A solution to the problem of Collision Detection in large environments

Marta Franquesa–Niubó Pere Brunet

20th June 2000

Abstract

Collision detection between several objects in a closed space is an interesting research topic in all those areas where objects must not penetrate one another, such as Computer Graphics, Computer Vision, Virtual Reality. In this paper, we describe a new data structure design that is useful for collision detection in the environment of ship design applications and several algorithms that solve the collision problem in such environment. The paper is focused on collision detection in complex systems with a large number of objects.

1 Introduction

Whenever two objects attemp to interpenetrate each other, we call it a collision. Collision detection between objects is of fundamental relevance in order to achieve efficiency and realism in several areas such as robotics, motion planning, computer animation, dynamic simulation, virtual reality and in all those applications where objects can not interpenetrate one another. The two main issues involved are those detecting that a collision has ocurred (*collision detection*) and then describing a response as an effect of collision (*collision response*). This document is focused on collision detection properly and not on collision response that is a dynamic problem which involves phisical laws to predict the behaviour of the objects that would penetrate. For a discussion of response algorithms, see [Kam93], [Bar90], [MW88], [Can86] or [FHA90]. Collision detection is a geometric interference problem that depends on the spatial relationship of objects.

The collision detection problem can be attached from two different point of views, *static collision* and *dynamic collision* (where objects are moving). One way to treat the dynamic collision detection is sampling time and reducing the problem to multiple calls to static inteference test. This approach is called *multiple interference detection*. Dynamic collision can be achieved in other ways: swept volume interference, extrusion in 4D space and trajectory parameterization. In the present document, we are interested in static collision. For more information about dynamic collision approaches see [JTT98].

In general collision algorithms consist of two main stages. In the first, *broad stage*, an approximate test is performed to identify interfering objects in the entire workspace using a coarse representation of object shapes (simplified forms of objects such as bounding volumes). In the second, *narrow* or *fine stage*, a tightly representation of object shapes is used to accurately identify any object parts that cause interference and collision. In this last stage the algorithms focus its attention only on objects (or regions) that are likely to collide.

The goal of the broad stage is to restrict the application of the interference test to those object parts at which a collision can truly occur. For this purpose objects are usually modeled by hierarchies of simple shapes (such as bounding boxes or spheres). Hierarchical approximations allow to determine which regions are susceptible of interfering. Then fine interference test need only be applied within the relevant regions. In [FB98] there is a detailed description of models and algorithms for collision detection problem.

In this paper, we describe a new data structure design that is useful for broad collision detection in the environment of ship design applications and several algorithms that solve the collision problem in such environment. The paper is focused on collision detection in complex systems with a large number of objects.

The remainder of this paper is structured as follows. In section 2 the problem of collision detection depending of the objects involved is exposed. In section 3 some definitions and properties of the environment are introduced. Section 4 explains the new data structure for collision detection and the way to generate it. In sections 5 and 6 algorithms for collision detection are presented. Finally, section 7 summarizes the analysis of memory occupancy and complexity of algorithms.

2 The Problem

The problem of collision detection between a large number of objects in a closed space can be divided in two kind of different problems:

1. *All–versus–all*: This case is when we are looking for collision detection between all the objects of the space.
2. *One–versus–all*: When a large number of objects are located in a finite closed space and we want to insert a new object in this space, it is necessary to examine if this new element collide with an existing one. In this case we are speaking about collision detection between one element and all the others.

We have to take into account that collisions between object primitives with the same maximal set originally connected that them belong to has not to be either detected and computed.

3 Definitions and properties concerning of objects and their relationships

In this section, the objects and elements that can be involved in collision detection in ship design applications are described.

3.1 Class of elements

Objects: The objects that appear inside the ship can be classified in three specific types: pipes (1D), walls (2D) and equipment-elements (3D). To reference all the objects that belong to a compartment of the ship we will use the notation:

$$\text{set of objects: } \{O_1, ..., O_t\}$$

Each object has a flag indicating the maximal set originally connected that it belongs to.

Group: Is a group of elements. The elements can be objects or groups. In this way, a hierarchy (tree structure of elements) can be implemented. In other words, a group is a subtree in the hierarchy of objects. We reference the set of groups as:

$$\text{set of group: } \{G_1, ..., G_h\}$$

We will note an object O_i that belongs to a group G_j as $O_i \in Objects(G_j)$. The function $Objects(G_j)$ give us the collection of objects in the subtree of G_j.

3.2 Definitions

In this section a collection of definitions related to the environment are exposed. The definition 3.2.1 restricts the problem of collision detection to those cases that are needed. The definition 3.2.2 introduces the concept of multiresolution in the representation of elements. The geometric model used to approximate the geometry of objects associated to each resolution level is defined in definition 3.2.3. The thereshold of tolerance associated to each level of resolution, the volume distance between the geometric representation of objects and, the geometry of objects is defined in definition 3.2.4. Finally, the concept of a minimum box that approximate a representation is presented in definition 3.2.5.

Definition 3.2.1 *A collision between two or more objects is considered a real collision when the objects have non–null intersection and their flags are different.*

Definition 3.2.2 *The level of detail (resolution) of the object representation will be denoted by LOD . For each level exists a representation of each object. As the level increases the representation is more accurate. We will assume n levels of detail:*

$$\text{LOD} = \{\ d_1, ..., d_n\ \} : \text{Levels of detail}$$

Definition 3.2.3 *An* **offset–dop** *is a surrounding polyhedron of an element. Each object and each group has a set of offset-dops, one for each level of detail d_k (LOD). The types of offset-dops can be: AABB (axis aligned bounding box), K-dop, convex-hull, offset-polyhedra (an offset-polyhedron is a bounding approximation of the element with a small number of faces). The kind of offset-dop selected is depending on the type of object (see figure 1).*

$$Od(O_i, d_k)\text{: Is the offset-dop of object } O_i \text{ at the } d_k \text{ level of detail}$$

For each object O_i and for each level of detail d_k there exists one offset–dop:

$$O_i \longleftrightarrow Od(O_i, d_k)$$

As mentioned above the concept of offset–dop is also applied to groups. Thus $Od(G_j, d_k)$, is the offset–dop of the group G_j at the d_k level of detail (see figure 1).

When a object is non–convex its $Od(O, d_k)$ will be non–convex for high values of d_k. Then, in this case the $Od(O, d_k)$ is generated as a collection of convex components. In other words, its offset–dop becomes a BSPtree with Ncc convex components at the tree leaves. The cost of subdividing a model into convex components can be accomplished in $O(K + v^3)$, obtaining $O(v^2)$ convex components where v is the number of concave vertices and K the number of faces of the model [FB97].

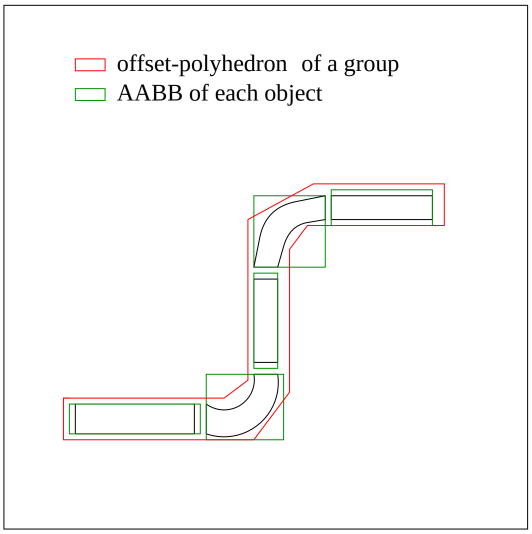

Figure 1: Example of offset–dops

Definition 3.2.4 *A maximum geometric error* ε_k *is associated to every* d_k*. The error bound* ε_k *measures the maximum allowed volume distance between objects* O *and their* $Od(O, d_k)$ *and between groups* G *and their* $Od(G, d_k)$*. Then, we will say that the volume distance between* O *and* $Od(O, d_k)$ *is less than* ε_k*,* $Dist(Od(O, d_k), O) < \varepsilon_k$*, iff:*

$$\forall p \in O \quad \exists\ p\prime \in Od(O, d_k) \mid dist(p, p\prime) < \varepsilon_k \quad and \quad \forall p\prime \in Od(O, d_k) \quad \exists p \in O \mid dist(p\prime, p) < \varepsilon_k$$

and, the volume distance between G *and* $Od(G, d_k)$ *is less than* ε_k*,* $Dist(Od(G, d_k), G) < \varepsilon_k$*, iff*

$$\begin{array}{c} \forall p \in Objects(G) \quad \exists\ p\prime \in Od(G, d_k) \mid dist(p, p\prime) < \varepsilon_k \\ and \quad \forall p\prime \in Od(G, d_k) \quad \exists p \in Objects(G) \mid dist(p\prime, p) < \varepsilon_k \end{array}$$

Definition 3.2.5 *We call a* $MinBox^*$ *the mimimum AABB box that bounds an offset–dop* $Od(E, d_k)$ *enlarged with an offsset* ε_{k+1}*. It is denoted as* $MinBox^*(Od(E, d_k))$*, where* E *can be an object or a group.*

$$MinBox^*(Od(E, d_k)) = MinAABB(Offsset(Od(E, d_k), \varepsilon_{k+1}))$$

This definition permitting us to guarantee that $\forall E$:

$$Od(E, d_{k+1}) \subset MinBox^*(Od(E, d_k))$$

3.3 Properties

As a conseqüence of the last definitions there exist some properties that we expose in that follows. Thus, the number of faces of offset–dops satisfy the property 3.3.1. The volum distance between objects and their offset–dops are restricted to satisfy the property 3.3.2. The $MinBox^*$ of offset–dops at different levels of detail have to satisfy the property 3.3.3

Property 3.3.1 *The number of faces of offset–dops must satisfy:*

$$\begin{array}{l} Numfaces(Od(O_i, d_k)) > Numfaces(Od(O_i, d_{k-1})) \\ and, \\ Numfaces(Od(G_j, d_k)) > Numfaces(Od(G_j, d_{k-1})) \end{array}$$

This property makes sure a monotonous increasing complexity of collision detection algorithms when the resolution d_k increases.

Property 3.3.2 *The distance measured at different levels of detail, satisfy:*

$$Dist(Od(O_i, d_k), O_i) < Dist(Od(O_i, d_{k-1}), O_i)$$

and,

$$Dist(Od(G_j, d_k), G_j) < Dist(Od(G_j, d_{k-1}), G_j)$$

In other words,

$$\varepsilon_k < \varepsilon_{k-1}$$

Property 3.3.3 *The $MinBox^*$ of each Od at different levels of detail, satisfy:*

$$MinBox^*(Od(O_i, d_k)) \subset MinBox^*(Od(O_i, d_{k-1}))$$

and,

$$MinBox^*(Od(G_j, d_k)) \subset MinBox^*(Od(G_j, d_{k-1}))$$

This property, that results as a consequence of definition 3.2.5 and property 3.3.2, guarantees that if a collision has not been detected for a low value of d_k, then it means that there is no collision for next value of d_k. In other words, the boxes ($MinBox^*$) corresponding to high levels of detail are allways contained into boxes ($MinBox^*$) associated to low levels of detail.

4 Description and construction of the Data Structures

4.1 Description

In this section the data structures to represent the whole system are presented. The main data structure is based on a *Kdtree*, where the leaves are lists of *container–sets*. Figure 2 shows an example in 2D of the data structure that the algorithm for collision detection is based on.

Kdtree: Is a 3-d tree (K=3) built in a top–down fasion. The main difference with a typical Kdtree is the kind of leaves. In our case, each leaf corresponds to one compartment of the ship. Each leaf of the Kdtree points to a list of container–sets. There will be as many lists as leaves in the Kdtree. In the case of the figure 2 there will be five lists associated to the five leaves marked as A, B, C, D, E. In the same figure, the list of container–sets corresponding to the F terminal node of the Kdtree is presented.

Container–set: Is a set of containers. There will be as many container–sets in each list as values $\{ d_1, ..., d_n \}$ levels of detail (LOD) (see fig 2).

Container: A container C is a data structure containing the geometry of the offset–dops of the scene objects within a certain $AABB$ box associated to C at the corresponding d_k LOD. The memory occupancy of the offset–dops contained in every container C is less than a memory page size. Each container of a set:

- Is contained into one and only one leaf node (corresponding to a compartment of the ship) of the Kdtree.

- Will contain the offset–dops (at an specified LOD d_k) of a set of objects contained in its associated box.

The attributes associated to each container of the container–sets are:

1. Kdtree node that contains it.
2. Max–Min coordinates of the container box.
3. d_k level of detail of the offset–dops and the ε_k geometric error associated to the d_k.
4. Explicit parent–child container pointers to the containers of the next level of detail d_{k+1}.
5. The geometry of the offset–dops of objects contained in the container.

In this way, each container–set will contain the list of attributes of their containers.

Figure 3 shows an example of the structure of container–sets in 2D. In this case, at the level d_1 of detail there is a single container that contains the offset–dops of objects. At this level those offset–dops are axis aligned bounding boxes (AABB). At level d_2, where the representation is more accurated, the offset–dops are bounding–polyhedra. In the same figure it is shown how the information is distributted in containers, and the relationship between the container C_{11} and its childs C_{21} and C_{22}.

4.2 Algorithm for building the Container–Set–List

In this section the algorithm for building the data structures is presented. In order to understant the algorithm we have to take into account that:

1. The algorithm is applicable to the elements belonging to an specified compartment of the ship. Thereby the objects and the groups mentioned here are referenced to all elements of that compartment.
2. The algorithm calculates for each LOD d_k a container–set associated to it. As mentioned above, those containers C_{kj} will contain a set of offset–dop representations:
$$d_k \longrightarrow \{C_{k_1}, ..., C_{k_m}\}$$
3. We will say that an element E belongs to a C_{kp} container iff its $Od(E, d_k)$ is represented in C_{kp} ($Od(E, d_k) \subset C_{kp}$). We note this as $E \in C_{kp}$ (where E can be either an object or a group).

Algorithm stages: The algorithm has two main stages.

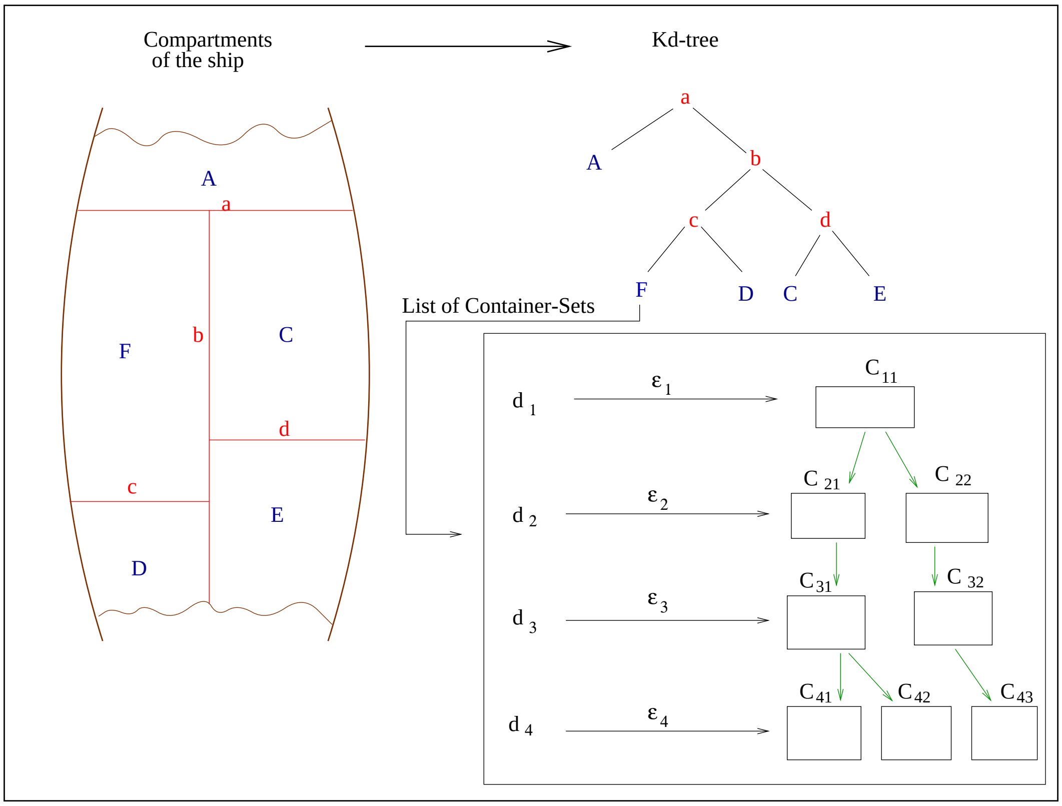

Figure 2: Data structure for collision detection

1. In the first one the C containers corresponding to the first level of detail, $d_{k_{min}}$, are generated.
2. In the second, the rest of C containers associated to the remandings levels of details, up to $d_{k_{max}}$, are computed and filled.

The algorithm to generate and fill a container–set is as follows:

1. First, a list of offset-dops of elements $List\_Od\_Elem\_d_k$ at a determined level of detail is generated. This list will contain the offset–dops of objects or the offset–dops of groups depending on their memory occupancy.
2. Second, the container–sets associated to the remanding levels of detail, are computed filling each one with the offset–dops of the $List\_Od\_Elem\_d_{k-1}$ list.

   While processing, the algorithm takes into account the size of memory page. That means that the information contained inside each C container must fit in a memory page. This will increase the speed of the collision detection algorithm.

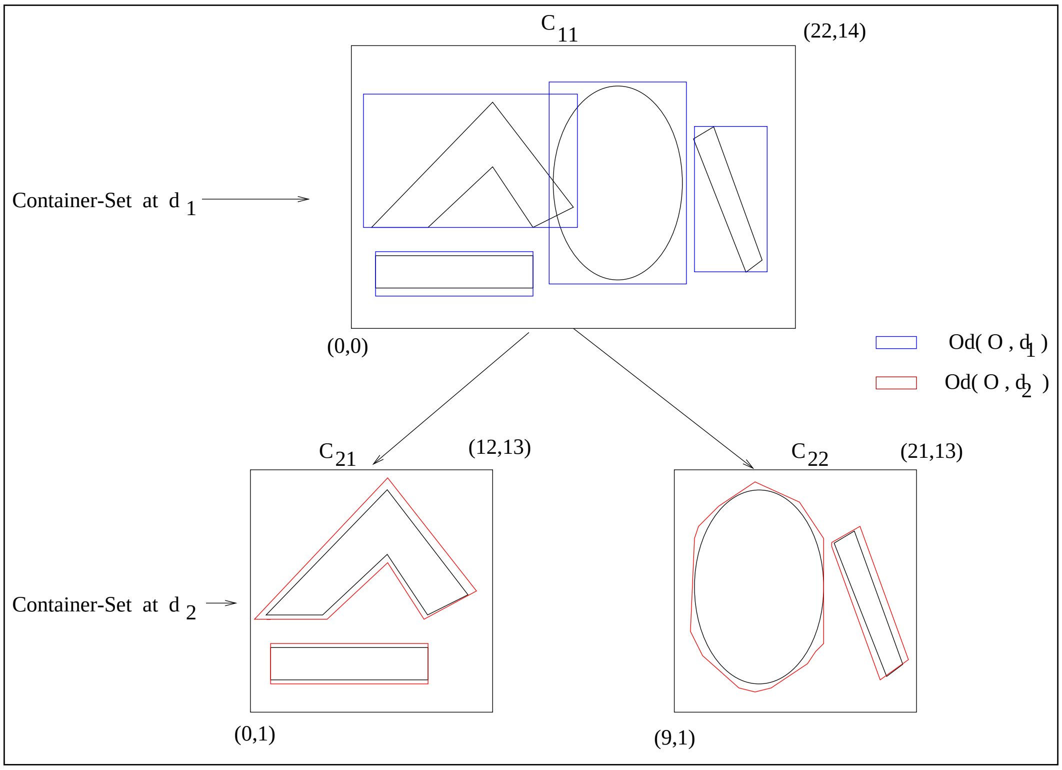

Figure 3: Example of container-sets at levels d_1 and d_2, in 2D

Algorithm Container–set–List–computation

algorithm $Build\_Containers\_set\_List$
  $\{First\ stage:\ generate\ and\ fill\ Containers\ at\ level\ d_{k_{min}}\}$
  $d_k := d_{k_{min}}$
  $InitializeList(List\_Od\_Elem\_d_k)$ $\{List\_Od\_Elem\_d_k:\ list\ of\ Ods\ of\ the\ elements\}$
  for each G_j **do** $\{post-order\ path\}$
    $ComputeOdsListElem(d_k, G_j, List\_Od\_Elem\_d_k)$
  endfor
  $\{Insert\ the\ offset-dops\ of\ elements\ at\ level\ d_k\ into\ C\ containers\}$
  $p := 0$
  $InsertOdIntoContainers(p, List\_Od\_Elem\_d_k, d_k, Container\_Set)$

  $\{Second\ stage:\ generate\ and\ fill\ the\ rest\ of\ the\ C\ containers\ associated\ to\ the\ remanding$
  $levels\ of\ detail: d_{k_{min+1}}\ to\ d_{k_{max}}\}$
  for d_k **in** $[d_{k_{min+1}}..d_{k_{max}}]$ **do**
    for each p **do** $\{For\ each\ Container\ at\ level\ d_{k-1}\}$
      $OpenMotherContainer(C_{d_{k-1},p})$
      $InitializeList(List\_Od\_Elem\_d_k)$

for each $Elem \in C_{d_{k-1},p}$ **do**
if
$IsGroup(Elem) \rightarrow ComputeOdsListElem(d_k, Elem, List\_Od\_Elem\_d_k)$
no $IsGroup(Elem) \rightarrow InsertToList(List\_Od\_Elem\_d_k, Od(Elem, d_k))$
endif
endfor
$\{Insert\ the\ offset - dops\ of\ elements\ at\ level\ d_k\ into\ C\ containers\}$
$InsertOdIntoContainers(p, List\_Od\_Elem\_d_k, d_k, Container\_Set)$
endfor
endfor
falgorithm

procedure $ComputeOdsListElem($**in** $d_k : LOD,$ **in** $G : Group,$ **out** $List\_Od\_Elem\_d_k : LOdE)$
$\{Generate\ the\ Od(E, d_k)\ of\ elements\ E_i \in G\ and\ insert\ them\ to\ a\ list\}$
$InitializeList(List\_Od\_G\_Elements\_d_k)$
$\{Compute\ the\ memory\ size\ of\ Od(E_i, d_k)\ of\ elements\ E_i \in G\}$
$MemSize\_Od\_E := 0$
for each $E_i \in G$ **do**
$InsertToList(List\_Od\_G\_Elements\_d_k, Od(E_i, d_k), d_k)$
$MemSize\_Od\_E := MemSize\_Od\_E + MemSize(Od(E_i, d_k))$
endfor
$\{Insert\ into\ list\ List\_Od\_Elem\_d_k\ the\ Ods\ of\ elements\ or\ the\ Od\ of\ the\ G$
$depending\ on\ memory\ occupancy\}$
if
$MemSize(Od(G, d_k)) \leq MemSize\_Od\_E \rightarrow InsertToList(List\_Od\_Elem\_d_k, Od(G_j, d_k))$
$MemSize(Od(G, d_k)) > MemSize\_Od\_E \rightarrow InsertIoList(List\_Od\_Elem\_d_k, List\_Od\_G\_Elements\_d_k)$
endif
endprocedure

procedure $InsertOdIntoContainers($**in** $p :$ **integer**, **in** $List\_Od : LOdE,$ **in** $d_k : LOD,$ **out** $Container\_Set : CS)$
var
$ll_1, ll_2 : LOdE$
$j :$ **integer**
endvar
$j := 1$
while $MemSize(List\_Od) > MemPag$ **do**
$DivideList(List\_Od, ll_1, ll_2)$ $\{Post:\ MemSize(ll_1) < MemPag\ \}$
$NewContainer(C_{d_k,j}, d_k, p)$
$Fill(C_{d_k,j}, ll_1)$
$List\_Od\_Elem\_d_k := ll_2$
$j := j + 1$
endwhile
$NewContainer(C_{d_k,j}, d_k, p)$
$Fill(C_{d_k,j}, List\_Od)$
endprocedure

Comments:

- The procedure *NewContainer* initializes $C_{d_k,j}$ with the attribute pointer to empty when $d_k = d_{k_{min}}$. Otherwise, the item pointer from parent to the new container is added.

- The procedure *DivideList* uses an heuristics to minimize the complexity of collision detection algorithms (sections 5 and 6) when obtaining the lists ll_1 and ll_2. Each group with objects inside ll_1 and ll_2 is subdivided in two groups.
- We assume, in this paper, a previous heuristics to group objects in a hierarchical and optimum way. Then, it makes possible to reduce:
  - The total memory occuppied by all the containers and,
  - The interference algorithms complexities

---

5 Algorithms for Collision Detection from Containers Collision Evaluation

In this section a collection of algorithms that perform collision detection is presented. The algorithms are based on containers–set collision evaluation at different levels of detail. In the next section another collection of algorithms is presented. In this second case, the algorithms perform the collision detection inside an specified container, and the collision evaluation is based on the offset–dops of the elements.

Three different algorithms for collision detection, depending on the inputs and the results that we are looking for, are classified in figure 4.

The algorithms have to be understand as follows:

- *Algorithm A*: will be used when it is only necessary to detect overlapping between different containers at a determined level d_k LOD, given us the spaces ($MaxMin$ coordinates) of overlapping containers and the objects that are located in them.
- *Algorithm B*: will be used when the space of colliding containers, the objects identifiers and the faces of the Offset–dops representations that belong to the interference space are needed.
- *Algorithm C*: will be used only when the exact colliding faces of the Offset–dops are looked for.

Description of the outputs of algorithms:

- *List_Colliding_Containers*: List of colliding spaces where a collision between containers has been detected at an specified LOD d_k. We call this spaces as *colliding–areas*. The *colliding–areas* are computed from containers $MaxMin$ coordinates comparation. Thus, each *colliding–area* in the list is represented by the next atributtes:

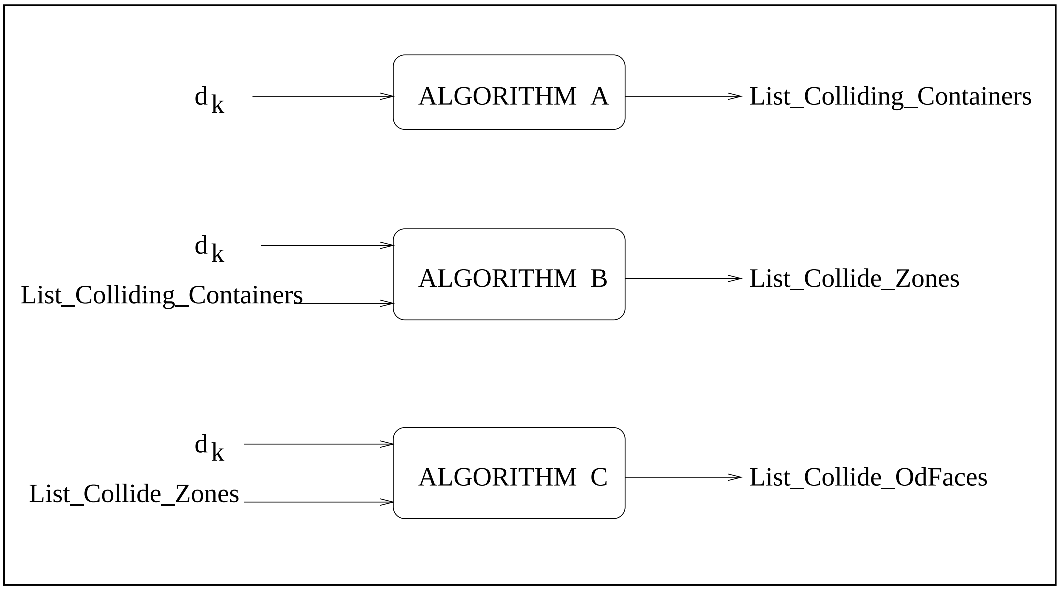

Figure 4: Collision Detection algorithms with their inputs and outputs

 - Pair of identifier containers that collide
 - Area: $MaxMin$ coordinates of overlapping space between containers

- *List_Collide_Zones*: Is a list of *zones* that collide at an specified level of detail d_k. Each *collide_zone* of the list is associated to one *colliding–area*. When a *collide_zone* is empty it means that there is no collision between elements in the *colliding–area*. The information of each zone is:
  - *colliding–area*: Pair of containers that collide and $MaxMin$ coordinates of overlapping space between them
  - Offset–dops: Pointers to Od that are inside the colliding–area
  - Pointers to elements that their Od faces belong to the colliding–area
  - Faces: Equations of faces of the Od that are located in the colliding–area.
- *List_Collide_OdFaces*: Equations of faces of the offset–dops that collide in the colliding–area.

5.1 Algorithm A

The follow algorithm computes a list of areas where containers overlap at a determined level of detail. In those areas it is possible that a collision between elements appears. The output list will contain a list of collision_areas with the atributtes mentioned above.

```
procedure ContainersCollision(in d_k : LOD, out List_Colliding_Containers : LCC)
  var
    i, p : integer
    area : MaxMin
    Collision_Area : CA
  endvar
  Initialize(List_Colliding_Containers) {Post : List_Colliding_Containers has its level attribute set to d_k}
  p := 1
  while p ≤ max do   {number of containers p := 1, .., max at d_k}
        i := p + 1
        while i ≤ max do
              MaxMinCoordinatesComparation(C_{d_k,p}, C_{d_k,i}, area)
              if
                no Empty(area) → Collision_Area.C_1 := C_{d_k,p}
                                 Collision_Area.C_2 := C_{d_k,i}
                                 Collision_Area.area := area
                                 Insert(List_Colliding_Containers, Collision_area)
                Empty(area) → {continue}
              endif
              i := i + 1
        endwhile
        p := p + 1
  endwhile
endprocedure
```

Figure 5 shows an example in 2D. If we call the procedure $ContainersCollision(d_2,$ $List\_Colliding\_Containers)$ in this situation, the $List\_Colliding\_Containers$ will contain the LOD (d_2) and only one component with:

- $d_k = d_2$
- Collision_Area:
  - Containers: C_{21}, C_{22}
  - Area: $\{(9, 1), (12, 13)\}$

5.2 Algorithm B

This algorithm computes a list of zones that collide, $List\_Collide\_Zones$, corresponding to collision–areas of the space where a collision has been detected (using the procedure $ContainersCollision$ of the algorithm A). As input the $List\_Colliding\_Containers$ contains pairs of containers that collide and the areas of collision.

```
procedure ZonesCollision(in d_k : LOD, in List_Colliding_Containers : LCC, out List_Collide_Zones : LCZ)
  var
    Collision_Area : CA
    Collide_Zone : CZ
    Ods_In_Area : OIA
  endvar
  InitializeList(List_Collide_Zones) {Post : List_Collide_Zones has its level attribute set to d_k}
  GetFromList(List_Colliding_Containers, Collision_Area)
  while no LastMember(Collision_Area) do
        SearchOdsInArea(d_k, Collision_Area, Ods_In_Area)
        if
          no Empty(Ods_In_Area) →
                                   Collide_Zone.CollisionArea := Collision_Area
                                   Collide_Zone.Ods := Ods_In_Area.Ods
                                   Collide_Zone.Elems := Ods_In_Area.Elems
                                   Collide_Zone.Faces := Ods_In_Area.Faces
                                   Insert(List_Collide_Zones, Collide_Zone)
          Empty(Ods_In_Area) → {continue}
        endif
        GetFromList(List_Colliding_Containers, Collision_Area)
  endwhile
endprocedure

procedure SearchOdsInArea(in d_k : LOD, in Collision_Area : CA, out Ods_In_Area : OIA)
  Initialize(Ods_In_Area)
  for each Elem ∈ Collision_Area.C_1 do
      Od_In_Collision_Area?(d_k, Elem, Collision_Area.area, Ods_In_Area)
  endfor
  for each Elem ∈ Collision_Area.C_2 do
      Od_In_Collision_Area?(d_k, Elem, Collision_Area.area, Ods_In_Area)
  endfor
endprocedure

procedure Od_In_Collision_Area?(in d_k : LOD, in E : ELE, in area : MaxMin, out Ods_In_Area : OIA)
  if
    Interfer(Od(E, d_k), area) → ComputeFacesInArea(Od(E, d_k), area, Faces)
                                 Ods_In_Area.Ods := Ods_In_Area.Ods + Od(E, d_k)
                                 Ods_In_Area.Elems := Ods_In_Area.Elems + E
                                 Ods_In_Area.Faces := Ods_In_Area.Faces + Faces
    no Interfer(Od(E, d_k), area) → {continue}
  endif
endprocedure
```

If we call at $ZonesCollision(d_2, List\_Colliding\_Containers, List\_Collide\_Zones)$ in the situation presented in figure 5, the outputs of the procedures will be:

Input:
$List\_Colliding\_Containers$ contains:

- $d_k = d_2$
- Collision_Area:
  - Containers: C_{21}, C_{22}
  - Area: $\{(9,1),(12,13)\}$

Output:

- $SearchOdsInArea(d_2, Collision\_Area, Ods\_In\_Area)$
  Ods_In_Area will contain:
  - $Ods\_In\_Area.Ods : Od(O_1, d_2), Od(O_2, d_2)$
  - $Ods\_In\_Area.Elems : O_1, O_2$
  - $Ods\_In\_Area.Faces : f_1, f_2, k_1, k_2, k_3, k_4$
- $Insert(List\_Collide\_Zones, Collide\_Zone)$:
  The $List\_Collide\_Zones$ has only one $Collide\_Zone$ and the level of detail:
  - $d_k = d_2$
  - $Collide\_Zone$:
    - $Collide\_Zone.CollisionArea$:
      - Containers: C_{21}, C_{22}
      - Area: $\{(9,1),(12,13)\}$
    - $Collide\_Zone.Ods : Od(O_1, d_2), Od(O_2, d_2)$
    - $Collide\_Zone.Elems : O_1, O_2$
    - $Collide\_Zone.Faces : f_1, f_2, k_1, k_2, k_3, k_4$

5.3 Algorithm C

This algorithm computes the faces of the offset–dops of elements that collide in a d_k and $Collide\_Zone$ determined. This procedure would be used by $ZonesCollision$ when the information of the faces that really collide would be needed.

```
procedure OdFacesCollision(in d_k : LOD, in Collide_Zone : CZ, out List_Od_Faces : LOdF)
  Initialize(List_Od_Faces)     {Post : List_Od_Faces has its level attribute set to d_k}
  CheckFaces(Collide_Zone.Faces, List_Faces)
  if
    no Empty(List_Faces) → Insert(List_Od_Faces, Collide_Zone.Elems, List_Faces)
    Empty(List_Faces) → {continue}
  endif
endprocedure
```

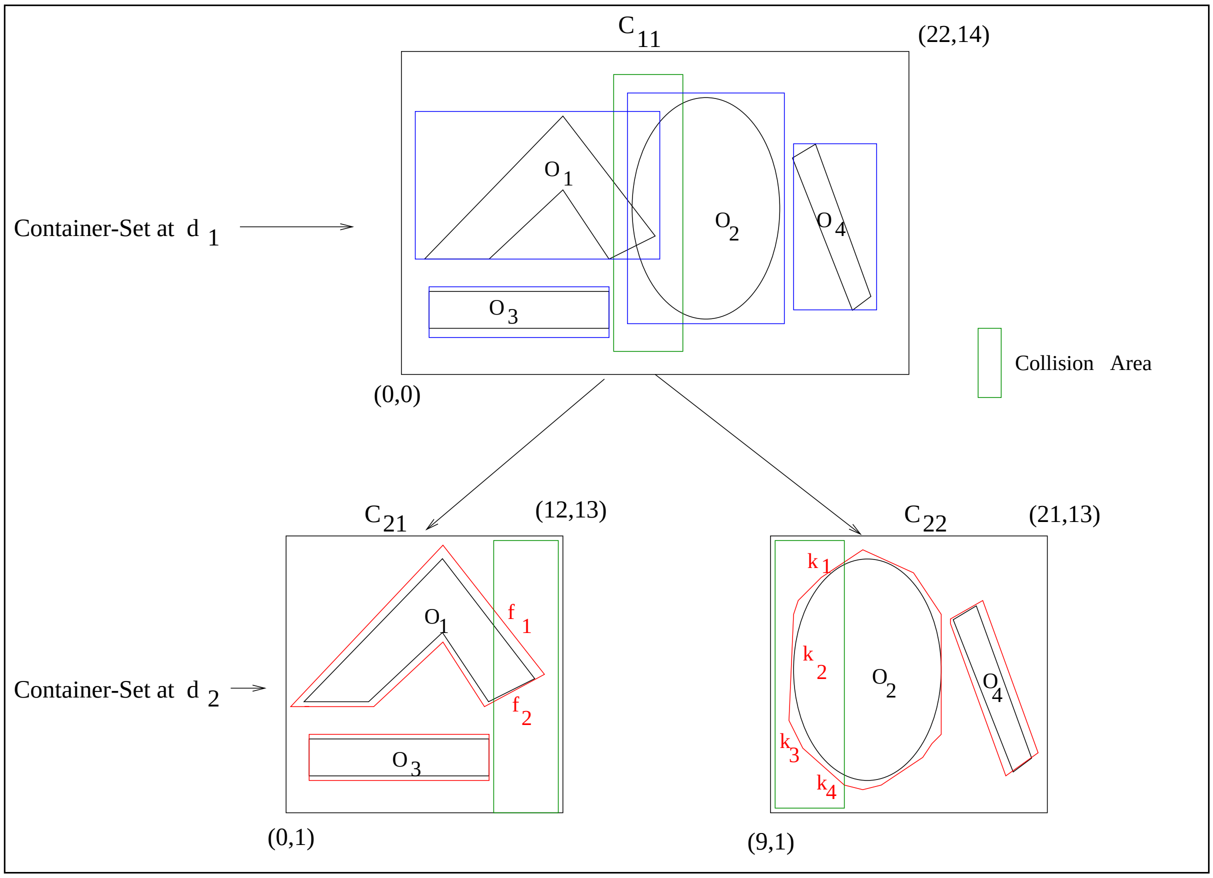

Figure 5: Example of collision in 2D

In the case of the figure 5, after calling $OdFacesCollision(d_2, Collide\_Zone, List\_Od\_Faces)$ we will obtain:

Input:
$Collide\_Zone$ with values:

- $d_k = d_2$
- $Collide\_Zone$:
  - $Collide\_Zone.CollisionArea$:
    - Containers: C_{21}, C_{22}
    - Area: $\{(9,1),(12,13)\}$
  - $Collide\_Zone.Ods$: $Od(O_1, d_2), Od(O_2, d_2)$

- $Collide\_Zone.Elems : O_1, O_2$
- $Collide\_Zone.Faces : f_1, f_2, k_1, k_2, k_3, k_4$

Ouput:
The $List\_Od\_Faces$ will contain:

- $d_k = d_2$
- Pointers to elements: O_1, O_2
- Od–Faces: Faces that collide: f_1, f_2, k_2

5.4 Application of A, B and C algorithms to the case presented in figure 6

Looking at the situation presented in figure 6, we can say that the collision between $Od(O_2, d_1)$ and $Od(O_4, d_1)$ will not be detected using the algorithms A, B or C. In fact, this collection of algorithms perform the collision detection evaluation from container boxes intersection, and the objects O_2 and O_4 belong to the same container box. This kind of collision problem, when objects belong to the same container, is solved by the algorithms exposed in the next section.

Using the procedures of the algorithms A, B, an C in the figure 6 we obtain:

Algorithm A: $ContainersCollision(d_2, List\_Colliding\_Containers)$

$List\_Colliding\_Containers$ will contain:

- $d_k = d_2$
- Collision_Area:
  - Containers: C_{21}, C_{22}
  - Area: $\{(9,1),(12,14)\}$

Algorithm B: $ZonesCollision(d_2, List\_Colliding\_containers, List\_Collide\_Zones)$

$List\_Collide\_Zones$ will contain

- $d_k = d_2$
- $Collide\_Zone$:
  - $Collide\_Zone.CollisionArea$:
    - Containers: C_{21}, C_{22}
    - Area: $\{(9,1),(12,14)\}$
  - $Collide\_Zone.Ods : Od(O_1, d_2), Od(O_2, d_2), Od(O_5, d_2)$

- $Collide\_Zone.Elems : O_1, O_2, O_5$
- $Collide\_Zone.Faces : f_1, f_2, k_1, k_2, k_3, k_4, a_1, a_2, a_3, a_4, a_5, a_6$

Algorithm C: $OdFacesCollision(d_2, Collide\_Zone, List\_Od\_Faces)$

Input:

- $d_k = d_2$
- $Collide\_Zone$:
  - $Collide\_Zone.CollisionArea$:
    - Containers: C_{21}, C_{22}
    - Area: $\{(9,1), (12,14)\}$
  - $Collide\_Zone.Ods : Od(O_1, d_2), Od(O_2, d_2), Od(O_5, d_2)$
  - $Collide\_Zone.Elems : O_1, O_2, O_5$
  - $Collide\_Zone.Faces : f_1, f_2, k_1, k_2, k_3, k_4, a_1, a_2, a_3, a_4, a_5, a_6$

Output: the $List\_Od\_Faces$ will contain

- $d_k = d_2$
- Pointers to elements: O_1, O_2
- Od–Faces: Faces that collide: f_1, f_2, k_2

Note that the faces of $Od(O_5, d_2) : \{a_1, a_2, a_3, a_4, a_5, a_6\}$ do not appear in the list $List\_Od\_Faces$ of algorithm C. The reason is because, even though the object O_5 belongs to the $Collision\_Area$ and to the $Collide\_Zone$, it does not collide with any other object.

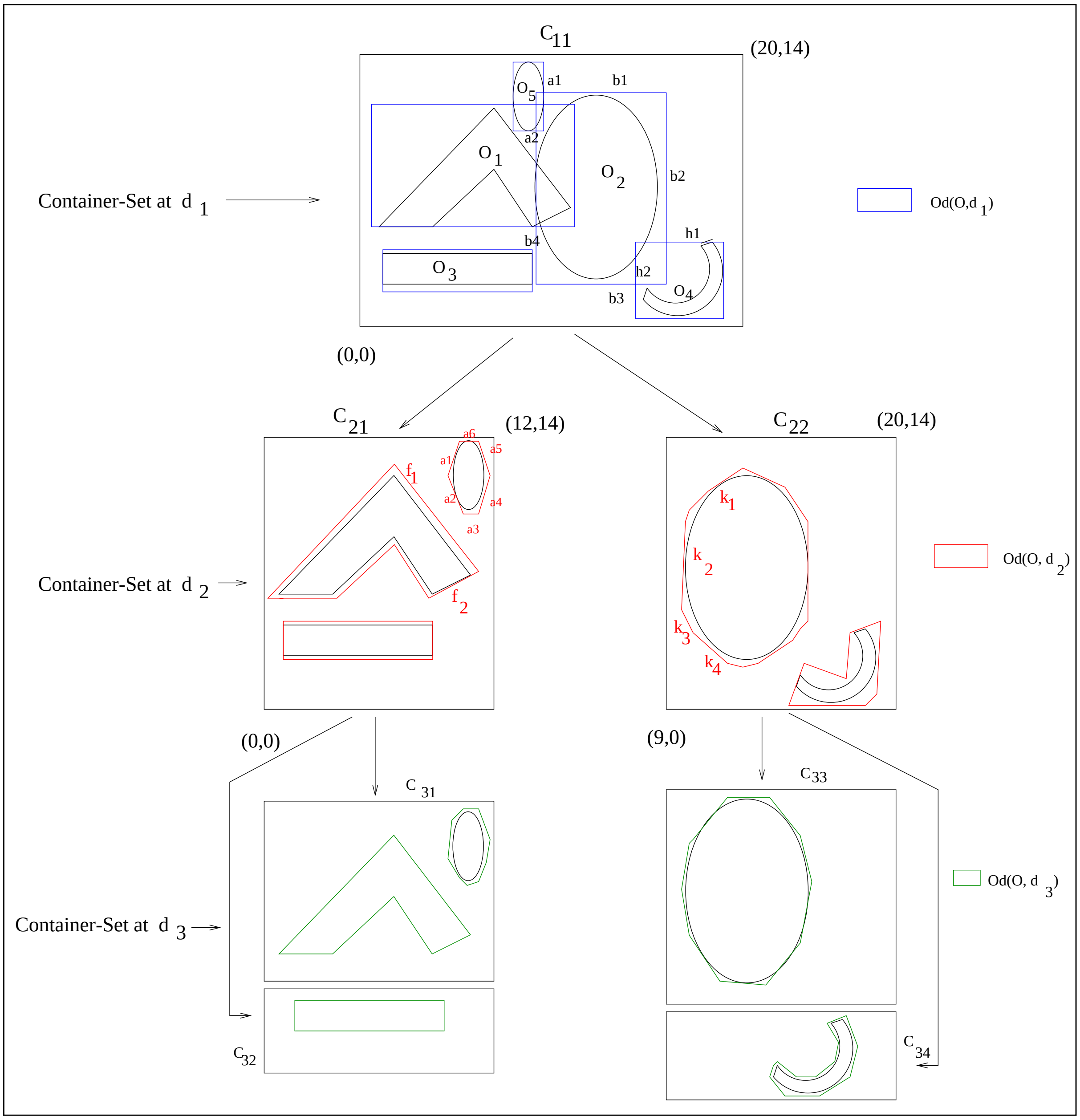

Figure 6: Example of collision in 2D and 3 levels

6 Algorithms for Collision Detection from Offset–Dops

In this section a collection of algorithms for collision detection based on the offset–dops of the elements associated to a determined container (at a d_k specified) is presented. The algorithms perform the collision detection inside the containers. Thus the collision is evaluated beginning from the offset–dops of the elements.

In figure 7 a classification of algorithms for collision detection depending on the inputs and the results are showed.

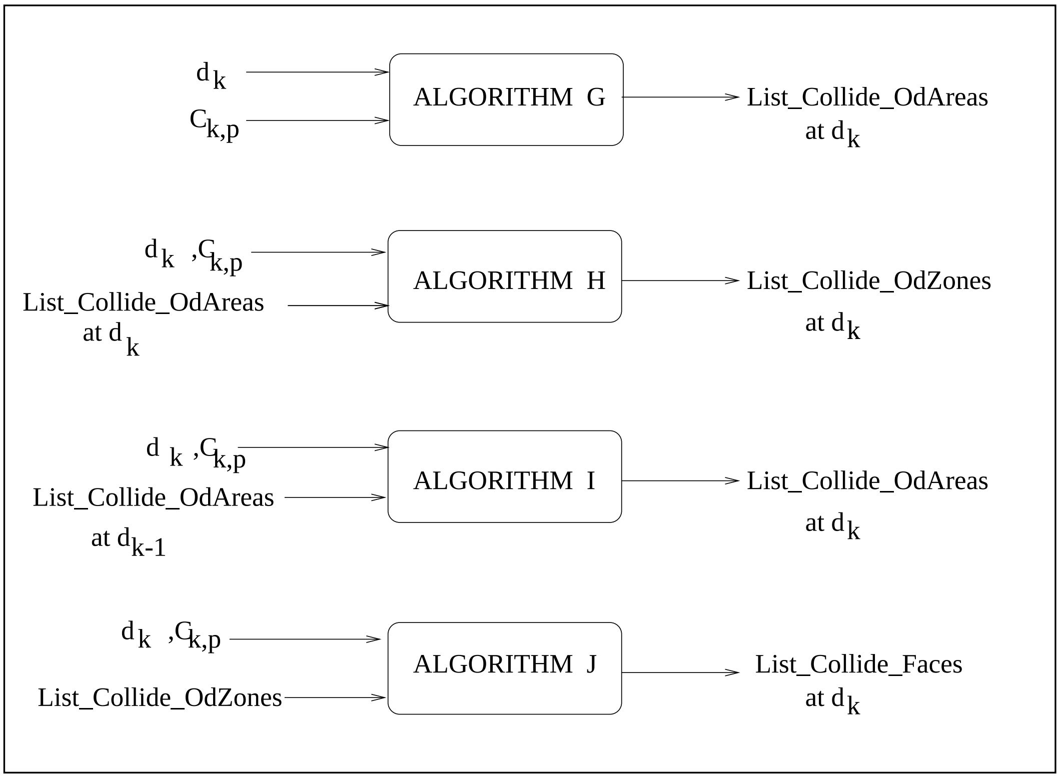

Figure 7: Collision Detection algorithms with the inputs and outputs from Offset–dops evaluation

Description of the outputs of the algorithms:

- *List_Collide_OdAreas*: List of subspaces where a collision is detected. Each OdArea is computed from the offset–dops representacions of elements at an specified LOD d_k. Each area in the list contains the next information:
  - LOD: d_k

  - Container Identifier: $C_{k,p}$
  - OdArea: $MaxMin$ coordinates
  - Pointers to elements that their offset–dops collide

- *List_Collide_OdZones*: Is a list of colliding zones. This list is associated to one level of detail d_k. When the *List_Collide_OdZone* is empty it means that there is no collision between the offset–dops of elements. The information of each zone is:
  - LOD: d_k
  - Container Identifier: $C_{k,p}$
  - Collision OdAreas: $MaxMin$ coordinates of the collision spaces and pointers to elements that their offset–dops collide
  - Offset–dops: Pointers to Od that collide
  - Faces: Equations of faces of the offset–dops that are in the collision_OdArea.

- *List_Collide_Faces*: Is a list of the faces that belongs to the candidate faces of the $List\_Collide\_OdZones$ and really collide.

6.1 Algorithm G

This algorithm computes the *List of collide areas* at an specified level d_k in a determined container $C_{k,p}$. The areas of collision are calculated from the $MinBox^*$ of the Od of the objects belonging to $C_{k,p}$. Thereby, each component of the output list $List\_OdAreas$ will contain the $MaxMin$ coordinates of the overlapping space between the $MinBox^*$ of the Od that collide and the pointers to the Od involved in the collision.

procedure $OdAreasCollision(\textbf{in}\ d_k : LOD, \textbf{in}\ C_{k,p} : Container, \textbf{out}\ List\_OdAreas : LOdA)$
  var
    $areaOD : MaxMin$
    $OdArea : OdA$
    $collide$: **boolean**
  endvar
  $Initialize(List\_OdAreas)$ $\{Post : d_k, C_{k,p} \in\ List\_OdAreas\}$
  $GetFirstElemFromContainer(E_1, C_{k,p})$
  while $E_1 \neq LastElem(C_{k,p})$ **do**
      $GetElemFromContainer(E_2, C_{k,p})$
      while $E_2 \neq LastElem(C_{k,p})$ **do**
          $Inilialize(areaOD)$
          $Check(MinBox^*(Od(E_1, d_k)), MinBox^*(Od(E_2, d_k)), collide)$
          if
            $collide \rightarrow$

$ComputeIntersectionArea(MinBox^*(Od(E_1,d_k)), MinBox^*(Od(E_2,d_k)), areaOD)$
$OdArea.area := areaOD$
$OdArea.E_1 := E_1$
$OdArea.E_2 := E_2$
$Insert(List\_OdAreas, OdArea)$
no *collide* $\rightarrow \{continue\}$
endif
$GetElemFromContainer(E_2, C_{k,p})$
endwhile
$GetNextElemFromContainer(E_1, C_{k,p})$
endwhile
endprocedure

In figure 8 a situation with several cases of collision in 2D is shown. In figures 9, 10, 11, the container–sets at differents levels of LOD are presented. For clearness they are only printed some coordinates that will serve to reader as reference.

Looking at the example of figure 8 (see also figures: 9, 10, 11) after calling $OdAreasCollision(d_k, C_{k,p}, List\_OdAreas)$, for different values of d_k and $C_{k,p}$, the $List\_OdAreas$ will contain:

- $d_k = d_1, \quad C_{k,p} = C_{1,1}$: $List\_OdAreas$
  - $d_k = d_1$
  - $C_{k,p} = C_{1,1}$
  - OdAreas:
    - $OdArea_1$:
      - area: $\{(5,13)(7,17)\}$
      - $E_1 : O_1$
      - $E_2 : O_2$
    - $OdArea_2$:
      - area: $\{(9,2)(10,6)\}$
      - $E_1 : O_5$
      - $E_2 : O_8$
    - $OdArea_3$:
      - area: $\{(19,5)(21,10)\}$
      - $E_1 : O_6$
      - $E_2 : O_7$
- $d_k = d_2, \quad C_{k,p} = C_{2,1}$: $List\_OdAreas$
  - $d_k = d_2$
  - $C_{k,p} = C_{2,1}$

- OdAreas:
  - $OdArea_1$:
    - area: $\{(8,2)(11,5)\}$
    - $E_1 : O_5$
    - $E_2 : O_8$

- $d_k = d_2, \quad C_{k,p} = C_{2,2}$: $List\_OdAreas$
  - $d_k = d_2$
  - $C_{k,p} = C_{2,2}$
  - OdAreas:
    - $OdArea_1$:
      - area: $\{(18,7)(20,10)\}$
      - $E_1 : O_6$
      - $E_2 : O_7$

- $d_k = d_3, \quad C_{k,p} = C_{3,1}$: $List\_OdAreas$
  - $d_k = d_3$
  - $C_{k,p} = C_{3,1}$
  - OdAreas: Empty

- $d_k = d_3, \quad C_{k,p} = C_{3,2}$: $List\_OdAreas$
  - $d_k = d_3$
  - $C_{k,p} = C_{3,1}$
  - OdAreas:
    - $OdArea_1$:
      - area: $\{(8,2)(10,5)\}$
      - $E_1 : O_5$
      - $E_2 : O_8$

- $d_k = d_3, \quad C_{k,p} = C_{3,3}$: $List\_OdAreas$
  - $d_k = d_3$
  - $C_{k,p} = C_{3,1}$
  - OdAreas: Empty

- $d_k = d_3, \quad C_{k,p} = C_{3,4}$: $List\_OdAreas$

- $d_k = d_3$
- $C_{k,p} = C_{3,1}$
- OdAreas: Empty

6.2 Algorithm H

This algorithm computes the interference faces of the Offset–dops that belong to the space of collision and the colliding areas. Those areas have been previously computed and are in the $List\_OdAreas$.

procedure $OdZonesCollision($**in** $d_k : LOD,$ **in** $C_{k,p} : Container,$ **in** $List\_OdAreas : LOdA,$
  out $List\_Collide\_OdZones : LCOZ)$
  var
    $OdZone : OdZ$
    $OdArea : OdA$
    $OdFaces : SetOfFaces$
  endvar
  $InitializeList(List\_Collide\_OdZones)$ $\{Post : d_k, C_{k,p} \in\ List\_Collide\_OdZones\}$
  for each $OdArea \in List\_OdAreas$ **do**
    $OdZone.OdArea := OdArea$
    $OdZone.Ods := \{Od(OdArea.E_1, d_k), Od(OdArea.E_2, d_k)\}$
    $ComputeCanditateFaces(OdArea, OdFaces)$
    $OdZone.Odfaces := Odfaces$
  endfor
endprocedure

procedure $ComputeCandidateFaces($**in** $OdArea : OdA,$ **out** $OdFaces : SetOfFAces)$
  for each $Face \in Od(OdArea.E_1, d_k)$ **do**
    if
      $Interfer(Face, OdArea.area) \rightarrow Insert(OdFaces, Face)$
      no $Interfer(Face, OdArea.area) \rightarrow \{continue\}$
    endif
  endfor
  for each $Face \in Od(OdArea.E_2, d_k)$ **do**
    if
      $Interfer(Face, OdArea.area) \rightarrow Insert(OdFaces, Face)$
      no $Interfer(Face, OdArea.area) \rightarrow \{continue\}$
    endif
  endfor
endprocedure

Looking at the example of the figure 8 (see also figures: 9, 10, 11) after calling $OdZonesCollision(d_k, C_{k,p}, List\_Collide\_OdZones)$, for different values of d_k and $C_{k,p}$ the $List\_Collide\_OdZones$ will contain:

See figure 9 to check outputs

$d_k = d_1, \quad C_{k,p} = C_{1,1} : List\_Collide\_OdZones$

- $d_k = d_1$
- $C_{k,p} = C_{1,1}$
- OdZones:
  - $OdZone_1$:
    - OdArea:$\{(5, 13)(7, 17), O_1, O_2\}$
    - Ods: $Od(O_1, d_1), Od(O_2, d_1)$
    - OdFaces: $\{f1, f2, f3, k1\}$
  - $OdZone_2$:
    - OdArea: $\{(9, 2)(10, 6), O_5, O_8\}$
    - Ods: $Od(O_5, d_1), Od(O_8, d_1)$
    - OdFaces:$\{s1, w1, w2, w3\}$
  - $OdZone_3$:
    - OdArea: $\{(19, 5)(21, 10), O_6, O_7\}$
    - Ods:$Od(O_6, d_1), Od(O_7, d_1)$
    - OdFaces: $\{h1, h2, t1, t2\}$

See figure 10 to check outputs

$d_k = d_2, \quad C_{k,p} = C_{2,1} : List\_Collide\_OdZones$

- $d_k = d_2$
- $C_{k,p} = C_{2,1}$
- OdZones:
  - $OdZone_1$:
    - OdArea: $\{(8, 2)(11, 5), O_5, O_8\}$
    - Ods: $Od(O_5, d_2), Od(O_8, d_2)$
    - OdFaces: $\{w1, w2, s1, s2\}$

$d_k = d_2, \quad C_{k,p} = C_{2,2} : List\_Collide\_OdZones$

- $d_k = d_2$
- $C_{k,p} = C_{2,2}$
- OdZones:

- $OdZone_1$:
  - OdArea: $\{(18,7)(20,10), O_6, O_7\}$
  - Ods: $Od(O_6, d_2), Od(O_7, d_2)$
  - OdFaces: $\{h1, h2, t1\}$

See figure 11 to check outputs

$d_k = d_3, \quad C_{k,p} = C_{3,1} : List\_Collide\_OdZones$

- $d_k = d_3$
- $C_{k,p} = C_{3,1}$
- OdZones: Empty

$d_k = d_3, \quad C_{k,p} = C_{3,2} : List\_Collide\_OdZones$

- $d_k = d_3$
- $C_{k,p} = C_{3,2}$
- OdZones:
  - $OdZone_1$:
    - OdArea:$\{(8,2)(10,5), O_5, O_8\}$
    - Ods: $Od(O_5, d_3), Od(O_8, d_3)$
    - OdFaces: $\{w1, w2, s1, s2\}$

$d_k = d_3, \quad C_{k,p} = C_{3,3} : List\_Collide\_OdZones$

- $d_k = d_3$
- $C_{k,p} = C_{3,3}$
- OdZones: Empty

$d_k = d_3, \quad C_{k,p} = C_{3,4} : List\_Collide\_OdZones$

- $d_k = d_3$
- $C_{k,p} = C_{3,4}$
- OdZones: Empty

6.3 Algorithm I

This algorithm computes the $OdAreas$ where a collision can be occurred from the inputs: $d_k, C_{k,p}$ and the $List\_OdAreas$ at level d_{k-1}. Knowing the $Collision\_OdAreas$ at level d_{k-1} it gives speed to select the areas of collision at level d_k, due to that we have only to evaluate for collision in those areas. In fact, the information that is useful to algorithm is the set of elements that their $Od(O, d_{k-1})$ collide. The input $List\_OdAreas$ corresponds to d_{k-1} and $C_{k-1,p}$, where $C_{k-1,p}$ is the Mother Container of $C_{k,p}$. Thus, we have to take into account that the mother $C_{k-1,p}$ could have been divided into several containers (at the next level: d_k) and not all the OdAreas have to belong to the actual container. It is solved by the procedure $GetOdAreaFromList$.

procedure $OdAreasCollision\_I($**in** $d_k : LOD,$ **in** $C_{k,p} : Container,$ **in** $List\_OdAreas\_d_{k-1} : LOdA,$
  out $List\_OdAreas : LOdA)$
  var
    $areaOD :$ **MaxMin**
  endvar
  $Initialize(List\_OdAreas)$ $\{Post : d_k, C_{k,p} \in\ List\_OdAreas\}$
  $GetOdAreaFromList(List\_OdAreas\_d_{k-1}, C_{k,p}, OdArea)$
  while **no** $LastOdArea(OdArea)$ **do**
    $Inilialize(areaOD)$
    $ComputeIntersectionArea(MinBox^*(Od(OdArea.E_1, d_k)), MinBox^*(Od(OdArea.E_2, d_k)), areaOD)$
    if
      no $Empty(areaOD) \rightarrow OdArea.area := areaOD$
        $Insert(List\_OdAreas, OdArea)$
      $Empty(areaOD) \rightarrow \{continue\}$
    endif
    $GetOdAreaFromList(List\_OdAreas\_d_{k-1}, OdArea)$
  endwhile
endprocedure

procedure $GetOdAreaFromList($**in** $List\_OdAreas : LOdA,$ **in** $C_{k,p} : Container,$ **out** $OdArea : OdA)$
  $GetOdArea(List\_OdAreas, OdArea)$
  while no $OdArea.area \subset\ C_{k,p}$**and no** $LastArea(OdArea)$ **do**
    $GetOdArea(List\_OdAreas, OdArea)$
  endwhile
endprocedure

Application of the procedure $OdAreasCollision\_I$ to different cases shown in figure 8 (see also figures: 9, 10, 11):

See figures 10 and 11

$$OdAreasCollision\_I(d_3, C_{3,2}, List\_OdAreas\_d_2, List\_OdAreas)$$

With $List\_OdAreas\_d_2$ (corresponding to the MotherContainer: $C_{2,1}$):

- $d_k = d_2$
- $C_{k,p} = C_{2,1}$
- $OdAreas$:
  - $OdArea_1$:
    - $area : \{(8,2),(11,5)\}$
    - $E_1 : O_5$
    - $E_2 : O_8$

With the above inputs we will get as outputs: $List\_OdAreas$

- $d_k = d_3$
- $C_{k,p} = C_{3,2}$
- $OdAreas$:
  - $OdArea_1$:
    - $area : \{(8,2),(10,5)\}$
    - $E_1 : O_5$
    - $E_2 : O_8$

See figures 10 and 11

$OdAreasCollision\_I(d_3, C_{3,3}, List\_OdAreas\_d_2, List\_OdAreas)$

With $List\_OdAreas\_d_2$ (corresponding to the MotherContainer: $C_{2,2}$):

- $d_k = d_2$
- $C_{k,p} = C_{2,2}$
- $OdAreas$:
  - $OdArea_1$:
    - $area : \{(18,7),(20,10)\}$
    - $E_1 : O_6$
    - $E_2 : O_7$

With the above inputs we will get as outputs: $List\_OdAreas$:

- $d_k = d_3$
- $C_{k,p} = C_{3,3}$
- $OdAreas$: Empty

Note that the result of the collision test is empty in this second call to the procedure ($OdAreasCollision\_I(d_3, C_{3,3}, List\_OdAreas\_d_2, List\_OdAreas)$). This means that at the d_2 LOD it was detected a collision between Ods that was a false collision.

6.4 Algorithm J

This algorithm computes the faces that really collide at an specified d_k and in a determined $C_{k,p}$. The inputs to algorithm include the $List\_Collide\_OdZones$. In other words, from a list of candidate faces that has been computed by algorithm H, the actual algorithm computes the faces that really collide.

```
procedure FacesCollision(in d_k : LOD, in C_{k,p} : Container, in List_Collide_OdZones : LCOZ,
            out List_Collide_Faces : LCF)
  var
    Faces_Collide : FC
  endvar
  InitializeList(List_Collide_Faces)  {Post : d_k, C_{k,p} ∈  List_Collide_Faces}
  for  each OdZone ∈ List_Collide_OdZones  do
      ComputeInterferenceFaces(OdZone.E_1.OdFaces, OdZone.E_2.OdFaces, Faces_Collide)
      if
        no Empty(Faces_Collide) → Insert(List_Collide_Faces, Faces_Collide)
        Empty(Faces_Collide) → {continue}
      endif
  endfor
endprocedure

procedure ComputeInterferenceFaces(in F_1 : SetOfFaces, in F_2 : SetOfFaces,
          out Faces_Collide : Set_Of_Colliding_Faces)
  for  each Face_i ∈ F_1  do
      for  each Face_j ∈ F_2  do
          if
            Interfer(Face_i, Face_j) → Insert(Face_i, Face_j, Faces_Collide)
            no Interfer(Face_i, Face_j) → {continue}
          endif
      endfor
  endfor
endprocedure
```

Application of the procedure $FacesCollision$ to different cases shown in figure 8 (see also figures: 9, 10, 11):

See figures 10 and 11

$$FacesCollision(d_3, C_{3,2}, List\_Collide\_OdZones, List\_Collide\_Faces)$$

With $List\_Collide\_OdZones$

- $d_k = d_3$
- $C_{k,p} = C_{3,2}$
- $OdZones$:

- $OdZone_1$:
  - $OdArea : \{(8,2),(10,5)\}$
  - $E_1 : O_5$
  - $E_2 : O_8$
  - $OdFaces : w1, w2, s1, s2$

With the above inputs we will get as outputs: $List\_Collide\_Faces$:

- $d_k = d_3$
- $C_{k,p} = C_{3,2}$
- OdZones:
  - Faces: $\{w1, w2, s1, s2\}$

7 Analysis of Algorithms

7.1 Analysis of memory swaping

Algorithms for Collision Detection From Containers Collision Evaluation

- *Algorithm A*: To compute the $CollisionAreas$ it is only needed the $MaxMin$ coordinates comparation between the container boxes. Keeping this information indepently of the container boxes it isn´t necessary to swap boxes to memory. Then, we have,

  $$MemorySwap(Algorithm\_A) = 0$$

- *Algorithm B*: In this case it is needed to intersect 2 containers in memory each time that $SearchOdsInArea$ procedure is called. The number of callings to this procedure is depending on the number of $Collision\_Areas$, N_{ca_k} at d_k, then we have:

  $$MemorySwap(Algorithm\_B) = N_{ca_k} * 2$$

  In fact the number of swaping container boxes in memory is depending on the order of elements in the $List\_Colliding\_Containers$, generated by $ContainersCollision$ procedure. Thus, the $ContainersCollision$ procedure has to use an heuristics to generate the list minimizing the number of swaps in memory (at this moment, the procedure computes the collision evaluation between containers keeping one container in memory and comparing it with the others, then the first container rests in memory as time as it is needed).

- *Algorithm C*: In this case it must swap 2 containers, then
$$MemorySwap(Algorithm\_C) = 2$$

Algorithms for Collision Detection From Offset–dops Collision Evaluation

- *Algorithm G*: This algorithm computes the collision evaluation inside a container box, then only one container has to be in memory. The main procedure only has to load one container at the moment that it is called. Thus,
$$MemorySwap(Algorithm\_G) = 1$$
- *Algorithm H*: For the same reason that the algorithm G,
$$MemorySwap(Algorithm\_H) = 1$$
- *Algorithm I*: Idem reasoning that algorithms G and H,
$$MemorySwap(Algorithm\_I) = 1$$
- *Algorithm J*: For the same reasons than the algorithms G, H, and I:
$$MemorySwap(Algorithm\_J) = 1$$

In table 1 there are presented the memory swap of the algorithms. The algorithms in the table have been grouped in two classes.

7.2 Complexity analysis

The names and description of the total parameters that are mentioned in this section are summarized in table 2

Algorithms for Collision Detection From Containers Collision Evaluation

- *Algorithm A*: This algorithm computes a list of areas where containers overlap at a determined level of detail d_k. To compute these areas the intersection evaluation between all the containers have to be performed. Due to that the containers are parallelepipeds the intersection between two of them is of constant order $O(1)$. However, this operation has to be done between all of them, N_{c_k}. The resultant cost of the algorithm is of order:
$$C_A = O(N_{c_k}^2)$$
- *Algorithm B*: This algorithm computes a list of zones that collide, corresponding to colliding–areas where a collision has been detected. The *Offset–dops* that belong to the colliging–areas, previously found, are sought. The cost of the algorithm can be calculated as follows:

Memory Swap	Class of algorithms	
	From Containers Collision	From Offset–dops Collision
Algorithm A	0	
Algorithm B	$N_{ca_k} * 2$	
Algorithm C	2	
Algorithms: G, H, I,J		1

Table 1: Memory Swaping of algorithms. Where N_{ca_k} is the number of colliding–areas

Parameter	Description
N_{c_k}	Maximum number of containers at d_k
N_{fO_k}	Average number of faces of an Od at d_k
n_{v_k}	Average number of vertices of an $Od(E, d_k)$
N_{ca_k}	Maximum number of $colliding\_areas$ at d_k
N_{o_k}	Maximum number of Od_k of two colliding containers
N_{ec_k}	Number of $E \in C_{k,p}$
N_{z_k}	Number of $OdZones$ at d_k and for one $C_{k,p}$
N_{cc_k}	Number of convex components

Table 2: Parameter description

– The evaluation cost, C_{oa}, to know if an specified $Od(E, d_k)$ with n_{v_k} vertices belongs to a colliding–area is of order:

$$C_{oa} = O(n_{v_k})$$

– This cost applied to all the N_{ca_k} colliding–areas and for all of Offset–dops, N_{o_k}, belonging to the colliding containers:

$$C_B = O(N_{ca_k} * N_{o_k} * n_{v_k})$$

- *Algorithm C*: This algorithm computes the faces of the Offset–dops of elements that collide in an specified d_k and a determined colliding–zone. Thus, the cost of computing the intersection between the faces of the Offset–dops belonging to the colliding–zone is of order:

– *Convex Od*: When the Offset–dops are convex (for lower values of d_k), the computation yield an $O(n_{v_k} * log\ n_{v_k})$ algorithm [PS85] [BKOS00]. Where n_{v_k} is the number of vertices of the Od at d_k.
– *Non–convex Od*: When the Offset–dops are non–convex we have a collection of convex components that represent the Od. Then, *Collision evaluation of the convex elements* cost can be computed as $O(N_{cc_k} * n_{v_k} log\ n_{v_k})$. Where N_{cc_k} is the number of convex components and is of order of $O(v_c^2)$ (see section 3.2).

$$C_C = O(N_{cc_k} * n_{v_k} * log\ n_{v_k})$$

Algorithms for Collision Detection From Offset–dops Collision Evaluation

- *Algorithm G*: This algorithm computes the list of areas where $MinBox^*(Od(O_i, d_k))$ overlap at a determined level of detail d_k and in a specified container $C_{k,p}$. To compute these areas the intersection evaluation between the $MinBox^*(Od(O_i, d_k))$ for all $O_i \in C_{k,p}$ have to be performed. Due to that the $MinBoxes$ are parallelepipeds the intersection evaluation between two of them is of constant order $O(1)$. However, this operation has to be done between all of the elements that belong to the $C_{k,p}$ container, N_{ec_k}. Then, resultant cost of the algorithm is of order:

$$C_G = O(N_{ec_k}^2)$$

- *Algorithm H*: This algorithm search the faces of the pairs of Offset–dops $Od(O_i, d_k)$ and $Od(O_j, d_k)$ that belong to the collide_area and previously sough and defined by $collide\_area = OverlapingArea(MinBox^*(Od(O_i, d_k)), MinBox^*(Od(O_j, d_k)))$. The cost can be calculated as:

  ComputeCandidateFaces: For each face of the $Od(O_i, d_k)$ a comparation with the $MinBox$ coordinates is done (and the same for $Od(O_j, d_k)$), then this operation yields in a cost, C_{fM}, of order:

$$C_{fM} = O(N_{fO_k})$$

  where N_{fO_k} is the number of faces of an Od. However, this operation have to be performed for each *collide_area*, N_{ca_k}, of the *List_OdAreas*, the final cost of the algorithm is of order:

$$C_H = O(N_{ca_k} * N_{fO_k})$$

- *Algorithm I*: This algorithm computes the $OdAreas$ collision from the $List\_OdAreas$ at level d_{k-1}. The information that is useful to algorithm is the list of elements that their $Od(O, d_{k-1})$ belong to those $Areas\_d_{k-1}$. Then, for each component of the $List\_OdAreas\_d_{k-1}$ it is computed a new $collide\_area$ corresponding to the intersection space between the $MinBox^*(Od(O_i, d_k))$ and $MinBox^*(Od(O_j, d_k))$ at level d_k. As the $MinBox$ are parellepiped the operation is done in $O(1)$ order. But, this operation have to be performed for all the $OdAreas$ of the $List\_OdAreas\_d_{k-1}$. Then to compute the cost of the algorithm:

  $$C_I = O(N_{ca_{k-1}})$$

  Where $N_{ca_{k-1}}$ is the number of collide areas at level d_{k-1}. Well, in the case that the information of the *mother container* $C_{k-1,p}$ have been divided into two containers we will have to consider the cost of swaping one container from to memory. In the worse case, when all the collide areas of level d_{k-1} are distributed in two containers, the cost of the algorithm has to be increased.

- *Algorithm J*: This algorithm computes the $Faces$ of the offset–dops that really are colliding from the candidated list of colliding faces of the list $List\_Collide\_OdZones$. Then, to compute the cost of the algorithm we have to take into account the number of canditate faces, N_{fO_k} belonging to an Od, that have to be examined , and the total number of the $OdZones$ that a collision can be done N_{z_k}. Then we get,

  $$C_J = O(N_{z_k} * N^2_{fO_k})$$

A summary of the algorithm's cost for collision detection are exposed in table 3.

	Cost of Algorithms
Algorithm A	$O(N^2_{c_k})$
Algorithm B	$O(N_{ca_k} * N_{o_k} * n_{v_k})$
Algorithm C	$O(N_{cc_k} * n_{v_k} * log\ n_{v_k})$
Algorithm G	$O(N^2_{ec_k})$
Algorithm H	$O(N_{ca_k} * N_{fO_k})$
Algorithm I	$O(N_{ca_{k-1}})$
Algorithm J	$O(N_{z_k} * N^2_{fO_k})$

Table 3: *Summary of costs of the Collision Detection algorithms*

References

[Bar90] D. Baraff. Curved surfaces and coherence for non–penetrating rigid body simulation. In *ACM SIGGRAPH Conf. Proc.*, pages 19–28, August 1990.

[BKOS00] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry. Algorithms and Applications.* Springer-Verlag, 2000. ISBN 3-540-65620-0.

[Can86] J. Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(2):200–209, March 1986.

[FB97] M. Franquesa and P. Brunet. Analysis of methods for generating octree models of objects from their silhouettes. In *Proceedings of the MICAD*, volume 12, pages 33–65, 1997. ISBN 0298–0924.

[FB98] M. Franquesa and P. Brunet. Collision detection: Models and algorithms. Technical report, Software Dept. Universitat Politecnica de Catalunya, 1998. Ref: LSI–98–11–R.

[FHA90] A. Foisy, V. Hayward, and S. Aubry. The use of awareness in collision prediction. In *IEEE Proceedings Interna. Conf. on Robotics and Automation*, pages 338–343, 1990.

[JTT98] P. Jimenez, F. Thomas, and C. Torras. *Book: Robot Motion. Planning and Control. Chapter: Collision Detection Algorithms for Motion Planning*, pages 305–343. Springer-Verlag, 1998. ISBN 3–540–76219–1.

[Kam93] V.V. Kamat. A survey of techniques for simulation of dynamic collision detection and response. *Computers and Graphics*, 17(4):379–385, 1993.

[MW88] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *ACM Computer Graphics*, 22(4):289–298, August 1988.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational Geometry, An Introduction*. Springer-Verlag, 1985. ISBN 0-387-96131-3.

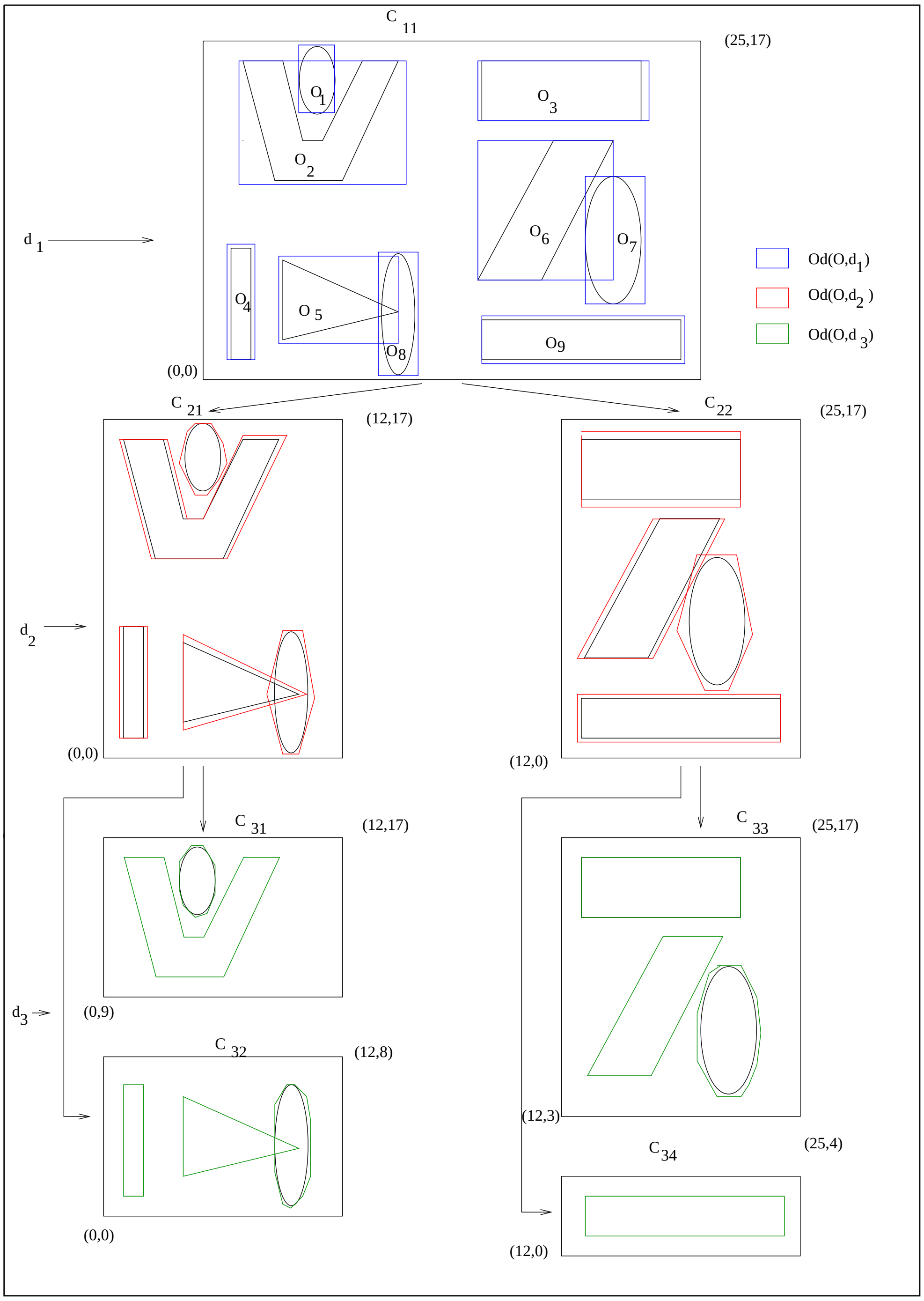

Figure 8: Example of several collision at different level of details, note that there are some false collisions

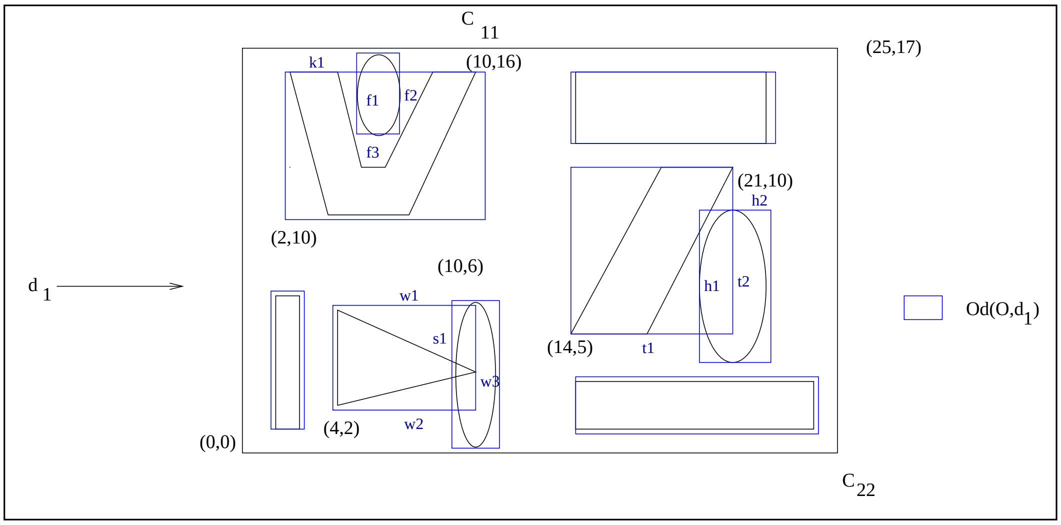

Figure 9: Zoom of the figure 8 at level d_1 of LOD

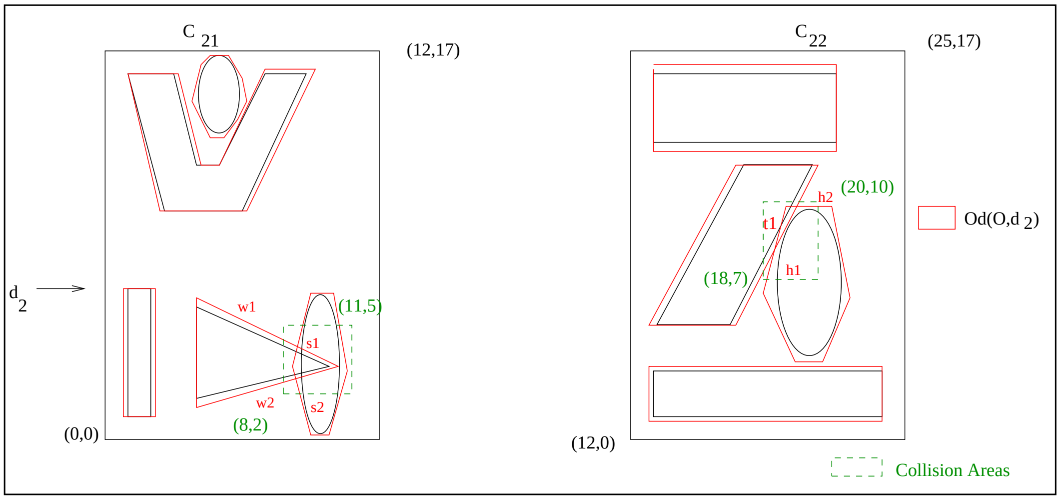

Figure 10: Zoom of the figure 8 at level d_2 of LOD

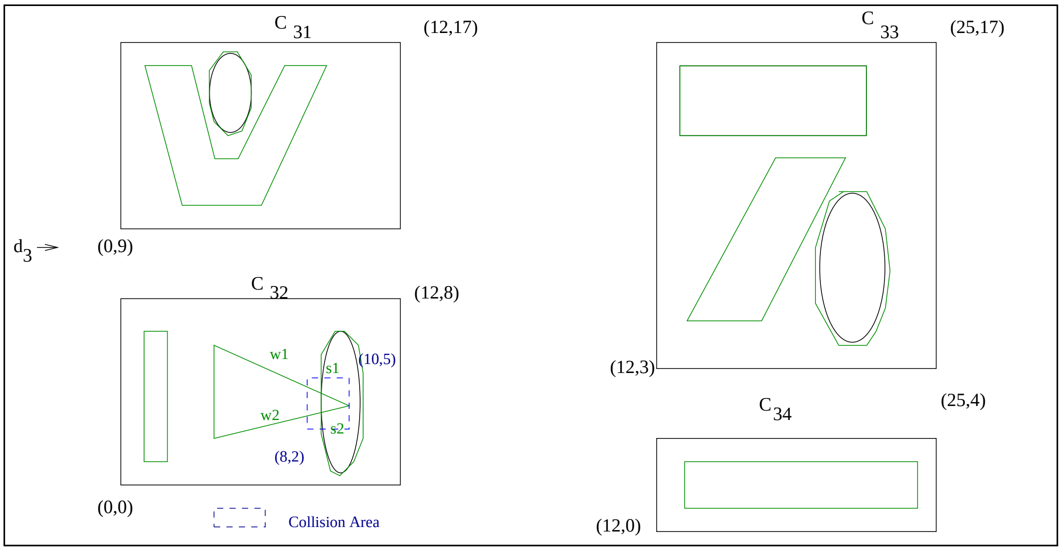

Figure 11: Zoom of the figure 8 at level d_3 of LOD