Lecture Notes

Managing and Mining Multiplayer Online Games

Summer Term 2019

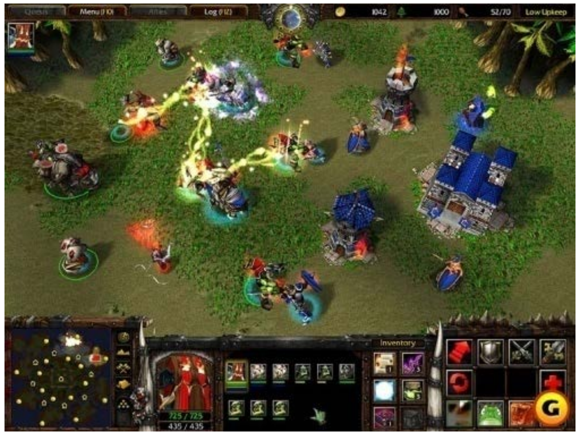
Chapter 2: The Game Core

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

Chapter Overview

- Modelling the state of a game (Game State)
- Modelling time (turn- and tick-system)
- Handling actions
- Interaction with other game components
- Spatial management and distributing the game state

Internal Representation of games


user view

ID	Type	PosX	PosY	Health	…
412	Knight	1023	2142	98	…
232	Soldier	1139	2035	20	…
245	Cleric	1200	2100	40	…
…					

Game State

Good Design: strict separation of data and display (Model-View-Controller Pattern)

- MMO-Server: Managing the game state / no visualization necessary
- MMO-Client: only parts of the game state but need for I/O and visualization. supports the implementation of various clients for the same game

Game State

All data representing the current state of the game

- object, attribute, relationship, …
  (compare ER or UML models)
- models all changing information
- lists all game entities
- contains all attributes of game entities
- information concerning the whole game

not necessarily in the Game State:

- static information
- environmental models/maps
- preset attributes of game entities

Game Entities

Game entities = objects in the game

examples for game entities:

- units in a RTS-Game
- squares or figures in a board game
- characters in a RPG
- items
- environmental objects (chests, doors, ...)

Attributes and Relationships

Properties of a game entity that are relevant for the rules equal attributes and relationships.

examples:

- current HP (max. HP only if variable)
- level of a unit in a RTSG
- enviromental objects: open or closed doors
- relationships:
    - character A has item X in her inventory (1:n)
    - A and B are part of the same team (n:m)
    - A is fighting C (n:m)
    - A has weapon W in his right hand (1:1)

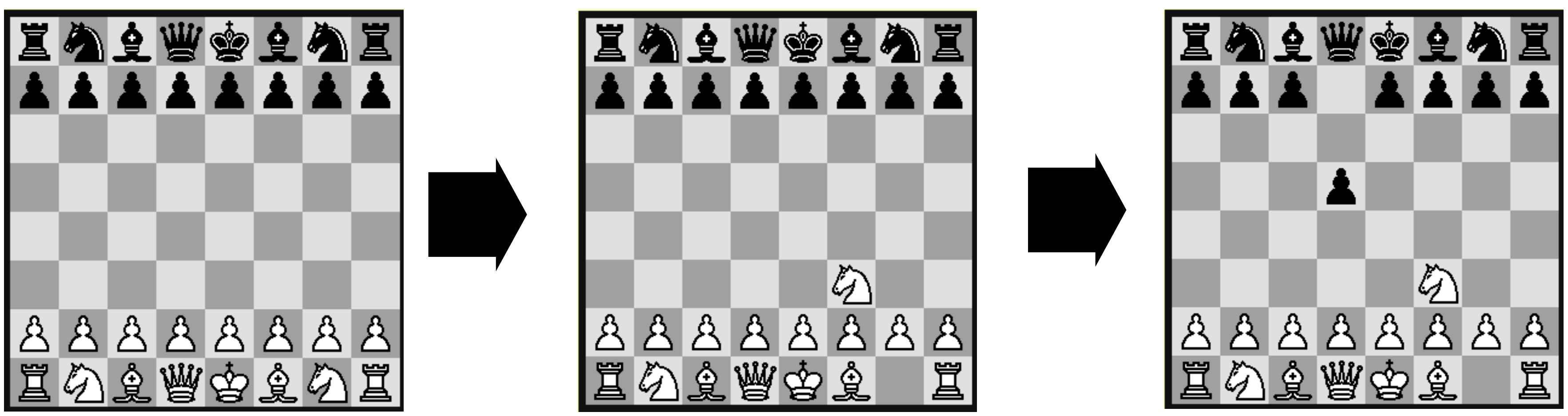
Information concerning the whole game

- every piece of information about the game that is not accessable via entities
- ingame time of the day (morning, noon, etc.)
- the map for the current game
- player field of view in an RTS
  (in case there is no abstract entity for a player)
- server type of an MMORPG (PVP/PVE/RP)
- …

Important:

Information can be modelled as game state attributes or separate entities.

Example: Chess



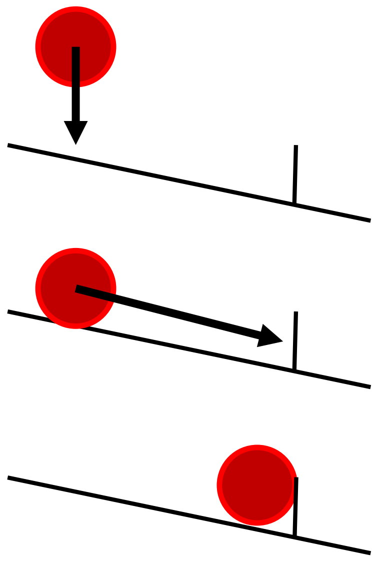
- game information:
  - players and assigned colors (black or white)
  - game mode: with chess clock or without
- game state:
  - positions of all figures / occupation of fields
    (entities encompass either figures or fields)
  - player who is next
  - time left for both players (dependant on game mode)

Actions

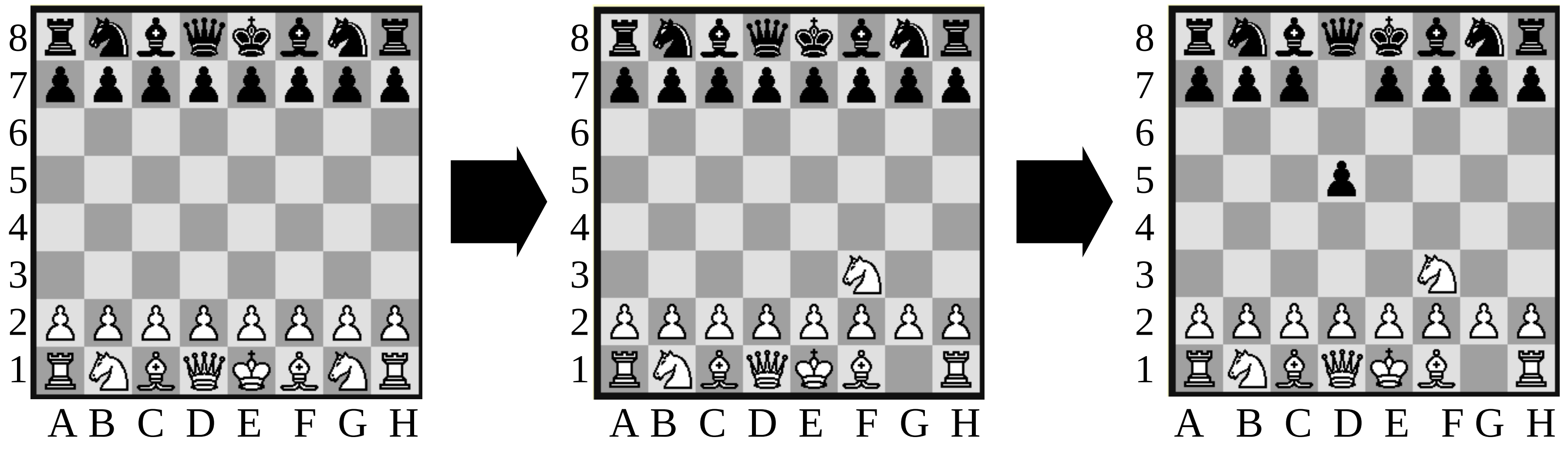
- actions transfer a valid game state into a new game state
- actions implement the rules of the games
- game core organizes their creation via:
  - players (user input)
  - control of NPCs (AI controlled)
  - environmental model

Example:

- a ball is placed on a slope
- environmental model decides that the ball will roll assisted by the physics engine
- actions changing the position and state of the ball (acceleration) are triggered



Example: Chess



- actions change figure positions/allocation of fields
  - knight from G1 to F3
  - pawn from D7 to D5
- actions enforce the game rules:
  - black pawn is allowed 2 squares ahead if it has not been moved yet
  - knight is allowed 2 squares ahead and one to the left (among other)

Actions and Time

- controls the point in time an action is performed
- game time (processed actions/time unit) to realtime (wall-clock time) ratio
- synchronization with other game components:
  - rendering (graphics/sound)
  - handling user input
  - AI calls for NPCs
  - ...
- handling actions which cannot be processed yet:
  - deleting (two moves in a row by one player in chess)
  - delaying (processing an action as soon as it is valid)

=> solutions strongly depend on the game

Time Models for Action Processing

Turn-based Games: Action type and sequence are predetermined and are managed by the game core.

- game core calls action creators in a fixed order
- implemented by loops, state machine, ...
- no concurrency possible

examples:

- Chess
- Civilization
- Settlers of Catan
- Turn-based RPGs

Disadvantages:

- player attendance is needed
- gameplay may allow for simultaneous player actions (reduced waiting time)

Time Models for Action Processing

Real-time/transaction system

- the game does not control action creation times
- players are able to take asynchronous actions
- NPC/environmental model can act in independent threads
- concurrency can be implemented similar to transaction systems (block, reset, …)

examples:

- certain browser games

Time Models for Action Processing

advantages:

- can be based on standard solutions (e.g. DBS)
- allows concurrency:
  - reduces waiting time for other players (game might demand waits)
  - system distribution is straight forward

disadvantages:

- no synchronization between game and real time => game time (actions per minute) might stall
- no control of max. amount of actions per time unit
- simultaneous actions are impossible (serialization)

Time Models for Action Processing

Tick-System (Soft-Real-Time Simulation)

- actions are only processed at fixed ticks.
- actions can be created at any moment.
- one tick has a minimum length (e.g., 1/24 s).
  => real time and game time are synchronized
- all actions in one tick are treated as simulatneous
  (no serialization)
- the next game state is created by a cumulated processing of all actions (no isolation)
- model used in rendering because it requires fixed framerates and concurrent changes

Time Models for Action Processing

Advantages:

- synchronizes game-time and real-time
- fair rules for actions per time-unit
- concurrency

Disadvantages:

- *handling lags*
  (server does not finish computing a tick in time)
- *conflict resolution* for concurrent and contradictory actions
- *chronological order*
  (all actions generated within one tick are considered concurrent)

Time Models for Action Processing

other important aspects of the tick-system:

- several factors influence computing time required for one tick:
  - hardware
  - game state size
  - number of actions
  - complexity of actions
  - synchronization and handling subsystem tasks

  (for example, saving the game state to the persistence layer)

Actions vs.Transactions

moves/actions are very similar to DBS transactions

- atomicity: move/action will be executed as a whole or not carried out
  example: player A makes a move, chess clock for player A stops,
  chess clock for player B resumes
- consistency: a valid game state is transferred to another valid game state
- durability: the game state has fixed transaction results and they are (at least partially) sent to the persistence layer.

furthermore:
Actions have to be consistent to rules of the game.
(maintaining integrity)

- static: game state is rule-consistent
- dynamic: actions is rule-consistent

Actions vs. Transactions

temporal aspects of action processing are important

- action handling should be as fair as possible
  - a player's actions should not be delayed
  - every player has an amount of possible actions per time unit
- concurrent actions should be theoretically possible (soft-realtime simulation)
- a time limit for processing is necessary for smooth game play
- possible elimination of actions when the time limit is exceeded
- synchronizing game time and real time should be possible

Actions vs.Transactions

no obligatory single user operation (Isolation)

- concurrent actions must be computed interdependently (not serializable)
- *example:*
    - Character A has 100/100 HP (=Hit Points)
    - At t_j, A suffers 100 HP damage from character B’s attack
    - Simultaneously at t_j, A is being healed for 100 HP by character C

outcome under isolation:

- healing first (overheal) followed by damage
  => A has 0 HP left and dies
- 100 HP damage first => A dies and can no longer be healed

result of concurrent actions:

- A suffers 100 HP damage and receives healing for 100 HP : the effects cancel each other

The Game Loop

- actions are applied to the game state in this continuous loop to ensure consistent transitions. (action handling)
- time model starts each iteration.
- other functions, that are dependant on the game loop

$$\left.\begin{array}{l}\bullet\ \text{handling user input(=> player actions)}\\ \bullet\ \text{calling NPC AIs (=> NPC-actions)}\\ \bullet\ \text{calling the environmental model}\end{array}\right\}\text{actions creation}$$

  - graphics and sound rendering
  - saving certain game aspects in the persistency layer
  - transmitting data to the network
  - update supporting data structures
    (spatial index structures, graphics-buffer, …)
  - …

Implementing a game loop

- one game loop for all tasks:
  - no overhead due to synchronization => efficient
  - poor abstraction of the architecture: a change in one aspect requires a change in the game core

- different game loops for each subsystem (e.g.: AI loop, network loop, rendering loop, …)
  - well layered architecture
  - subsystems can be turned off in a server-client architecture
    - client needs no dedicated NPC-control
    - server has no need of a rendering-loop
  - game loops must be synchronized

Communicating with the game loop

- game loop calls other modules
  - solution for systems that are in sync with or slower than the game loop
  - ill-suited for multithreading

  examples: persistence layer, network, sound rendering, …
- game loop messages subsystems
  - allows multithreading
  - call frequency is a multiple of game loop pace

  examples: NPC control, client synchronization, sound rendering, …
- synchronization via read only access to the game state
  - in fast paced systems, the sub-system needs its own loop
  - multithreading with comprehensive access to the game state
  - read date must be consistent (not yet changed)

  examples: graphics rendering, persistence-layer, …

Handling actions

- action handling: turns game actions (run, shoot, jump, ...) into changes to the game state
- game mechanics are implemented via action-handling
- valid actions follow calculation rules
- read operations on the game state
- write operations on the game state
- use of subsystems possible

  for example, spatial management module
  or physics engine

Consistency during action handling

- tick-system: concurrent actions are possible
- actions within a tick are independent of sequence

problem: reading already changed data

solution:

- shadow memory:
  - there are two game states G1 and G2
  - G1 holds the last consistent game state (active)
  - G2 is changed during current iteration (inactive)
  - on completion of the tick, G1 will be set to inactive and G2 will be set to active
- fixed sequence of read and write operations for actions
  - break down and rearrange the necessary action components
  - all actions are being handled simultaneously

Conflicts during Concurrency

- concurrency causes conflicts (e.g. simultaneously picking up a gold coin)
- **problem**: result of an action cannot be calculated in isolation (If A gets the coin, B cannot get the coin)
- **conflict resolution**:
  - deleting both actions (undo both)
    => conflict detection and possible reset of data
  - random pick of an action and deleting the other (random)
    => conflict detection and possible reset of data
  - first action is executed (natural order)
    => this solution is not necessary fair

input order is not necessarily equal to the execution order
but: division into ticks can already influence order of actions

Implementing Actions

How to implement actions?

- direct implementation using the programming language
  - **advantage**: high efficiency
  - **disavantages**:
    - redudant code for the same mechanics
    - inconsistencies are possible

- encapsulate parts of action processing in modules and subsystems :
  - Physics Engine (collision testing, acceleration, objects bouncing, ...)
  - Spatial Management Module (nearest neighbors, field of view, ...)
  - AI Engine (routing, swarm movement, ...)

Implementing actions

- Scripting Engine
  - offers a standardized implementation interface
  - encapsulates access to the game state (better consistency)
  - entities and their behavior are programed on the same basis
  - advantages: some designs require changing the scripting engine
  - **example**: LUA (http://http://lua-users.org/wiki/ClassesAndMethodsExample)

```
require("INC_Class.lua")
--=========================
 cAnimal=setclass("Animal")

function cAnimal.
     methods:init(action, cutename)
     self.superaction = action
     self.supercutename = cutename
end
```

```
--==========================
cTiger=setclass("Tiger", cAnimal)

function cTiger.methods:init(cutename)
         self:init
        super("HUNT (Tiger)", "Zoo Animal (Tiger)")
        self.action = "ROAR FOR ME!!"
        self.cutename = cutename
 end
```

Physics Engines

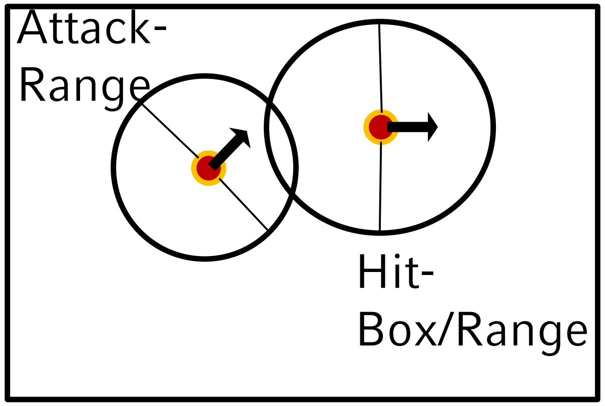
- implements solid state physics and classical mechanics
- game entity must provide all necessary parameter
  - spatial extension (polygon mesh, simplificatios: cylinders, MBRs)
  - movement vectors
  - mass
  - …
- uses differential equations
- realistic effects require large tick rates and detailed models
- large computational effort:
  - precomputation
  - numerical approximations

Physics Engines und MMO-Server

- majority of the results from a classical physics engine are only required for a realistic display
  e.g. particle filters, rag-doll animations,..
- joint computation of game state and graphic display often makes sense because they are based on the same effect
- use on the client side because display is available anyway
- on sever side mechanics might be too detailed to be computed for all game entities
- simplifications on the server side are often sufficient to implement game design
- use physics engines to determine parameters and approximations on the server side

Spatial Management in Game Servern

- majority of games takes place in a spatial environment (2D/3D maps, ...)
- action processing, NPC control and network layer requires spatial query processing:
  - Which other game entities are within interaction range? (AoI = Area of Interest)
  - supports collision detection (cmp. Physics Engine) and area intersections (prefiltering)
  - Which other game entity is closest?
  - Does a player enter the aggro range of an NPC?



Spatial Management for Game Servers

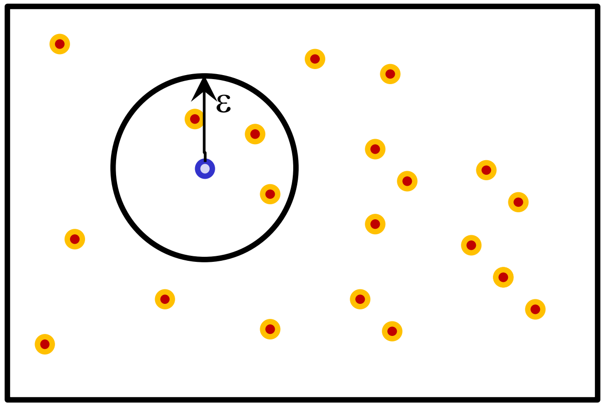
- for small game worlds with limited game entities
  => organize spatial position in a list
- process queries by sequential scans
- with high query frequencies and large numbers of moving objects query processing becomes expensive
  example: 1000 game entities in one zone, 24 ticks/s
  => naive AoI computation requires 24.000.00 distance computations per second
- **conclusion**: the cost for spatial query processing strongly increases with the size of the game state
  (number of potential interacting entities grows quadratic)

Spatial Queries (1)

Here spatial queries w.r.t Euclidian distance in IR^2

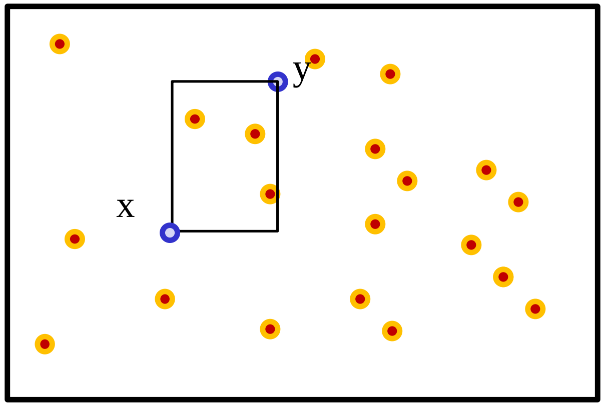
- ε-Range Queries

$$RQ(q,\varepsilon) = \left\{v \in GS \middle| \sqrt{(q_1 - v_1)^2 + (q_2 - v_2)^2} \leq \varepsilon\right\}$$



- Box-Query

$$BQ(q,\varepsilon) = \{v \in GS | x_1 \leq v_1 \leq y_1 \wedge x_2 \leq v_2 \leq y_2\}$$

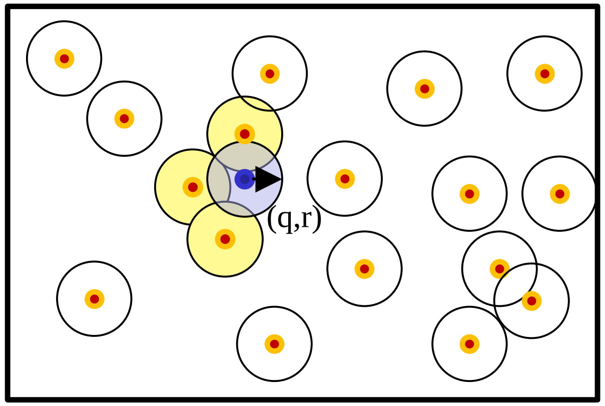


Spatial Queries (2)

- Intersection Query

$SIQ(q,r) =$

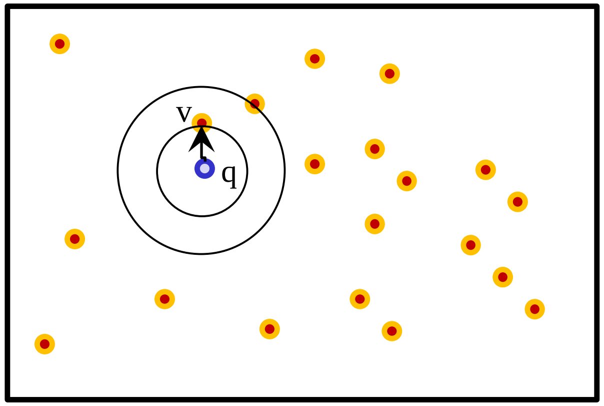
$$\left\{(v,s) \in GS \times \mathbb{R} \mid \sqrt{(q_1 - v_1)^2 + (q_2 - v_2)^2} \le r + s\right\}$$



- Nearest Neighbor Query

$NN(q) =$

$$\left\{v \in GS \mid \forall x \in GS\colon \sqrt{(q_1 - v_1)^2 + (q_2 - v_2)^2} \le \sqrt{(q_1 - x_1)^2 + (q_2 - x_2)^2}\right\}$$



Efficiency Tuning for Spatial Queries

- methods to reduce the number of considered objects (pruning)
  - distribute the game world (zoning, instancing, sharding, …)
  - index structures (BSP-Tree, KD-Tree, R-Tree, Ball-Tree)
- reduce the number of spatial queries
  - reduce query ticks
  - spatial publish subscribe
- efficient query processing
  - nearest-neighbor queries
  - ε-range Join (simulaneously compute all AoIs)

Sharding and Instantiation

- copying a region for a specific group
- any number of the same region exists
- instances and shards were primarily created for game design purposes
  (e.g., 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

problems:
- objects of bordering zones need to be considered
- uneven distribution of players

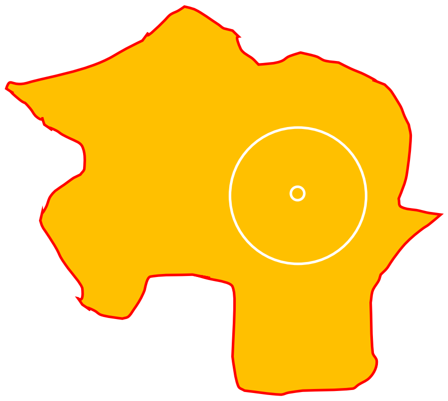
Netherstorm
Blade's Edge Mountains
Zangarmarsh
Hellfire Peninsula
Nagrand
Terokkar Forest
Shadowmoon Valley
Outland

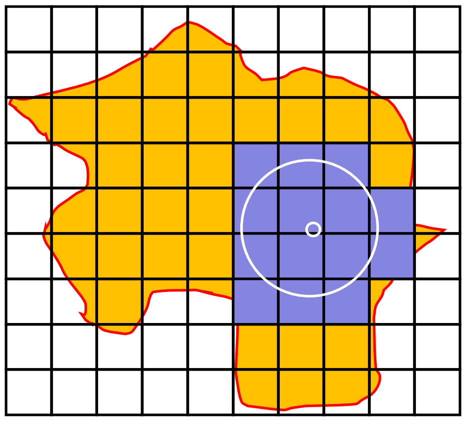
CLICK ANYWHERE ON THE MAP TO EXPLORE THE GALAXY IN 3D!

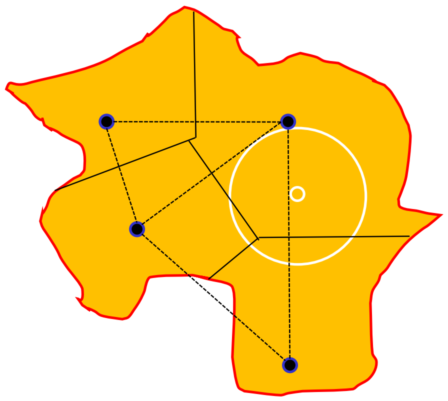
Stretching across more than 30,000 parsecs, the known galaxy is comprised of billions of star systems. The charting of these systems is of vital importance for both interplanetary travel and for military strategy. The galaxy map provided here is updated with the latest coordinates for heavily trafficked systems as well as other notable worlds of interest.

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

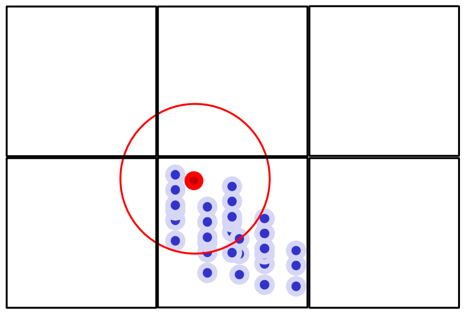
Advantages:

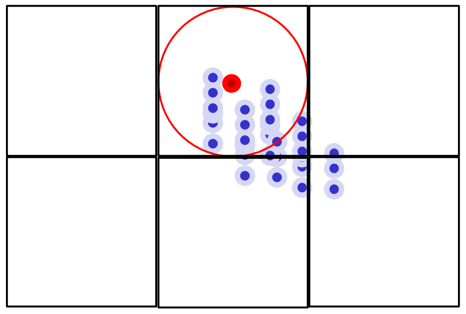
- objects close by can be determined efficiently
- changes can be passed on to subscribers
  (no regular queries necessary)

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



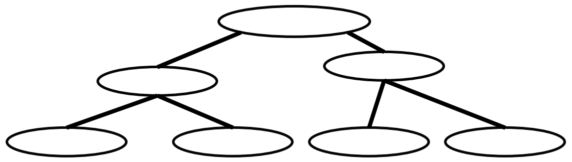


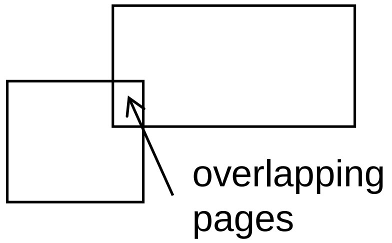
Classic Index Structures

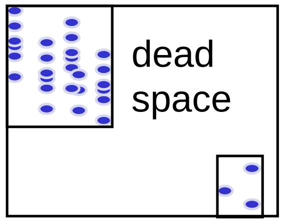
- managing spatial objects can also be done via spatial search trees
- search trees tailor their region pages (zones) to data distribution
    - ⇒ one maximally filled region pages/zone is guaranteed
    - ⇒ reducing the number of objects in question increases search performance
    - ⇒ adjusting the 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

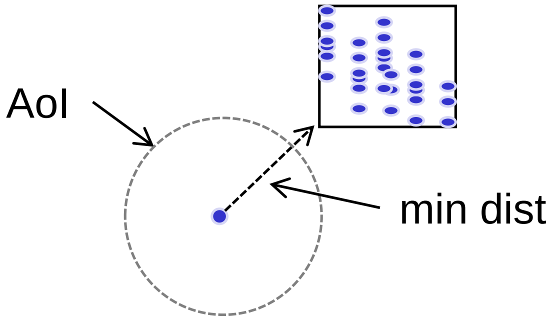
Important Features of Search Trees

- *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

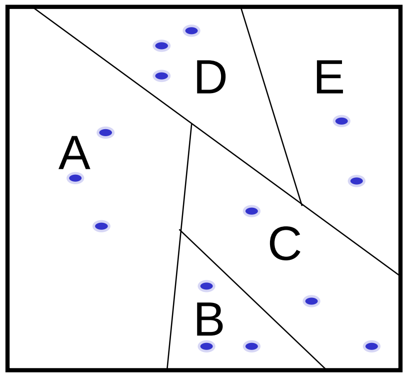
- generally the whole tree is stored within main memory
- high volatility, i.e. every change of a game entity's position
    - dependent on the game, up to one change per tick per entity
    - trees might degenerate in their structure/costly balancing required
- many queries per time unit
- support for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

conclusions:

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

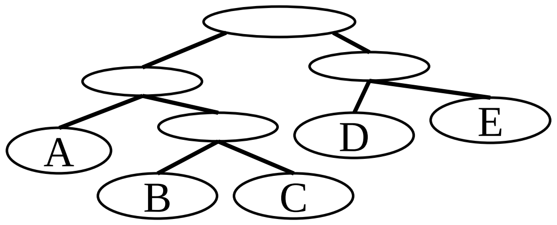
Binary Space Partitioning Trees (BSP-Tree)

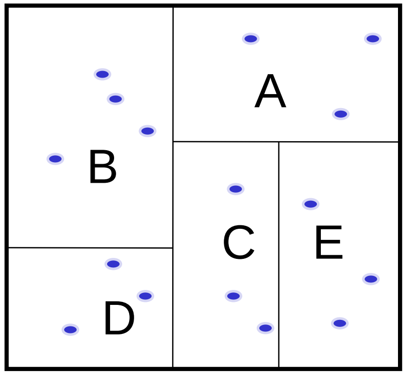
- root contains the whole data space
- every inner node has two successors
- data objects are stored in leaf nodes



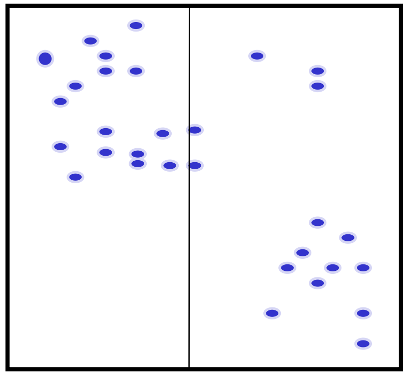
most popular type: ***kD-tree***

- 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



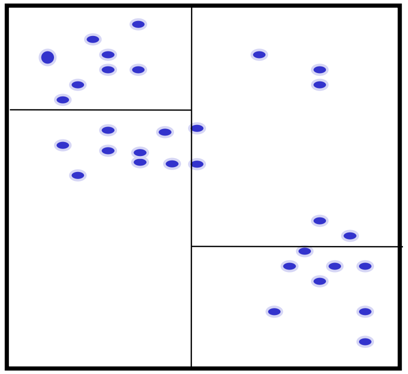


Binary Space Partitioning Trees (BSP-Tree)



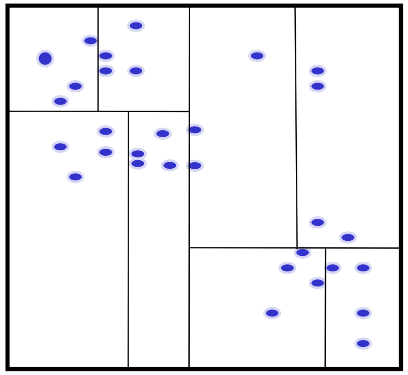
problem with dynamic behavior:

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



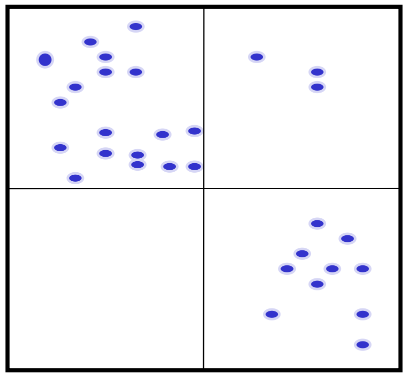
Bulk-Load

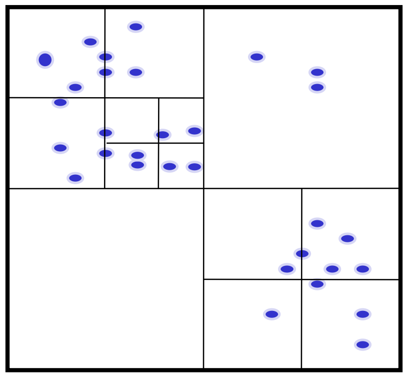
- **assumption**: all data objects are known
- **creation**: recursively distributing objects with a 50/50 split until every leaf contains less than M objects
- bulk-load always creates a balanced tree
- a data page of a tree of size *h* containing *n* objects contains at least $\left\lfloor \frac{n}{2^h}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

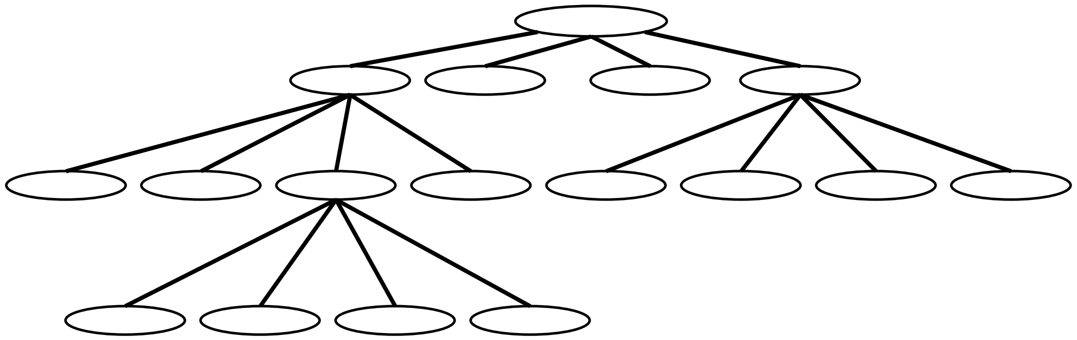


Quad-Tree

- 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





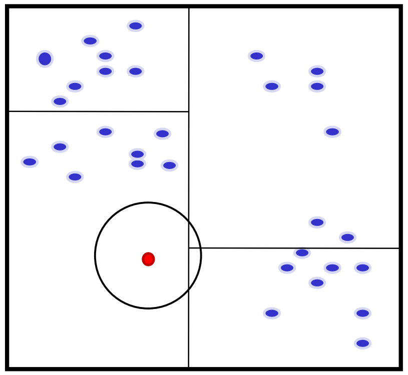


Data Partitioning Index Structures

space partitioning procedures:

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

  => potentially bad search performance for spatial queries



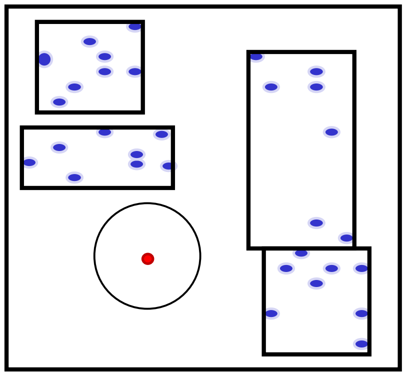
range query on BSP-Tree

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



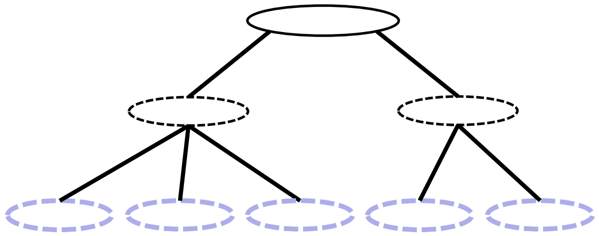
range query on R-Tree

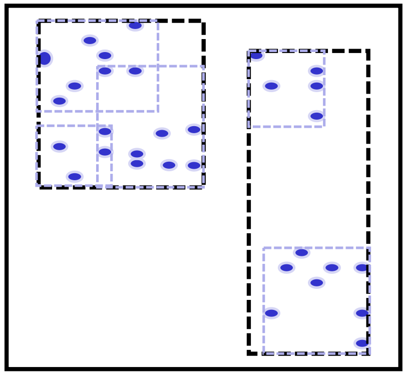
R-Tree

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 $|K_2| \geq m$

Basic algorithm in $O(n^2)$

- 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∪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 *O(n)*

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.

Bulk-Loads within R-T

- **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))*

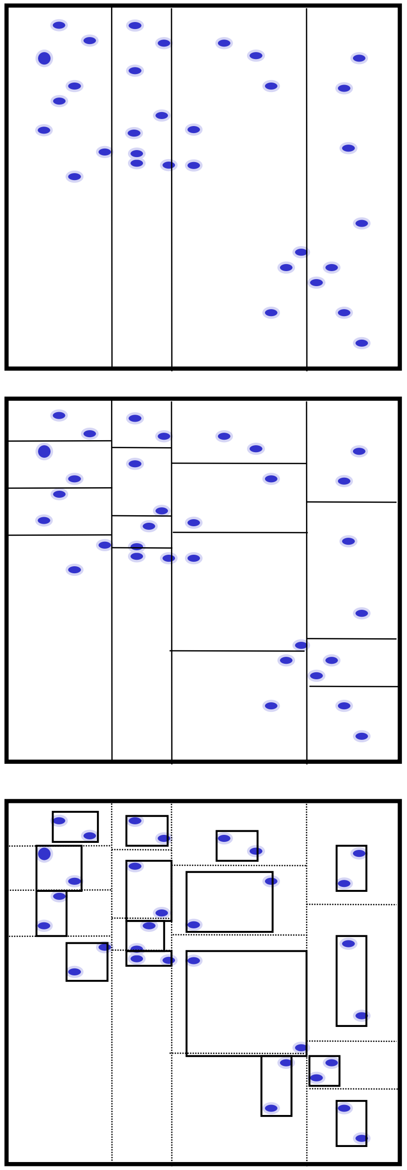
Sort-Tile Recursive

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·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($q$,$\varepsilon$):
List  C // list of candidates (MBRs/Objects)
List  Result // list of all objects within $\varepsilon$-range of q
C.insert(root)
WHILE(not C.isEmpty())
    E := C.removeFirstElement()
    IF E.isMBR()
        FOREACH F ∈ E.children()
          IF minDist(F,q) < $\varepsilon$
            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
- priority queues are usually implemented via heap-structures (cf.heapsort)

Spatial Joins

Idea: Define Joins on spatial predicates.

advantage: Parallel processing of multiple queries.

Example: *ε-Range-Join*

Let G and S sets of spatial objects G,S $\subseteq$ D, let *dist:D×D→IR be a distance function and $\varepsilon \in$ IR*. Then, the ε-range join between G and S is defined as:

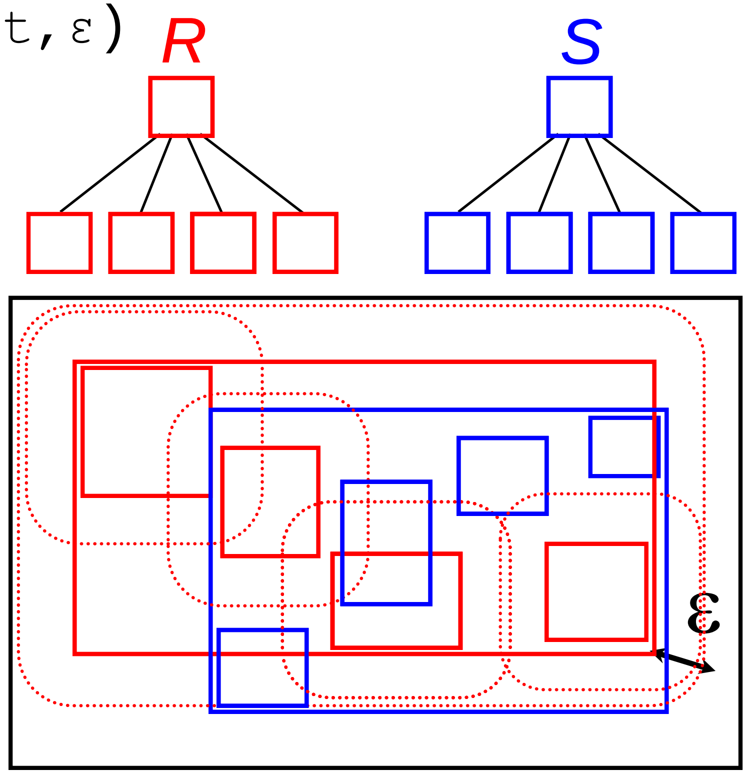
$$S \underset{dist(s,r)<\varepsilon}{\bowtie} G = \{(g,s) \in G \times S \mid dist(g,s) < \varepsilon\}$$

Remark: The objects within the area of interest (AoI) for each entity can be determined with a single ε-range-join.

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: Algorithm 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.