COLLISION DETECTION ALGORITHMS

ALEŠ DANEU

Faculty of computer and information science

University of Ljubljana

2002

Abstract

This thesis deals with the problem of collision detection between 3D objects. The thesis is divided into four parts. The first chapter – Introduction shortly explains the problem of collision detection, why collision detection algorithms are needed and different approaches to tackle the it. Second chapter is a short overview of existing algorithms. Seven different algorithms are described: algorithms using sphere trees, SOLID, RAPID, QuickCD, SWIFT++, I-COLLIDE and Q-COLLIDE. Third chapter deals with our collision detection library - CD. Algorithm is described into detail and some performance measures are given at the end of chapter. The last chapter delves into use of collision detection algorithms. A simple game example shows how to use our collision detection library CD (or any other collision detection algorithm). Finally, appendix (CD API) will be of some interest for those interested in using CD library.

Key words: *collision detection, hierarhical trees, bounding volumes*

Acknowledgements

First of all I would like to thank my menthor prof. dr. Saša Divjak for his remarks and directions during my work on the thesis. Thanks to James Thomas Klosowski (State University of New York at Stony Brook) for letting me see the code of QuickCD algorithm. It made my understanding of that algorithm better. Finally I would like to thank to all that helped me in any way to finish my thesis.

Contents

1. Introduction

The problem of collision detection is very important in various areas such as computer graphics, simulation of real world systems, virtual reality, robotics, object modelling and many others. The fact that two solid objects can not occupy the same space is the basic reason why we need such algorithms.

Collision detection can be divided into *collision detection* and *collision detection and report*. In second case we usually also need to solve the problem of *collision detection response* which will be shortly described in chapter 4. All algorithms described in chapters 2 and 3 are of *collision detection and report* type. Collision detection can be further divided into *static collision detection, pseudo dynamic collision detection* and *dynamic collision detection*. Static collision detection means that we simply check for collision between two objects in certain position and orientation. Pseudo dynamic collision detection is similar to static, but the checks are done on a set of discrete pairs position/orientation. Dynamic collision detection stands for comparison of volumes carved into space by movement of objects.

Collision detection algorithm must be fast, accurate and robust. Fast beacuse of the fact that in some systems we need to perform up to 1000 tests per second. Accurate because that is the basic condition for accurate collision detection response which is near to real world response. And finally robust because often there are errors on object models and we want the algorithm to work properly in those cases too.

Objects are usually composed from several thousand triangles. To test all pairs of triangles from two objects would simply take too much time. Algorithms that use a combination of some bounding volume and hierarhical tree are used in practice. There are also algorithms that are based on decomposition of object into convex pieces which compose hierarhical tree. The root of such tree is the whole object. Those algorithms are combination of hierarhical trees and fast algorithms for collision detection between convex shaped objects. More about that in chapter 2.

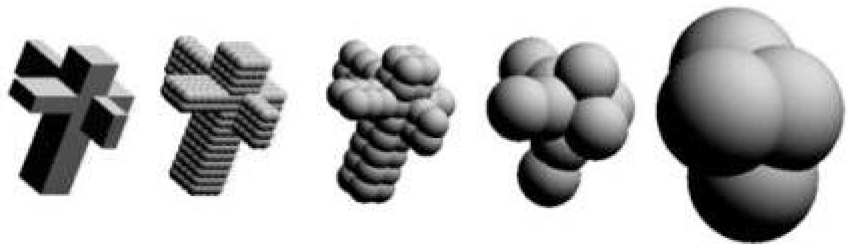
The purpose of this thesis was to write a collision detection library for use in games, although use for other purposes is not excluded. Algorithm and it's usage are described in chapters 3 and 4.

2. Overview of existing algorithms

There are many different collision detection algorithms. Based on their use they can be divided into two groups. The ones meant for collision detection only between convex shaped objects and those which are not limited by the shape of the object. The type of bounding volume used is also one property where algorithms differ. This chapter shortly describes some of the algorithms from both groups. For details you will need to look into their respective technical documents.

2.1. Sphere trees

Spheres are often used as aproximation for objects in computer graphics. One of the main reason is that it is very simple to check if two spheres overlap, the second even more important reason is that spheres are independent from rotation. Because of that last property we have a very simple task when two nodes from different trees are compared. All we need to do is translate and rotate one point – sphere's centre and then check for overlap. But spheres also have downsides, they are often a very bad aproximation for object's shape which leads us to big tree hierarchies if we want to have good aproximation of object. See [4] for more about this type of bounding volume and sphere trees.



Picture 1 : Example of sphere tree hierarchy (summarized from [4]).

2.2. SOLID

SOLID is one of the simpler representatives of algorithms that use hierarhical trees in combination with bounding volume. In this case the bounding volume is a box whose sides are aligned with X, Y and Z axis of object's coordinate system (AABB or *axis aligned bounding boxes*). Advantage of this algorithm is the speed of building the tree. It's disadvantage is the speed of collision detection. In [2] it is suggested that because of the speed in which tree is adapted to represent object's new shape such algorithm is very useful when objects are deformable. See [2] for more about this algorithm.

2.3. RAPID

RAPID also uses hierarhical tree in combination with bounding volume. Bounding volume is a box which is oriented in such way that it best fits the object's shape (OBB or *oriented bounding boxes*). Authors have developed a fast overlap test (called *SAT*, see more in [3]), therefore the orientation of box is not a limitation in terms of speed. OBB trees perform very good in cases where the objects are very close to each other (*close proximity*). RAPID is asimptoticaly much faster compared to algorithms that use spheres or AABBs as bounding volumes. See [3] for more.

2.4. QuickCD

QuickCD bounding volume is a convex polytop which has it's sides determined by as set of *k* orientations. In [1] such polytopes are called *discrete orientation polytopes* or shorter k-dop. QuickCD implements algorithms for 6-dop (which is basicly the same as AABB), 14-dop, 18-dop and 26-dop. The algorithm is based on assumption that one object is static while the second object is moving.
As all algorithms described until now this one also uses hierarhical trees. Algorithm also uses some tehnicques which exploit time coherence to speed up collision detection. Read [1] for details about QuickCD. Also, some details of this algorithm are further discussed in chapter 3 since this is the algorithm that was choosed as the basis for CD collision detection algorithm.

2.5. SWIFT++

SWIFT++ is algorithm that can also perform queries other than just collision detection. Those queries are (see descriptions of each query in [5]):

(a) Intersection detection (is a pair of polyhedra intersecting or not)

(b) Tolerance verification (is the distance between pair of polyhedra below tolerance or not)

(c) Exact minimum distance (returns the mimimum distance between pair of polyhedra if the distance is less than tolerance)

(d) Approximate minimum distance (returns the mimimum distance between pair of polyhedra such that the minimum distance is within given error bounds and the distance is less than tolerance)

(e) Disjoint contact determination (returns a set of pairs of features such that each pair represents local minimum and the distance is less than tolerance)

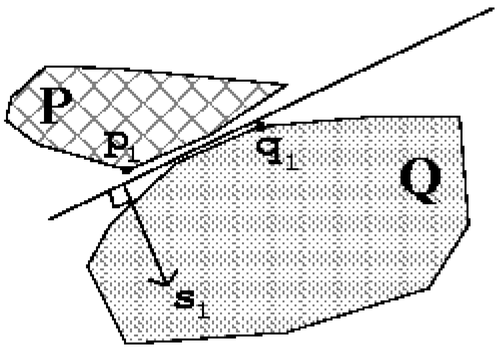
Algorithm is based on decomposition of object to convex parts. Those parts are convex hulls of convex patches. Convex parts are then combined together and form a hierarhical tree which has the whole object as it's root. See [5] for detailed description of SWIFT++. Not only that this algorithm can perform various queries, according to [5] this algorithm is also the fastest in group of algorithms that are not limited by object's shape.

2.6. I-COLLIDE

This algorithm together with Q-COLLIDE is a representative of algorithms which are primarily meant to detect collisions between convex shaped objects. Since the algorithm is limited to convex shaped objects it can exploit this property for faster collision detection. The algorithm searches for pair of closest triangles and uses local search to do that. In [6] it is described exactly how this is done. I-COLLIDE was adapted to detect collision between non convex shaped objects too. Adaptation was done with use of hierarhical trees. Tree leafs contain convex parts. By merging those objects together and calculating bounding volume for merged objects the tree is built all the way to the top. I-COLLIDE also includes a method to determine necessary tests between objects in enviroment. This method is shortly described in chapter 4.1.3. See more about I-COLLIDE in [6].

2.7. Q-COLLIDE

Q-COLLIDE is an enhancement of I-COLLIDE. The algorithm does not search for the pair of closest triangles. Instead algorithm searches for separating plane between two objects. If a separating plane is not found collision is reported. Point of collision is calculated from previous positions of objects. As noted in [7] this is a big performance improvement in terms of speed, simpler implementation and memory usage. Algorithm is described in [7].



Picture 2 : Separating plane between two convex objects (summarized from [7]).

3. CD – a collision detection library

This chapter is about CD collision detection library. It's target area of use are games, so the imlementation is optimized for such use.

3.1. Algorithm basics

Algorithm is based on QuickCD algorithm. QuickCD uses k-dop bounding volume. QuickCD gave best results when 18-dop bounding volume was used (see [1]), so this bounding volume is also used in CD. Some other decisions taken during design of CD collision detection library were based on results from [1]. The main differences between CD and QuickCD are:

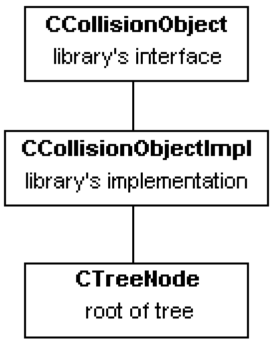
(a) QuickCD is designed for collision detection between one static object (enviroment) and one dynamic object. CD can also handle collision detection between two dynamic objects.

(b) QuickCD's leafs contain triangles. Static object's tree has only one triangle in it's leafs while dynamic objects's tree leafs contain up to a defined maximum number of triangles. CD is different – it's leafs contain convex patches which are obtained with object decomposition. The number of triangles on convex patches is also limited by a defined maximum number.

CD exploits convexity on last stage of collision detection algorithm: triangle-triangle tests. Due to convexity of patches we can use local search. That speeds up the procedure of finding an intersection between two convex patches and also limits the number of necessary triangle-triangle tests. More about that in 3.4.

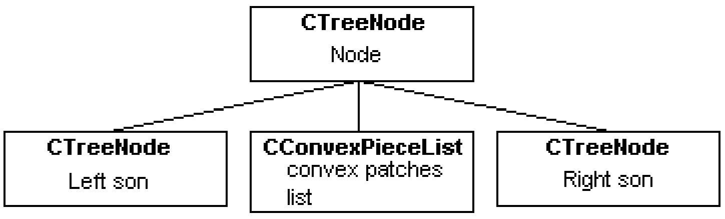
3.2. Data structure

Class that represents an object and is visibile to library's user is *CCollisionObject*. It's functionality is implemented in class *CCollisionObjectImpl*. This last class contains all data about hierarhical tree, objects's position and methods for building the tree, collision detection, updating object's position and reading test results. Read more about those methods and theirs usage in appendix.



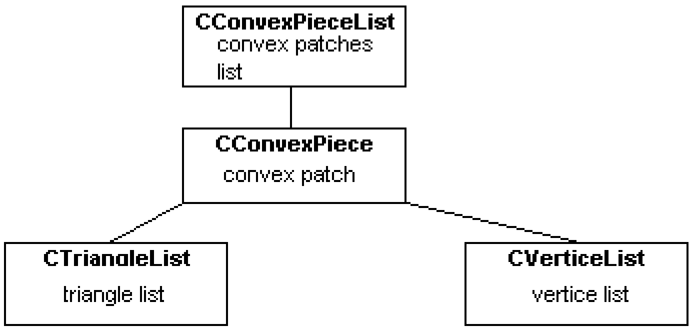
Picture 3 : CCollisionObject represents library functionality that is available to user. CCollisionObjectImpl is implementation of whole library functionality, data needed for collision detection test is stored in class CTreeNode.

CTreeNode class represents a node in tree. A node contains data about 18-dop's boundaries, node's sons, a list of convex patches it contains and their number. Operations on this class are: addition of triangles, addition of convex patches, bounding volume's boundaries calculation and node splitting.



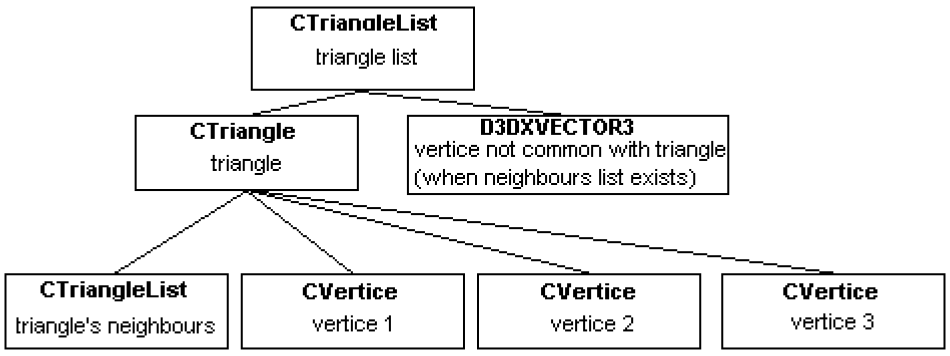
Picture 4 : Tree node structure. Node has a left and a right son and a list of convex patches in node.

CConvexPieceList represents a list of convex patches contained in the node. The following operations are possible for list of convex patches: addition of new triangles (it is added on one of the existing convex patches or on a new convex patch if it can't be added to any of the existing ones) and addition of new convex patches. A convex patch is represented by class *CConvexPiece*. This class supports the following operations: addition of triangles on convex patch (addition is sucessfull if triangle actually belongs to this convex patch – all conditions are fullfilled) and calculation of convex patch center. Data that represents convex patch are: center of convex patch (mean value of all vertices on patch is used as center), boundaries of 18-dop, list of vertices on convex patch (*CVerticeList*) and a list of triangles on convex patch (*CTriangleList*).



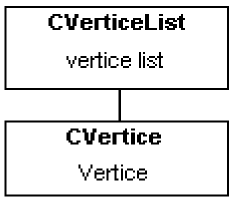
Picture 5 : Structure of convex patches list. A convex patch contains data about triangles on it and vertices that define those triangles.

A list of triangles is implemented by class *CTriangleList*. It allows addition of new triangles into list, copying triangles from other lists and deletion of first element in list. Element of list contains data about triangle (class *CTriangle*). If the list represents a list of neighbours for a given triangle then the elements also contain vertice that is not common with triangle to which the list of neighbours belongs. Class *CTriangle* contains data about triangle vertices (pointers on vertices saved in list of vertices (*CVerticeList*) for current convex patch), a list of neighbours (*CTriangleList*) and some temporary values used during collision detection.



Picture 6 : Structure of triangle list. Associated with triangle are data about it's neighbours and vertices. If a list represents a list of neighbours for some triangle then each element also contains data about vertice not common with triangle to which the list of neighbours belongs.

A list of vertices is represented with classes *CVerticeList* and *CVertice*. The only possible operation is addition of new vertices into list, coordinates of vertices are the only data saved.



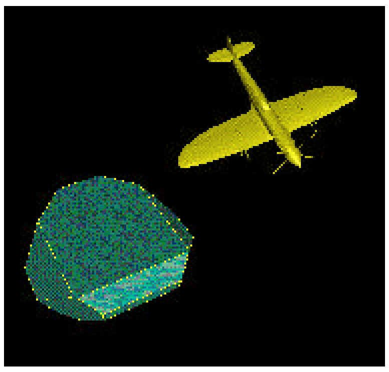
Picture 7 : Structure of vertice list.

3.3. Building hierarchy tree

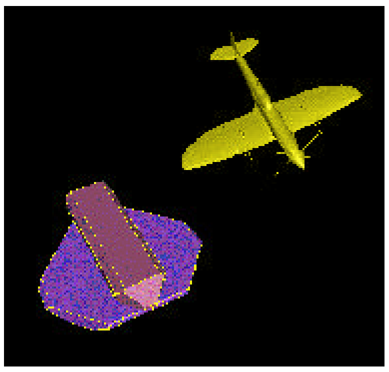
Two steps are performed while building hierarchy tree. In first one we add all objects's triangles into root of the tree. The second is building tree hierarchy (node splitting).

1	2	3	4	5	6	7	8	9
(1,0,0)	(0,1,0)	(0,0,1)	(1,1,0)	(1,0,1)	(0,1,1)	(1,-1,0)	(1,0,-1)	(0,1,-1)

Table 1 : Nine directions (in (X,Y,Z) format), in which the boundaries of 18-dop are calculated.



Picture 8 : Spitfire plane and it's 18-dop on first level of tree hierarchy (root). Summarized from [1].



Picture 9 : Second level of tree hierarchy for Spitfire plane. Summarized from [1].

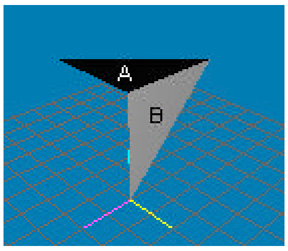
3.3.1. Adding object's triangles into root of the tree

We need to find a corresponding convex patch for each triangle added into tree. Triangle is part of new convex patch if such convex patch is not found. Some data which is needed later (while building tree hierarchy and during collision detection) are also calculated when a triangle is added. This data are list of neighbours for added triangle, new boundaries of convex patch (after the new triangle was added) and new center of convex patch. Any new vertices are added in list of vertices. Triangles are added with the following pseudo code:

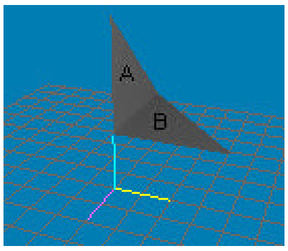
```
procedure AddTriangle(v1,v2,v3); //v1,v2,v3 are vertices
begin
  //search for piece
  Piece = PieceList.First;
  PieceFound = false;
  while not (Piece = NULL) do
  begin
    //is threshold for number of triangles already reached?
    if not (Piece.TriangleNum > MaxTriangleNum) then
    begin
      //compare triangle to other triangles
      Triangle = Piece.TriangleList.First;
      TriangleAdded = true;
      while not (Triangle = NULL) do
      begin
        //check convexity conditions
        if not FormsConvexPiece(Triangle,v1,v2,v3) then
        begin
          TriangleAdded = false;
          break;
        end else
          //store triangle to new triangle's neighbours list
          StoreToTriangleList(Neighbours,Triangle);
        Triangle = Triangle.Next;
      end;
      //if all convexitiy checks were successfull do the insert
      if TriangleAdded then
      begin
        //set data for new triangle
        newTriangle = Piece.TriangleList.Insert(v1,v2,v3,
                                                Neighbours);
        //update existing triangles neighbours lists
        Neighbour = Neighbours.First;
        while not (Neighbour = NULL) do
        begin
          //determine which vertice of new triangle is different
          distinctVertice = CheckVertices(Neighbour,v1,v2,v3);
          //insert triangle into neighbours list
          Neighbour.NeighboursList.Insert(newTriangle,
                                          distinctVertice);
        end;
        //update piece data (k-dop limits, center)
        Piece.UpdateInternalData();
        //stop piece list search
        PieceFound = true;
        break;
      end;
    end;
    Piece = Piece.Next;
  end;

  //if corresponding piece was not found create new piece and
  //calculate its internal data
  if not PieceFound then
  begin
    newPiece = PieceList.CreateNewPiece(v1,v2,v3);
    newPiece.UpdateInternalData();
    inc(PieceCount);
  end;
end;
```

Algorithm goes through list of convex patches and checks if a triangle can be added to current convex patch. New triangle has to be checked against all triangles on current convex patch. Three conditions must be met:

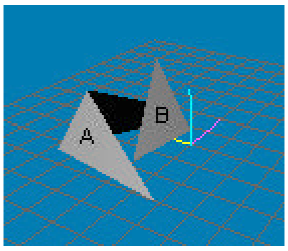
(a) normals of both triangles are facing on the same side of the patch,
(b) the surface of other triangle must not be visible from the first triangle and vice versa,
(c) if triangles are not touching then the triangle being added must not intersect the plane formed by second triangle.



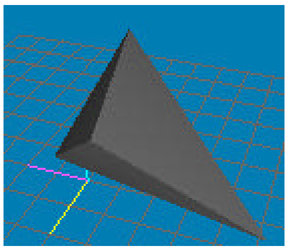
Picture 10 : Example of breaching the first rule: normals of triangles A and B are not facing on the same side.



Picture 11 : Second condition violated: triangle B can be seen from triangle A and vice versa.



Picture 12 : Triangle does not conform to third rule: if triangle A is added the plane formed by triangle B would be intersected.



Picture 13 : An example of convex patch formed by three triangles.

Current convex patch is not the one to which the new triangle belongs if one of the conditions above is not satisfied. New 18-dop's boundaries and new center of convex patch need to be calculated if the new triangle was sucessfully added. A new convex patch on which the new triangle is added is created in case when the new triangle doesn't belong to any of the already existing convex patches.

3.3.2. Building tree hierarchy

After all triangles were added the second step can begin. Tree hierarchy is built recursively. Recursion is stopped when all nodes that weren't splited yet contain only one convex patch. Tree hierarchy is built using the following algorithm:

```
procedure DivideTree(node);
begin
  //divide only if there are more than one piece in this node
  if (node.PieceCount > 1) then
  begin
    //compute mean and variance of all piece centers in node
    //(computed separately for X,Y and Z axis)
    ComputeVariance(node,mean_x,mean_y,mean_z,var_x,var_y,var_z);
    //create left and right son
    Left = new TreeNode;
    Right = new TreeNode;
    //choose direction to split (splatter method)
    sort(var_x,var_y,var_z,outArray);
    if (var_x = outArray[0]) then //var_x is largest
    begin
      piece = node.PieceList.First;
      while not (piece = NULL) do
      begin
        if (Piece.Center.x < mean_x) then begin
          Left.InsertPiece(Piece);
          Left.ProcessNode();
        end else if (Piece.Center.x > mean_x) then begin
          Right.InsertPiece(Piece);
          Right.ProcessNode();
        end else begin //pieces with Center.x on mean_x
          //put first, third, fifth etc in left node
          //others in right node
          if evenPos then begin
            Left.InsertPiece(Piece);
            Left.ProcessNode();
          end else begin
            Right.InsertPiece(Piece);
            Right.ProcessNode();
          end;
        end;
      end;
    end else if (var_y = outArray[0]) then var_y is largest
    begin
      piece = node.PieceList.First;
      while not (piece = NULL) do
      begin
        if (Piece.Center.y < mean_y) then begin
          Left.InsertPiece(Piece);
          Left.ProcessNode();
        end else if (Piece.Center.y > mean_y) then begin

          Right.InsertPiece(Piece);
          Right.ProcessNode();
        end else begin //pieces with Center.y on mean_y
          //put first, third, fifth etc in left node
          //others in right node
          if evenPos then begin
            Left.InsertPiece(Piece);
            Left.ProcessNode();
          end else begin
            Right.InsertPiece(Piece);
            Right.ProcessNode();
          end;
        end;
      end;
    end else begin //var_z is largest
      piece = node.PieceList.First;
      while not (piece = NULL) do
      begin
        if (Piece.Center.z < mean_z) then begin
          Left.InsertPiece(Piece);
          Left.ProcessNode();
        end else if (Piece.Center.z > mean_z) then begin
          Right.InsertPiece(Piece);
          Right.ProcessNode();
        end else begin //pieces with Center.z on mean_z
          //put first, third, fifth etc in left node
          //others in right node
          if evenPos then begin
            Left.InsertPiece(Piece);
            Left.ProcessNode();
          end else begin
            Right.InsertPiece(Piece);
            Right.ProcessNode();
          end;
        end;
      end;
    end;
  end;
end;
```

The described algorithm splits a node into two nodes (left and right son). The recursion itself is executed one level higher (where described procedure is called) and is finished when no nodes can be divided any more (all nodes contain only one convex patch). Similar to QuickCD, one of the axis X, Y and Z is used as splitting axis. The axis on which variance of projections of convex patch centers is the greatest is selected, this method is called splatter in [1]. With splatter method we project centers of convex patches to axis X, Y and Z, then variance of projected points is calculated. The axis with biggest variance is selected. Splitting point is mean value of projected points on selected axis. In case when convex patch center's projection is exactly the same as mean value of all projections we add every second such convex patch into right son, other such paches are added into left son.

Now *ProcessNode* procedure, which is called from described algorithm will be described. Boundaries of node's bounding volume change when a convex patch is added into node. By using the already computed bounding volume of convex patch (calculated in first step of building the hierarchy tree – adding object's triangles into root of the tree). We don't have to check all triangles in node again, only boundaries of node's bounding volume and bounding volume of newly added convex patch need to be merged. This is a simple comparison of boundaries and selection of bigger or smaller value.

3.4. Collision detection

Collision detection algorithm first compares hierarchy trees of two possibly colliding objects. Triangle-triangle tests need to be performed if tree leafs are overlaping.

3.4.1. Hierarchy tree comparison

Tree comparison uses recursion, algorithm also allows interuption in cases when an optionaly given timeout for execution is exceeded. The heuristics used for selection of node that will be first unfolded is as follows:

(a) If both nodes are leafs do a triangle-triangle test.

(b) If one node is leaf and the other an internal node, unfold internal node with the smaller son being visited first.

(c) If both nodes are internal, unfold the larger node first.

Pseudo code for hierarchy tree comparison:

```
function TraverseTree(node1,node2) : boolean;
begin
  //check if allowable time elapsed
  if Timeouted then Exit;
  //transform limits of object two
  TransformLimits(transformMatrix,node2.Limits,Limits1);
  //check if intervals overlap
  if not IntervalsOverlap(Limits1,node1.Limits) then
    return false;
  else begin

    if Leaf(node1) and Leaf(node2) then
      CheckTriangles(node1,node2);
    else if Node(node1) and Leaf(node2) then
    begin
      c1 = TraverseTree(SmallerSon(node1),node2);
      if not StopCondition then
      begin
        c2 = TraverseTree(LargerSon(node1),node2);
        collision = c1 or c2;
      end;
    end else if Leaf(node1) and Node(node2) then
    begin
      c1 = TraverseTree(node1,SmallerSon(node2));
      if not StopCondition then
      begin
        c2 = TraverseTree(node1,LargerSon(node2));
        collision = c1 or c2;
      end;
    end else begin //both nodes are internal
      if Volume(node1) > Volume(node2) then begin
        Larger = node1; Smaller = node2;
      end else begin
        Larger = node2; Smaller = node1;
      end;
      c1 = TraverseTree(Larger.Left,Smaller);
      if not StopCondition then
      begin
        c2 = TraverseTree(Larger.Right,Smaller);
        collision = c1 or c2;
      end;
    end;
    result = collision;
  end;
end;
```

Algorithm first checks if timeout is exceeded. In the next step 18-dop bounding volume of second node is transformed using transformation matrix. This transformation matrix sets 18-dop of second node into the same relative position to 18-dop of first node as it would be if both 18-dops would be transformed by using their respective transform matrices. The transformation matrix is precomputed and stored. It is computed by multiplying the transform matrix of second node with inverse transform matrix of first node. By precomputing such matrix we gain on algorithm speed as this matrix is used very often through whole algorithm. The transformed 18-dop is then compared to 18-dop of first node. Search in current part of tree is aborted if 18-dops do not overlap on at least one of nine axis. In other case the search is continued using heuristics above up to the point when a conclusion can be made if objects collided or not. *StopCondition*, which is used in algorithm above is true when maximum number of collision points that can be returned by collision detection algorithm was already found. CD library has a preset maximum number of collision points that can be returned in the case when objects collided. The current version of library has this number set to 2, which is enough for use in simple games.

3.4.2. Triangle-triangle tests

Tree's leaf contains one convex patch, which is composed of several triangles. By using the fact that the patch is convex we can avoid checking each triangle with each triangle from other convex patch for intersection. From starting triangle the algorithm moves closer to plane spanned by triangle on other convex patch. If the sign of distance from plane is changed then we do the actual intersection test. The algorithm:

```
function CheckTriangles(node1,node2) : boolean;
begin
  collision = false;
  //select starting triangle for outer loop
  outertriangle = node2.PieceList.Piece.TriangleList.First;
  while not (outerTriangle = NULL) do
  begin
    //transform vertices according to transform matrix
    TransformVertices(outerTriangle,transV1_o,transV2_o,transV3_o,
                      transformMatrix);
    //compute plane on which triangle resides (normal is plane
    //normal, d is fourth parameter from plane equation)
    ComputePlane(transV1_o,transV2_o,transV3_o,normal,d);
    //choose starting triangle for inner loop
    innerTriangle = node1.PieceList.Piece.TriangleList.First;
    //put triangle into test queue
    testQueue.CopyTriangleToList(innerTriangle);
    //compute starting minimal distance from plane on which
    //outerTriangle resides, indicate if innerTriangle
    //intersects that plane
    mindist = MinimumDistance(normal,d,
                              testQueue.Triangle.V1,
                              testQueue.Triangle.V2,
                              testQueue.Triangle.V3,
                              testQueue.Triangle.IntersectsPlane);
    while not (testQueue.Triangle = NULL) do
    begin
      if testQueue.Triangle.IntersectsPlane then
      begin
        //check if triangles truly intersect, coplanar indicates
        //that both triangles share all three points which define
        //them, P1 and P2 are intersection points
        if TrianglesIntersect(transV1_o,transV2_o,transV3_o,
                              testQueue.Triangle.V1,
                              testQueue.Triangle.V2,
                              testQueue.Triangle.V3,
                              P1,P2,coplanar) then
        begin
          if coplanar then //any two points of triangle will do
          begin
            point[PointCount] = transV1_o;
            inc(PointCount);
            point[PointCount] = transV2_o;
            inc(PointCount);
          end else begin
            point[PointCount] = P1;
            inc(PointCount);
            point[PointCount] = P2;
            inc(PointCount);
          end;
          //indicate that collision occured and check if we need
          //to stop detection
          collision = true;
          if StopCondition then
            break;
        end;

        //current triangle was checked, we need to check it's
        //neighbours
        tempTriangle = testQueue.Triangle.Neighbours.First;
        //initialize starting values for neighbours
        Initialize(mindist,testQueue.Triangle.MinDist,
                   IntPlane,testQueue.Triangle.IntersectsPlane);
        //now we can remove current triangle from test queue
        testQueue.RemoveFirstFromList;
        while not (tempTriangle = NULL) do
        begin
          if NotAlreadyChecked(tempTriangle) then begin
            if (tempTriangle.DistinctVertice <> NULL) then
            begin
              temp = PointDistance(normal,d,
                                   tempTriangle.DistinctVertice);
              tempTriangle.IntersectsPlane =
                                    SignChanged(temp,mindist);
            end else begin
              temp = mindist;
              tempTriangle.IntersectsPlane = IntPlane;
            end;
            //check if we are any closer to plane
            if Closer(temp,mindist) then
            begin
              tempTriangle.MinDist = temp;
              testQueue.CopyTriangleToList(tempTriangle);
            end else begin
              tempTriangle.MinDist = mindist;
              if tempTriangle.IntersectsPlane then
                testQueue.CopyTriangleToList(tempTriangle);
            end;
            MarkTriangleAsChecked(tempTriangle);
          end;
          tempTriangle = tempTriangle.Next;
        end;
      end;
    end;
    //check stop condition
    if StopCondition then
      break;
    outerTriangle = outerTriangle.Next;
  end;
  //return result
  return collision;
end;
```

The outer loop loops through triangles from the second node. Vertices of those triangles need to be transformed, after transformation they must be in the same relative position to vertices from the first node as they would be if vertices in both nodes would be transformed. The transformation matrix mentioned in 3.4.1 is used here too. Next step is to calculate the plane on which the currently selected triangle form outer loop lies. In inner loop, the distance between the currently selected triangle in this loop and plane of current triangle in outer loop is calculated. Whole triangle-triangle test is performed when a triangle from inner loop intersects the plane. If all points of collision haven't been found yet (*StopCondition* is *false*), the algorithm starts to look at neighbours of current triangle in inner loop. Triangles with smaller or equal distance from plane spanned by current triangle in outer loop and triangles that intersect that plane are stored in queue with triangles that need to be tested. By doing that we are getting closer and closer to the plane with each iteration. If a neighbouring triangle which intersects the plane is found we mark such triangle. Those triangles are always added to the top of the queue. Whole triangle-triangle test is performed when such triangle is selected in loop. The inner loop ends when the queue is empty and next triangle is selected in outer loop. The whole procedure is ended when *StopCondition* becomes *true* or all of the triangles in outer loop were compared against triangles from inner loop.

The triangle-triangle test will be shortly described next. Algorithm first checks if vertices of first triangle are on opposite sides of plane spanned by second triangle (this check is already included in algorithm described above). If this is true the same think is checked in other way (does the second triangle intersects the plane spanned by first triangle). If this is also true then intersection points are calculated for both cases which gives us two lines. Lines are then compared and the result is returned (do triangles intersect and the points where they intersect). More about triangle-triangle and other intersection tests can be found in [9]. The test from [12] was used in CD. It was slightly modified to remove calculation of the already calculated information.

3.5. Results

CD was tested with simultaion of moving one car through the other and by checking for collision between two spheres where one sphere was a bit smaller than the other and positioned inside the bigger sphere. Average times for detection of collision and average times for confirmation that there is no collision were measured. Ten different versions were tested – the difference between them was the maximum number of triangles on convex patch (1 to 10). Times were measured on the following computer configuration:

? Intel Celeron 433, 128Mb RAM

? Windows98, DirectX 8.0 SDK (Retail library version)

3.5.1. Measurements

The following table contains data for test where one car moves through the other. The moving car is composed from 905 triangles, the non moving car from 878 triangles. The moving car first approached the non moving car in a straight line, when they were positioned on the same location the moving car performed a 360? rotation and the continued movement in straight line all the way to the finishing point .

Max. number of triangles	Collision confirm [ms]	No collision confirm [ms]	Average [ms]
1	0.626	0.033	0.429
2	0.475	0.034	0.320
3	0.553	0.030	0.349
4	0.459	0.032	0.296
5	0.513	0.031	0.333
6	0.422	0.033	0.274
7	0.460	0.037	0.296
8	0.459	0.035	0.297
9	0.467	0.032	0.309
10	0.469	0.033	0.313

Table 2 : First test timings.

The results shows that convex patches with maximum six triangles perform best. Time for confirmation that no collison occured is almost the same for all versions due to the fact that the test usually finishes on top levels of tree hierarchy. The results are very much dependant on object's shape and the number of triangles from which the object consists. This means that it is good to test various versions (different maximum number of triangles) of algorithm before using it to get the optimal performance.

The second test was a test with two spheres, one slightly smaller than other and positioned inside the bigger one. Both spheres are composed from 1984 triangles. A large portion of both hierarchy trees must be searched before the algorithm can conclude that there is no collision. Measured times are given in the following table.

Max. triangles	1	2	3	4	5	6	7	8	9	10
Time [ms]	2.614	1.819	1.767	1.266	2.419	1.537	3.030	1.792	3.240	2.590

Table 3 : Second test measurements.

It is obvious that this is worst case scenario, so the times measured are higher than in first test. It is interesting that convex patches with even maximum perform better, but then again that could be down to the number of triangles in sphere, so we can again conclude that before using the algorithm it is good to do some testing to determine which maximum number of triangles is best.

3.5.2. Conclusion

The current version of CD uses convex patches composed of maximum 6 triangles. This can be easily changed in code by changing a single constant. This number is used because it best suits the simple game described in next chapter. Algorithm could be improved with use of time coherence as this would reduce the amount of time when comparing two hierarchy trees twice in consecutive frames. At the moment the algorithm uses from 60 to 80% of all time during collision detection for comparing trees. It is obvious that this is the area where the biggest improvements can be gained. Improvements are possible on the last stage too, that is triangle-triangle tests. As you could see convexity is exploited only on one patch, if it would be exploited on other patch too the number of necessary tests would decrease even further. Those improvements were not done due to lack of time.

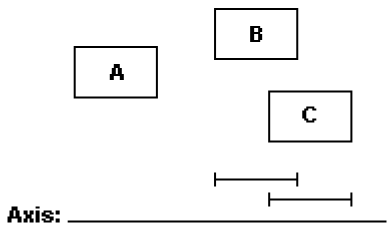
4. Use of collision detection algorithms

4.1. Necessary collision tests

In general, a collision detection algorithm is used to check if there is a collision between two objects in space. But in reality we have many objects that can collide with each other. So we have to come up with a method that determines which objects really need to be tested as testing all object against each other would be time consuming. First improvement that comes to our mind is that we don't really need to do tests between pairs of static objects. So we have at most *(N(N-1)(N-3)...(3)(1)) + NM* necessary collision tests where N is the number of dynamic objects and *M* is the number of static objects. Some methods for determining necessary collision tests are described in this chapter.

4.1.1 Axis sort

The idea behind this method is to project bounding volumes of all objects onto some selected axis. If projections of two distinct bounding volumes overlap we need to use collision detection algorithm to precisely determine if there really is a collision. This method behaves very good in cases where the scene is sparse but it's performance quickly deteriorates when the scene is dense.



Picture 14 : Axis sort, pair of objects B and C needs to be tested for collision.

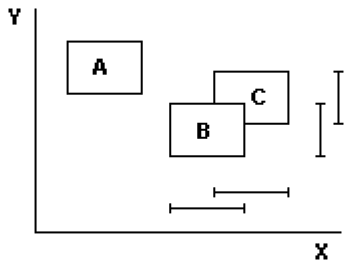
4.1.2 Grid

This method uses more information about object's location in space. We can use 2D or 3D grids. The size of the cell in grid must be bigger than the biggest bounding volume of any object. This condition assures that the objects for which there is a possibility of collision are spaced at most one cell apart. Objects are sorted into cells based on their bounding volume's (usually AABB or sphere) center point. As a consequence of that at most half of a given bounding volume can be in neighbouring cell. This also means that objects spaced at least two cells apart can not collide. There are two simple rules to create a list of necessary tests:

(a) if cell contains more than one object then all possible pairs of objects are added to necessary tests list,

(b) if the cell is not empty and some of it's neighbouring cells are also not empty we add all possible pairs of objects in those cells to necessary tests list.

With proper searching method we avoid double checking of any cell (for a 2D grid we would start searching the grid in upper left corner and check only the cells below and right of the current cell before moving to next cell and repeating the same procedure). Algortithm for 3D grid would be similar we just need to take the third dimension into account. If we consider the fact that a given object does not change it's cell too often, we can improve this method by updating the necessary tests list only when at least one object has changed the cell to which it belongs.



Picture 15 : Example of 2D grid, test between B and C is needed.

4.1.3 Comparison of intervals on X, Y and Z axis

This algorithm maintains a list of active pairs of objects (objects for which we need to do collision test) and a sorted list of intervals for X, Y and Z axis. AABB is used as bounding volume. Each pair of AABB's has three flags. Those indicate if the pair overlaps on X, Y or Z axis. If all three flags are set it means that AABBs are overlaping. Based on the values of flags we have one of the following situations:

(a) All three dimensions of AABBs are overlaping thus we add this pair of objects to list of active pairs.

(b) AABBs were overlaped before but they do not overlap at the moment. Corresponding pair of objects is removed from list of active pairs.

(c) AABBs are still not overlaped.

When list of active pairs is completed all pairs of objects in list are tested for collision. This algorithm is used I-COLLIDE and is described more into detail in [6].

4.1.4 Use of human perception

This method is based on relatively unknown area of human perception of enviroment. Objects that are likely to be spoted first (are bigger, closer to us) have bigger priority. Other factors can be also considered. Algorithm uses a simple model of human perception described in [4]. Two lists are maintained: active collision list (objects whose bounding volumes are overalped and need collision test) and a real collision list (objects that actually collided). Active collision list is filled with pairs of objects whose bounding volumes overlap. Pairs of objects are removed from active collision list when collision test for them is finished. If a pair of objects actually collided it is added to real collision list. Collision tests are scheduled with various methods. Simpler methods that don't use human perception information are *round-robin scheduling* and *sequential scheduling*. Methods that use perceptual information are *perceptualy sorted sequential scheduling* and *priority queue scheduling*. It has been shown that *priority queue scheduling* is the most effective scheduling mechanism. For description of the last two methods see description in [4]. The real collision list is used only for object's response to collision.

4.2 Object's collision response

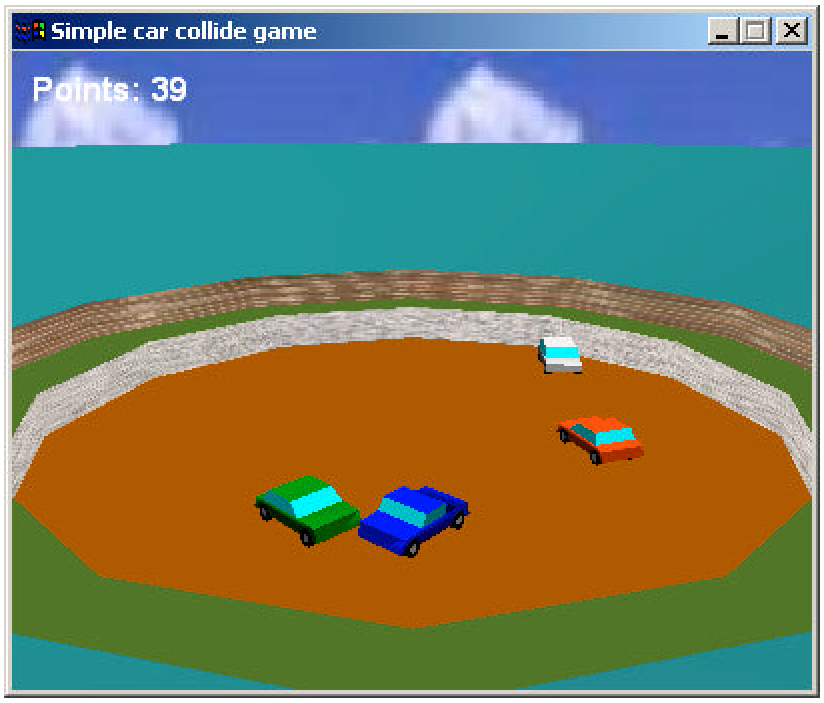
Usually we need object's collision response. Accuracy of collision detection algorithm's result and physical model determine how accurate the response will be. A simple physical model should include at least data about mass, velocity and if possible object's acceleration. Velocity and acceleration data is usually given as a vector which tells us the velocity (acceleration) in all three directions (X, Y and Z axis). Collision detection algorithm's result is needed to calculate new movement directions of both colliding objects (a simple approach is to calculate the vector starting at point of collision and ending in object's center). New speeds and accelerations are calculated afterwards. Better physicals models also consider other factors such as:

(a) gravity
(b) air resistance
(c) friction
(d) and many more

See more about basic physical models in [10].

4.3 Game example

To demonstrate the use of CD library I made a simple car colliding game. The goal is to collide as many times as possible in 30 seconds. Some methods described in this chapter are used in this simple game.



Picture 16 : Car colliding game.

4.3.1 Data structure

Only the important parts of code will be described - classes *CCollidableMesh* and *CGrid*. *CCollidableMesh* contains all data about object. Data included are mass, velocity and some internal data structures needed while testing for collision. For each object we can check whether it collided with another object and if it so collision response can be calculated (new direction of movement, new speed) – see 4.3.3. *CGrid* is a class that implements a 2D grid which we use to determine the necessary tests – see more in 4.3.2.

4.3.2 Necessary collision tests

The grid method is used in game. It is a 2D grid of 3x3 size. Position of car in grid is calculated each time when it moves. Algorithm is very simple:

```
procedure SaveGridPosition(object);
begin
  for each_cell_in_grid do
  begin
    if IsInCell(object) then
      StorePosition(Object);
  end;
end;
```

The search begins in grid's top left corner and is finished when the corresponding cell is found.

After grid positions are calculated grid can be used to determine necessary collision tests. Pseudo code of algorithm that calculates necessary tests for a given car:

```
procedure DetermineTests(object);
begin
  cells = DetermineCellsToCheck(object);
  for cells do
  begin
    if NeighbourFound then begin
      StoreInObjectList(Neighbour);
      inc(numberOfTestNeeded);
    end;
  end;
end;
```

In next step we calculate available time for each collision test. Amount of time available for each test depends on number of necessary tests for current car, maximum posssible number of collisions for this car and number of collision tests which were not performed for previous car. We have *TOTALTIME* time available for all tests. *TOTALTIME* is set to 30 miliseconds. We have 4 cars and this means at most six different collisions. For human car collisions we have *TOTALTIME/(3+numberOfTestNeeded)* available time for each test. That would be at most *TOTALTIME/3* (but there is no test to be done in this case) and at least *TOTALTIME/*6. In second step (one computer car against remaining two computer cars) the available time is calculated with this formula: *TOTALTIME/(4–numberOfTestNotDoneInFirstStep+numberOfTestNeeded).* In this case we have at most *TOTALTIME* time (in case when human car did not have any collision and there is no collisions in this step either) and at least *TOTALTIME/6*. The available time for the last possible collision is calculated with the following formula: *TOTALTIME/(6–numberOfTestNotDoneBefore).* That means from *TOTALTIME/6* to *TOTALTIME* time available. In every step collision tests are performed immediately after allowable time is calculated. If allowable time is exceeded the test is interupted. In this case it is assumed that there is a collision and the points of collision are taken from car's bounding volume.

The purpose of described timeout use serves only as a demonstration of use. Actually, we do not need to use timeouts in our simple game as CD is fast enough for such use. Collisions of car against arena are not checked since it is simpler just to limit the car movement as the arena is roundly shaped.

4.3.3 Collision response

Collision response is implemented in *CCollidableMesh*. Only mass and velocity are used to calculate the new velocity. Pseudo code:

```
...
point = ReadPointOfCollision

dir1 = Object1.Position – point;
Normalize(dir1);

dir2 = Object2.Position – point;
Normalize(dir2);

momentum1 = Length(Object1.Speed) * Object1.mass;

momentum2 = Length(Object2.Speed) * Object2.mass;

SetSpeed(Object1,(momentum2/Object1.mass)*dir1);
SetSpeed(Object2,(momentum1/Object2.mass)*dir2);
...
```

Point of collision which is needed to calculate the new movement directions of both cars is obtained first. New car velocities in all three directions (X, Y and Z) are calculated next on the basis of new directions, previous velocities and car masses.

Literature

[1] Efficient collision detection for interactive 3D graphics and virtual enviroments (James Thomas Klosowski; State University of New York at Stony Brook; 1998)

[2] Efficient collision detection of complex deformable models using AABB trees (Gino Van Den Bergen; Eindhoven University of Technology; 1998)

[3] OBBTree: A hierarchical structure for rapid interference detection (S. Gottschalk, M. C. Lin, D. Manocha; University of North Carolina; 1996)

[4] Real-time collision detection and response using sphere-trees (O'Sullivan, C. Dingliana; J. Image Synthesis group, Trinity College Dublin)

[5] Accurate and fast proximity queries between polyhedra using convex surface decomposition (Stephen A. Ehmann, Ming C. Lin; University of North Carolina; 2001)

[6] I-COLLIDE: An interactive and exact collision detection system for large scale enviroments (Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, Madhav K. Ponamgi; University of North Carolina)

[7] An efficient collision detection algorithm for polytopes in virtual enviroments (Kelvin Chung Tat Leung; University of Hong Kong; 1996)

[8] The QuickHull algorithm for convex hulls (C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa; 1996)

[9] ERIT – A collection of efficient and reliable intersection tests (Martin Held; Institut fur Computerwissenschaften Universitat Salzburg)

[10] Beginning DIRECT3D game programming (Wolfgang F. Engel, Amir Geva; Prima Tech's game development series; 2001)

[11] Collision detection : Algorithms and applications (Ming C. Lin, Dinesh Manocha, Jon Cohen, Stefan Gottschalk; Army Research Office, University of North Carolina)

[12] http://www.acm.org/jgt/papers/Moller97/tritri.html (Tomas Moller, 1997)

Apendix: CD API

To use CD library in your application you need to include *CollisionObject.h* header. Library's interface is quite simple. The main library's class is *CCollisionObject* which includes the following methods:

```
void addTriangle(D3DXVECTOR3 v1,D3DXVECTOR3 v2,
                 D3DXVECTOR3 v3)
void processTriangles()
void SetTransformMatrix(D3DXMATRIX* m)
bool CheckCollision(CCollisionObject* other,int timeout=0)
void GetCollisionPoint(D3DXVECTOR3 *point[MaxPointNum],
                       int* pointsDetected)
```

Methods are described in following paragraphs. There is another class that is available to user - *CInconsistency*. It's purpose is error handling. E.g. Exception *CInconsistency* is thrown if user calls *CheckCollision* before data was prepared with call to *processTriangles*. It can also be thrown when other data related problems occur. User can catch the exception and handle it. Example of use:

```
...
try {
  if (obj1->CheckCollision(obj2)) {
    D3DXVECTOR3 *p[MaxPointNum], temp;
    int num;

    cout << "collision!\n";
    obj1->GetCollisionPoint(p,&num);
    cout << "Number of detected points: " << num << "\n";
    for (int i=0; i<num; i++) {
      temp = *p[i];
      cout << "P" << i <<": x: " << temp.x << " y: " <<
      temp.y << " z: " << temp.z << "\n";
      cout.flush();
      delete *p[i];
    }
  } else cout << "no collision!\n";
}
catch (CInconsistency) {
  cout << "Data error!\n";
}
...
```

Library's interface also includes the following function, which is not a member of *CCollisionObject* class:

```
CCollisionObject* newCollisionObject()
```

This function returns a new instance of *CCollisionObject* object. It has to be called in your application before any of methods in *CCollisionObject* can be used. Example:

```
...
CCollisionObject* obj = newCollisionObject();
...
```

CCollisionObject's methods are described next.

**void addTriangle(D3DXVECTOR3 v1, D3DXVECTOR3 v2,
D3DXVECTOR3 v3)**

Parameters:

D3DXVECTOR3 v1,
D3DXVECTOR3 v2,
D3DXVECTOR3 v3 : triangle vertices, the order of vertices is important since the algorithm uses this information to calculate normal of plane spanned by triangle.

How it works:

Method is used to add triangle in *CCollisionObject's* internal data structure. See example for method *processTriangles*.

void processTriangles()

How it works:

This method must be called after all triangles were added to *CCollisionObject's* internal data structure by using method *addTriangle*. Example:

```
HRESULT CCollidableMesh::Create(LPDIRECT3DDEVICE8 pd3dDevice,
        TCHAR* strFilename,
        bool createtree)
{
  HRESULT rc=CD3DMesh::Create(pd3dDevice,strFilename);
  //read triangles to tree structure
  if (SUCCEEDED(rc) && createtree) {
    m_ColModel = newCollisionObject();
    LPD3DXMESH clone;
    if (SUCCEEDED(m_pSysMemMesh->CloneMeshFVF
                  (D3DXMESH_SYSTEMMEM,D3DFVF_XYZ,
                   pd3dDevice,&clone))) {
      int vnum=clone->GetNumVertices();
      int fnum=clone->GetNumFaces();
      CUSTOMVERTEX* v;
      if (SUCCEEDED(clone->LockVertexBuffer
                    (D3DLOCK_READONLY,(BYTE**)&v))) {
        WORD* ind;
        if (SUCCEEDED(clone->LockIndexBuffer
                      (D3DLOCK_READONLY,(BYTE**)&ind))) {
          for(int i=0;i<fnum;i++) {
            m_ColModel->addTriangle((float*)&v[ind[i*3+0]],
                                    (float*)&v[ind[i*3+1]],
                                    (float*)&v[ind[i*3+2]]);
          }
          m_ColModel->processTriangles();
          clone->UnlockIndexBuffer();
        }
        clone->UnlockVertexBuffer();
      }
      clone->Release();
    }
  }
  return rc;
}
```

void SetTransformMatrix(D3DXMATRIX* m)

Parameters:

D3DXMATRIX m* : matrix of size 4x4 which contains position and rotation of object's local coordinate system with regards to world coordinate system.

How it works:

Always when orientation and/or position of object in space changes we need to reset *CCollisionObject's* internal transform matrix. Results will be inaccurate if this is not done. Example:

```
...
D3DXMATRIX tr;

tr._11=1.0f; tr._12=0.0f; tr._13=0.0f; tr._14=0.0f;
tr._21=0.0f; tr._22=1.0f; tr._23=0.0f; tr._14=0.0f;
tr._31=0.0f; tr._32=0.0f; tr._33=1.0f; tr._14=0.0f;
tr._41=-5.0f; tr._42=-5.0f; tr._43=-5.0f; tr._14=1.0f;

obj->SetTransformMatrix(&tr);
...
```

bool CheckCollision(CCollisionObject* other, int timeout = 0)

Parameters:

CCollisionObject other* : pointer to object which is being checked for collision with this object
int timeout : time available for calculation of the result, no time limit is applied if timeout is not given.

Result:

true if objects collided otherwise *false*.

How it works:

This method checks whether two objects collided. Timeout after which the result must be returned can be applied. *True* is returned if exact result could not be obtained in given time and two points from current node's bounding volume are returned as points of collision. Algorithm always executes completely if timeout is not given and returns *true* if objects collided or *false* if objects do not collide. Example:

```
...
timeout = 10; //10 ms timeout
//call with timeout setting...
bool Colliding1 = m_ColModel->CheckCollision(other->m_ColModel,
                                             timeout);
//...and without it...
bool Colliding2 = m_ColModel->CheckCollision(other->m_ColModel)
...
```

**void GetCollisionPoint(D3DXVECTOR3 *point[MaxPointNum],
int* pointsDetected)**

Parameters:

*D3DXVECTOR3 *point[MaxPointNum]* : array of pointers to structure D3DXVECTOR3 which will be filled with points of collision. *MaxPointNum* is defined in header *CollisionObject.h* and determines maximum number of collision points that are searched for by method *CheckCollision*.
int pointsDetected : pointer to number of collision points found.

How it works:

This method is used if we need to know where objects collided. Example:

```
...
m_ColModel->GetCollisionPoint(p,&num);

D3DXVECTOR3 dir1=D3DXVECTOR3(m_Transform(3,0),m_Transform(3,1),
                             m_Transform(3,2)) - *p[0];
dir1 /= D3DXVec3Length(&dir1);

float momentum1 = D3DXVec3Length(&m_Speed) * m_Mass;
float momentum2 = D3DXVec3Length(&other->m_Speed) * other->m_Mass;

D3DXVECTOR3 dir2=D3DXVECTOR3(other->m_Transform(3,0),
                             other->m_Transform(3,1),
                             other->m_Transform(3,2)) - *p[0];
dir2 /= D3DXVec3Length(&dir2);

SetSpeed((momentum2/m_Mass)*dir1); //first object
other->SetSpeed((momentum1/other->m_Mass)*dir2); //second object
...
```