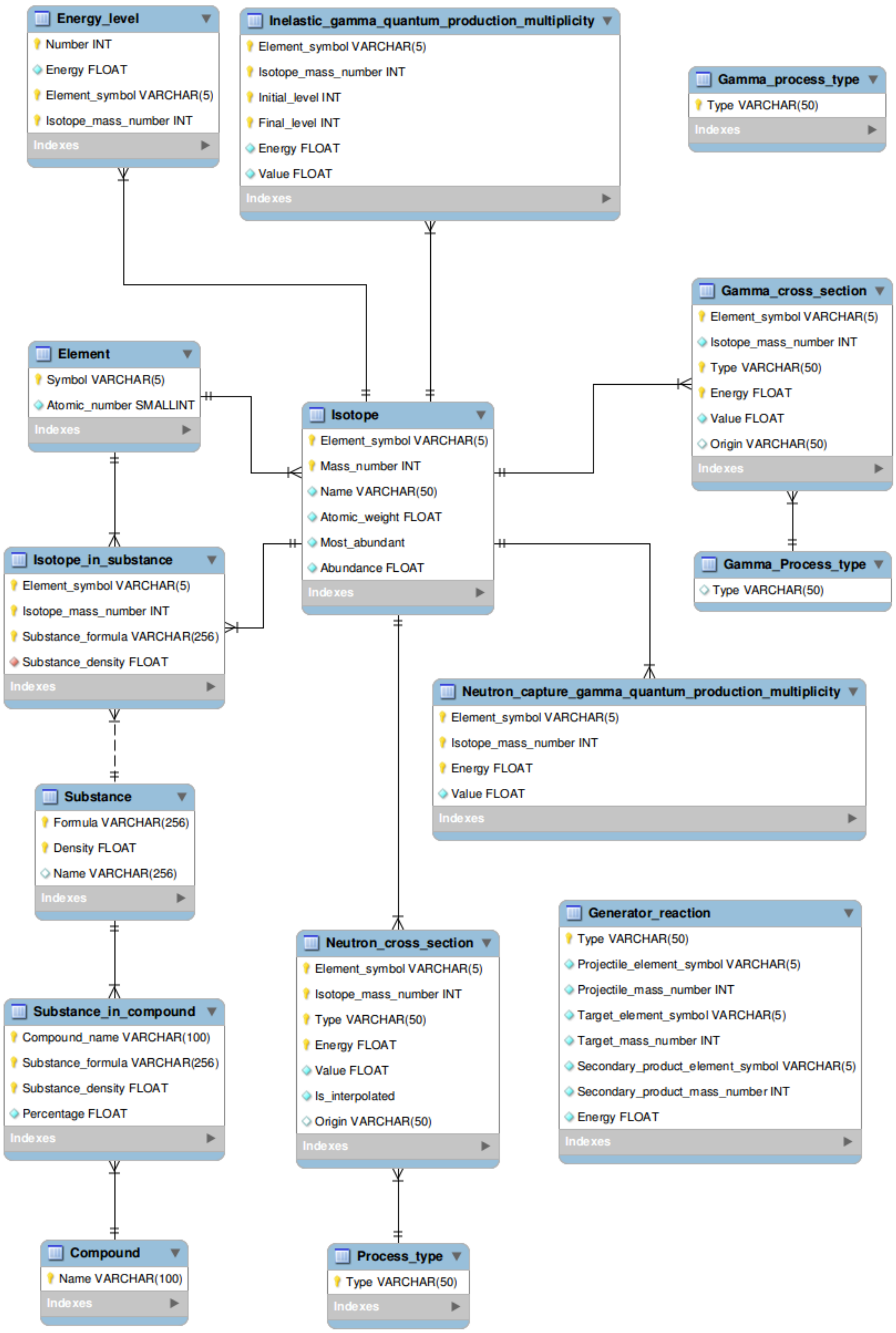


Figure A.2: Enhanced entity–relationship (EER) diagram of the database schema.

Appendix B

DVD with the SABAT package source code and documentation

This thesis is distributed with the DVD containing the whole simulation C++ source code with its documentation generated with the help of Doxygen. The documentation is held in the directory `docs` within the root folder.

Listings

List of Figures

Bibliography

[1] SABAT — Stoichiometry Analysis by Activation Techniques: http://koza.if.uj.edu.pl/sabat/, [Online; accessed 28-July-2015].

[2] M. Silarski and P. Moskal. *Atometria jako metoda wykrywania substancji niebezpiecznych.* Foton, 112:15-22, Wiosna 2011.

[3] P. Moskal. *Nuclear physics in medicine, minefield and kitchen.* Annales UMCS Physica 66:71-88, 2012.

[4] M. Silarski and P. Moskal. Patent application No. P409388 (2014); PCT/PL2015/050021

[5] M. Silarski for the SABAT Collaborration. *Application of Neutron Activation Spectroscopy.* Acta Physica Polonica B, 6:1061-1066, 2013.

[6] M. Silarski, D. Hunik, P. Moskal, M. Smolis and S. Tadeja. *Project of the underwater system for chemical threat detection.* Acta Physica Polonica A, 127:1543, 2015.

[7] M. Silarski, D. Hunik, P. Moskal, M. Smolis and S. Tadeja. *Design of the SABAT system for underwater detection of dangerous substances.* Acta Physica Polonica B, 47(2):497-502, 2016.

[8] Compact Neutron Generator. Phoenix Nuclear Labs, website: http://phoenixnuclearlabs.com/library-article/compact-neutron-generator/, [Online; accessed 09-July-2016].

[9] M. Smolis. Magister's Thesis: *Zrównoleglenie i optymalizacja algorytmów w pakiecie symulacji wykrywacza materiałów wybuchowych.* Jagiellonian University, 2015.

[10] A. Ferrari, P.R. Sala, A. Fasso, J. Ranft. *FLUKA: a multi-particle transport code.* CERN-2005-10, INFN/TC_05/11, SLAC-R-773, 2005.

[11] T.T. Böhlen, F. Cerutti, M.P.W. Chin, A. Fassò, A. Ferrari, P.G. Ortega, A. Mairani, P.R. Sala, G. Smirnov and V. Vlachoudis. *The FLUKA Code: Developments and Challenges for High Energy and Medical Applications.* Nuclear Data Sheets, 120:211-214, 2014.

[12] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, ... and D. Zschiescheaf. *Geant4 — a simulation toolkit.* Nuclear Instruments and Methods in Physics Research, 506:250–303, 2003.

[13] J. Allison, K. Amako, J. Apostolakis, H. Araujo, P.A. Dubois, M. Asai, ... and H. Yoshida. *Geant4 Developments and Applications.* IEEE Transactions on Nuclear Science, 53:270-278, 2006.

[14] N.V. Mokhov and C.C. James. *The MARS Code System User's Guide Version 15(2014)*. Google Inc. Google C++ Style Guide, website: http://www-ap.fnal.gov/MARS/m1514-manual.pdf, [Online; accessed 06-August-2015].

[15] Cartesian coordinate system — Wikipedia, The Free Encyclopedia: https://upload.wikimedia.org/wikipedia/commons/6/64/Coord_XYZ.svg, [Online; accessed 17-May-2016].

[16] Linear combination — Wikipedia, The Free Encyclopedia: https://upload.wikimedia.org/wikipedia/commons/f/fd/3D_Vector.svg, [Online; accessed 17-May-2016].

[17] Zbigniew H. Nitecki *Calculus in 3D: Geometry, Vectors, and Multivariate Calculus* Tufts University, August 19, 2012.

[18] P. Bourke, website: http://paulbourke.net/geometry/, [Online; accessed 02-July-2016].

[19] D. Sunday. *Geometry Algorithms Home*, website: http://geomalgorithms.com/, [Online; accessed 02-July-2016].

[20] CSG tree — Wikipedia, The Free Encyclopedia: https://upload.wikimedia.org/wikipedia/commons/8/8b/Csg_tree.png, [Online; accessed 21-June-2016].

[21] Mesh overview — Wikipedia, The Free Encyclopedia: https://upload.wikimedia.org/wikipedia/commons/6/6d/Mesh_overview.svg, [Online; accessed 16-June-2016].

[22] Dolphin triangle mesh — Wikipedia, The Free Encyclopedia: https://upload.wikimedia.org/wikipedia/commons/f/fb/Dolphin_triangle_mesh.png, [Online; accessed 16-June-2016].

[23] Cuboid — Wikipedia, The Free Encyclopedia: https://commons.wikimedia.org/wiki/File:Cuboid_simple.svg, [Online; accessed 16-June-2016].

[24] Truncated square pyramid — Wikipedia, The Free Encyclopedia: https://upload.wikimedia.org/wikipedia/commons/6/67/Pyramide_tronquee.svg, [Online; accessed 16-June-2016].

[25] Tomas Nikodym. Bachelor's thesis: *Ray Tracing Algorithm For Interactive Applications*. Czech Technical University in Prague, 2010.

[26] KD Trees for Faster Ray Tracing. Blog FrogSlayer: http://blog.frogslayer.com/kd-trees-for-faster-ray-tracing-with-triangles/, [Online; accessed 24-June-2016].

[27] J. D. Hunter. *Matplotlib: A 2D graphics environment*. Computing In Science & Engineering 3(9):90-95, EEE COMPUTER SOC, 2007.

[28] AutoCAD, website: http://www.autodesk.com/products/autocad/overview, [Online; accessed 11-June-2016].

[29] Blender, website: https://www.blender.org/, [Online; accessed 11-June-2016].

[30] fcmp website: http://fcmp.sourceforge.net/, [Online; accessed 27-Ma-2016].

[31] Google Inc. Google C++ Style Guide, website: https://google.github.io/styleguide/cppguide.html, [Online; accessed 28-May-2016].

[32] Doxygen, website: http://www.stack.nl/~dimitri/doxygen/, [Online; accessed 09-September-2015].

[33] Ray tracing diagram — Wikipedia, The Free Encyclopedia: https://upload.wikimedia.org/wikipedia/commons/8/83/Ray_trace_diagram.svg, [Online; accessed 31-July-2015].

[34] T.Möller, B.Trumbore *Fast, Minimum Storage Ray/Triangle Intersection.* Journal of Graphics Tools, 1997.

[35] Möller-Trumbore intersection algorithm — Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm, [Online; accessed 12-June-2016].

[36] GNU gprof profiler: https://sourceware.org/binutils/docs/gprof/, [Online; accessed 17-June-2016].

[37] SQLite database engine, official website: https://www.sqlite.org/index.html, [Online; accessed 28-July-2015].

[38] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM 13 (6): 377, 1970.

[39] Oracle Corporation. MySQL database engine, official website: https://www.mysql.com/, [Online; accessed 28-July-2015].

[40] Oracle Corporation. MariaDB database engine, official website: https://mariadb.com/, [Online; accessed 28-July-2015].

[41] N. Otuka, E. Dupont, V. Semkova, B. Pritychenko, A.I. Blokhin, M. Aikawa, ... and Y. Zhuang. *Towards a More Complete and Accurate Experimental Nuclear Reaction Data Library (EXFOR): International Collaboration Between Nuclear Reaction Data Centres (NRDC).* Nuclear Data Sheets, 120:272–276, 2014.

[42] NIST Physical Measurement Laboratory. NIST XCOM: Photon Cross Sections Database, website: http://physics.nist.gov/PhysRefData/Xcom/Text/intro.html, [Online; accessed 31-July-2015].

[43] Oracle Corporation. MySQL Workbench: https://www.mysql.com/products/workbench/, [Online; accessed 18-May-2016].

[44] Scott Chacon, Ben Straub. *Pro Git 2nd ed. 2014 Edition.* Pro Git online version: https://git-scm.com/book/en/v2, [Online; accessed 19-May-2016].

[45] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, ... and T. S. Woodall. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation.* In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004. Official website: https://www.open-mpi.org/, [Online; accessed 19-May-2016].

[46] Tanner Lovelace. *CMake: The Cross Platform Build System.* Linux Magazine, July 2006.

[47] Scott Chacon, Ben Straub. *Pro Git 2nd ed. 2014 Edition.* ROOT website: https://root.cern.ch/, [Online; accessed 30-May-2016].