Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Lecture Notes for

Managing and Mining Multiplayer Online Games

for the Summer Semester 2017

Chapter 2: The Game Core (part 2)

http://www.dbs.ifi.lmu.de/cms/VO_Managing_Massive_Multiplayer_Online_Games

Sharding and Instantiation

- copying a region for a specific group
- any number of the same region exist
- instances and shards were primarily created for game design purposes
  (Limiting the number of players for a quest)
- but: The more players are in an instance, the less performance issues in the open world.

Complications:

- does not solve the underlying Problem(no connected MMO-World)
- storing local game states, even if there are no more players in the instance

  => instance management can cause additional expenses
  (worst case: 1000 parallel game states for 1000 players)

Zoning

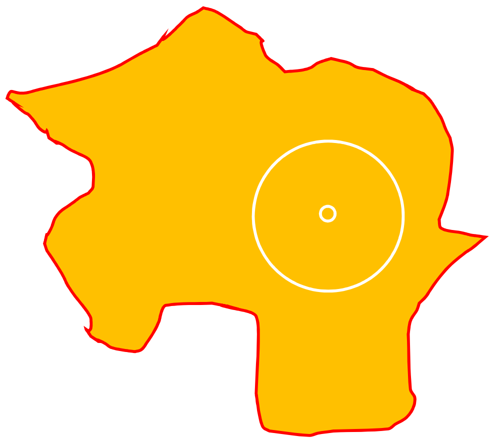
- Splitting the open World into several fixed areas
- Only objects in the current zone need to be considered for a query
- Does not only partition space, but also the game state
- Makes it easier to distribute the game world onto several computers

Complications:

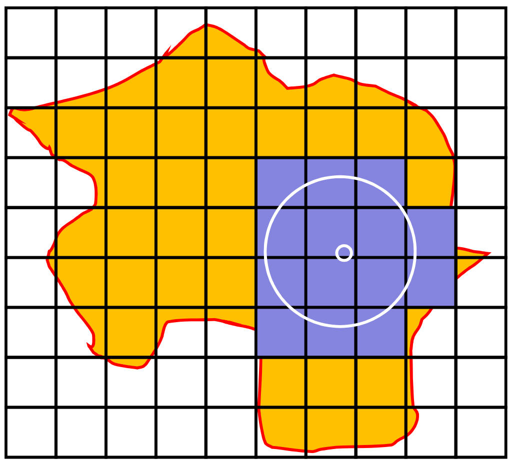
- In peripheral areas may necessitate taking objects of bordering zones into account
- Uneven distribution of players

Micro-Zoning

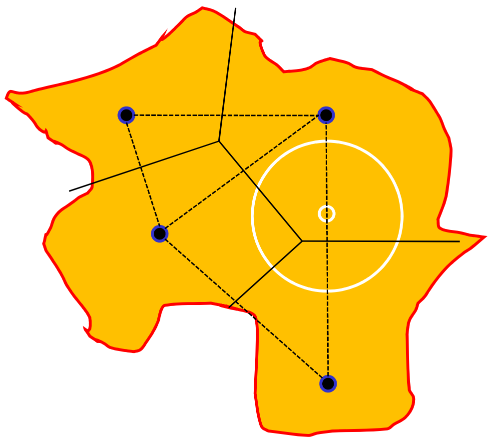
- game world is partitioned into several small areas (micro zones)
- only game entities within the actual micro zone are being managed
- only micro zones that intersect the AoI are relevant
- sequential search within the region
- zones can be created with different methods (grids, Voronoi-cells, ...)



zoning



micro zoning (grid-based)



micro zoning (Voronoi based)

Spatial Publish-Subscribe

- combination of micro-zoning and a subscriber systems
- game entities are registered in their current micro zone (publish)
- game entities subscribe to the information of all micro zones that intersect their AoI (subscribe)
- list of all game entities within AoI is created by merging all entries of subscribed micro zones

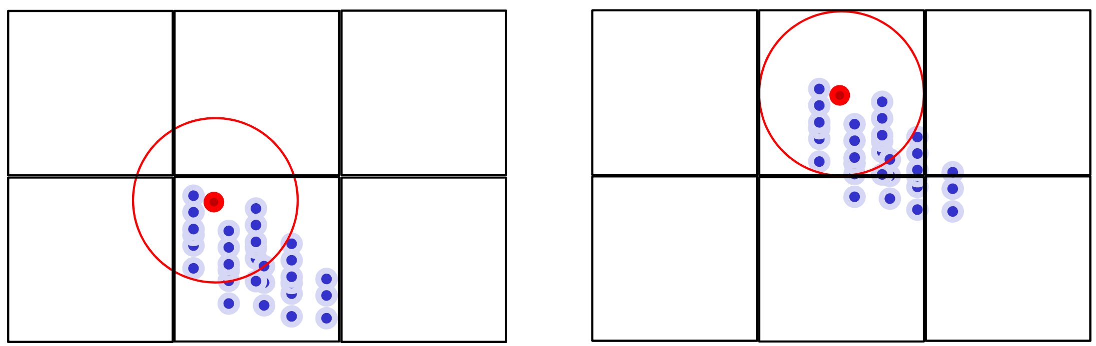
Advantages:

- objects close by can be determined efficiently
- changes can be passed on to subscribers (no regular queries necessary)

Micro Zoning and Spatial Publish-Subscribe

Disadvantages:

- Even Micro Zones can be overcrowded
  => the smaller the area, the more likely it is
- Overhead for changing zones increases if they are too small
  => the smaller the zone, the more frequent a change
- Location of Zone borders may lead to extreme fluctuations of observed objects.
- High rates of change extremely increase Overhead.
  => Many subscribe- and unsubscribe-operations inhibit the system

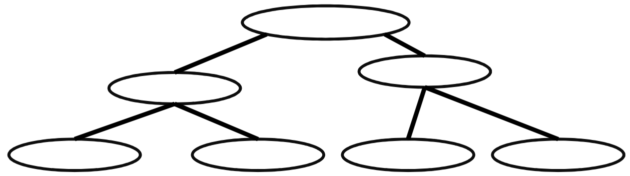


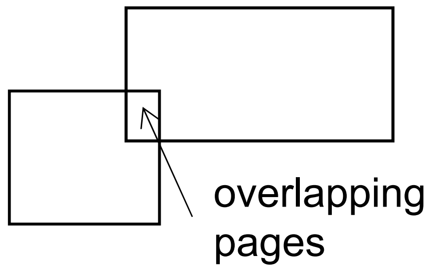
Classic Index Structures

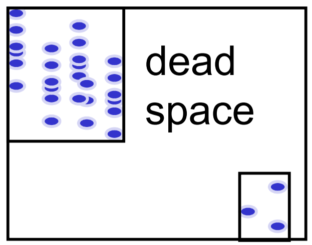
- Managing spatial objects can also be done via spatial search trees
- Search trees tailor their region pages (zones) to data distribution
  - ⇒ One maximally filled region pages/zone is guaranteed
  - ⇒ Reducing the number of objects in question increases search performance
  - ⇒ Adjusting the Search Tree causes calculation effort
- Adaption via recursive partition of space (Quad-Tree, BSP-Trees)
- Adaption via distribution of data to minimal surrounding page regions

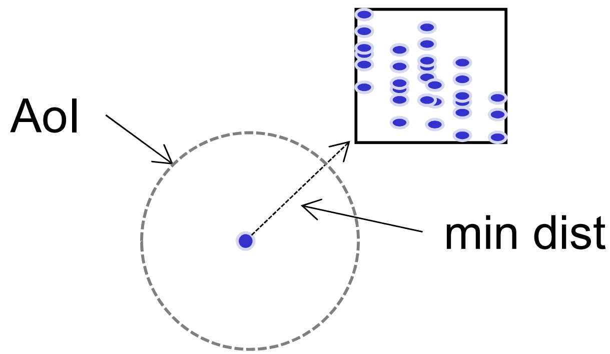
Important Features of Search Trees

- *region page*: surrounding approximation of several objects
- *balancing*: addressing different path lengths, from root to leaf notes, of branches
- *page capacity*: minimum and maximum number of objects within a region page
- *overlap*: intersecting regions between pages
- *dead space*: space without region pages/objects
- *pruning*: exclusion of all objects within one region page via testing for region pages









Requirements for an MMO Server

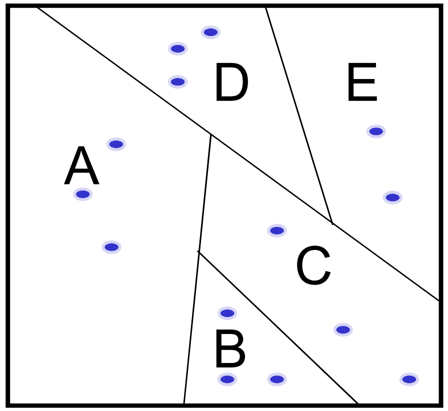
- generally the whole tree is stored within main memory
- high volatility, i.e. every change of a game entity's position
  - dependent on the game, up to one change per tick per entity
  - trees might degenerate in their structure/costly balancing required
- many queries per time unit
- support for multiple queries during one tick
- objects have either 2 or 3 dimensions
- objects have volume (spatial extension, hitbox, …)

conclusions:

- data structures optimizing pages accesses are ill suited (Tree is stored in main memory)
- **runtime increase ate query processing must compensate for the time for index creation/update**

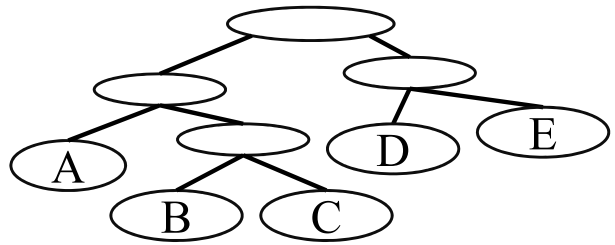
Binary Space Partitioning Trees (BSP-Tree)

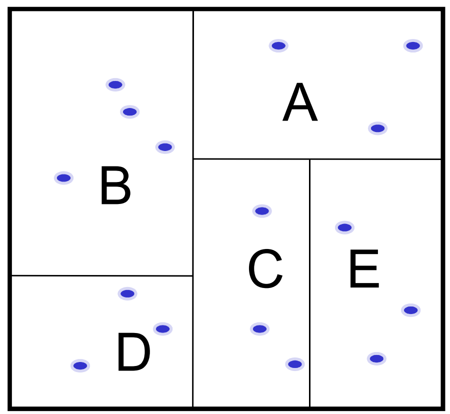
- root contains the whole data space
- every inner node has two successors
- data objects are stored in leaf nodes



most popular type: ***kD-Tree***

- max. page capacity are *M* entries
- min. page capacity are *M/2* entries
- at overflow => splitting w.r.t. an axis
- axis for the split changes after every split
- data is distributed 50%-50%
- at deletion: merge sibling nodes

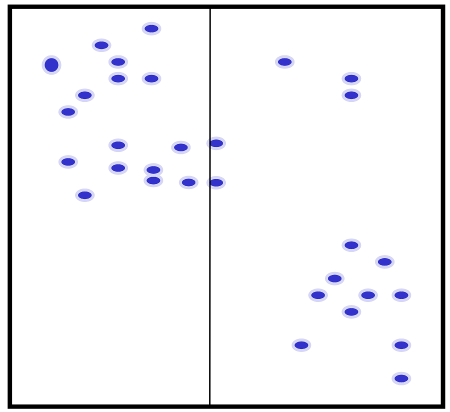




Binary Space Partitioning Trees (BSP-Tree)

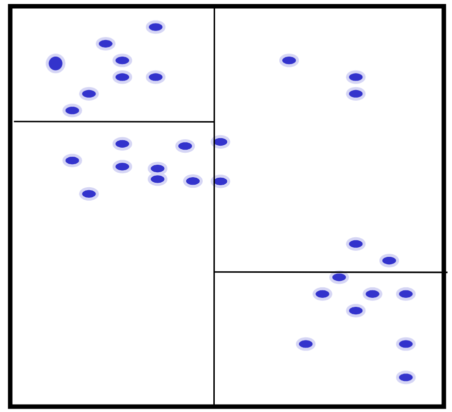
Problem with dynamic behavior:

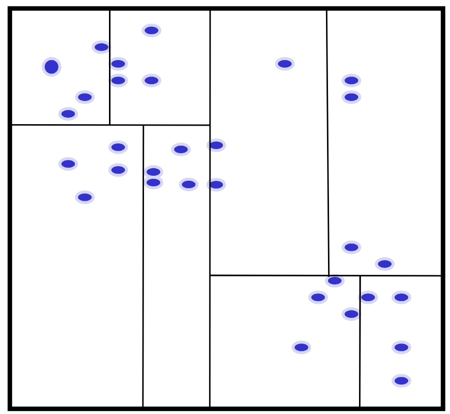
- no balancing (tree might degenerate)
- rebalancing is possible but very expensive
  => high update complexity



Bulk-Load

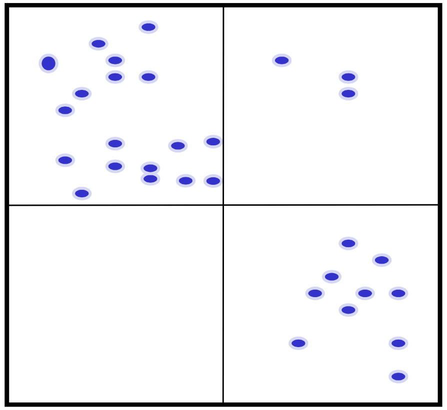
- assumption: all data objects are known
- creation: recursively distributing objects with a 50/50 split until every leaf contains less than M objects
- bulk-load always creates a balanced tree
- a data page of a tree of size *h* containing *n* objects contains at least $\left\lfloor \frac{n}{2^h} \right\rfloor$ objects and at most $\left\lfloor \frac{n}{2^h} \right\rfloor + 1$ objects

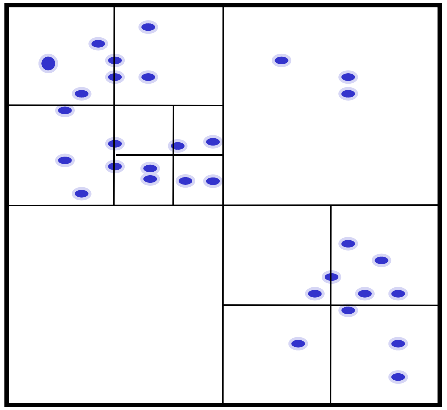


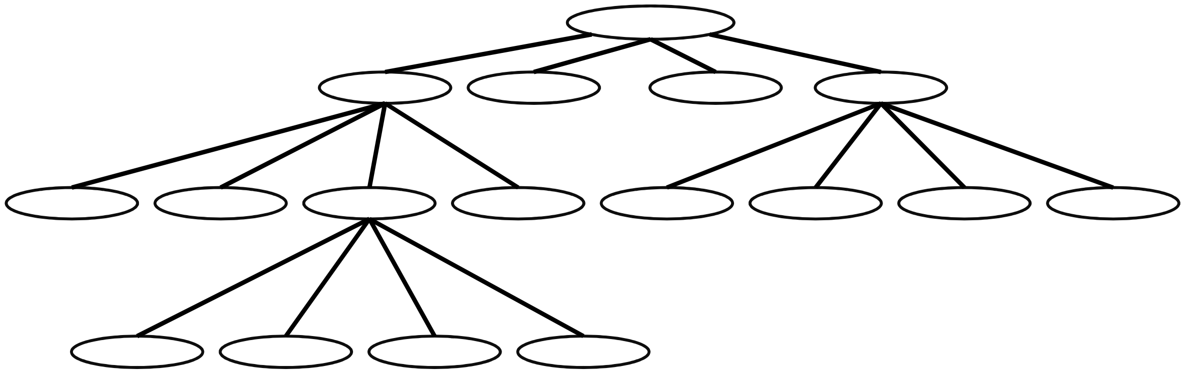


Quad-Tree

- root represents the whole data space
- every inner node has four successors
- sibling nodes split their parents space in four equal parts
- as a rule Quad-Trees are not balanced
- pages have a maximum filling ratio M, but no minimum
- leaves contain data objects





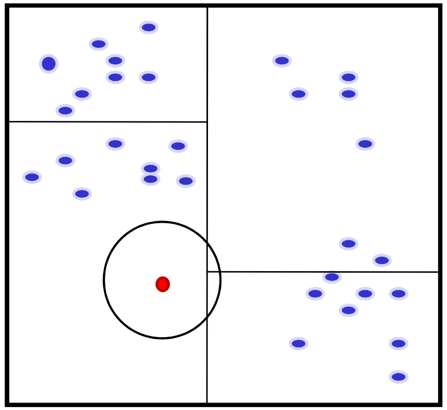


Data Partitioning Index Structures

Space partitioning procedures:

- partitioning the data space via dimensional splits
- page regions include dead space

  => potentially bad search performance for spatial queries

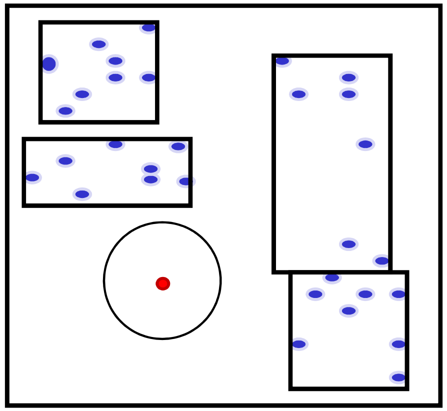


range query on BSP-Tree

Data partitioning procedures:

- page regions are defined by their minimum bounding Region (e.g. rectangles)

  => better pruning performance
- page regions may overlap
  => degeneration w.r.t. overlap
- split- and insert-algorithms minimize:
  - overlap between page regions
  - dead space within pages
  - balancing w.r.t. filling degree



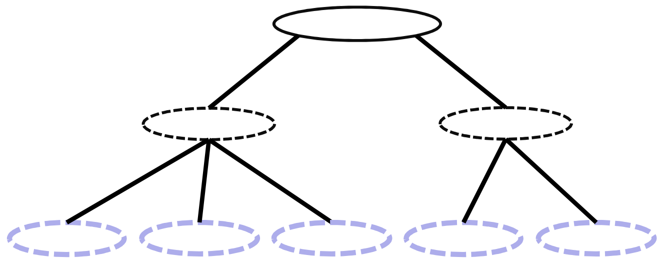
range query on R-Tree

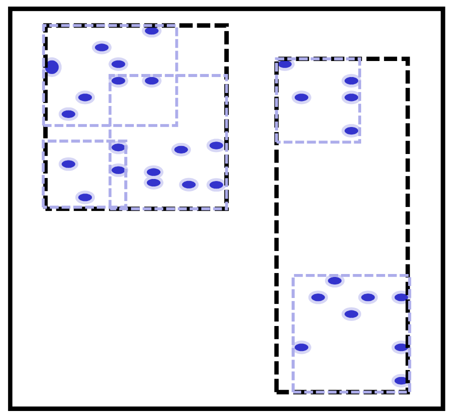
R-Tree

R-Tree structure:

- root encompasses the complete data space and contains a maximum of M entries
- page regions are modeled by minimal bounding rectangles (MBR)
- inner nodes have between *m* and *M* successors (where m ≤ M/2)
- the MBR of an successor node is completely contained within the predecessor's MBR
- all leafs are at the same height
- leafs contain data objects

  possible date objects:
  - points
  - rectangles





Inserting into an R-Tree

Object x is to be inserted into an R-Tree

Due to overlap, there are three possible cases

- Case 1: x is contained the directory rectangle D
  ⇒ Insert x into subtree of D
- Case 2: x is contained in several directory-rectangles $D_1, \dots, D_n$
  ⇒ Insert x into subtree D_i with the smallest area
- Case 3: x is not contained in any directory-rectangle D
  ⇒ Insert x into subtree D which suffers the smallest area increase to contain x (in doubt, choose the one with the smaller area)
  ⇒ extend *D accordingly*

Split-Algorithm within a R-Tree

(for the following we consider the case of inner nodes: objects are MBRs)

node K has an overflow $|K| = M+1$

$\Rightarrow$ divide K into two nodes K_1 and K_2, so that $|K_1| \geq m$ and $|K_2| \geq m$

square algorithm

- choose the pair of rectangles (R_1,R_2) with the largest "dead space" within the MBR, in case both R_1 and R_2 fall into Node K_i

  $d(R1, R2) := area(MBR(R1 \cup R2)) - area(R1) - area(R2)$

- Set $K_1 := \{R_1\}$ and $K_2 := \{R_2\}$
- repeat the following until STOP:
  - all R_i are assigned: STOP
  - if all remaining R_i are necessary to minimally fill the smaller node: assign them all and STOP
  - else, choose the next R_i and allocate it to the node whose MBR will experience the smallest area increase. In doubt, prefer the K_i with the smaller MBR area or rather with fewer entries.

Faster Split Strategy for R-Tree (1)

Linear Algorithm

The linear algorithm is identic to the square algorithm with the exception of choosing the initial pair (R_1,R_2).

Choosing the pair (R_1,R_2) with the *"greatest distance"*, or more precise:

- Identify the rectangle with the lowest maximum value and the rectangle with the largest minimum value, for every dimension (*maximum distance*).
- Normalize the maximum distance in every dimension by dividing it by the sum of the expansions of all R_i in this dimension (*setting the maximum distance in relation to their extension*).
- Choose the pair of rectangles with the greatest normalized distance in all dimensions. Set $K_1 := \{R_1\}$ and $K_2 := \{R_2\}$.

- This algorithm has linear complexity concerning the number of rectangles (*2m+1)* and the number of dimensions d.

Split algorithm within a R*-Tree

Idea for the R*-Tree split algorithm

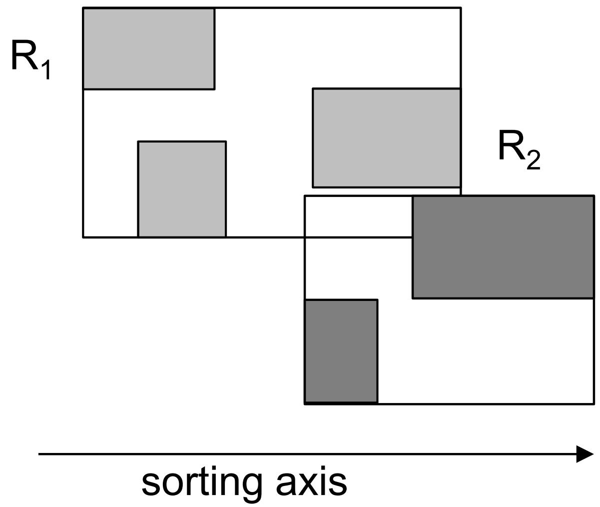
- Sort the rectangles in each dimension by their two corner points and only look at subsets of adjacent rectangles in this system
- Time complexity is O(*d · M · log M) for d* dimensions and *M rectangles*

Determining the Split Dimension

- For every dimension sort the rectangles according to both extreme values (lower and upper bound)
- For every axis:
  - Sort the entries by the lower and then by the upper vales of their rectangles and determine M-2m+2 distributions of the M+1 rectangles, such that the first group contains m-1+j rectangles and the second group contains the remaining rectangles
  - Compute S, the sum of all margin-values of the different distributions

⇒ Choose the dimension with the smallest S as split dimension.

Split algorithm within a R*-Baum



2. Partitioning
(M=4, m=2)

UG =
perimeter R_1+
perimeter R_2

Determining distribution

- Given the split dimension, R_1 *and* R_2 are selected to minimize overlap.
- In doubt, the distribution of R_1 *and* R_2 with the smallest coverage of *dead space* is chosen.

⇒ Best results were empirically determined for $m = 0{,}4 \cdot M$.

Bulk-Loads within R-Space

- **Advantage**:
  - faster creation
  - structure usually allows for faster query processing
- **Criteria for optimization**:
  - greatest possible filling ratio of both sides (low height)
  - little overlap
  - small dead space

Sort-Tile-Recursive:

- Assembling the R-Tree bottom-up
- No overlap for point objects at leaf level
- Time complexity: $O(n \log(n))$

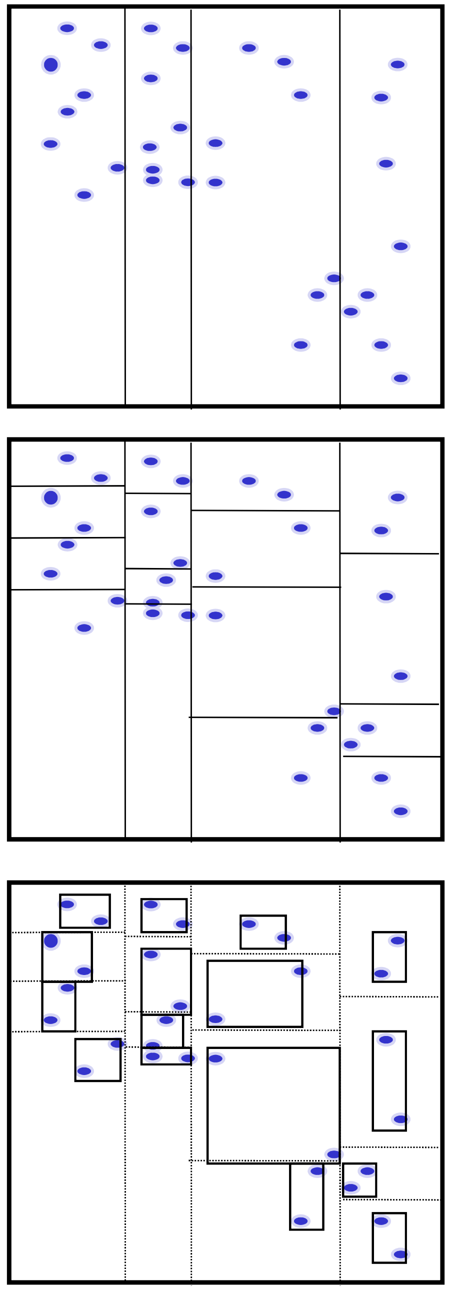
Sort-Tile Recursive

Algorithm:

1. Set DB to the set of objects P with |P| = n
2. Calculate the quantile: $q = \left\lceil \sqrt{\frac{n}{M}} \right\rceil$
3. Sort data elements in dimension 1
4. Generate quantile after $q \cdot M$ objects in dimension 1
5. Sort objects of every quantile into dimension 2
6. Generate quantile after M objects in dimension 2
7. Create a MBR around the points within each cell
8. Restart the algorithm with the set of derived MBRs or stop in case of q < *2*
   (all remaining MBRs fall into the root)

Note:

1. MBRs without overlap are created for points
2. For rectangles overlap may occur
3. For rectangles, calculation of the quantile via minimum values, maximum values or complex heuristics is possible
4. If the number of objects is not sufficient to completely fill all pages, only the last node is not maximally filled.



Deletions in R-Trees

Object x needs to be deleted from the R-Tree.

Delete:

- Test page S for underflow after deleting x: $|S| < m$
- If there is no underflow, delete x and STOP
- If there is an underflow, determine which predecessor nodes would have an underflow in case of deletion
- For every node with an underflow:
  - Delete the under flowed page from its predecessor node.
  - Insert the remaining elements of the page into the R-Tree.
  - In case of the root containing a single child, the child becomes the new root (height is reduced).

Note:

- deletion is not limited to one path with this algorithm
- makes the insertion of a subtree on layer 1 into the R-Tree necessary
- very expensive in worst case

Search Algorithms for Trees

Range Query:

```
FUNCTION List RQ(q,ε):
List  C // list of candidates (MBRs/Objects)
List  Result // list of all objects within ε-range of q
C.insert(root)
WHILE(not C.isEmpty())
    E := C.removeFirstElement()
    IF E.isMBR()
       FOREACH F ∈ E.children()
         IF minDist(F,q) < ε
           C.insert(F)
    ELSE
       Result.insert(E)
RETURN Result
```

Note: BOX and intersection queries follow the same principle.

Nearest Neighbor Queries

NN-query: Top-Down Best-First-Search

```
FUNCTION Object NNQuery(q):
  PriorityQueue  Q // objects/pages to investigate,
  sorted by mindist
  Q.insert(0, root)
  WHILE(not Q.isEmpty())
     E := Q.removeFirstElement()
     IF E.isMBR()
       FOREACH F ∈ E.children()
            Q.insert(mindist(F,q), F)
     ELSE
       RETURN E
```

Notes:

- mindist(R,P) is minimal distance between two points in R and P. If R and P are points, mindist = dist
- PriorityQueues are usually implemented via heap-structures (cf.heapsort)

Spatial Joins

Idea: defining join request by spatial attributes

Advantage: parallel processing of several requests during one pass.

Example: ε -Range-Join

Let G and S be sets of spatial objects with $G,S \subseteq D$, *$dist: D \times D \rightarrow \mathbb{R}$ as distance function and $\varepsilon \in \mathbb{R}$*.

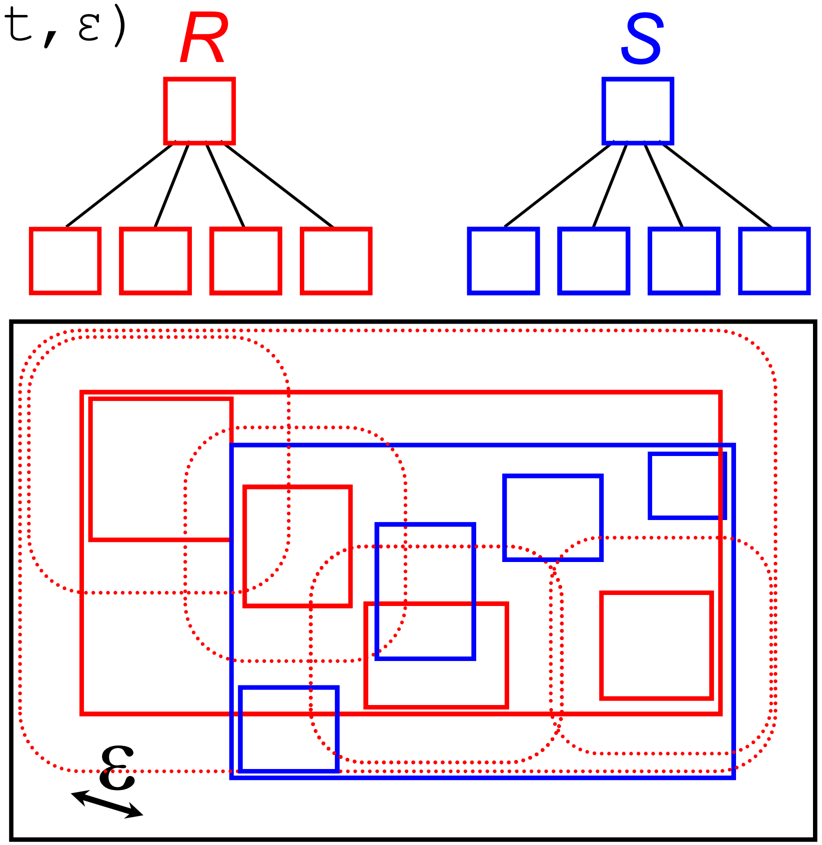
$S \bowtie_{dist(s,r)<\varepsilon} G = \{(g,s) \in G \times S \mid dist(g,s) < \varepsilon\}$ is called ε-Range-Join of G and S.

Use: Determine AoI for all player entities in one tick.

R-tree Spatial Join (RSJ)

Algorithm:

```
FUNCTION rTreeSimJoin (R, S, result, ε)
  IF R.isDirectoryPage() or S. isDirectoryPage()
   FOREACH r ∈ R.children()
      FOREACH s ∈ S.children()
         IF minDist(r,s) ≤ ε
           rTreeSimJoin(r,s,result,ε)
 //assume R,S are both DataPoints
  ELSE
    FOREACH p ∈ R.points
      FOREACH q ∈ S.points
       IF dist(p,q) ≤ ε
         result.insertPair(p,q)
RETURN  result
```



Note: Algorithms expects trees of the same height!
(can be extended to more general cases)

Problems of Data Volatility

Problems caused by spatial movement of all objects:

In games the majority of objects move several times per second.

- Changing position by deleting and inserting
  - dynamic changes may negatively influence data structures (miss-balance, more overlap, overfilling a micro-zone)
  - changes cause big overhead (search for object, follow up inserts, underflow- and overflow-handling)
- Changing position via dedicated operations
  - expansion of page regions: page overlap may extremely increase *(only possible in cases of data partitioning)*
  - moving objects between page regions:
    - *might have a negative instance to tree balance*
    - *overflow or underflow possible*

Conclusion: dynamic calculation either has a huge computational overhead or might degenerate data structures.

Throw-Away Indices

Idea:

- For highly volatile data changing existing data structures is more expensive than rebuilding with bulk load.
- Similar to the game state, use 2 index structures:
  - Index I_1 represents positions of the last consistent tick and is used for query processing
  - Index I_2 is created simultaneously:
    - Created via Bulk-Load: little concurrency, but fast creation, good structure
    - Dynamic creation: higher calculation effort and possibility of worse structure, but potential creation for every new position
  - At the start of the new tick, I_2 is used for query processing, I_1 is deleted and subsequently build on the new positions.

Conclusion: Use a tree if time for tree creation and query processing on the tree is faster than brute force query processing.

Game Design

Spatial problems are very dependent on Game-Design:

- Number and distribution of spatial objects
- Number and distribution of players
- Environmental model, fields, 2D or 3D
  (3D Environment does not necessitate 3D-Indexing)
- Movement type and speed of objects

What you should know by now..

- Game state and game entities
- Actions and time modelling
- Game loop and synchronization with other sub-systems
- Exemplary processing steps of an iteration
- Connection to scripting-engine, physics engine and spatial management
- Zoning, Sharding and Instantiation
- Micro-Zoning and Spatial-Publish-Subscribe
- BSP-Tree, KD-Tree, Quad-Tree and R-Tree
- Insert, Delete, Bulk-Load
- Query Processing: Range-Query, *NN*-Query and Range-Join
- Problems of highly volatile data

Literature and Material

- Shun-Yun Hu, Kuan-Ta Chen
  VSO: Self-Organizing Spatial Publish Subscribe
  In 5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2011, Ann Arbor, MI, USA, 2011.

- Jens Dittrich, Lukas Blunschi, Marcos Antonio Vaz Salles
  Indexing Moving Objects Using Short-Lived Throwaway Indexes
  In Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases, 2009.

- Hanan Samet. 2005. **Foundations of Multidimensional and Metric Data Structures** *(The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.