Collision detection for 3D objects based on the Fourier-Motzkin elimination method

P. Baillehache

July 10, 2013

Abstract

This article describes how to perform efficient and accurate detection of collision of 3D objects approximated by bounding parallelepipeds and tetrahedrons whose intersection volume is calculated by the Fourier-Motzkin elimination method.

1 Introduction

3D video games represent physical objects using geometries approximating their shapes more or less accurately due to memory and computational power limitations on the machine running the game. These geometries are generally a collection of primitives like polygon meshes, spheres or parallelepipeds which provides an efficient way to store the geometry in memory and apply calculus on it.

To simulate the physics rules of the game's world, one must perform several calculus on these geometries. They are generally very expensive in terms of computation time and one must find a way to perform them in a limited time to give the impression of a real-time reaction to user's or environment's actions. Nowadays, ordinary machines have not enough computational power to perform them in this limited time as soon as the number of geometries exceeds a rather low quantity compare to what's needed to simulate the real world. One must then find methods to calculate the best approximation of the solution in the available time.

One of the physical phenomena whose simulation is needed is the detection of collisions between objects. Binary space partitioning or bounding volumes are common methods to approximate a solution to this problem. In these methods, the space is recursively divided by hyperplanes or volumes down to primitives. The tree structure associated with this space division is

then used to identify primitives susceptible to collide and perform collision detection only on these ones, which reduces greatly the number of test by avoiding trying each pair of primitives. If necessary, the collision test can also be approximated by the test on a higher level in the tree, instead of going down to the primitives, which makes these methods adaptable to variation of available computational time.

This article describes the Fourier-Motzkin elimination method applied to a bounding volumes (precisely, parallelepipeds and tetrahedrons) space partitioning to test collision between these volumes. I will first introduce a detailed description of the problem, then introduce the Fourier-Motzkin elimination method and how to apply it to the collision detection problem. Next, I will give an implementation of the resolution of the problem using this method. Finally I will give validation results and performance estimate.

2 Problem description

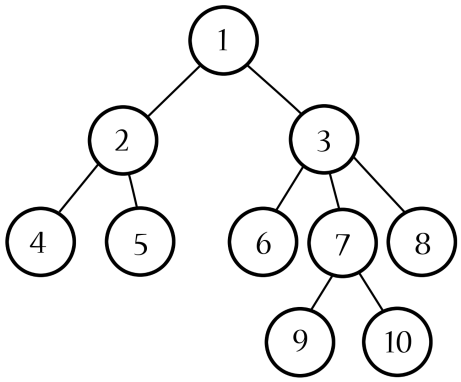
2.1 Space partitioning

With the view to detect collision between objects, the space is divided in volumes where there are objects susceptible to collide with others. These volumes are defined by a collection of mathematical objects approximating objects' geometry, while being handy for the collision test calculus. Space is divided at several levels of accuracy organized hierarchically. The hierarchy between accuracy levels is such as it is possible to know at one level the several bounding volumes which refine one bounding volume at an upper level. At one given level of accuracy, the bounding volumes describing one given object may be overlapping but as much as possible should be choosen in a way they are not. If a volume is contained by several bounding volumes, the collision test will occur as many time as there are bounding volumes containing it, which is unnecessary and should be avoided.

In pratical terms, the space partition is a recursive partition by volumes (parallelepipeds and tetrahedrons) stored in a tree structure. A the top level there is a first node representing the whole world (Fig. 1, (1)). From the point of view of this article, this node has the only purpose to give a root to the tree structure and a general basis to express coordinates of all volumes. On the second level, there is one node per object (Fig. 1, (2) and (3)). These nodes refer to the smallest volume bounding entirely each object. From the third level, each node refers to a set of volumes bounding more and more accurately the object geometry (Fig. 1, (4) to (10)). On each level the union of the volumes with a common parent entirely bounds the volume of

the object bounded by this parent (Fig. 1, (9) and (10) entirely contain the volume of (3) covered by (7)), and the volumes of this level are entirely included in their parent (Fig. 1, (9) and (10) are entirely included in (7)). Each object may have a different number of bounding volumes, organized in a different number of sublevels.

Figure 1: Space partition tree example for 2 objects



Due to the inclusion relation between parents and childs' nodes, if two parents are not colliding all of their childs aren't colliding neither. Thus, if the collision test fails for two nodes, test between all of their childs is unnecessary (Fig. 1, if (2) and (3) don't collide with each other, neither do (4) to (10)). Lets make the hypothesis that an object can't collide with itself, thus it's unnecessary to test two objects having a common ancestor under the first level (Fig. 1, (6) to (10) never collide with each others). Also, it can be noted that one or several node level could be inserted between the first and second level on Fig. 1 to define groups of objects which never intersect with a member of the same group. This would allow to avoid even more tests. For example, in a game where a character moves in a building, a node level can be defined to represent each room. Then the first collision test on the highest level would return true only for the room the character is currently located in, and following collision test would be performed only on objects of this room. However I will only consider in this article the case where all objects are directly on the second level.

The information stored about the volume $\mathcal{A}$ are the coordinates of one vertex ($\overrightarrow{A}$), the three edge's vectors ($\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w}$) from this vertex, the type of volume (parallelepiped or tetrahedron) and its bounding area. Vertex and vectors' coordinates must be expressed in the world's basis. Vectors must not be normalized. The bounding volume is defined as the 6 values $x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}$, such as

$$\forall (x,y,z) \in [0.0,1.0]^3 : \overrightarrow{(x_{min}, y_{min}, z_{min})} \leq \overrightarrow{A} + x.\overrightarrow{u} + y.\overrightarrow{v} + z.\overrightarrow{w} \leq \overrightarrow{(x_{max}, y_{max}, z_{max})}$$

n.b. : From here to the end of this article the inequality between two vectors holds componentwise.

$$\overrightarrow{(a,b,c)} \leq \overrightarrow{(d,e,f)} \Leftrightarrow \left\{ \begin{array}{l} a \leq d \\ b \leq e \\ c \leq f \end{array} \right.$$

2.2 Collision detection

Once the space has been partitioned in volumes describing objects, the collision detection consists of looking for intersections between them. Lets consider two volumes $\mathcal{A}$ and $\mathcal{B}$, respectively defined by the vertices and edge vectors $(\overrightarrow{A}, \overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w})$ and $(\overrightarrow{B}, \overrightarrow{u'}, \overrightarrow{v'}, \overrightarrow{w'})$, with $\overrightarrow{u} = (u_1, u_2, u_3)$, $\overrightarrow{v} = (v_1, v_2, v_3)$, $\overrightarrow{w} = (w_1, w_2, w_3)$, $\overrightarrow{u'} = (u'_1, u'_2, u'_3)$, $\overrightarrow{v'} = (v'_1, v'_2, v'_3)$, $\overrightarrow{w'} = (w'_1, w'_2, w'_3)$, $\overrightarrow{A} = (A_1, A_2, A_3)$, $\overrightarrow{B} = (B_1, B_2, B_3)$. If $\mathcal{A}$ is a parallelepiped, the volume bounded by $\mathcal{A}$ is defined by

$$\left((x,y,z) \in [0.0,1.0]^3 : \overrightarrow{A} + x.\overrightarrow{u} + y.\overrightarrow{v} + z.\overrightarrow{w} \right) \tag{1}$$

If it's a tetrahedron, it becomes

$$\left((x,y,z) \in [0.0,1.0]^3 : \left\{ \begin{array}{l} \overrightarrow{A} + x.\overrightarrow{u} + y.\overrightarrow{v} + z.\overrightarrow{w} \\ x + y + z \leq 1.0 \end{array} \right. \right) \tag{2}$$

And, if $\mathcal{B}$ is a parallelepiped, the volume bounded by $\mathcal{B}$ is defined by

$$\left((x',y',z') \in [0.0,1.0]^3 : \overrightarrow{B} + x'.\overrightarrow{u'} + y'.\overrightarrow{v'} + z'.\overrightarrow{w'} \right) \tag{3}$$

If it's a tetrahedron, it becomes

$$\left((x',y',z') \in [0.0,1.0]^3 : \left\{ \begin{array}{l} \overrightarrow{B} + x'.\overrightarrow{u'} + y'.\overrightarrow{v'} + z'.\overrightarrow{w'} \\ x' + y' + z' \leq 1.0 \end{array} \right. \right) \tag{4}$$

If I express $\mathcal{B}$ in $\mathcal{A}$'s basis as $(\overrightarrow{B}_{\mathcal{A}}, \overrightarrow{u'_{\mathcal{A}}}, \overrightarrow{v'_{\mathcal{A}}}, \overrightarrow{w'_{\mathcal{A}}})$, the area bounded by $\mathcal{B}$ in $\mathcal{A}$'s basis becomes

$$\left((x',y',z') \in [0.0,1.0]^3 : \left\{ \begin{array}{l} \overrightarrow{B_{\mathcal{A}}} + x'.\overrightarrow{u'_{\mathcal{A}}} + y'.\overrightarrow{v'_{\mathcal{A}}} + z'.\overrightarrow{w'_{\mathcal{A}}} \\ x' + y' + z' \leq 1.0 \text{ if } \mathcal{B} \text{ is a tetrahedron.} \end{array} \right. \right) \tag{5}$$

where

$$\left(\begin{pmatrix} \\ \overrightarrow{u'_{\mathcal{A}}} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{v'_{\mathcal{A}}} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{w'_{\mathcal{A}}} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{B_{\mathcal{A}}} \\ \\ \end{pmatrix}\right) =$$
$$\left(\begin{pmatrix} \\ \overrightarrow{u} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{v} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{w} \\ \\ \end{pmatrix}\right)^{-1} . \left(\begin{pmatrix} \\ \overrightarrow{u'} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{v'} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{w'} \\ \\ \end{pmatrix}\begin{pmatrix} \\ \overrightarrow{B} - \overrightarrow{A} \\ \\ \end{pmatrix}\right) \tag{6}$$

Then $\mathcal{A}$ and $\mathcal{B}$ intersect if and only if

$$\begin{array}{l} \exists (x', y', z') \in [0.0, 1.0]^3 : \\ \left\{ \begin{array}{l} \overrightarrow{0} \leq \overrightarrow{B_{\mathcal{A}}} + x'.\overrightarrow{u_{\mathcal{A}}} + y'.\overrightarrow{v_{\mathcal{A}}} + z'.\overrightarrow{w_{\mathcal{A}}} \leq \overrightarrow{1} \\ \\ \left(\overrightarrow{B_{\mathcal{A}}} + \overrightarrow{(x', y', z')} \circ (\overrightarrow{u_{\mathcal{A}}} + \overrightarrow{v_{\mathcal{A}}} + \overrightarrow{w_{\mathcal{A}}})\right) . \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \leq 1.0 \text{ (a)} \\ \\ x' + y' + z' \leq 1.0 \text{ (b)} \end{array} \right. \\ \text{(a) : if } \mathcal{A} \text{ is a tetrahedron.} \\ \text{(b) : if } \mathcal{B} \text{ is a tetrahedron.} \end{array} \tag{7}$$

Lets define $\overrightarrow{m} = (m_1, m_2, m_3)$ and $\overrightarrow{n} = (n_1, n_2, n_3)$ as

$$\begin{array}{c} \overrightarrow{m} = -\overrightarrow{B_{\mathcal{A}}} \\ \overrightarrow{n} = \overrightarrow{1} - \overrightarrow{B_{\mathcal{A}}} \end{array} \tag{8}$$

Then (7) is equivalent to

$$\begin{array}{l} \exists (x', y', z') \in [0.0, 1.0]^3 : \\ \left\{ \begin{array}{l} m_1 \leq x'.u'_{\mathcal{A}1} + y'.v'_{\mathcal{A}1} + z'.w'_{\mathcal{A}1} \leq n_1 \\ m_2 \leq x'.u'_{\mathcal{A}2} + y'.v'_{\mathcal{A}2} + z'.w'_{\mathcal{A}2} \leq n_2 \\ m_3 \leq x'.u'_{\mathcal{A}3} + y'.v'_{\mathcal{A}3} + z'.w'_{\mathcal{A}3} \leq n_3 \\ x'.\left(u'_{1\mathcal{A}} + v'_{1\mathcal{A}} + w'_{1\mathcal{A}}\right) + \\ y'.\left(u'_{2\mathcal{A}} + v'_{2\mathcal{A}} + w'_{2\mathcal{A}}\right) + \\ z'.\left(u'_{3\mathcal{A}} + v'_{3\mathcal{A}} + w'_{3\mathcal{A}}\right) \leq 1.0 + m_1 + m_2 + m_3 \text{ (a)} \\ x' + y' + z' \leq 1.0 \text{ (b)} \end{array} \right. \\ \text{(a) : if } \mathcal{A} \text{ is a tetrahedron.} \\ \text{(b) : if } \mathcal{B} \text{ is a tetrahedron.} \end{array} \tag{9}$$

Finally the volumes $\mathcal{A}$ and $\mathcal{B}$ intersects if and only if the following system of linear inequalities has a solution, and the solutions of this system defines

the intersection volume between $\mathcal{A}$ and $\mathcal{B}$ in $\mathcal{A}$'s basis:

$$
\begin{array}{l}
(x_1, x_2, x_3) \in \Re^3 : \\
\left\{ \begin{array}{llll}
a_{11}.x_1 & +a_{12}.x_2 & +a_{13}.x_3 & \leq b_1 \\
a_{21}.x_1 & +a_{22}.x_2 & +a_{23}.x_3 & \leq b_2 \\
a_{31}.x_1 & +a_{32}.x_2 & +a_{33}.x_3 & \leq b_3 \\
a_{41}.x_1 & +a_{42}.x_2 & +a_{43}.x_3 & \leq b_4 \\
a_{51}.x_1 & +a_{52}.x_2 & +a_{53}.x_3 & \leq b_5 \\
a_{61}.x_1 & +a_{62}.x_2 & +a_{63}.x_3 & \leq b_6 \\
x_1 & & & \leq 1.0 \\
-x_1 & & & \leq 0.0 \\
 & x_2 & & \leq 1.0 \\
 & -x_2 & & \leq 0.0 \\
 & & x_3 & \leq 1.0 \\
 & & -x_3 & \leq 0.0 \\
(\sum_{i=1}^{3} a_{1i}).x_1 & +(\sum_{i=1}^{3} a_{3i}).x_2 & +(\sum_{i=1}^{3} a_{5i}).x_3 & \leq 1.0 - b_2 - b_4 - b_6 \text{ (a)} \\
x_1 & +x_2 & +x_3 & \leq 1.0 \text{ (b)}
\end{array} \right. \\
\text{(a) : if } \mathcal{A} \text{ is a tetrahedron.} \\
\text{(b) : if } \mathcal{B} \text{ is a tetrahedron.}
\end{array}
\tag{10}
$$

with

$$
\begin{array}{cccc}
a_{11} = u'_{\mathcal{A}1} & a_{12} = v'_{\mathcal{A}1} & a_{13} = w'_{\mathcal{A}1} & b_1 = n_1 \\
a_{21} = -u'_{\mathcal{A}1} & a_{22} = -v'_{\mathcal{A}1} & a_{23} = -w'_{\mathcal{A}1} & b_2 = -m_1 \\
a_{31} = u'_{\mathcal{A}2} & a_{32} = v'_{\mathcal{A}2} & a_{33} = w'_{\mathcal{A}2} & b_3 = n_2 \\
a_{41} = -u'_{\mathcal{A}2} & a_{42} = -v'_{\mathcal{A}2} & a_{43} = -w'_{\mathcal{A}2} & b_4 = -m_2 \\
a_{51} = u'_{\mathcal{A}3} & a_{52} = v'_{\mathcal{A}3} & a_{53} = w'_{\mathcal{A}3} & b_5 = n_3 \\
a_{61} = -u'_{\mathcal{A}3} & a_{62} = -v'_{\mathcal{A}3} & a_{63} = -w'_{\mathcal{A}3} & b_6 = -m_3
\end{array}
\tag{11}
$$

3 Fourier-Motzkin elimination method

The Fourier-Motzkin elimination method has been introduced by J.J.-B. Fourier in 1827 [1], and described in the Ph.D. thesis of T.S. Motzkin in 1936 [2]. This is a generalization of the Gaussian elimination method to linear systems of inequalities. This method consists of eliminating one variable of the system and rewrite a new system accordingly. Then the elimination operation is repeated on another variable in the new system, and so on until we obtain a trivial system with only one variable. From there, a solution for each variable can be obtained if it exists. The variable elimination is performed as follow.

Lets write the linear system of inequalities $\mathcal{I}$ as

$$\left\{\begin{array}{llll} a_{11}.x_1 & +a_{12}.x_2 & +\cdots & +a_{1n}.x_n \quad \leq b_1 \\ a_{21}.x_1 & +a_{22}.x_2 & +\cdots & +a_{2n}.x_n \quad \leq b_2 \\ & & \vdots & \\ a_{m1}.x_1 & +a_{m2}.x_2 & +\cdots & +a_{mn}.x_n \quad \leq b_m \end{array}\right. \tag{12}$$

with

$$\begin{array}{c} i \in 1, 2, ..., m \\ j \in 1, 2, ..., n \\ x_i \in \Re \\ a_{ij} \in \Re \\ b_j \in \Re \end{array} \tag{13}$$

To eliminate the first variable x_1, lets multiply each inequality by $1.0/|a_{i1}|$ where $a_{i1} \neq 0.0$. (10) becomes

$$\left\{\begin{array}{llll} x_1 & +a'_{i2}.x_2 & +\cdots & +a'_{in}.x_n \quad \leq b'_i \quad (i \in \mathcal{I}_+) \\ & a_{i2}.x_2 & +\cdots & +a_{in}.x_n \quad \leq b_i \quad (i \in \mathcal{I}_0) \\ -x_1 & +a'_{i2}.x_2 & +\cdots & +a'_{in}.x_n \quad \leq b'_i \quad (i \in \mathcal{I}_-) \end{array}\right. \tag{14}$$

where $\mathcal{I}_+ = \{i : a_{i1} > 0.0\}$, $\mathcal{I}_0 = \{i : a_{i1} = 0.0\}$, $\mathcal{I}_- = \{i : a_{i1} < 0.0\}$, $a'_{ij} = a_{ij}/|a_{i1}|$ and $b'_i = b_i/|a_{i1}|$. Then $x_1, x_2, \cdots, x_n \in \Re^n$ is a solution of $\mathcal{I}$ if and only if

$$\left\{\begin{array}{c} \sum_{j=2}^{n}((a'_{kj} + a'_{lj}).x_j) \leq b'_k + b'_l \quad (k \in \mathcal{I}_+, l \in \mathcal{I}_-) \\ \sum_{j=2}^{n}(a_{ij}.x_j) \leq b_i \quad i \in \mathcal{I}_0 \end{array}\right. \tag{15}$$

and

$$\max_{l \in \mathcal{I}_-}(\sum_{j=2}^{n}(a'_{lj}.x_j) - b'_l) \leq x_1 \leq \min_{k \in \mathcal{I}_+}(b'_k - \sum_{j=2}^{n}(a'_{kj}.x_j)) \tag{16}$$

The same method is then applied on the system defined by (15) to eliminate the second variable x_2, and so on until we reach the inequality

$$\max_{l \in \mathcal{I}''^{\cdots'}_-}(-b''^{\cdots'}_l) \leq x_n \leq \min_{k \in \mathcal{I}''^{\cdots'}_+}(b''^{\cdots'}_k) \tag{17}$$

If (17) has no solution, then neither the system $\mathcal{I}$. If it has a solution, the minimum and maximum are the bounding values for the variable x_n. One can get a particular solution to the system $\mathcal{I}$ by choosing a value for x_n between these bounding values, which allow us to set a particular value for the variable x_{n-1} using (16), and so on recursively up to x_1. Eventually (16) may lead to no solution for some variables, then neither the system $\mathcal{I}$ has a solution.

4 Application of the Fourier-Motzkin elimination method to collision detection

The Fourier-Motzkin elimination method can be applied as it is to the system (10) defining the collision detection problem. Then, the bounding values for x_n in (17) defines the largest interval of values possible for x_n if the system has a solution. Then if one doesn't look for a particular solution to the system but for a set of bounding values for each variables, one can apply the elimination method from (12) again with a different set of n-1 variables eliminated to obtain the bounding values for another variable. And so on until we obtain the bounding values for each variables. The set of bounding values defines a volume containing all the solutions of the system if it has a solution, eventually bigger than the actual volume of solutions of the system.

It is sufficient from the point of view of the collision detection algorithm which doesn't look for a particular solution. However one still needs to determine if the system actually has a solution. Rely only on the existence of these bounding values can generate false positives (collision detected where there isn't) as shown in the example of the two following tetrahedrons :

$$\begin{array}{l|l} \vec{A} = (0.5, 0.5, 0.5) & \vec{B} = (-0.6, -0.6, -0.6) \\ \vec{u} = (-1.0, 0.0, 0.0) & \vec{u'} = (1.0, 0.0, 0.0) \\ \vec{v} = (0.0, -1.0, 0.0) & \vec{v'} = (0.0, 1.0, 0.0) \\ \vec{w} = (0.0, 0.0, -1.0) & \vec{w'} = (0.0, 0.0, 1.0) \end{array} \tag{18}$$

In this case there are bounding values satisfying the inequations system for each variable independently ((0.3-0.8),(0.3-0.8),(0.3-0.8)), but there is actually no solution to the system.

As the bounding values respect the inequations of the system independently, they define a volume included in the paralellepipeds $(\vec{A}, \vec{u}, \vec{v}, \vec{w})$ and $(\vec{B}, \vec{u'}, \vec{v'}, \vec{w'})$. If the volumes we are testing for collision are actually parallelepipeds, it means the solution given by these bounding values is necessarily included in both parallelepipeds, then it can't be a false positive. So, only the case of tetrahedrons needs to be checked, meaning it only needs to be checked if it exists values in the set of bounding values verifying the inequation(s) (a) and/or (b) in (10).

If a collision is detected, the volume defined by the bounding values locates where the collision occurs. This can be used to reduce the number of nodes to be tested in the next level of the space partitionning tree.

5 Implementation

The resolution of the problem of collision detection consists of scanning the tree representing the space partition and applying the collision test on pairs of nodes using the Fourier-Motzkin elimination method. Thanks to test results, tree structure and node information, the algorithm avoids as much as possible unnecessary collision tests.

The detailed algorithm depends on the context it is used. For example, should the algorithm look for all collisions in the world, or only those with one particular object, or should it tests for collision existence only or returns the list of nodes collided. The algorithm introduced here as example searches collisions between one given object and all the others, and return the list of the deepest pairs of nodes collided.

5.1 Data structures

The tree node's data structure is defined as follow:

```
structure TreeNode
   Volume Vol
   TreeNode Child
   TreeNode Next
end TreeNode
```

The volume's data structure is defined as follow (coordinates in world's basis):

```
structure Volume
   Real A[0], A[1], A[2]
   Real U[0], U[1], U[2]
   Real V[0], V[1], V[2]
   Real W[0], W[1], W[2]
   Real Ui[0], Ui[1], Ui[2]
   Real Vi[0], Vi[1], Vi[2]
   Real Wi[0], Wi[1], Wi[2]
   BoundingArea BdgArea
   Integer Type
end Parallelepiped
```

Vectors `Ui`, `Vi`, `Wi` are used to store the inverse of the matrix composed by vectors `U`, `V`, `W`. `Type` represents the type of volume, parallelepiped or tetrahedron.

The bounding area's data structure is defined as follow:

```
structure BoundingArea
   Real Min[0], Min[1], Min[2]
   Real Max[0], Max[1], Max[2]
end BoundingArea
```

The details of the list's data structure are not introduced here and left to the responsibility of the programmer.

In what follows, the value of the member `M` of the structure `S` in the instance `I` is noted `I.M`.

5.2 Algorithm to search in the tree

Here is the algorithm of the procedure *SearchCollisions*:

```
01  Procedure SearchCollisions( P, T, L, Cl ) is
02    M:=FirstChildOf( T )
03    while Exist( M )
04      if Different( P, M ) is true then
05        if QuickTestCollision( M, Cl )
06          if TestCollisionBetween( P.Vol, M.Vol, CoW ) is true
07            if HasChild( P ) is true
08              N:=FirstChildOf( P )
09              while Exist( N ) is true
10                if QuickTestCollision( N.Vol, CoW ) is true
11                  SearchCollisions( N, M, L, CoW )
12                end if
13                N:=NextNode( N )
14              end while
15            else
16              if HasChild( M ) is true
17                SearchCollisions( P, M, L, CoW )
18              else
19                AddToCollisionList( L, P, M, CoW )
20              end if
21            end if
22          end if
23        end if
```

```
24      end if
25      M:=NextNode( M )
26    end while
27  end SearchCollisions
```

Comments on *SearchCollisions*:

(01) The procedure *SearchCollisions* takes 4 parameters in argument. P is the highest bounding box in the tree for the object to be compared to other objects. T is the root node of the tree structure. L is the list to store the collisions detected. Cl is a bounding box used by the algorithm, it must bound the whole world when the algorithm starts.

(02) Start the search with the first node M in the tree T.

(03) Perform the search as long as there is node to test against P.

(04) If the current object M is the object P the algorithm doesn't perform the test collision as we are interesting only in finding collisions with other objects.

(05) Perform a quick test to check if the object M may be in the bounding area Cl. If not, the object M can't collide the object P and there is no need to perform the collision test. The detailed of *QuickTestIntersection* is introduced below.

(06) If the algorithm reach here, it means there is a possibility for P and M to collide. Perform the test collision, which returns true if there is a collision and store in CoW the intersection's bounding volume expressed in world's basis, or return false if there is no collision. The detailed of *TestCollisionBetween* is given in the next subsection.

(07) If there has been a collision, the algorithm continues the test between child nodes of P and M. First it checks if P has childs.

(08) If P has childs, starts with the first child N.

(09) As long as there is child of P to test.

(10) If the child N of P is not in the collision's bounding area CoW, it can't be part of the collision between P and M and doesn't need to be tested agains M's childs.

(11) If the child N may be in CoW, search for collisions of N against M's childs (the subtree whose root is M), add the eventual collisions in L, and restrain the search to M's child inside the collision's bounding volume CoW.

(13) Continue the search with the next child of P.

(15) If P has no child

(16) If M has childs, continue the search between P and M's childs.

(17) Search for collisions of P against M's childs (the subtree whose root is M), add the eventual collisions in L, and restrain the search to M's child in the collision's bounding volume CoW.

(18) If M also has no childs, the algorithm has reached two deepest nodes on the tree which collide each other.

(19) Store the two collided nodes P and M references, and the bounding volumes of the collision in each node's basis in the list L.

(25) Continue the search with the next child of T

(27) End the procedure *SearchCollisions*. If collisions have been detected, the list L contains the collisions' information. If not, the list L is empty.

The procedure *QuickTestIntersection* performs an imprecise but very fast test of intersection between a node's volume and a bounding volume. It returns false if the intersection is necessarily empty, and true if the intersection is not necessarily empty. The principle is as follow. A volume doesn't intersect another if its minimum and maximum values are both inferior to the minimum values or superior to the maximum values of the other volume on at least one axis of the basis. Given one volume in 3D, it defines 6 areas where, if the other volume is entirely located, there can't be intersection between them. Using this test to define exclusion areas whose entirely included nodes doesn't need to be checked for collision against nodes entirely included in the bounding volume allows to reduce the number of collision test.

Here is the algorithm of the procedure *QuickTestIntersection*:

```
01  Procedure QuickTestIntersection( P, Cl ) is
02    for i:=0..2
03      if (P.BdgArea.Min[i]<Cl.Min[i] and P.BdgArea.Max[i]<Cl.Min[i])
..        or (P.BdgArea.Min[i]>Cl.Max[i] and P.BdgArea.Max[i]>Cl.Max[i])
04          return false
05      end if
06    end for
07    return true
08  end QuickTestIntersection
```

5.3 Algorithm to test collision of two nodes

The main procedure is given below and the subprocedures are following in alphabetical order.

```
01  Procedure TestCollisionBetween( P, Q, Co ) is
02    NbEq:=6
03    ChangeBasis( P, Q, Qp )
04    for i:=0..2
```

```
05      N[ 2*i ]:=1.0-Qp.A[ i ]
06      A[ 6*i ]:=Qp.U[ i ]
07      A[ 6*i+1 ]:=Qp.V[ i ]
08      A[ 6*i+2 ]:=Qp.W[ i ]
09      N[ 2*i+1 ]:=Qp.A[ i ]
10      A[ 6*i+3 ]:=-1.0*Qp.U[ i ]
11      A[ 6*i+4 ]:=-1.0*Qp.V[ i ]
12      A[ 6*i+5 ]:=-1.0*Qp.W[ i ]
13    end for
14    for i:=0..2
15      N[ NbEq ]:=1.0
16      A[ NbEq*3 ]:=0.0
17      A[ NbEq*3+1 ]:=0.0
18      A[ NbEq*3+2 ]:=0.0
19      A[ NbEq*3+i ]:=1.0
20      NbEq=NbEq+1
21      N[ NbEq ]:=0.0
22      A[ NbEq*3 ]:=0.0
23      A[ NbEq*3+1 ]:=0.0
24      A[ NbEq*3+2 ]:=0.0
25      A[ NbEq*3+i ]:=-1.0
26      NbEq=NbEq+1
27    end for
28    if IsTetrahedron( Q )
29      N[ NbEq ]:=1.0
30      A[ NbEq*3 ]:=1.0
31      A[ NbEq*3+1 ]:=1.0
32      A[ NbEq*3+2 ]:=1.0
33      NbEq:=NbEq+1
34    end if
35    if IsTetrahedron( P )
36      N[ NbEq ]:=1.0-Qp.A[0]-Qp.A[1]-Qp.A[2]
37      A[ NbEq*3 ]:=Qp.U[0]+Qp.V[0]+Qp.W[0]
38      A[ NbEq*3+1 ]:=Qp.U[1]+Qp.V[1]+Qp.W[1]
39      A[ NbEq*3+2 ]:=Qp.U[2]+Qp.V[2]+Qp.W[2]
40      NbEq:=NbEq+1
41    end if
42    if TestTrivialCases ( A, N, NbEq, 3 ) is false
43      return false
44    end if
45    ElimVar( 0, A, N, NbEq, 3, Ap, Np, Nb )
```

```
46    if TestTrivialCases ( Ap, Np, Nb, 2 ) is false
47      return false
48    end if
49    ElimVar( 0, Ap, Np, Nb, 2, App, Npp, Nbp )
50    if TestTrivialCases ( App, Npp, Nbp, 1 ) is false
51      return false
52    end if
53    GetBound( Co, Nbp, App, Npp, 2 )
54    if Co.Min[ 2 ]>=Co.Max[ 2 ]
55      return false
56    end if
57    ElimVar( 1, Ap, Np, Nb, 2, App, Npp, Nbp )
58    if TestTrivialCases ( App, Npp, Nbp, 1 ) is false
59      return false
60    end if
61    GetBound( Co, Nbp, App, Npp, 1 )
62    if Co.Min[ 1 ]>=Co.Max[ 1 ]
63      return false
64    end if
65    ElimVar( 1, A, N, NbEq, 3, Ap, Np, Nb )
66    if TestTrivialCases ( Ap, Np, Nb, 2 ) is false
67      return false
68    end if
69    ElimVar( 1, Ap, Np, Nb, 2, App, Npp, Nbp )
70    if TestTrivialCases ( App, Npp, Nbp, 1 ) is false
71      return false
72    end if
73    GetBound( Co, Nbp, App, Npp, 0 )
74    if Co.Min[ 0 ]>=Co.Max[ 0 ]
75      return false
76    end if
77    if IsTetrahedron( Q )
78      if Co.Min[ 0 ]+Co.Min[ 1 ]+Co.Min[ 2 ]>=1.0
79        return false
80      end if
81    end if
82    if IsTetrahedron( P ) and ConfirmSolution( Co, Qp )=false
83      return false
84    end if
85    ConvertToWorldBasis( Co, Q )
86    for i:=0..2
```

```
87      if Co.Min[ i ]<P.BdgArea.Min[ i ]
88        Co.Min[ i ]=P.BdgArea.Min[ i ]
89      end if
90      if Co.Max[ i ]>P.BdgArea.Max[ i ]
91        Co.Max[ i ]=P.BdgArea.Max[ i ]
92      end if
93    end for
94    return true
95  end TestCollisionBetween
```

Comments on *TestCollisionBetween*:

(01) The procedure *TestCollisionBetween* takes 3 arguments. P and Q are the two volumes to be tested for collision. Co contains the collisions' information at the end of the procedure in case there was a collision. In this case the procedure return the value true, else it returns the value false and the content of Co is undefined.

(02) Initialize a variable to store the number of inequation.

(03) Get the basis of Q expressed in P's basis.

(04)-(27) Initialize the coefficients of the matrix representing the system (10).

(28)-(34) If Q is a tetrahedron add the inequation (b) of the system (10).

(35)-(41) If P is a tetrahedron add the inequation (a) of the system (10).

(42)-(44) Test for trivial cases where the system has no solutions. The procedure *TestTrivialCases* is described below.

(45) Eliminate the first variable in the initial system. The procedure *ElimVar* is described below.

(46)-(48) Test for trivial cases where the new system has no solutions.

(49) Eliminate the first variable in the new system (corresponding to the second variable in the initial system).

(50)-(52) Test for trivial cases where the new system has no solutions.

(53) The resulting system as only one variable (the third one in the initial system). Calculate the bounding values for this variable. The procedure *GetBound* is described below.

(54)-(56) Check if the third variable of system has a solution. If not return false.

(57) Eliminate the second variable (corresponding to the third variable in the initial system) in the system resulting from line 45.

(58)-(60) Test for trivial cases where the new system has no solutions.

(61) The resulting system as only one variable (the second one in the initial system). Calculate the bounding values for this variable.

(62)-(64) Check if the second variable of the system has a solution. If not return false.

(65) Eliminate the second variable in the initial system.

(66)-(68) Test for trivial cases where the new system has no solutions.

(69) Eliminate the second variable in the new system (corresponding to the third variable in the initial system).

(70)-(72) Test for trivial cases where the new system has no solutions.

(73) The resulting system as only one variable (the first one in the initial system). Calculate the bounding values for this variable.

(74)-(76) Check if the first variable of the system has a solution. If not return false.

(77)-(84) Ensure the solution is not a false positive. The procedure *ConfirmSolution* is detailed below.

(85) Convert the bounding values in P's basis to bounding values in world's basis. The procedure *ConvertToWorldBasis* is detailed below.

(86)-(93) The set of bounding values is potentially bigger than the actual intersection area. By clipping this set with the bounding area of P, the algorithm can improve the bounding values.

(94) If we reached here it means there has been a collision detected and the information about it are stored in Co.

```
01  Procedure ChangeBasis( P, Q, Qp ) is
02  Qp.U[ 0 ]:=
..    P.Ui[ 0 ]*Q.U[ 0 ]+P.Vi[ 0 ]*Q.U[ 1 ]+P.Wi[ 0 ]*Q.U[ 2 ]
03  Qp.V[ 0 ]:=
..    P.Ui[ 0 ]*Q.V[ 0 ]+P.Vi[ 0 ]*Q.V[ 1 ]+P.Wi[ 0 ]*Q.V[ 2 ]
04  Qp.W[ 0 ]:=
..    P.Ui[ 0 ]*Q.W[ 0 ]+P.Vi[ 0 ]*Q.W[ 1 ]+P.Wi[ 0 ]*Q.W[ 2 ]
05  Qp.U[ 1 ]:=
..    P.Ui[ 1 ]*Q.U[ 0 ]+P.Vi[ 1 ]*Q.U[ 1 ]+P.Wi[ 1 ]*Q.U[ 2 ]
06  Qp.V[ 1 ]:=
..    P.Ui[ 1 ]*Q.V[ 0 ]+P.Vi[ 1 ]*Q.V[ 1 ]+P.Wi[ 1 ]*Q.V[ 2 ]
07  Qp.W[ 1 ]:=
..    P.Ui[ 1 ]*Q.W[ 0 ]+P.Vi[ 1 ]*Q.W[ 1 ]+P.Wi[ 1 ]*Q.W[ 2 ]
08  Qp.U[ 2 ]:=
..    P.Ui[ 2 ]*Q.U[ 0 ]+P.Vi[ 2 ]*Q.U[ 1 ]+P.Wi[ 2 ]*Q.U[ 2 ]
09  Qp.V[ 2 ]:=
..    P.Ui[ 2 ]*Q.V[ 0 ]+P.Vi[ 2 ]*Q.V[ 1 ]+P.Wi[ 2 ]*Q.V[ 2 ]
10  Qp.W[ 2 ]:=
..    P.Ui[ 2 ]*Q.W[ 0 ]+P.Vi[ 2 ]*Q.W[ 1 ]+P.Wi[ 2 ]*Q.W[ 2 ]
11  Ap[ 0 ]:=Q.A[ 0 ]-P.A[ 0 ]
```

```
12  Ap[ 1 ]:=Q.A[ 1 ]-P.A[ 1 ]
13  Ap[ 2 ]:=Q.A[ 2 ]-P.A[ 2 ]
14  Qp.A[ 0 ]:=
..    P.Ui[ 0 ]*Ap[ 0 ]+P.Vi[ 0 ]*Ap[ 1 ]+P.Wi[ 0 ]*Ap[ 2 ]
15  Qp.A[ 1 ]:=
..    P.Ui[ 1 ]*Ap[ 0 ]+P.Vi[ 1 ]*Ap[ 1 ]+P.Wi[ 1 ]*Ap[ 2 ]
16  Qp.A[ 2 ]:=
..    P.Ui[ 2 ]*Ap[ 0 ]+P.Vi[ 2 ]*Ap[ 1 ]+P.Wi[ 2 ]*Ap[ 2 ]
17  end ChangeBasis
```

Comments on *ChangeBasis*:

(01) The procedure *ChangeBasis* takes 3 arguments. It converts Q in P's basis and store the result in Qp.
(02)-(16) Calculate as detailed in (6)

```
01  Procedure ConfirmSolution( Co, Q ) is
02    for x:=(0.0, 1.0)
03      for y:=(0.0, 1.0)
04        for z:=(0.0, 1.0)
05          val:=0.0
06          for i:=0..2
07            val:=val+Q.A[ i ]
..              +(x*Co.Min[ 0 ]+(1.0-x)*Co.Max[ 0 ])*Q.U[ i ]
..              +(y*Co.Min[ 1 ]+(1.0-y)*Co.Max[ 1 ])*Q.V[ i ]
..              +(z*Co.Min[ 2 ]+(1.0-z)*Co.Max[ 2 ])*Q.W[ i ]
08          end for
09          if val<1.0
10            return true
11          end if
12        end for
13      end for
14    end for
15    return false
16  end ConfirmSolution
```

Comments on *ConfirmSolution*:

(01) The procedure *ConfirmSolution* takes 2 arguments. Co is the intersection's bounding volume expressed in Q's basis.
(02)-(15) To check if the solution verify (a) in (10), I search if there is at

least one edge of the bounding volume which verify (a).

```
01  Procedure ConvertToWorldBasis( Co, Q ) is
02    Max[ 0 ]=Q.A[ 0 ]+Co.Min[ 0 ]*Q.U[ 0 ]
..      +Co.Min[ 1 ]*Q.V[ 0 ]+Co.Min[ 2 ]*Q.W[ 0 ]
03    Max[ 1 ]=Q.A[ 1 ]+Co.Min[ 0 ]*Q.U[ 1 ]
..      +Co.Min[ 1 ]*Q.V[ 1 ]+Co.Min[ 2 ]*Q.W[ 1 ]
04    Max[ 2 ]=Q.A[ 2 ]+Co.Min[ 0 ]*Q.U[ 2 ]
..      +Co.Min[ 1 ]*Q.V[ 2 ]+Co.Min[ 2 ]*Q.W[ 2 ]
05    Min[ 0 ]=Max[ 0 ]
06    Min[ 1 ]=Max[ 1 ]
07    Min[ 2 ]=Max[ 2 ]
08    for x:=(0.0, 1.0)
09      for y:=(0.0, 1.0)
10        for z:=(0.0, 1.0)
11          A[ 0 ]:=Q.A[ 0 ]
..            +(x*Co.Min[ 0 ]+(1.0-x)*Co.Max[ 0 ])*Q.U[ 0 ]
..            +(y*Co.Min[ 1 ]+(1.0-y)*Co.Max[ 1 ])*Q.V[ 0 ]
..            +(z*Co.Min[ 2 ]+(1.0-z)*Co.Max[ 2 ])*Q.W[ 0 ]
12          A[ 1 ]:=Q.A[ 1 ]
..            +(x*Co.Min[ 0 ]+(1.0-x)*Co.Max[ 0 ])*Q.U[ 1 ]
..            +(y*Co.Min[ 1 ]+(1.0-y)*Co.Max[ 1 ])*Q.V[ 1 ]
..            +(z*Co.Min[ 2 ]+(1.0-z)*Co.Max[ 2 ])*Q.W[ 1 ]
13          A[ 2 ]:=Q.A[ 2 ]
..            +(x*Co.Min[ 0 ]+(1.0-x)*Co.Max[ 0 ])*Q.U[ 2 ]
..            +(y*Co.Min[ 1 ]+(1.0-y)*Co.Max[ 1 ])*Q.V[ 2 ]
..            +(z*Co.Min[ 2 ]+(1.0-z)*Co.Max[ 2 ])*Q.W[ 2 ]
14          for i:=0..2
15            if Min[ i ]>A[ i ]
16              Min[ i ]:=A[ i ]
17            end if
18            if Max[ i ]<A[ i ]
19              Max[ i ]:=A[ i ]
20            end if
21          end for
22        end for
23      end for
24    end for
25    for i:=0..2
26      Co.Min[ i ]:=Min[ i ]
27      Co.Max[ i ]:=Max[ i ]
```

```
28    end for
29  end ConvertToWorldBasis
```

Comments on *ConvertToWorldBasis*:

(01) The procedure *ConvertToWorldBasis* takes 2 arguments. Co is the intersection's bounding volume initially expressed in Q's basis and to be expressed in world's basis. To convert from Q's basis to world's basis, the procedure calculate the maximum and minimum values of Co along each axis of the world's basis. The coordinates in world's basis are given by $x.\vec{u}+y.\vec{v}+z.\vec{w}$ with $(x, y, z) \in (Min[0]..Max[0], Min[1]..Max[1], Min[2]..Max[2])_{Co}$.

(02)-(07) Initialize the bounding values in world's basis with the coordinates of $(Min[0], Min[1], Min[2])_{Co}$

(08)-(24) For each corner of the paralellepiped defined by $(Min[0]..Max[0], Min[1]..Max[1], Min[2]..Max[2])_{Co}$ calculate the coordinates in world's basis and update the minimum and maximum along each axis when necessary.

(25)-(28) Update the value of Co with the result.

```
01  Procedure ElimVar( indexVar, M, A, N, NbIneq, NbVar,
..    Mp, Ap, Np, NbIneqp ) is
02    NbIneqp:=0
03    for i:=0..NbIneq-1
04      if IsNull( A[ i*NbVar+indexVar ] )
05        l:=0
06        for j:=0..NbVar-1
07          if j<>indexVar
08            Ap[ NbIneqp*( NbVar-1 )+l ]:=A[ i*NbVar+j ]
09            l:=l+1
10          end if
11        end for
12        Np[ NbIneqp ]:=N[ i ]
13        NbIneqp:=NbIneqp+1
14      end if
15    end for
16    for i:=0..NbIneq-2
17      for j:=i..NbIneq-1
18        if Sign( A[ i*NbVar+indexVar ] )<>
..          Sign( A[ j*NbVar+indexVar ] )
..          and IsNotNull( A[ i*NbVar+indexVar ] )
..          and IsNotNull( A[ j*NbVar+indexVar ] )
19          l:=0
```

```
20          for k:=0..NbVar-1
21            if k<>indexVar
22              Ap[ NbIneqp*( NbVar-1 )+l ]:=
..                A[i*NbVar+k]/AbsoluteValue(A[i*NbVar+indexVar])
..                +A[j*NbVar+k]/AbsoluteValue(A[j*NbVar+indexVar])
23              l:=l+1
24            end if
25          end for
26          Np[ NbIneqp ]:=
..            N[ i ]/AbsoluteValue( A[ i*NbVar+indexVar ] )
..            +N[ j ]/AbsoluteValue( A[ j*NbVar+indexVar ] )
27          NbIneqp:=NbIneqp+1
28        end if
29      end for
30    end for
31  end ElimVar
```

Comments on *ElimVar*:

(01) The procedure *ElimVar* takes 10 arguments. IndexVar is the index of the variable to be eliminated. M, A and N are the coefficients of the system $M \leq A \leq N$ in which there are NbIneq inequations and NbVar variables. After eliminating the variable, the new system's coefficients are stored in Mp, Ap and Np, and the number of inequations is stored in NbIneqp.

(02) Initialize the number of inequation in the new system.

(03)-(15) For each inequation in the initial system, if the coefficient of the indexVar variable is null, add the inequation to the new system as it is, omitting the null coefficient of the removed variable.

(16)-(30) For each pair of inequations where the coefficients of the indexVar variable are of opposite signs and not null, calculate the coefficients of the new system according to equation (15).

```
01  Procedure GetBound( Co, Nb, A, N, indexVar ) is
02    Co.Min[ indexVar ]:=0.0
03    Co.Max[ indexVar ]:=1.0
04    for i:=0..Nb-1
05      if IsNotNull( A[ i ] )
06        N[ i ]:=N[ i ]/AbsoluteValue( A[ i ] )
07      end if
08    end for
09    for i:=0..Nb-1
```

```
10      if A[ i ]>0.0
11        if Co.Max[ indexVar ]>N[ i ]
12          Co.Max[ indexVar ]:=N[ i ]
13        end if
14      end if
15      if A[ i ]<0.0
16        if Co.Min[ indexVar ]<-1.0*N[ i ]
17          Co.Min[ indexVar ]:=-1.0*N[ i ]
18        end if
19      end if
20    end for
21  end GetBound
```

Comments on *GetBound*:

(01) The procedure *GetBound* takes 5 arguments. Co is the bounding area were the value will be stored. Nb is the number of inequations. A and N are the coefficients of the inequations. IndexVar if the index of the variable these bounding values refer.

(02)-(22) Calculate the bounding values according to equation (17).

```
01  Procedure TestTrivialCases( A, N, NbEq, NbParam ) is
02    for i:=0..NbEq-1
03      val:=0.0
04      sp:=1
05      n:=0
06      for j:=0..NbParam
07        if IsNotNull( A[ i*NbParam+j ]
08          n=n+1
09        else
10          if A[ i*NbParam+j ]<0.0
11            sp:=0
12            val=val+A[ i*NbParam+j ]
13          end if
14        end if
15      end for
16      if n<>NbParam
17        if sp=1 and N[ i ]<0.0
18          return false
19        end if
20        if N[ i ]<val
```

```
21          return false
22        end if
23      end if
24    end for
25    return true
26  end TestTrivialCases
```

Comments on *TestTrivialCases*:

(01) The procedure *TestTrivialCases* takes 4 arguments. A is the coefficients of the left part of the inequations system. N is the coefficient of the right part of the system. NbEq is the number of inequations in the system. NbParam is the number of variables in the system.

(02)-(25) For each inequation in the system, I check the two following trivial cases. As each variable is between 0.0 and 1.0, if all the coefficients on the left part are positive and the coefficient on the right part is negative, the system has no solution. And if the sum of negative coefficients on the left part is superior to the coefficient on the right part, the system has no solution.

Finally, here is the procedure used to calculate the inverse of the matrix of P.

```
01  Procedure CalculateInverse( P ) is
02    Det:=P.U[ 0 ]*(P.V[ 1 ]*P.W[ 2 ]-P.V[ 2 ]*P.W[ 1 ])
..      -P.V[ 0 ]*(P.U[ 1 ]*P.W[ 2 ]-P.U[ 2 ]*P.W[ 1 ])
..      +P.W[ 0 ]*(P.U[ 1 ]*P.V[ 2 ]-P.U[ 2 ]*P.V[ 1 ])
03    P.Ui[ 0 ]:=(P.V[ 1 ]*P.W[ 2 ]-P.W[ 1 ]*P.V[ 2 ])/Det
04    P.Ui[ 1 ]:=(P.W[ 1 ]*P.U[ 2 ]-P.W[ 2 ]*P.U[ 1 ])/Det
05    P.Ui[ 2 ]:=(P.U[ 1 ]*P.V[ 2 ]-P.U[ 2 ]*P.V[ 1 ])/Det
06    P.Vi[ 0 ]:=(P.W[ 0 ]*P.V[ 2 ]-P.W[ 2 ]*P.V[ 0 ])/Det
07    P.Vi[ 1 ]:=(P.U[ 0 ]*P.W[ 2 ]-P.W[ 0 ]*P.U[ 2 ])/Det
08    P.Vi[ 2 ]:=(P.U[ 2 ]*P.V[ 0 ]-P.V[ 2 ]*P.U[ 0 ])/Det
09    P.Wi[ 0 ]:=(P.V[ 0 ]*P.W[ 1 ]-P.W[ 0 ]*P.V[ 1 ])/Det
10    P.Wi[ 1 ]:=(P.U[ 1 ]*P.W[ 0 ]-P.W[ 1 ]*P.U[ 0 ])/Det
11    P.Wi[ 2 ]:=(P.U[ 0 ]*P.V[ 1 ]-P.V[ 0 ]*P.U[ 1 ])/Det
12  end CalculateInverse
```

Comments on *CalculateInverse*:

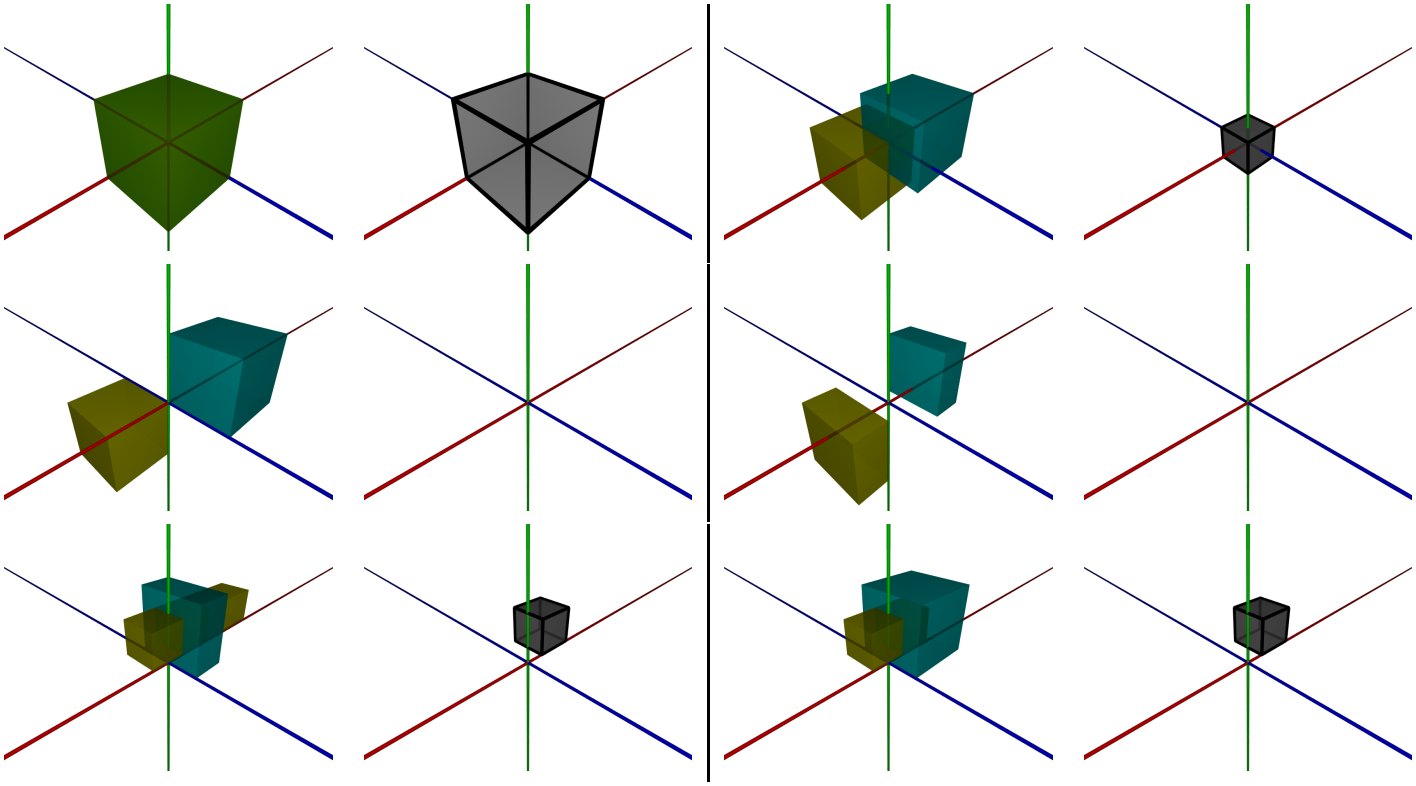
(01)-(12) The procedure *CalculateInverse* takes 1 argument. P is the volume whose inverse matrix of edges vector must calculated.

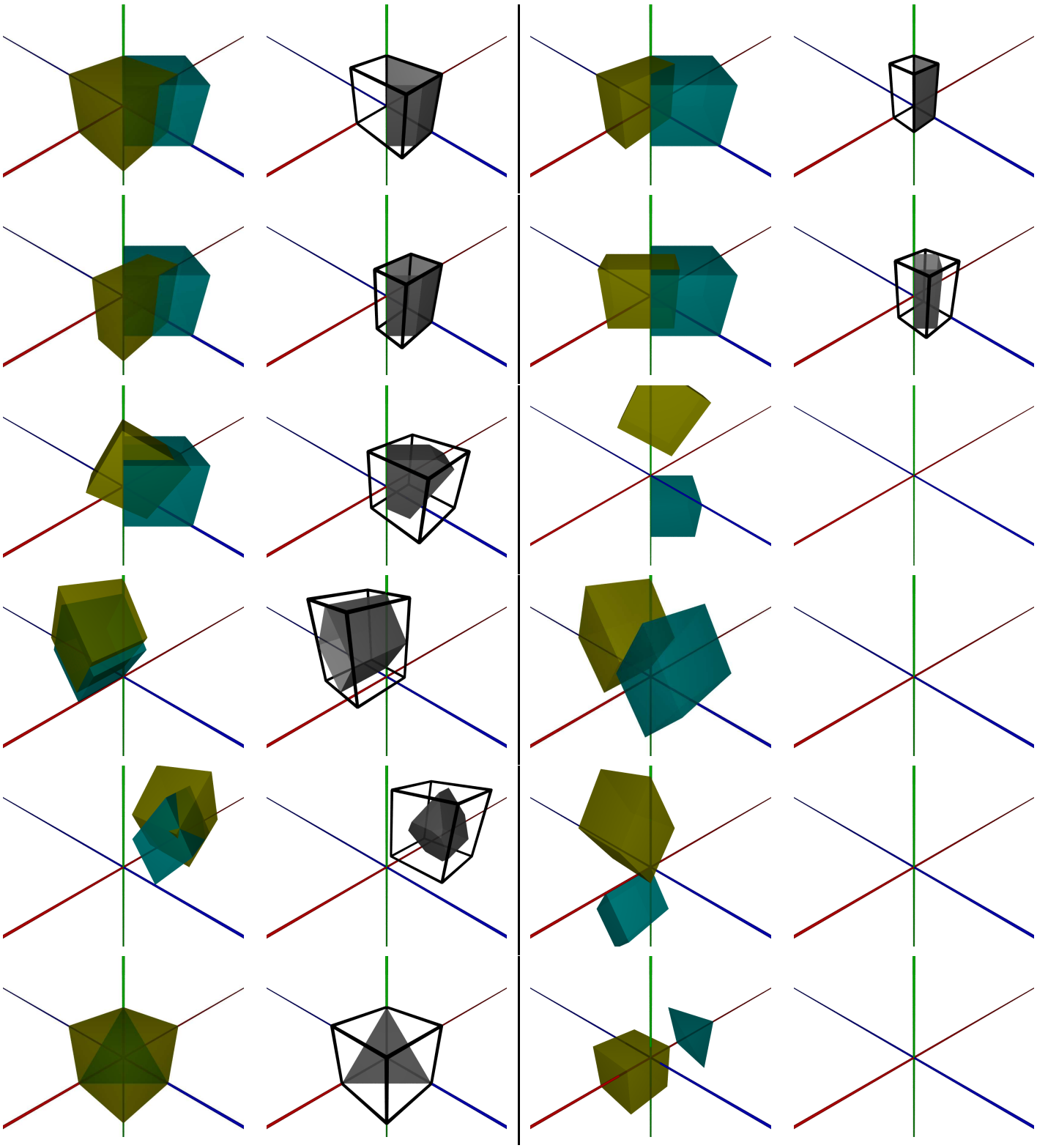
6 Results

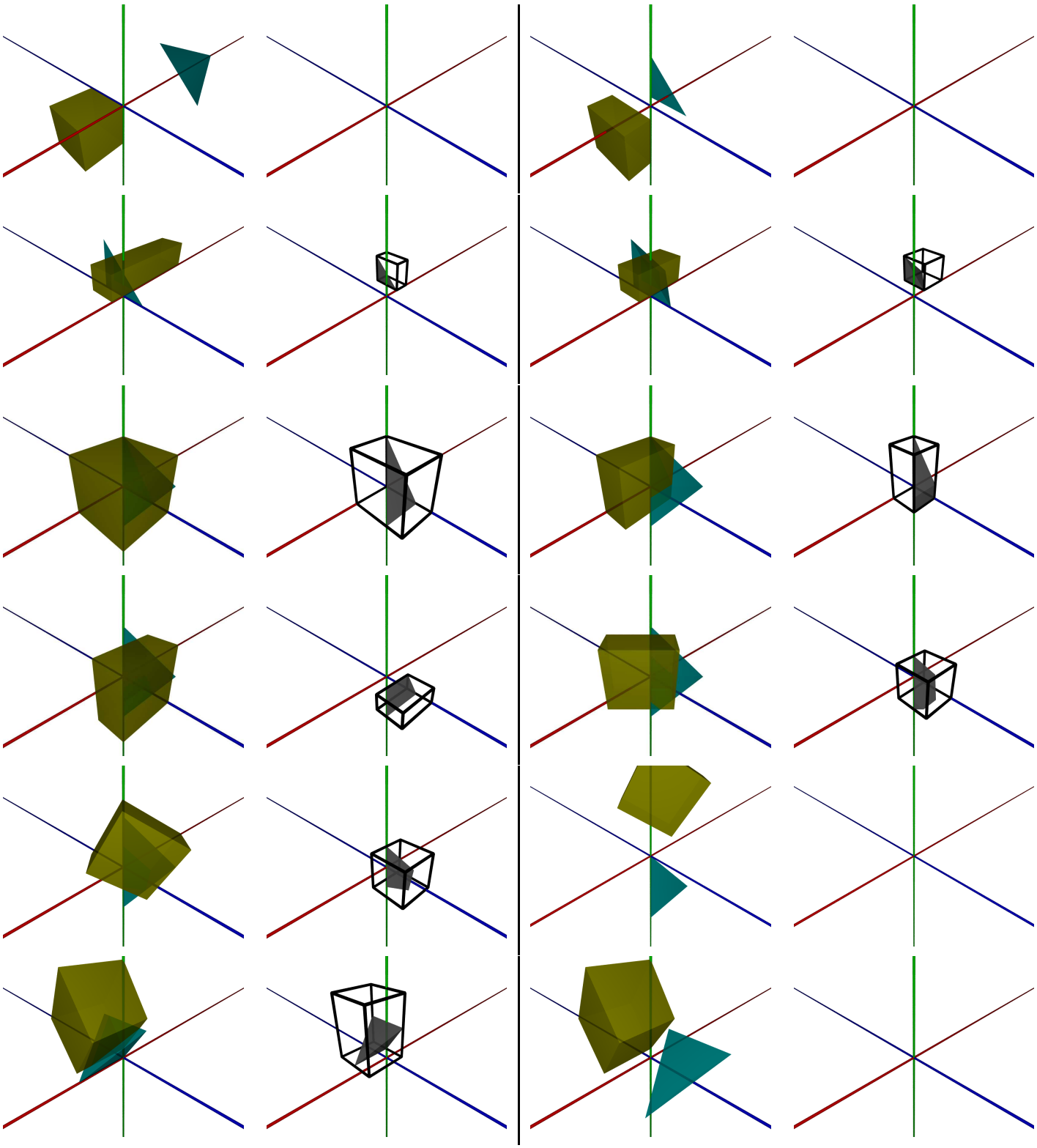
6.1 Validation

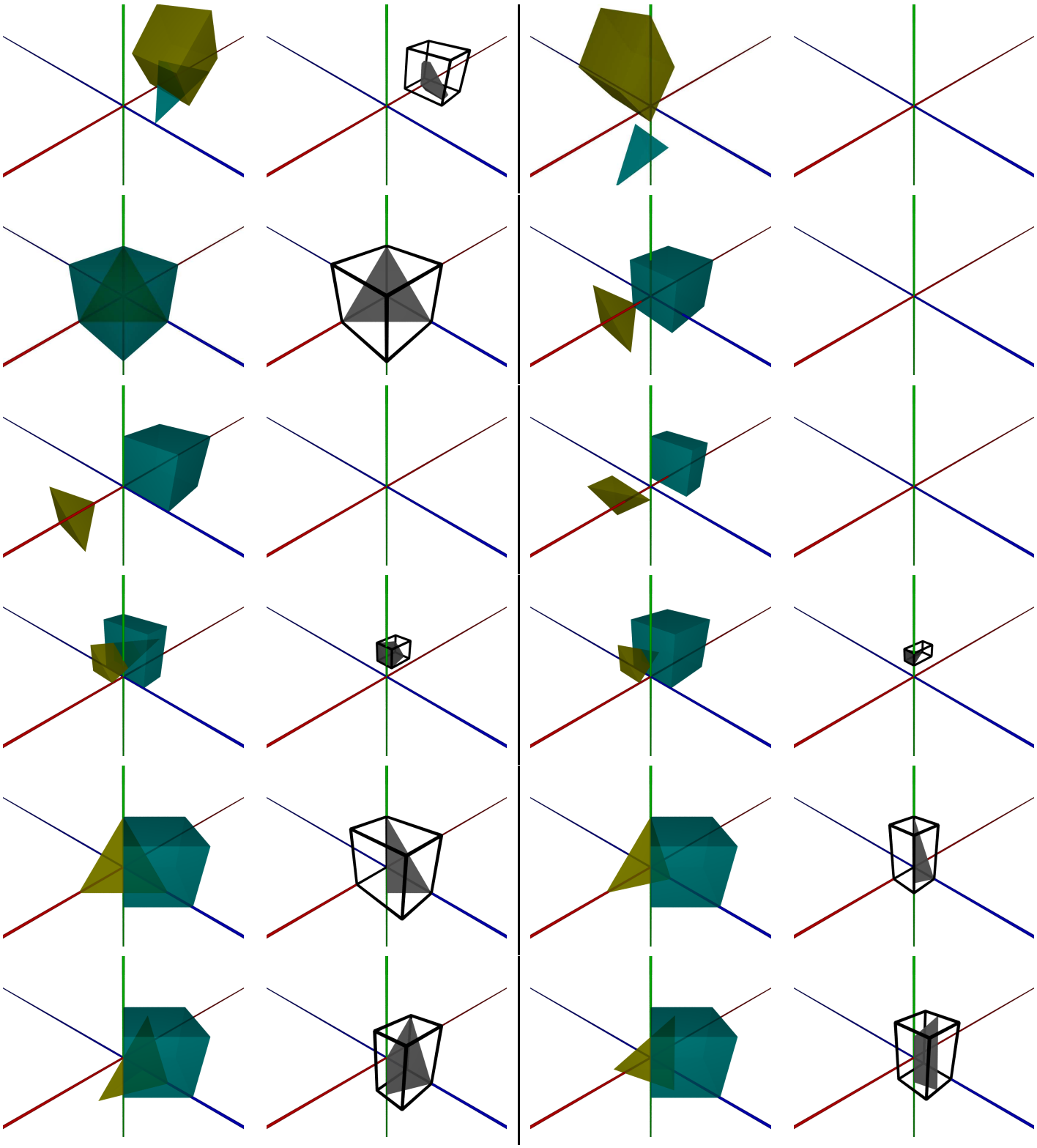
6.1.1 Procedure *TestCollisionBetween*

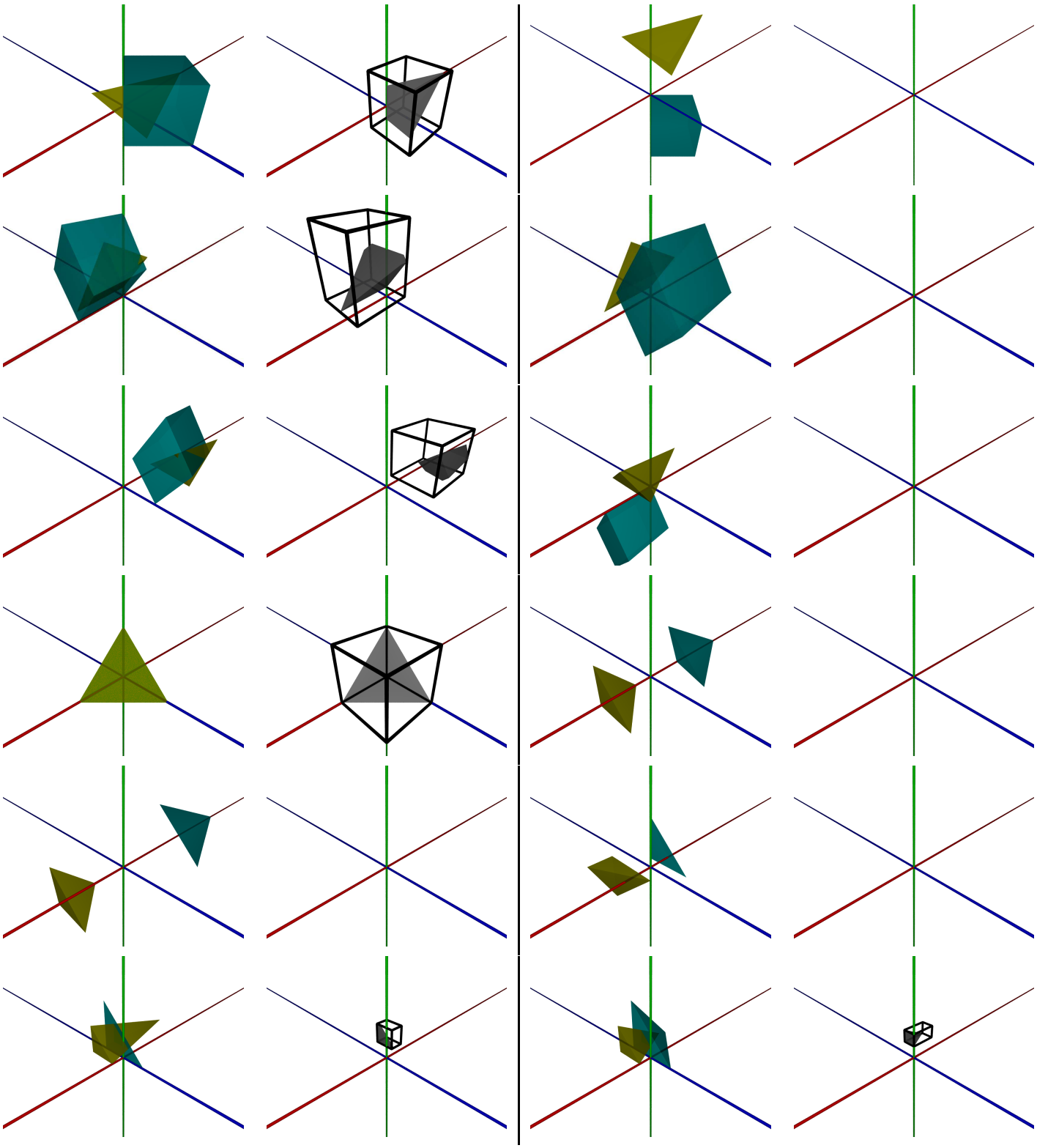
To validate the procedure *TestCollisionBetween* I have used 16 cases for each combination parallelepiped/tetrahedron, and checked if the collision or the absence of collision was properly detected. To check the results I have automatically generated POV-Ray scripts for the two objects, their intersection and the eventual bounding box. The pictures generated are shown below in two columns, on the left side of each column the two objects, on the right side the intersection calculated by POV-Ray (in gray) and the bounding box calculated by *TestCollisionBetween* (in black). The red, green and blue lines represent respectively the X, Y and Z axis for better estimate of each picture.

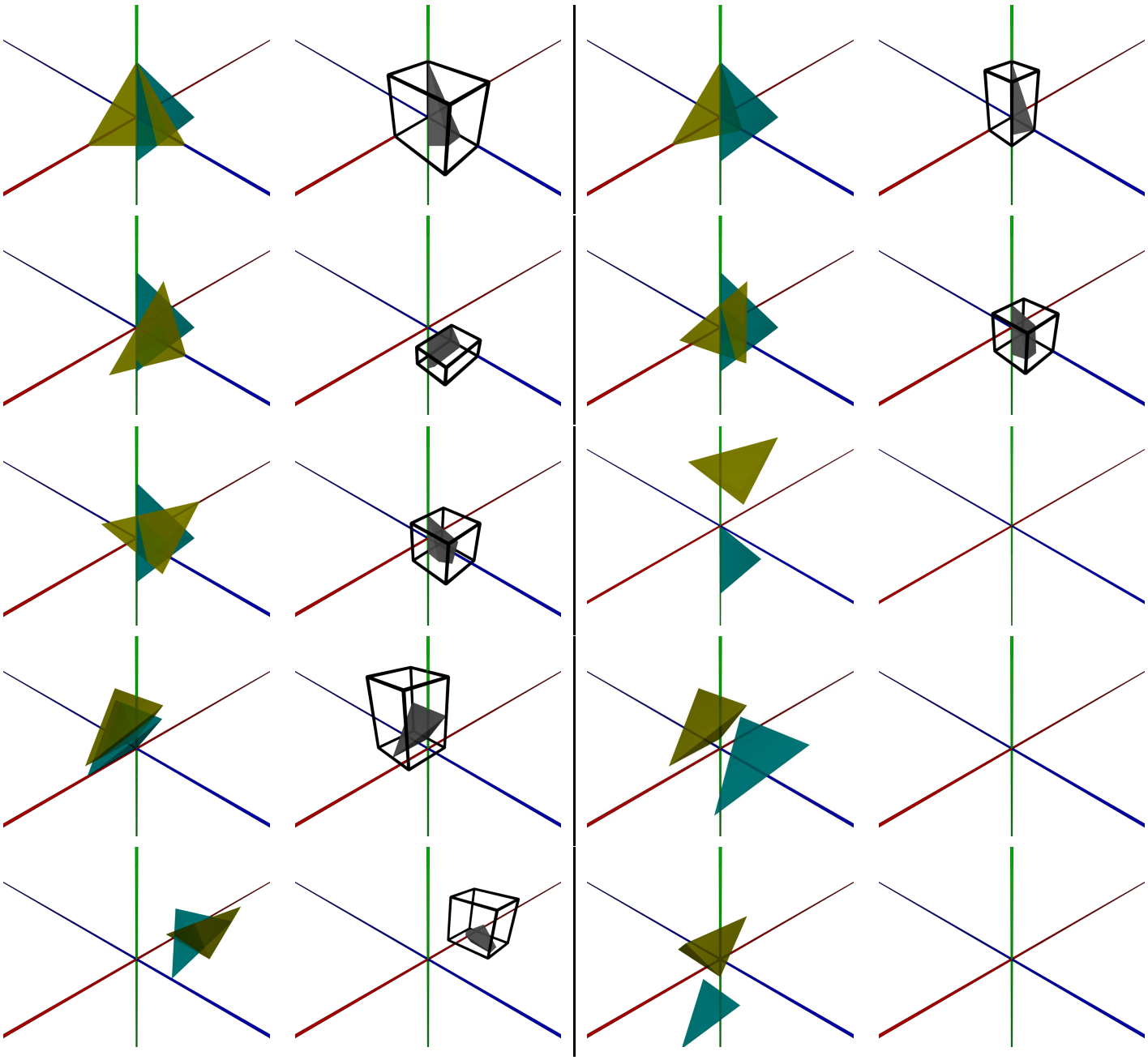








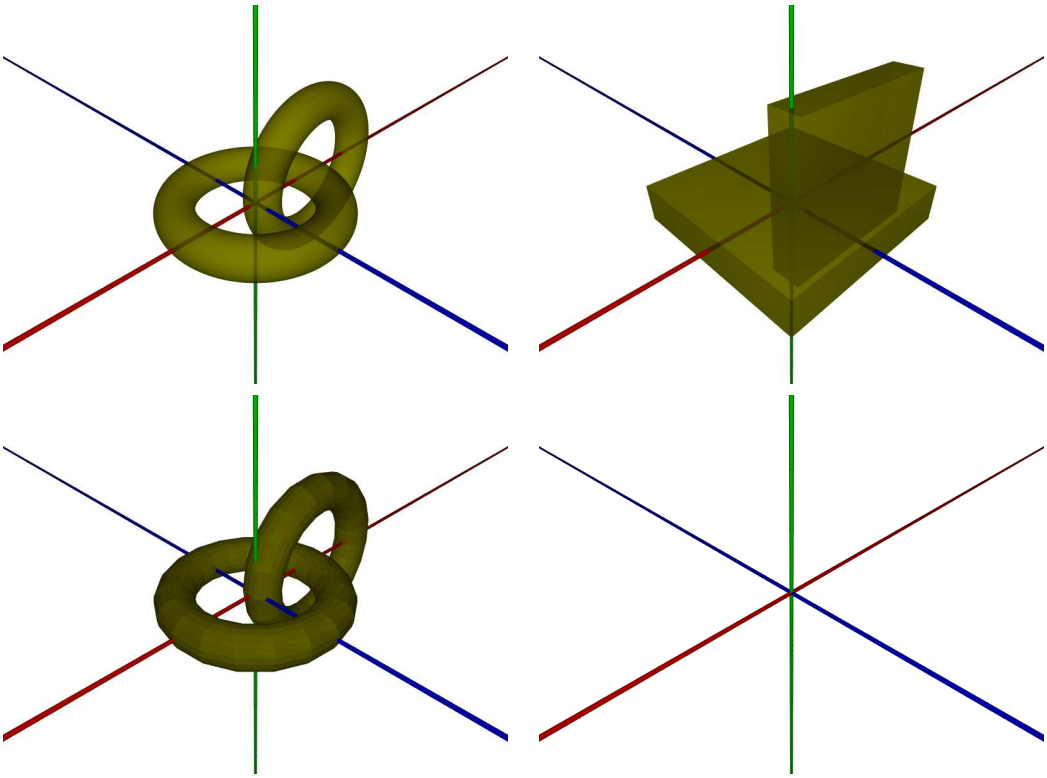


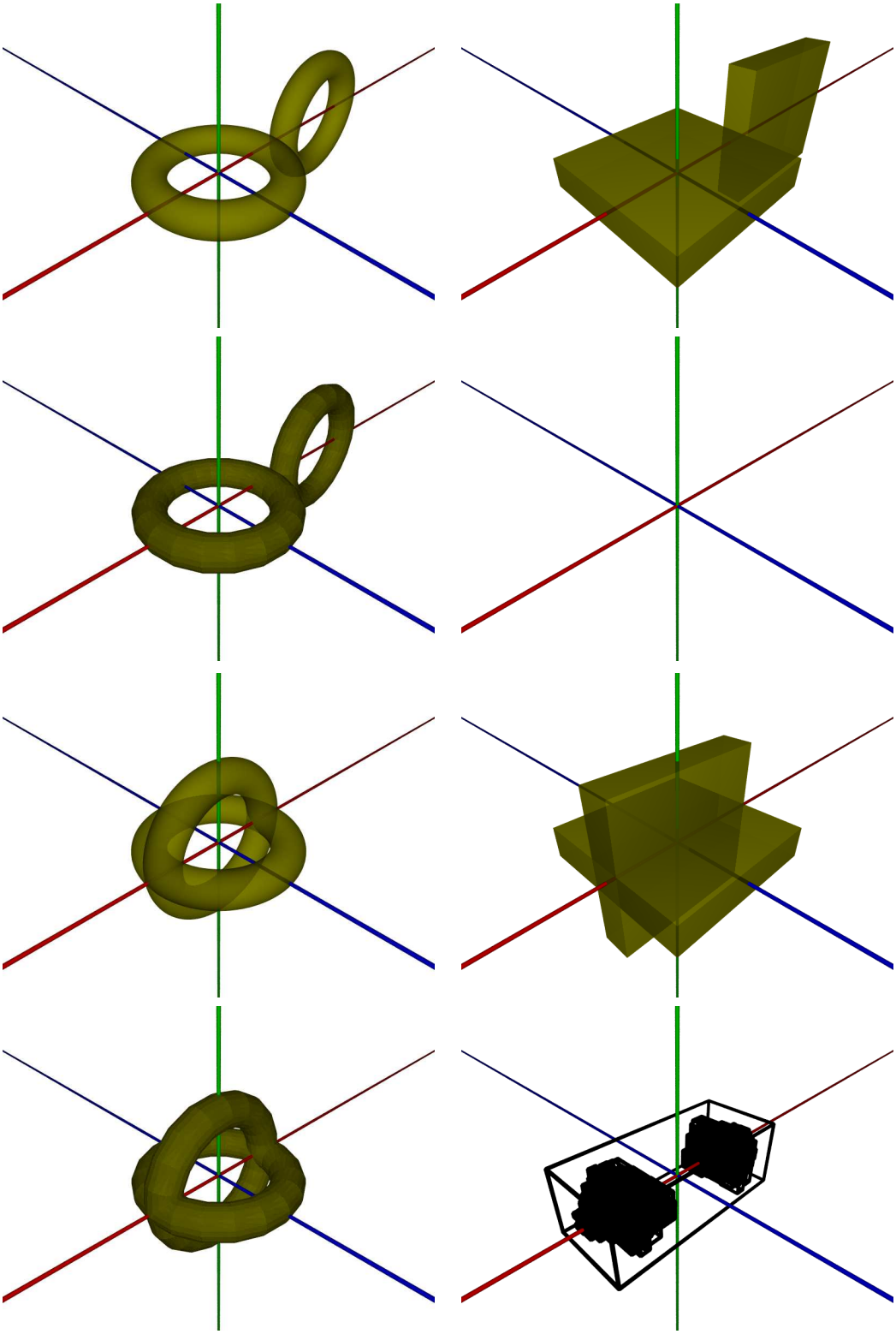


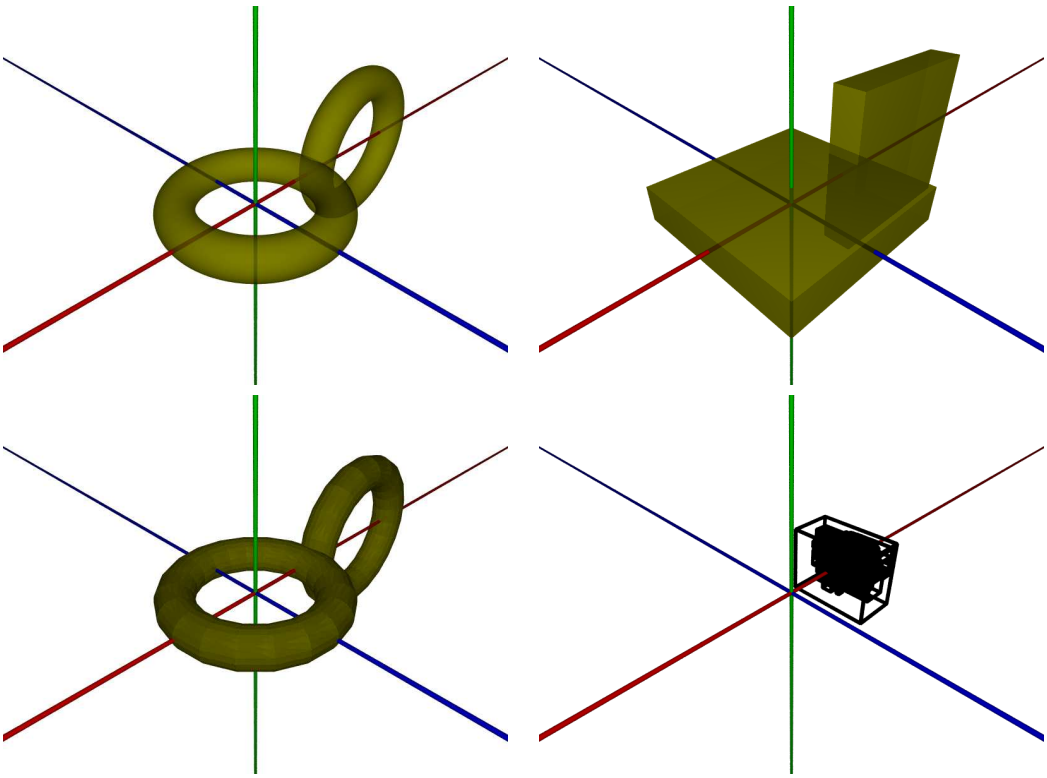
6.1.2 Procedure *SearchCollisions*

To validate the procedure *SearchCollisions* I have tested collisions between two torus in 4 cases. In the first case, they are intricated in each other but not colliding. In the second case, they are side by side and not colliding. In the third case, they are colliding in two places. In the fourth case, they are

colliding in one place. Each torus is defined by two levels of approximation. The first level is one simple parallelepiped surrounding the torus. The second level is a set of 800 tetrahedrons surrounding the torus. As for *TestCollision-Between*, pictures of the results have been obtained thanks to POV-Ray. I show below for each case, the real torus, its approximation on level 1, its approximation on level 2, and the bounding boxes for each pair of nodes if there was, plus one bounding box surrounding all the other bounding boxes.







6.2 Performance

The algorithms have been implemented in C++, compiled with g++ version 4.6 and the option -O3, run under Ubuntu 12.04 LTS on a Intel Core i3-2120 CPU @ 3.30GHz x 4. The time execution has been measured with the command 'time'.

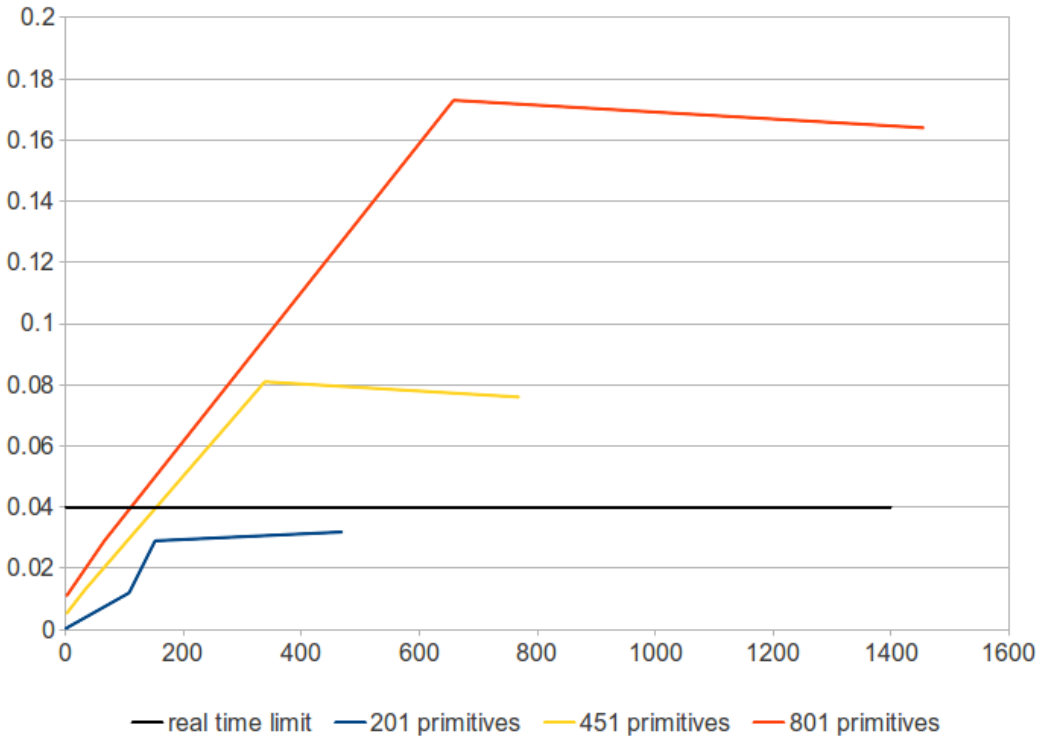
6.2.1 Procedure *SearchCollisions*

To calculate the execution time of one call to the procedure *TestCollisionBetween*, I have repeated 10.000 times the collisions detection test on the 64 cases used for validation, once with the real procedure and once with a fake procedure which simply return false (to calculate the time spent outside of *TestCollisionBetween*). Then I have calculated the average of the difference between execution time in both case. The result is approximatively 6.5e-06 seconds.

6.2.2 Procedure *SearchCollisions*

To calculate the execution time of one call to the procedure *SearchCollisions*, I've created 100 of the torus introduced in previous subsection and looked for the collisions between one of them and all the others. Each torus was randomly generated in term of location, size and orientation. Thus, the execution time shown in what follows is an average of several runs and an approximation of the exact time which vary with the exact configuration. As previously, I have run the test once with a fake procedure simply returning false and once with the real procedure, and calculated the average of the difference between execution times. The results are shown on the graphic below.

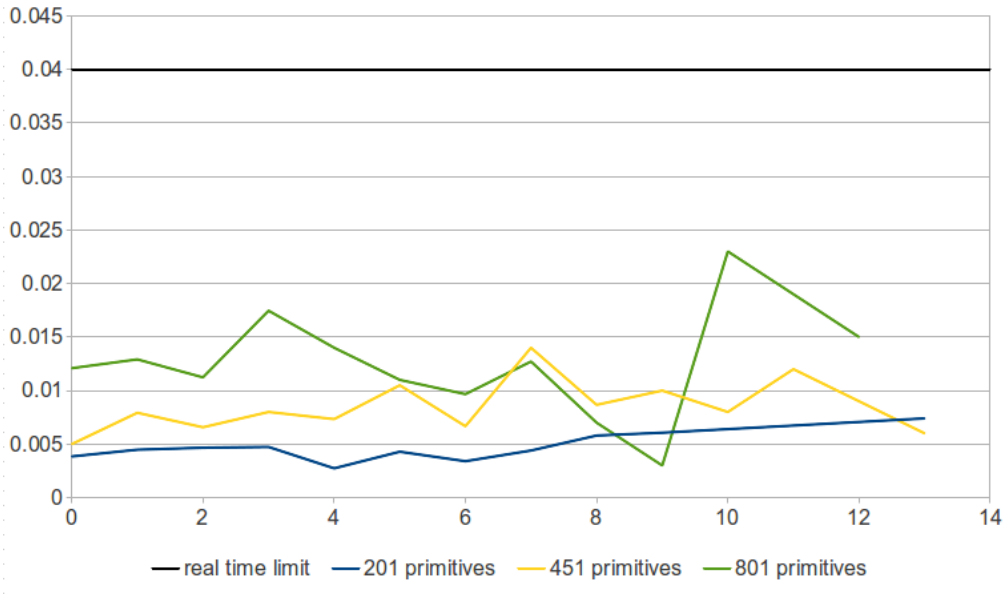
I've repeated the test with several amounts of primitives on the second level of each torus. Also, with the view to use this algorithm in a real time video game, it's interesting to compare the execution time with the limit of 0.04 seconds. This limit is equivalent to 25 steps per second, which is the slowest pace a picture must be refreshed to appear smoothly animated. Here I only consider the algorithm for collision detection, however a real application would have to manage many other things during each step or may use a different pace to refresh pictures. This limit is then only to be seen as a subjective reference to estimate the weight of the algorithm in a more global environment. For example if the algorithm runs in 0.02 seconds, it would mean it takes approximately 50% of the total available time and leaves approximately 0.02 seconds to the other algorithms to keep the whole application running smoothly.



The abscissa represents the number of nodes collided, the ordinate represents the execution time in second.

With only few tests we can see that the execution time is almost immediately over the real time limit, which means one can't expect to use this algorithm in a real time application.

However, the algorithm introduced in this article looks for the complete list of nodes in intersection. In a video game, knowing if two objects are colliding can be sufficient. Then we can modify the algorithm to stop the search between two objects as soon as it has found a collision, and make it returns the list of collided objects instead of the list of collided nodes. Results obtained with this modified algorithm are shown below.



The abscissa represents the number of torus collided, the ordinate represents the execution time in second.

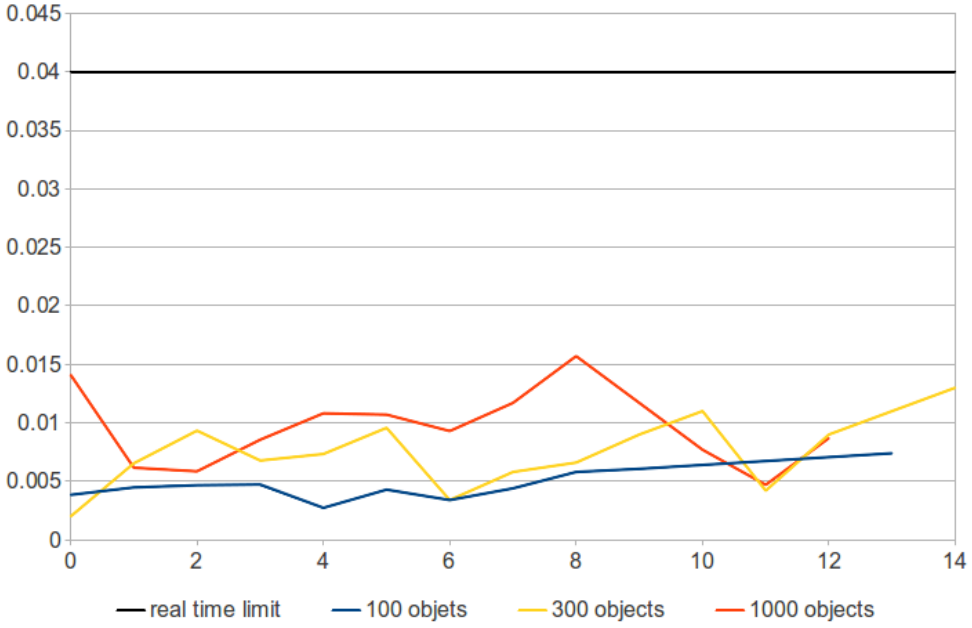
Now we see that the execution time is below the time limit, and increases slowly with the number of primitives. Then one can expect to use this algorithm in a real time application as long as the amount of primitives defining each object is limited.

We also notice that the execution time doesn't vary much with the number of collisions, but can be very different while the numbers of collisions are similar (for example between 9 and 10 collisions when there are 801 primitives). To understand well how to interpret these results one must remind how the algorithm works. To detect the collision it tests each pair of nodes in the set of nodes of each objects, and stop as soon it has found one collision. When two objects are colliding, in the best case the first pair of nodes tested for these two objects will be in intersection and the test will end immediately for these objects. But in the worst case, only the last pair of nodes will be in intersection. If the number of primitives is high, the difference between best case and worst case may be quite significant. If the number of primitive is low, the difference won't be significant and the execution time will be more stable as we can see on the graphic.

Also, thanks to the first level of definition of the torus, those not colliding will be generally detected in one step of *SearchCollisions*, and in best cases objects colliding will be detected in two steps (one for the first level and one for the second level). Thus, in best cases the total number of tests will be the sum of number of objects and number of collisions. As the number of collisions is necessarily significantly lower than the number of objects, the

number of collision won't have a significant impact on the execution time. Even if it's not the best case, we've seen that if the number of primitives is low the difference between best case and worst case will be low. So, globally the execution time will be fairly stable, which we can verify on the graphic in case of 201 primitives per torus.

Finally, I wished to show the importance of defining levels in the definition of objects. As written in the previous paragraph, non-colliding objects are almost always detected in just one step of *SearchCollisions*. Only situation where the approximation of two objects on first level are in intersection and it takes a verification on the second level to realize they actually don't collide causes more than one test. Then, for a given number of collisions, the total number of objects should have limited impact on the execution time. To verify this, I've measured the execution time for 100, 300 and 1000 torus, each defined by 201 primitives. Results are shown below.



The abscissa represents the number of torus collided, the ordinate represents the execution time in second.

Results are impressive. The number of objects seems to have almost no effect on the execution time. Multiply the number of objects by 10 only multiply the average execution time by 2.

This illustrates how efficient the tree structure is in the search for collisions. In this article I only made tests with two levels per object, but it would be interesting to look further about the influence of the number of levels and their architecture on speed. For example, one can immediately guess from what I've shown here that defining an object in a way each nodes at one level

would have only a limited number of childs on the next level would reduce the effect of worst cases and improve the speed of the algorithm.

7 Conclusion

In this article I've given a solution to the collision detection problem in 3D by approximating objects as sets of parallelepipeds and tetrahedrons organized in a tree structure. The problem becomes then equivalent to the resolution of systems of linear inequalities for which the existence of a solution and an approximation of the solution can be obtained thanks to the Fourier-Motzkin elimination method. I've also shown that the algorithm which implements this solution gives accurate results and is fast enough to be used in a real time application such as a video game.

This article may be followed up by :

- a study of the best way to represent a given object with sets of parallelepipeds and tetrahedrons with the view to maximize accuracy and speed
- a study of the best way to define supergroup on top of objects in the space partitionning tree as suggested in section 2.1 with the view to maximize speed
- optimization work on the algorithms introduced here.

References

[1] J.J.-B. Fourier. Oeuvres II. Paris, 1890

[2] T.S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. Thesis, 1936. Reprinted in: *Theodore S. Motzkin: selected papers* (D.Cantor et al., eds,), Birkhäuser, Boston, 1983.