University of Zurich
Department of Informatics

SEMESTER THESIS - Visualization and MultiMedia Lab, Prof. Dr. Renato Pajarola

Spatial Structures and Multidimensional Data.

Semester Thesis in Informatics
submitted by

Florian Spöring
Zürich, Switzerland
Student ID: 01-450-030

Supervisor: Daniel Bisig, Jonas Bösch
Date of Submission: August 29th, 2008

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Table of content

1 Introduction

Since the ISO Framework was built and released the performance of it was quite limited. Although there was sufficient effort put into choosing a fast and open enough space partitioning technique which meets the requirements, the outcome was still quite limited. Because the whole ISO Framework is quite generic it is not possible to use specialized partitioning techniques which focus strongly lies on using inherent properties of the system to speed up the application and lessen the use of resources.

Because the goal of this application is to produce music, there is a need for a high number of dimensions which are representing properties of music like sequence, rhythm and other things. The other requirement needed is a high number of agents, because these agents are simulating the relations between these properties. Thus a high number of dimensions and a high number of agents are leading to a highly complex structure with a huge amount of calculating.

One goal by the creation of the ISO Framework was to build something highly customizable. Therefore the initial space partitioning structure was the use of an n-tree, which is applicable for any number of dimensions. The drawback of the chosen algorithm is that it is fairly time and resource consuming in building up and in updating. This lead to the fact, that the allowed number of participating agents is quite limited.

To overcome this limitations there are several assignments. As first, we introduce a simple fixed grid partitioning technique. This implementation should be used if not a big number of dimensions have to be simulated. By using this method, we should have a super fast algorithm for simulating a big number of agents. Another goal of this algorithm is to get in touch with the ISO Framework and learn how to use it.

A second assignment is to do a literature research for other useful space partitioning techniques and to evaluate their applicability for the ISO Framework.

2. Fundamentals

2.1. Introduction to space partitioning

In many applications it is necessary to deal with multidimensional data. That could be the case with Database Management System or with computer graphics or many others.

“These points can represent locations and objects in space, as well as more general records. […] Such records arise in database management systems and can be treated as points in multidimensional space albeit the different dimensions have different type units.” (Samet 2006, 1)

Therefore it is crucial to choose an appropriate data structure to deal with these data. By carefully choosing the best matching data structure it is possible to save a lot of resources. May it be time or memory or both at the same time.

By using the term multidimensional data one usually speaks about a collection of points in higher dimensional space. These points represent data with as many parameters as there are dimensions. Possible points are for example data records in a database. These records have different attributes. For example if we have a look at a person record, a record can have attributes like first name, last name, address, a birthday and a social id. This record would be stored in a 5 dimensional space, because it has 5 different attributes.

Why it becomes important to choose an appropriate data structure to deal with multidimensional data is because an application usually has to manipulate: to find, insert and delete this data.

“Bereits im täglichen Leben machen wir aber die Erfahrung, dass die richtige Organisationsform für eine Menge von Daten und damit die richtige Datenstrukturwahl ganz erheblichen Einfluss darauf hat, wie effizient sich bestimmte Operationen für die Daten ausführen lassen.” (P.Widmayer 2002, 15)

Depending when and how often these operations take place, a data structure will be more suitable than another one. For example if we consider static data we just have to build up the whole structure once. Therefore we can choose a data structure which is costly by creating, but very cheap for searching a particular data set or a range of data sets. On the other hand, if data is dynamic and chances a lot we maybe need to use a structure which is much cheaper to build up and insert new data, but also slower in finding data.

If one deals with spatial data a big class of data structures is based on space partitioning techniques. Space partitioning is done by division of space into two or more subsets. It divides the space into non overlapping regions. This process is repeated until every single data point lies in exactly one region and no other data point occupies the same region. Therefore, after partition the space, one can identify every data point by its region the point occupies.

By applying the same method recursively, a space partitioning technique leads to a hierarchical system in which the outcome of it – the created regions - can be represented in a Tree.

2.2. Description of ISO

To get in touch with ISO, I will firstly present an overview what the ISO Framework is and what it does. After building up the basic knowledge about ISO we begin in analyzing how ISO Space is built. In that chapter we make an in depth image of what the requirements are, what objects are involved, what space partitioning techniques are used and all other important facts one needs to know about it.

2.2.1 Introduction to ISO – What is it?

ISO is an abbreviation and stands for Interactive Swarm Orchestra. As the name is saying, the core business of ISO is a piece of software which is there to create music – if one is willing to say to compose music. Music in that context shouldn't be created according to how music used to be composed, but it should be a continuous process of creating music on the fly by reacting to changes in the environment.

"The ISO project is a manifestation of our belief, that practical and conceptual ideas from Artificial Live (ALife) provide an excellent foundation towards the establishment of a coherent approach to several important aspects of computer music (sound synthesis, composition and interaction). Our approach employs a generic swarm simulation as intermediary between musician(s), sound generation and acoustic projection. It is the simulated agents' behaviors that affect the mapping of the performer's activities into musical structure and its timbral, temporal and spatial development. We intend to shift the creative focus of a musician's work towards the design of properties, behaviors and interrelationships among agents and their musical dependencies."
(Daniel Bisig, Interactive Swarm Orchestra, 3)

A possible application for example would be that a region is observed by video cameras. The cam is passing its data to a tracking device, which is extracting motion information out of these pictures. These motion information then acts as input to influence the way how the music is created in background through ISO.

ISO consists of 3 main parts. First we have ISO Synth which is responsible for creating and manipulating sound. This is a synthetic device to alter sounds in a number of ways. There are different units, all of them having different properties and capabilities. By combining these units one can generate new sounds. But because it is in the nature of units that once two or more of them are connected to each other, they just keep doing what they are until they are released. Therefore, beside the units and basic functionality for music synthesis, there are Events. Events are metaphorically speaking action messenger sent to notify a unit at a distinct time step with a message. By using these Events one can alter and control these units.

Second, we have ISO Tracker which is responsible to track motion and pass the extracted information to the ISO Flock application. In these classes beside figuring as input to the agents, one also can specify special conditions under which Events are generated to influence Agents additional.

The last fundamental ISO Flock class deals with spatial calculations. This space class contains spatial partition algorithms for the calculation of Euclidian distances among parameters and thereby manages their neighborhood relationships.

“The main non-generic aspect of this library concerns its focus on simulating large numbers of agents each of which possesses a very simple morphology.”
(Daniel Bisig, Interactive Swarm Orchestra, 6)

The generic swarm library is the core piece of ISO where the most implementation effort was put in. In the ISO Flock part the simulating of the swarms is done. These swarms can influence each other and have behavior allocated to them. Due to these behaviors these swarms are reacting and creating Events to control the music which is generated.
One important concept to speak about is the agent which a swarm consists of. An agent is a labeled container for parameters and behaviors. By adding parameters and behaviors it is defined how an agent updates itself. A behavior defines a functional relationship among parameters. If a behavior is executed the agent updates its state according to its Input and the influence of its neighbors which are present in the neighbor group. All agents are updated in parallel thus the order in which they are updated doesn’t play a role. Beside the behaviors, parameters of an agent can also be changed through Events.

2.2.2. Analysis of ISO Space

Since there exist almost no documentation about what the different classes are representing, I try to give a short overview about what the most important classes are doing. All information given here refers to the official documentation which can be accessed through http://i-s-o.ch/doc/index.html .

The Space class is spanning up a space to simulate swarms for different parameters. A space can contain one or multiple swarms. These swarms are built up by agents.

An agent is a labeled container for parameters and behaviors. Every parameter is expressed through a space object which by itself has a position in space and a value, which expresses the parameter value for the moment. A space-object can have neighbors.

These neighbors have designated properties and are managed in neighbor-groups. Neighbor groups are built to manage and maintain Euclidian distance between two neighbors. This information is used later on to trigger certain behavior. To manage distance calculation between space object the ISO Framework provides 3 different distance calculation types. To use one of them, you pick the appropriate one through the neighbor group alg class.

Whenever a space object has to be inserted into a neighbor group a space proxy object is created and inserted into the neighbor group. This space proxy object is needed, since a space object can be present in more than one neighbor group.

As mentioned before an agent consists of parameters and behaviors whereas a parameter can express something like force or velocity and a behavior is triggered when a certain condition is satisfied e.g. a certain velocity is hit.

A behavior distinguishes between input parameters, internal parameters and output parameters. If a behavior is executed, it reads from its input parameter, its internal parameters as well as neighbor groups and writes the outcome to its output parameters. The output is buffered and updated at once for all agents, to prevent the order in which the output parameters are calculated are affecting the outcome.

A swarm itself is built of a group of agents with the same structure. This structure is built up of behaviors and parameter. Every agent in a swarm has the same parameter and behaviors with of course different values for each of them. Important to know is that it is possible to allocate parameters and behaviors to a swarm as well. In fact, a swarm can be treated as if it were an individual agent!

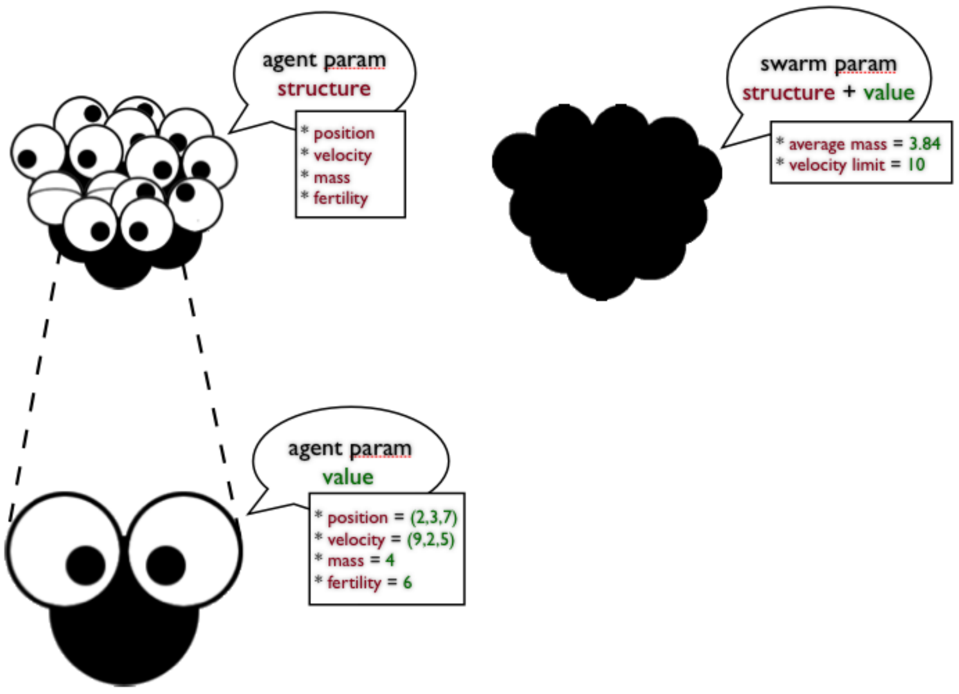

Figure 1
Relationship between agents and swarm(Source: http://i-s-o.ch/doc/index.html)

In the ISO Space there exist already a number of different spaces for different applications. Right now there is implemented a grid-space, a point-space and a shape-space.

“In its current implementation, the ISO Flock library provides a group of specialized spaces and behaviors that inherit from the generic base classes. The “point space” class manages distance calculations among point like spatial objects (i.e. parameters) via a quad tree, oc tree or higher dimensional space partitioning algorithm. The “shape space” class implements an R-Tree algorithm for the calculation of distances among objects in space that possess a shape (as opposed to point like objects). This allows agents to move along splines or on the surface of triangulated meshes. Such spatial objects can be employed to structure the environment within which agents exist. An example application transforms the tracked outline of people into a spline that serves as movement guide for agents. For a similar purpose another type of space manages the distribution of vectors on an n-dimensional regular grid. Such grids can serve for example as static or dynamic force fields and propel or slow down agents as they move through space. Another example application updates such a force field based on tracked visitor motion.”
(Daniel Bisig, I-S-O Documentation)

Because neighbor groups have to deal with distance between agents, at every step of the simulation it is necessary to build up the whole structure again. For every Agent the neighborhood calculation has to be done.

As we already mentioned, the point-space manages neighborhood calculations, by calculating the Euclidian distance. The existing point space spatial partitioning algorithm is rather slow because the neighborhood calculation within a multidimensional n-Tree is very time consuming. This limits the performance of the ISO Flock to a maximum number of agents depending on the number of dimension. To extend this number, we try to implement faster space partitioning techniques, which we are discussing in detail in the next chapter.

3. More detailed description of individual Space Partitioning techniques

Note: For simplicity reasons, we discuss all the trees in 2 dimensions with the following sample data

NAME	X	Y
Chicago	35	41
Mobile	52	10
Toronto	62	77
Buffalo	82	65
Denver	5	45
Omaha	27	35
Atlanta	85	15
Miami	90	5

Figure 2
Sample data for the different spatial partitioning algorithms

3.1. Fixed Grid

As a first improvement for performance there is a simple fixed grid implementation. A fixed grid method partitions the space into rectangular cells by overlaying it with a grid. Each cell is of a fixed size, whereas each dimension can have its own size. Beside the fix size, a cell contains a pointer to another structure (e.g. a linked list) which contains the set of agents which are present at a time step within this cell. The grid itself has to implement a method which determines to which cell an object is linked to. Usually the access structure to access the cells is a d-dimensional array or a tree where each leaf represents a cell.

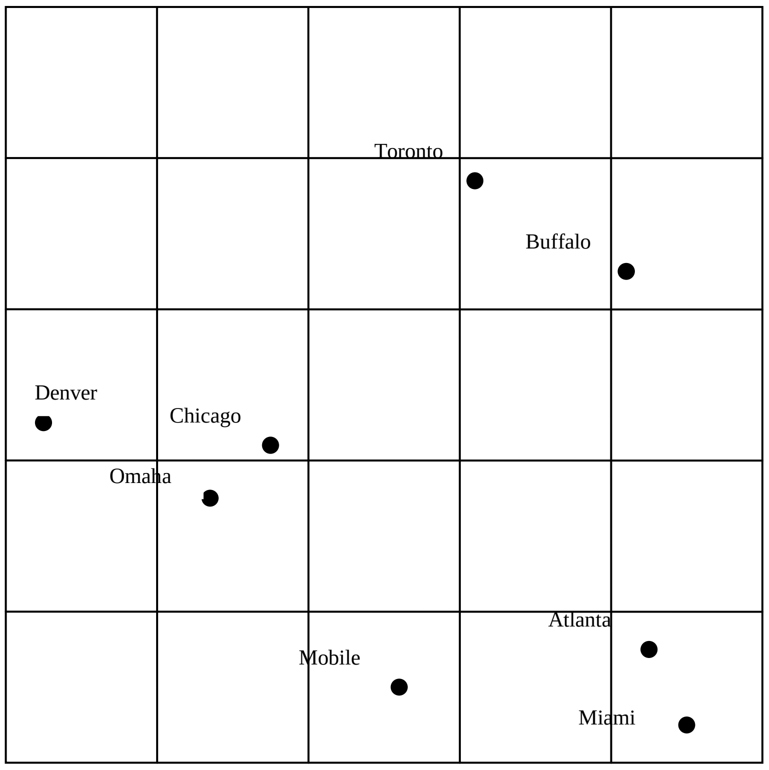

Figure 3
A fixed grid representation of the sample data in Figure 2

There are 2 ways to build up a fixed grid. One either can subdivide the space into equal sized grid cells, or one can subdivide it at arbitrary positions which are dependent on the underlying data. The difference between these two types is that in the first the access structure is quite easy whereas in the second type the access is quite difficult to realize since the data of the borders for the cell is different and has to be stored explicitly for every cell.
The advantage of an array access structure lies within the fact, that every cell is very easy to address and to access.

Building up such a fixed grid is quite fast, since one can calculate the assigned cell directly out of the information which is present in the object and does not have to check the whole structure. Thus, since every object should be present in the grid has to be inserted into it and assuming that there are N objects the needed time to build up the structure is O(N).

3.1.1 Insertion

To insert an object, one has to compute the cell index of the assigned cell. This is done by using the fact that the cell-size is known and that we know the position of the object which has to be inserted. Therefore insertion takes a constant amount of time an lies in O(k) where k addresses the constant computation steps one has to complete.

3.1.2. Deletion
Deleting an object is done in a similar fashion. As first, one again has to calculate the assigned cell through inherent data of the object. After the cell is known, depending on which data structure is used to store the present object of a cell, deleting the object takes as most O(k + m) time, whereas m designate the number of neighbor object in the same cell.

The use of an array access structure is fine as long as static data is concerned. But it has its drawback with dynamic data. If one is dealing with dynamic data the possibility gets bigger that a cell becomes too crowded while others stay empty and one looses the advantage of the easy access, because you need much calculation time to find the wanted object inside the crowded cell. These drawbacks are addressed with the introduction of variable sized cells.

3.2. Quadtree
While we were dealing in the case of a fixed grid with equally sized spatial structures which can produce a large number of empty cells in a dynamic environment, we are now introducing a structure which merges adjacent empty grid cells to larger ones. The resulting reduction of the number of cells results in a decrease of search time. If a cell gets too crowded, there will be a split and new cells with smaller number of objects are introduced.

A quadtree is a tree access structure on a grid. Its basic idea is to have a fast and easy way to adjust access structure which divides the cells the way that every cell only contains a maximal number of objects. If a cell is full but still a new object is added, the space will be further subdivided. If a cell becomes empty, the cell tries to merge with neighboring empty cells. The subdivision divides a region into 4 sub-regions, a northwest, northeast, southwest and southeast quadrant thus the name quadtree. A quadtree therefore is an access structure for 2 dimensional data, since the division takes place along 2 dimensions.

In general there are 2 types of quadtrees. A first type of a quadtree is the Point quadtree which is subdividing the space according to data point values.

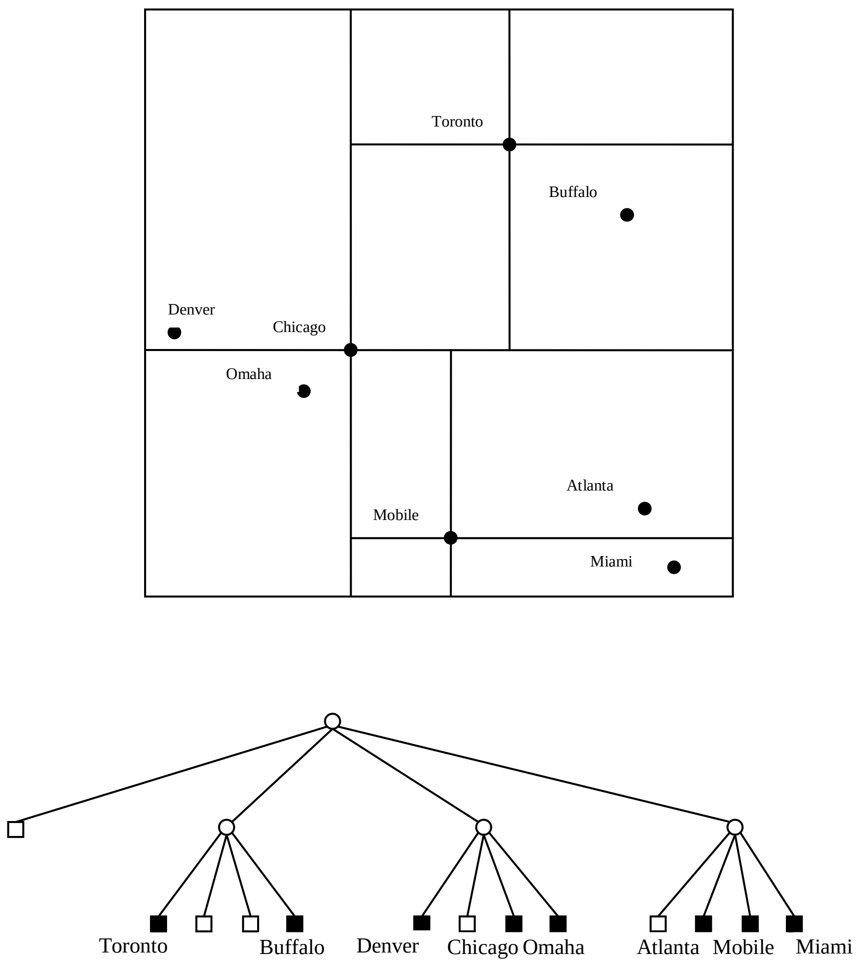

Figure 4
A point quad tree and the associated space partitioning of the sample data of Figure 2 (Source Samet, 2006, 28)

In the point quadtree of course the shape of the resulting tree depends on the sequence in which the objects are inserted into the tree.

The second type is called trie-based and decomposes the space into region based on the distribution of the data points. Trie is an abbreviation for information re**trie**val which shows that these particular trees are well especially well suited for search procedures.

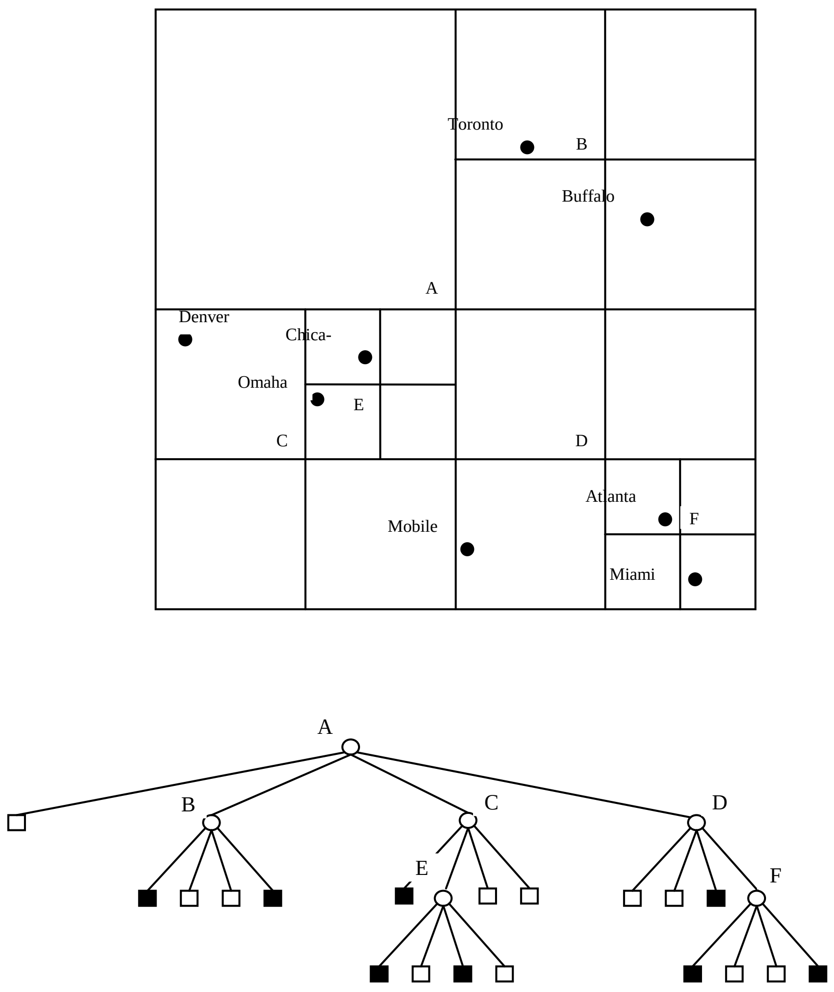

Figure 5
A point-region quad tree and the associated space partitioning of the sample data of Figure 2
(Source Samet 2006, 34)

3.2.1. Search

quadtrees are especially well suited for applications that involve proximity search. Typically these searches are of the type like search all cities given a data point which are lying within a certain search radius. The efficiency of such searches in Quad-trees is gained by the fact that with every search step one can prune a big region and therefore a lot of data records don't need to be examined. Since every object belongs to exactly one region all the other sub trees belonging to the other regions can be pruned away

3.2.2. Insertion

Inserting a node in a Quad-tree is quite simple. Since each record r has the key values (a, b) we have to search first for the value pair (a, b). If the tree is empty, we create a new node for the record. If not, we first search for the node h with values (a, b). If a node is found, we replace the node with the new record r. If the node is not found, we are getting a nil- child node in the 2 dimensional space of the type northwest, northeast, southwest or southeast. The only difference from a binary search tree is the need to perform at each node a four-way comparison within the tree. This four-way comparison is needed to distinguish in which quadrant the new node lies.

In the case of a trie-based quad-tree the insertion procedure is slightly different, because all the regions resulting from the subdivision process have to be of equal size. If the domain of the data points is discrete and we take every data point as a nonzero element, we have an image closely related to a matrix. Therefore this approach is called the MX quadtree. The difference in the inserting procedure is that if the search stops at a nil pointer the space is repeatedly subdivided until it is a 1 x 1 square. The length of that square is given by the number of subdivision and the domain which is spanning the space.

If the underlying space is not discrete and finite, this approach is not feasible anymore because the minimal number of separation between the data points is unknown. Therefore it is impossible to subdivide the spanning space into 1 x 1 squares. This leads to an alternative adaption of the region quadtree to point data. A leaf node is now corresponding to a region with at most one data point allocated to it. This approach is called PR quadtree whereas the P stands for point and the R stands for region. The insert procedure is now changed as follows. For a data point r we are searching for the region in which the data point belongs. If the region is already occupied by another data point s with different coordinate values, we must start splitting the space repeatedly until r and s no longer occupy the same block.

3.2.3. Deletion

The simplest way to delete a node is by reinsert the whole sub tree from the deleted node on. Of course this is very costly and time consuming if the deleted node is not a leaf node. There exists another way to delete a node and rearrange the tree, but since the deletion method is not of a particular interest the interesting reader is referred to Hanan Samet’s book Multidimensional and metric data structures page 31.

The delete procedure in a trie-based quadtree (MX- or PR- quad tree) is much simpler. Since here, data is always stored in a leave node. We simply have to delete that leaf node, and therefore there is no need to rearrange the tree. However, if a leaf node is deleted, we maybe have generated a lot of empty cells in both cases. If this is the case, we have to collapse cells. Collapsing cells means to merge empty cells until the state of a tree is again a valid one.

3.2.4. Comparison between point- and pr quadtree

Point quadtree	Point region quadtree
Subdivision based on point location	Subdivision into regular decomposition
Data points stored within the tree	Data points stored in leaf nodes
Deletion needs to update the tree	Simple leaf node deletion
No boundary	Fixed boundary
Shape and size is sensitive to the order of input	Size and shape independent of order in which the nodes are inserted
Coordinate information have to be stored within every data point	No need to store coordinates

3.3. K-d trees

For increasing dimensionality of the underlying space, each level of decomposing of the quad-tree results in many new cells. This results in a Fan out of around 2^d with d as the number of dimensions. This is not the case when using a variant of a K-d tree. Here, the space is partitioned at every level of the tree on the basis of just one attribute. In other words, at each level of the tree just one attribute value is tested instead of all attributes in the case of a quad-tree.

By restricting the number of tests at each level, we gain a lot. First, the overall resulting size of a tree is much smaller, because for every node, at most one additional split is necessary. This makes the algorithm a lot easier because at each level we only have two options to choose from, since the underlying space is divided into only two parts. A second advantage is that at each tree level only one test has to be done, instead of d in the case of a quad-tree. This fact alone speeds up the search procedure quite a lot. As a last point worth mentioning, one data structure can be used to represent a node for all values of d, since we don't need to care about what region we are processing right now.

But all these advantages mentioned do not come for free. By replacing all tests of the attributes by one test, we are creating a structure which becomes highly sensitive to the order in which the data points are inserted. Because only one attribute is tested at each level, the former parallel procedure of testing for attributes becomes a serial one.

3.3.1. Point K-d Tree

Although there are many variations of the point k-d tree whose exact structure is depending on details, we focus on the most common variant. Here the k-d tree partitions the underlying space at the data points and cycles through the different axes in a predefined and constant order. For simplicity reason, we again discuss the case of a two dimensional k-d tree. At each even node level we test along the x axis, at each uneven node we test along the y axis. If we are testing a data point P, then all data point with an x coordinate value less than P are in the left child of P and all those with a value equal or bigger than P are in the right child of P at an even level. Similar convention holds for P at uneven level.

3.3.2. Insertion

With that in mind, inserting records is very simple. First, if the tree is empty, we allocate a new node containing the data point r which we would like to store and we return the tree with r as its root node. If the tree is not empty, we are searching for a node h with a record having key values (a, b). If h exists, we replace the existing record associated with h. If the search reaches a nil node, we allocate a new node t containing r and make t a child of node c. It is important to understand, that at each node level we have to store what test we need to test for. This procedure is basically the same as for a quadtree or a binary tree.

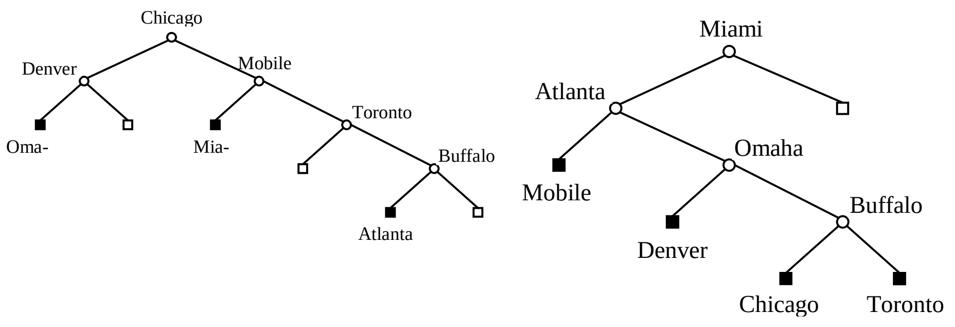

Figure 6
Point k-d tree by inserting Chicago, Mobile, Toronto, Buffalo, Denver, Omaha, Atlanta and Miamy. The tree on the right is created with reverse inserting order.

3.3.3. Deletion

As one can observe that not every sub tree of a k-d tree is itself a k-d tree, the deletion of a node becomes more complex than it is for binary search trees. We can not just simply remove a node and replace the node by the resulting sub tree since the sub tree itself can harm the integrity of a k-d tree. This is, because sub tree values might not have the same relative relationship to their new depths as they had before.

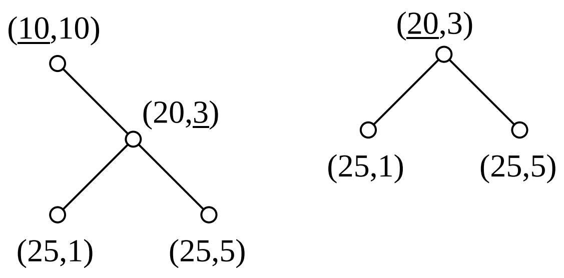

Figure 7
a) Example of a two-dimensional k-d tree whose (b) right child is not a k-d tree (Source Samet 2006, 52)

But still we can use a recursive process for deleting nodes in a k-d tree. This procedure goes as follows: If both sub trees of the node we wish to delete are empty, we replace the node by an empty tree. If that's not the case, we search for a suitable replacement node in one of its subtrees and recursively delete that node.

This procedure works because if we are looking at an even node that node is according to our definition an x-discriminator with the values (a,b).Now for every node of the left subtree got an x coordinate value smaller than a. Every node in the right subtree has an x coordinate equal or greater than a. It seems now as if we have a choice to replace (a,b) with the right subtree with the maximum x coordinate value, or to replace (a,b) with the minimal x coordinate value of the left subtree.

This is not the case. If we decide to replace (a,b) by its right subtree (c,d) and there exists another subtree with value (c,h) our definition of the tree is harmed. Therefore we have to look for the substitute always in the left subtree.

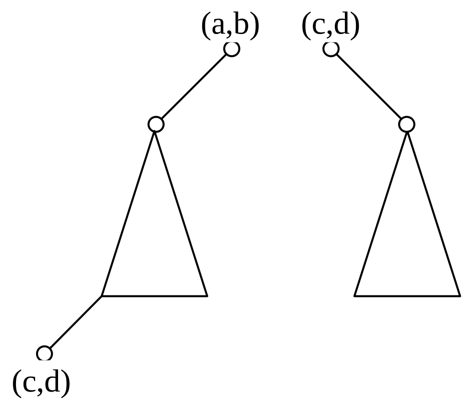

Figure 8a) Example k-d tre and b) the result of deleting (a,b) from it (Source Samet 2006, 53)

This leads us to the question what to do if the left sub-tree of a deleting node is empty. Since replacing the deleted node with a node of the right sub tree the integrity of our tree could be injured. To solve this problem one searches the right sub-tree for the smallest coordinate value of the deleted category (c,d) , exchange the left and the right subtree and replace the deleted node (a,b) with the found node and recursively apply the deletion procedure to the found node (c,d).

As an example of the deletion process, consider the tree of Figure 10. By assuming that A at (20, 20) is an x-discriminator, we see that C at (25,50) has the minimum value of the right subtree. Therefore, we replace A with C and continue with the deleting procedure at the former C node.

Since C's right subtree is empty, we have to search its left subtree for the node with a minimum value of a y coordinate. Thus D with the minimal y coordinate is attached as right subtree, replaces C's former position and is then deleted. Because D was an x-discriminator, we replace it by the node in its right subtree having a minimum x coordinate value.

H satisfies this minimum value condition and therefore is moved up in the tree.

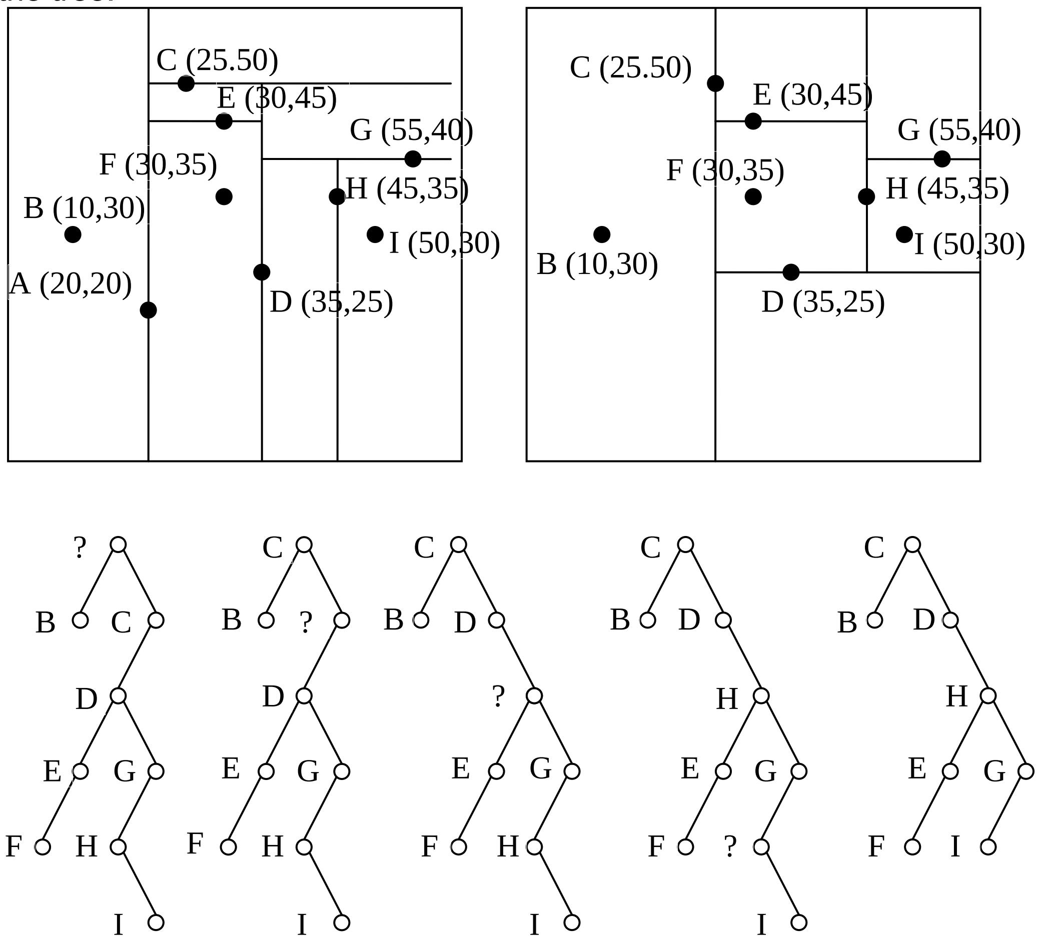

Figure 9
Example illustrating deletion in k-d trees where "?" indicates the node being deleted: (a) the original k-d tree, (b-f) successive steps in deleting node A, (g) the final k-d tree.
Source (Samet 2006, 54)

3.3.4. Search

Like a quad tree, a k-d tree is particularly useful in applications involving search. While seeking all nodes within a specified distance of a given point, a big amount of search can be pruned, which otherwise would be required. For the following discussion, we are interested in all points having a Euclidean distance from a given point less or equal to a search radius r.

From the assignment we know that for any point of interest the equation $r^2 \geq (a-x)^2 + (b-x)^2$ has to be satisfied, where r nouns the search radius, (a, b) the given point and (x, y) the point of interest. This equation leads us to the following properties.

Indicated through that equation we know that we are observing a circular region. The minimum x and y coordinate values of a node in this circle cannot be less than $a - r$ and $b - r$, respectively. The same property holds for the maximum coordinates but with changed sign. Therefore at every node level (e, f) we compare this point with the point$(a - r, b - r)$. If the resulting point lies within the right subtree we know that all data points in the right subtree have an Euclidean distance bigger than r. At the same point we also compare the point of interest (e, f) with the point $(a + r, b + r)$ if the resulting point lies in the left subree we do not need to search further in that tree.

3.3.5. PR K-d tree

If we have a big number of data points, it takes a lot of resources to build a point K-d tree, since in every node we need to store its location. To save memory, in that case a method similar to a PR Quadtree was developed. Like in the Quadtree case, the examined space is not divided on the basis of a point location but the space is just divided into halves. This is done recursively until every data point lies in its own region. This halving process is done in the same manner as a point k-d tree is built. The splitting is done by cycling through the different axis in a random but constant order.

A big disadvantage in many PR K-d trees is that they start having problems if the data points are not distributed uniformly. If many data points are clustering together, a lot of splitting is required producing many nodes and the resulting tree becomes unbalanced.

One Solution which is addressing that problem is to use buckets. A bucket has a predefined size b and splitting only takes place if a bucket is full, e.g. if a bucket contains already b data points. If a new data point is going to be inserted in such a bucket, the space is split again and the data points the bucket contained are distributed to the newly build buckets according the PR K-d tree rule.

3.4 Comparison between quad- and K-d tree

Quadtree	K-d tree
Test for every dimension at each level	At each level only 1 dimension is tested
Multitude of NIL links	Only few NIL pointers
parallel	Sequential process
Order in which the nodes are inserted does not influent shape and size of tree	Sensitive to the order of inserting nodes

4. Implementation Procedure

4.1. Fixed Grid

In order to realize this fixed grid, I created 2 classes. Classes iso_space_point_space_grid and iso_space_point_space_grid cell. If a fixed grid is initialized the size of the used array is calculated an array with of fixed grid cells with the calculated size is created.

```
//calculate size of array (space)
int tmpSize = 1;
for(unsigned int i=0; i<pSubdivisionCount.dim(); i++){
        tmpSize *= mSubdivisionCount[i];
}

//create cell array
mCells = QVector<FixedGridCell*>(tmpSize);
for(unsigned int j= 0;j<tmpSize; j++){
        FixedGridCell tmpCell = FixedGridCell();
        mCells[j] = &tmpCell;
}
```

The method *updateSpaceStructure()* is resetting the array to empty grid cells. Method *updateNeighbors()* which usually is called after updating the space structure then fills in all the space proxy objects.

To insert a space proxy object is easy. At first one has to check whether or not it can have neighbors. If it can, one creates the space proxy object as container for the space object. Then one has to determine fixed grid cell, where you have to insert the space proxy object.

To add neighbors, we implemented 2 different versions. A first very fast but also a bit limited version accepts only space proxy objects as neighbors which are lying in the same fixed grid cell. Since a cell can contain only one or a few space proxy object there is the possibility that our space proxy object doesn't get a lot of influence trough other neighbors, we give up most of the emergent properties of a swarm.

The second version we implement consists of a neighbor search within a certain radius. This radius is variable, but constant during runtime. Although the method addNeighbor() of the space-proxy-object is checking the condition against that radius too, we check that ourselves since we don't need to add all possible neighbors but only the one which are lying inside that radius. Although that check against the radius is done twice it is still much faster than just trying to insert all possible neighbors since to determine which other cells lie inside the radius does not require a lot of time.

4.2. Quad tree

Since an n tree (a variant of a quad tree for any number of dimensions) is already implemented, there is no need to implement this again.

4.3. PR K-d tree

For implementing a PR K-d tree we got the implementation by Jonas Bösch. The tree itself uses actually only one class iso_space_pr_kd_tree_node. Such a node has following interesting members:

_parent; parent node if any
_child_nodes[2]; its children if any
_bucket; a bucket with a static capacity
_split; the point where the next split will take place
_axis; dimension which has to be split
_aabb; the vectors spanning the actual region

_bucket_capacity;
_pool; a pool of pr kd tree nodes
_bucket_pool; a pool of buckets

If a node is being created all of its properties are set to NULL and remains out of any tree.

To insert a newly created node into a tree one has to call the insert method of the root node, or the newly created node store that root as root node. Calling the *insert(point)* method function as described in the PR K-d tree discussion.

```
if ( _has_children ){
        _child_nodes[ find_child ( point ) ]->insert ( point );
}
```

The *find_child(point)* function returns 0 or 1 depending on if the point's position belongs to the left or the right subree.

```
find_child(point){
        return ( point->position().c[ _axis ] < _split[ _axis ] ) ? 0 : 1;
}
```

As long as there are children, we travel trough the tree recursively by calling the *insert(point)* method on the child spanning the space partition our space proxy object belongs until we find a leave node. After the node is found, we are inserting the space proxy object into the bucket list.

```
if ( _bucket->size() == _bucket_capacity )
{
        split();
        _child_nodes[ find_child ( point ) ]->insert ( point );
}
```

If the bucket list is full already, we *split()* the space again into two halves otherwise we just configure the node the way that it now contains our inserted space proxy object.

The *split()* function creates 2 childnodes by calling *_create_child_nodes();* This method allocates memory for the new childnodes, and make them accessible through the node property *_child_nodes[0] , _child_nodes[1]* respectively.
Then the new childnodes are configured. The region which they are spanning now is set by the call of

```
_aabb.split_kd ( _axis, _split[ _axis ], child0->_aabb, child1->_aabb );
```

Followed by the call of to configure the rest of the properties

```
child0->_setup ( this, 0 ); child1->_setup ( this, 1 );
```

After the children are set up correctly, we have to divide the bucket of this node to the newly created child nodes. This is done by the call of *spill_bucket();* After that we do some clean up stuff because we don't need that bucket anymore since bucket are only used in leave nodes. Calling *_bucket_pool.destroy (_bucket);_bucket = 0*; and we are done.

The *spill_bucket()* function distributes all space proxy objects contained in a bucket by finding the appropriate node and inserting the space proxy object.

```
for ( ; it != itend; ++it ){
        _child_nodes[ find_child ( *it ) ]->insert ( *it );
}
```

Deletion a node is implemented too, but since we don't care about deleting a single node, I don't discuss this here in detail. The interesting reader can find the source code in the appendix.

For search queries we can recycle the *find_child(point)* method with a fake point as described at here and recursively call that method until a query returns a subtree which violates the search criteria.

5. Results

As first I would like to state, that the way how I was working on this wasn't particularly well suited. I didn't had much time to work on it in one bunch, so it took always a huge amount of time to get into it again.

One thing which made the work much harder than it actually is was the fact that almost no documentation exists. Although I had 2 or 3 meetings with Daniel Bisig who gave me an introduction to his code, the possibility to think about a concept myself and then read the information again was missing. This made it quite hard to get into the written source.

Many classes are named similar like GridSpace and SpaceGrid. This is a source of confusions. The problem would be smaller if there would be a comment of what the class is about and how it is used.

Generally, an overview of the different classes where it is explained what a class represents and how the class is used is missing. This makes it almost impossible to get the big picture. Even though the source code itself isn't that hard to read, it's quite hard to figure out how and for what the classes are used.

Maybe not everyone needs to see the big picture, but for me it is quite important. By understanding the big picture, it is much easier to find the different requirements. In the end, I just copied the point-space class and checked that I had all the methods implemented as they were implemented on the point space. I still cannot tell what properties the space really needs to have.

Overall, the assignment was maybe a little too difficult for me. I wasn't familiar with C++ and the environment, and I don't have a suitable background for space partitioning data structures. All this together made it quite time consuming and I did not proceed as fast as I thought I would.

This is sad, since I actually didn't had much time to develop my own ideas. I could not contribute as much as I wanted to the project.

Bibliography

Daniel Bisig, Martin Neukom, John Flury. „an artificial life approach to computer music.“ http://bitingbit.org/publications/pdf/iso_icmc_demo_2008.pdf (Zugriff am 27. 08 2008).

—. „Interactive Swarm Orchestra.“ http://bitingbit.org/publications/pdf/ISO_GA_2007.pdf (Zugriff am 24. 08 2008).

—. *I-S-O Documentation.* http://i-s-o.ch/doc/index.html (Zugriff am 22. 08 2008).

Mehlhorn, K. *Data structures and algorithms, Vol. 3: Multidimensional searching and computational geometry.* Berlin: Springer, 1984.

P.Widmayer, T.Ottmann /. *Algorithmen und Datenstrukturen.* Heidelberg Berlin: Spektrum Akademischer Verlag, 2002.

Samet, Hanan. *Foundations of Multidimensional and Metric Data Structures.* University of Maryland, College Park: Morgan Kaufmann, 2006.

TABLE OF FIGURES

Semester Thesis
Communication Systems Group
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://csg.ifi.unizh.ch

Appendix

```
/*
 *  iso_space_pk_kd_tree_node.h
 *  iso_space
 *
 *  Created by iso on 7/1/08.
 *  Copyright 2008 __MyCompanyName__. All rights reserved.
 *
 */


#ifndef _iso_space_pr_kd_tree_node_h_
#define _iso_space_pr_kd_tree_node_h_

#include <boost/pool/object_pool.hpp>

#include "iso_space_proxy_object.h"
#include "aabb.h"

#include <QVector>

#include <deque>
#include <list>

namespace iso
{
#define TREE_STATS

namespace space
{

class PrKdTreeNode
{

public:

	typedef	std::deque< SpaceProxyObject* >  bucket;

	/**
	\brief creates a tree node consisting with no parent, no children, marked as death
	and an emtpy bucket
	*/
	PrKdTreeNode();

	~PrKdTreeNode();

	/**
	\brief insert a space proxy object into the tree.
	\\	1. searches the tree according to the Space Proxy Object Position
	\\	2. adds the Space Proxy Object to the matching node if there is space in
	bucketlist
	\\	3. creates a split if it is neccesary
	*/
	void insert ( SpaceProxyObject* point );

	/**
	\brief divides the content of a bucket into the newly created childnodes.
	*/
	void	remove ( SpaceProxyObject* point );
```

```
		void	spill_bucket(); // sort bucket into child nodes

		/**
		 \brief splits the representated region along an axis into 2 region and creates the
		new children and set them up
		 */
		void	split();

		// get index of child in which the point belongs
		int  find_child ( SpaceProxyObject* point ) const;

		inline PrKdTreeNode* get_parent() const;
		inline PrKdTreeNode* get_child ( int num ) const;

		// child nodes queries
		inline bool	has_children() const;
		inline bool	is_leaf() const;

		// payload queries
		inline int   get_size() const;
		inline bool	is_empty() const;

		// bucket access
		inline bucket& get_bucket();
		inline const bucket& get_bucket() const;
		static void set_bucket_capacity ( const int capacity );
		static inline const int get_bucket_capacity();
		friend std::ostream& operator<< ( std::ostream& o, const PrKdTreeNode& node )
		{
			o << "pr_kd_tree_node\n"
			<< "  bucket size: " << ( ( node._bucket ) ? node._bucket->size() : 0 )
			<< "\n"
			<< "  " << node._aabb;
			if ( node.has_children() )
			{
				o << "child0 :\n  " << *node._child_nodes[ 0 ]
				<< "child1 :\n  " << *node._child_nodes[ 1 ];
			}
			o << std::endl;
			return o;
		}

#ifdef TREE_STATS
		const int	get_load_count();
		const int	get_node_count();
#endif

		inline const math::Vector<float> get_split() const;
		inline const int	get_axis() const;
		inline const aabb2f&	get_aabb() const;
		inline aabb2f&		get_aabb();
		inline void set_split ( const math::Vector<float>& split_ );
		inline void set_axis ( const int axis_ );
		inline void set_aabb ( const aabb2f& aabb_ );

protected:
		/**
		\brief creates new childnodes and sets the children flag to true
		*/
```

```
	void _create_child_nodes();

	void _destroy_child_nodes();

	/**
	\brief setup the node members (except _aabb)
	\param parent parent node of the newly built node
	\param pDim the number of
	*/
	inline void _setup ( PrKdTreeNode* parent, const unsigned int child_num );
	// check if this node is empty and all child nodes are empty
	bool	_zombie_check();
	PrKdTreeNode*	_parent;
	PrKdTreeNode*	_child_nodes[ 2 ];
	uint64_t	_path;
	bool		_has_children;
	bool		_dead;
	bucket*		_bucket;
	math::Vector<float>	_split; // the point where the next split will take place
	int	_axis; //number of dimension which has to be partitioned
	aabb2f	_aabb; // the vectors spanning the actual region

	static int _bucket_capacity;
	static boost::object_pool< PrKdTreeNode > _pool;
	static boost::object_pool< bucket >	_bucket_pool;
	static const int _max_children;

#ifdef TREE_STATS
	static int _node_count;
	static int _load_count;
#endif

};//class PrKdTreeNode




inline PrKdTreeNode*
PrKdTreeNode::get_child ( int num ) const
{
	assert ( num < 2 );
	return _child_nodes[ num ];
}



inline PrKdTreeNode*
PrKdTreeNode::get_parent() const
{
	return _parent;
}




inline int
PrKdTreeNode::get_size() const
{
	return _bucket ? _bucket->size() : 0;
```

```
}



inline bool
PrKdTreeNode::is_empty() const
{
	return _bucket ? _bucket->size() : 0;
}



inline bool
PrKdTreeNode::is_leaf() const
{
	return !_has_children;
}



inline bool
PrKdTreeNode::has_children() const
{
	return _has_children;
}



inline PrKdTreeNode::bucket&
PrKdTreeNode::get_bucket()
{
	assert ( _bucket );
	return *_bucket;
}


inline const PrKdTreeNode::bucket&
PrKdTreeNode::get_bucket() const
{
	assert ( _bucket );
	return *_bucket;
}



inline void
PrKdTreeNode::set_bucket_capacity ( int capacity )
{
	_bucket_capacity = capacity;
}
```

```
inline const int
PrKdTreeNode::get_bucket_capacity()
{
	return _bucket_capacity;
}



inline int
PrKdTreeNode::find_child ( SpaceProxyObject* point ) const
{
	assert ( point );
	return ( point->position().c[ _axis ] < _split[ _axis ] ) ? 0 : 1;
}



inline const math::Vector<float>
PrKdTreeNode::get_split() const
{
	return _split;
}



inline void
PrKdTreeNode::set_split ( const math::Vector<float>& split_ )
{
	_split = split_;
}



inline const int
PrKdTreeNode::get_axis() const
{
	return _axis;
}



inline void
PrKdTreeNode::set_axis ( const int axis )
{
	_axis = axis;
}



inline const aabb2f&
PrKdTreeNode::get_aabb() const
{
	return _aabb;
}



inline aabb2f&
```

```
PrKdTreeNode::get_aabb()
{
	return _aabb;
}



inline void
PrKdTreeNode::set_aabb ( const aabb2f& aabb_ )
{
	_aabb = aabb_;
}



inline void
PrKdTreeNode::_setup ( PrKdTreeNode* parent, const unsigned int child_num )
{
	assert ( parent );
	_parent = parent; // set parent
	memset ( _child_nodes, 0, sizeof ( void* ) * 2 ); //create space for children
	_path = ( _parent->_path << 1 ) | child_num; //set path to child
	_has_children = false;
	_dead = true;

	_split = _aabb.get_center();
	_axis = ( _parent->_axis + 1 ) % 2; //TODO change to n dimensions!
	// _aabb -> already setup in split() of the parent
}


#ifdef TREE_STATS
inline const int
PrKdTreeNode::get_load_count()
{
	return _load_count;
}



inline const int
PrKdTreeNode::get_node_count()
{
	return _node_count;
}
#endif

}; // namespace space
}; // namespace iso

#endif
```

```
/*
 *  iso_space_pk_kd_tree_node.cpp
 *  iso_space
 *
 *  Created by iso on 7/1/08.
 *  Copyright 2008 __MyCompanyName__. All rights reserved.
 *
 */

#include "iso_space_pr_kd_tree_node.h"

using namespace iso;
using namespace iso::space;

// static members
boost::object_pool< PrKdTreeNode >          PrKdTreeNode::_pool;
boost::object_pool< PrKdTreeNode::bucket >   PrKdTreeNode::_bucket_pool;

int PrKdTreeNode::_bucket_capacity = 2;
const int PrKdTreeNode::_max_children = 2;

#ifdef TREE_STATS
	int PrKdTreeNode::_node_count = 0;
	int PrKdTreeNode::_load_count = 0;
#endif


PrKdTreeNode::PrKdTreeNode()
	: _parent ( 0 )
	, _path ( 1 )
	, _has_children ( false )
	, _dead ( true )
	, _bucket ( _bucket_pool.construct() )
{
	memset ( _child_nodes, 0, sizeof ( void* ) * 2 );
}

PrKdTreeNode::~PrKdTreeNode()
{
	if ( _has_children )
		_destroy_child_nodes();
	if ( _bucket )
		_bucket_pool.destroy ( _bucket );
}

void
PrKdTreeNode::insert ( SpaceProxyObject* point )
{
	// if there are already children atached to this node, find the suitable place for the
	new SpaceProxy Object
	if ( _has_children )
	{
		_child_nodes[ find_child ( point ) ]->insert ( point );
	}
	else //there are no children
	{
		if ( _bucket->size() == _bucket_capacity ) //no space left in this bucket ->
		we need to split
		{
				split();
				_child_nodes[ find_child ( point ) ]->insert ( point );
			}
			else // try to add the spaceProxyObject into this node
			{
				_bucket->push_back ( point );
				#ifdef TREE_STATS
				++_load_count;
				#endif
				if ( _dead )
					_dead = false;
				//_bucket.sort( SpaceProxyObject_less( _position.get_offset(),
				_axis ) );
			}
		}
	}

	void
	PrKdTreeNode::remove ( SpaceProxyObject* point )
	{
		if ( _has_children )
		{
			_child_nodes[ find_child ( point ) ]->remove ( point );
		}
		else
		{
			bucket::iterator it = std::find ( _bucket->begin(), _bucket->end(), point );
			if ( it != _bucket->end() )
			{
				_bucket->erase ( it );
				#ifdef TREE_STATS
				--_load_count;
				#endif
				if ( _bucket->empty() )
				{
					if ( _has_children && _zombie_check() )
					{
						_destroy_child_nodes();
					}
					PrKdTreeNode* p = _parent;
					bool continue_removing = ( p != 0 );
					while ( continue_removing )
					{
						continue_removing = p->_zombie_check();
						if ( continue_removing )
						{
							p->_destroy_child_nodes();
							p = p->_parent;
							if ( p == 0 )
								continue_removing = false;
						}
					}
				}
			}
			else
			{
				// DEBUG
				std::cerr << "pos " << point->position() << std::endl;
//				throw exception ( "Could not remove payload from tree!" );
```

```
                }
        }
}




void
PrKdTreeNode::spill_bucket()
{
        bucket::iterator it     = _bucket->begin();
        bucket::iterator itend  = _bucket->end();
        for ( ; it != itend; ++it )
        {
                _child_nodes[ find_child ( *it ) ]->insert ( *it );
#ifdef TREE_STATS
                --_load_count;
#endif
        }
        _bucket->clear();
}




void
PrKdTreeNode::_create_child_nodes()
{
        assert ( ! _has_children );
        for ( int i = 0; i < _max_children; ++i )
        {
                _child_nodes[ i ] = _pool.construct();
#ifdef TREE_STATS
                ++_node_count;
#endif
        }
        _has_children = true;
}




void
PrKdTreeNode::_destroy_child_nodes()
{
        assert ( _dead && _has_children );
        for ( int i = 0; i < _max_children; ++i )
        {
                if ( _child_nodes[ i ]->_has_children )
                {
                        _child_nodes[ i ]->_destroy_child_nodes();
                }
                _pool.destroy ( _child_nodes[ i ] );
#ifdef TREE_STATS
                --_node_count;
#endif
        }
        _has_children = false;
        _bucket = _bucket_pool.construct();
```

```
}


bool
PrKdTreeNode::_zombie_check()
{
        if ( _has_children )
        {
                _dead = true;
                for ( int i = 0; _dead && i < _max_children; ++i )
                {
                        _dead = _child_nodes[ i ]->_zombie_check();
                }
                return _dead;
        }
        else
        {
                return _bucket->empty();
        }
}


void
PrKdTreeNode::split()
{
        _create_child_nodes(); // creates new childnodes (max childnodes hardcoded set to
2 )
        PrKdTreeNode* child0 ( _child_nodes[0] );
        PrKdTreeNode* child1 ( _child_nodes[1] );
        _aabb.split_kd ( _axis, _split[ _axis ], child0->_aabb, child1->_aabb );
        child0->_setup ( this, 0 );
        child1->_setup ( this, 1 );
        spill_bucket();
        // this node has child nodes now, so we don't need the bucket anymore
        _bucket_pool.destroy ( _bucket );
        _bucket = 0;
}
```