

Skript zur Vorlesung
Managing and Mining Multiplayer Online Games
im Sommersemester 2014

Kapitel 2: Der Game Core

Skript © 2012 Matthias Schubert

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

Kapitelübersicht

- Generelle Modellierung eines Spielzustandes (Game State)
- Zeit-Modellierung (Zug- und Tick-System)
- Aktionsverarbeitung
- Interaktion mit anderen Komponenten der Game Engine
- Räumliche Verwaltung und Aufteilung des Game States

Interne Darstellung von Spielen



Benutzersicht

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

Game State

Gutes Design: Strikte Trennung von Daten und Darstellung (Model-View-Controller Pattern)

- MMO-Server: Verwalten des Game State / keine Darstellung notwendig
- MMO-Client: Teile des Game States aber I/O und Darstellungskomponente
- Begünstigt die Implementierung unterschiedlicher Clients für dasselbe Spiel (unterschiedliche Graphikqualitäten)

3

Game State

Gesamtheit aller Daten, die den Spielzustand repräsentieren

- Modellierung mit ER-Modell oder UML möglich (Objekte, Attribute, Beziehungen, ...)
- Modell aller veränderlichen Informationen
- Liste aller Game Entities
- Attribute der Game Entities
- Informationen über das gesamte Spiel

Informationen, die nicht im Game State stehen müssen:

- Statische Informationen
- Umgebungsmodelle/Karten
- Standard-Attribute von Einheiten

4

Game Entitäten

Entsprechen den Objekten

Beispiele für Game Entities:

- Einheiten in einem RTS-Spiel
- Felder in einem Brettspiel
- Charaktere in einem RPG
- Gegenstände
- Umgebungsobjekte (Truhen, Türen, ...)

5

Attribute und Beziehungen

Regelrelevante Eigenschaften einer Game Entity

Entspricht Attributen und Beziehungen

Beispiele:

- Aktuelle HP (max. HP nur wenn veränderlich)
- Ausbaustufe von Einheiten in einem RTS
- Umgebungsobjekte: offene oder geschlossene Türen
- Beziehungen zwischen Objekten:
 - Charakter A hat Item X im Inventar (1:n)
 - A ist mit B in einer Gruppe (n:m)
 - A befindet sich im Kampf mit C (n:m)
 - A hält Waffe W in der rechten Hand (1:1)

6

Informationen über das gesamte Spiel

Alle Informationen über den Spielzustand, die nicht über Entitäten erfassbar sind

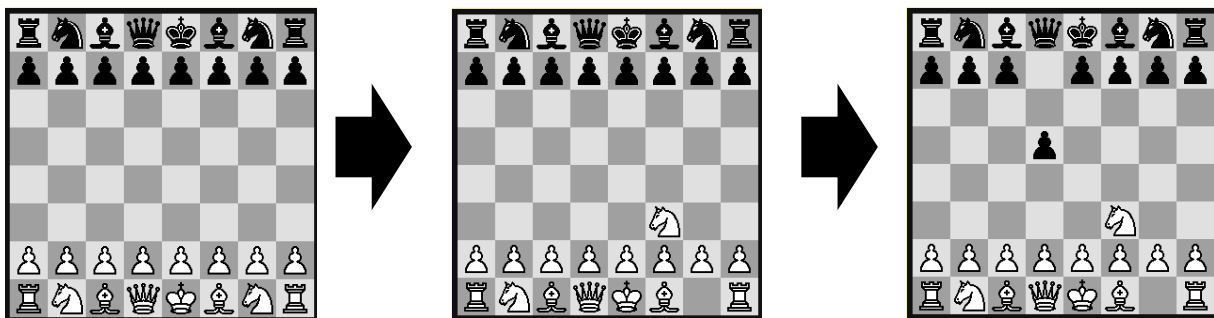
- Tageszeiten im Spiel
- Karte auf der gespielt wird
- Sichtbereich der Spieler in einem RTS Game (Falls für die Spieler kein abstraktes Entity erzeugt wird)
- Servertyp in einem MMORPG (PVP/PVE/RP)
- ...

Achtung:

Informationen können als Attribute des Game States oder auch als Entitäten modelliert werden.

7

Beispiel: Schach



- Information über das Spiel:
 - Spieler und Zuordnung der Seite (schwarz oder weiß)
 - Spielmodus: mit „Schachuhr“ oder ohne
- Game State:
 - Positionen aller Figuren/ Belegung aller Spielfelder (Entitäten sind hier entweder Figuren oder Felder)
 - Spieler, der gerade am Zug ist
 - verbleibende Zeit für beide Spieler (abhängig vom Spielmodus)

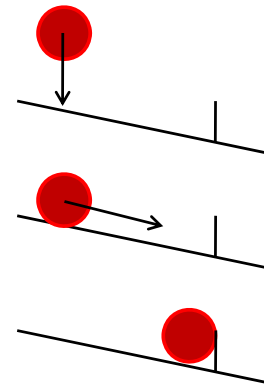
8

Aktionen

- Aktionen überführen einen gültigen Game State in einen anderen gültigen Game State
- Aktionen implementieren die Regeln des Systems
- Game Core organisiert die Entstehung durch:
 - Spieler (Benutzereingabe)
 - NPC-Steuerung (AI-Steuerung)
 - das Umgebungsmodell

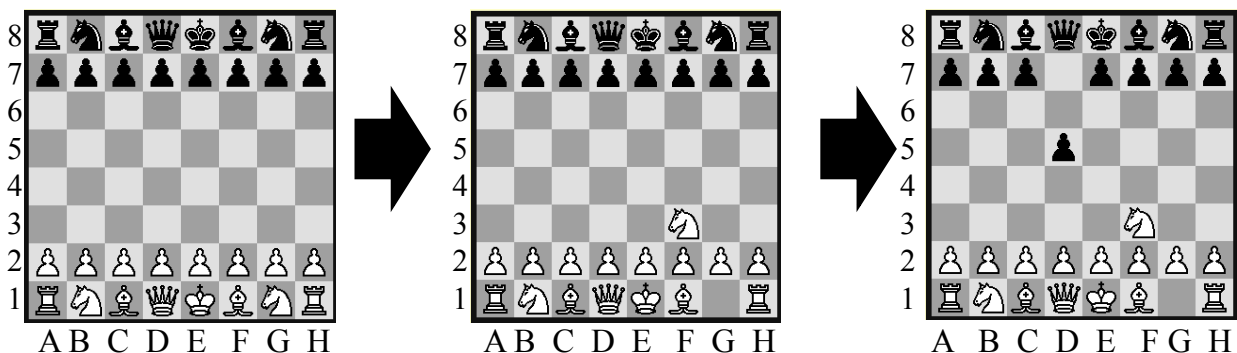
Beispiel:

- Ball wird auf eine Schräge gelegt
- Umgebungsmodell berechnet mit Hilfe der Physics Engine, dass Ball rollt
- Aktion, die sowohl die Position als auch den Zustand des Balls (Beschleunigung) ändert, wird ausgelöst



9

Beispiel: Schach



- Aktionen: ändern die Position der Figuren/Belegung der Felder
 - Pferd von G1 auf F3
 - Bauer von D7 auf D5
- Aktionen realisieren die Regeln:
 - schwarzer Bauer 2 Felder nach vorne falls er noch nicht bewegt wurde
 - Pferd darf 2 nach vorne und 1 nach links (unter anderem)

10

Zeitmodellierung der Aktionen

- Regelt den Ausführungszeitpunkt einer Aktion
- Verhältnis von Spielzeit (verarbeitete Aktionen/Zeiteinheit) zur Echtzeit (Wall-Clock Time)
- Synchronisation mit anderen Komponenten des Spiels:
 - Rendering (Grafik/Sound)
 - Abfrage von Benutzereingaben
 - Aufrufe der AI für NPCs
 - ...
- Umgang mit Aktionen, die noch nicht verarbeitet werden können
 - Löschen (2ter Zug in Folge von einem Spieler im Schach)
 - Verzögern (Ausführen der Aktion sobald sie erlaubt ist)
- Lösung hängt stark vom Spielprinzip ab

11

Aktionsverarbeitung bei Eventsteuerung

Rundenbasierte Spiele: Art und Reihenfolge der Aktionen ist festgelegt und wird vom Game Core verwaltet

- Game Core ruft Aktionserzeuger in fester Reihenfolge auf
- Realisierung über Schleifen, Zustandsautomaten, ...
- Keine Nebenläufigkeit möglich
- *Beispiele:*
 - Schach
 - Civilization
 - Siedler von Catan
 - Rundenbasierte RPGs

Nachteile:

- Anwesenheit der Spieler erforderlich
- Spielprinzip würde eventuell gleichzeitiges Ziehen mehrerer Spieler zulassen (weniger Wartezeit)

12

Aktionsverarbeitung bei Eventsteuerung

Realtime/Transaktionsystem

- Game verwendet keine feste Steuerung der Aktionserzeugung
- Spieler können Züge asynchron zueinander absetzen
- NPC/Umgebungsmodelle können in unabhängigen Threads agieren
- Nebenläufigkeit kann wie bei Transaktionssystemen realisiert werden (sperrern, rücksetzen, ...)
- Beispiel:
 - bestimmte Browser Games

13

Aktionsverarbeitung bei Eventsteuerung

Vorteile:

- Kann auf Standardlösungen aufsetzen (z.B. DBS)
- Vollständige Nebenläufigkeit:
 - wenig Wartezeit auf andere Spieler (Spiel kann Wartezeiten verlangen)
 - verteilte Realisierung ist einfach

Nachteile:

- Keine Synchronisation zwischen Spielzeit und Echtzeit
=> Spielzeit (Aktionen/Minute) kann stagnieren
- Keine Kontrolle über max. Handlungen pro Zeiteinheit
- Gleichzeitige Handlungen sind unmöglich (Serialisierbarkeit)

14

Realisierung der Zeitabhängigkeit

Tick-Systeme (Soft-Real-Time Simulation)

- Handlungen werden nur zu fest getakteten Zeitpunkten (Ticks) verarbeitet.
- Aktionen können zu beliebigen Zeitpunkten erzeugt werden
- Ein Tick hat eine Mindestdauer (z.B. 1/24 s)
=> feste Synchronisation von Echtzeit und Spielzeit
- Alle Aktionen innerhalb eines Ticks gelten als gleichzeitig (keine Serialisierung)
- Der nächste Game State entsteht aus einer kumulierten Betrachtung aller Aktionen (keine Isolation)
- Verwendetes Modell im Rendering, da hier feste Frameraten und gleichzeitige Änderungen notwendig sind

15

Zeit-Modellierung für Aktionen

Bewertung des Tick-Systems

Vorteile:

- Synchronisation zwischen Echt- und Spielzeit
- Faire Regelung der Aktionen pro Zeiteinheit
- Gleichzeitigkeit

Nachteile:

- Lag-Behandlung
(Server schafft Tick-Berechnung nicht rechtzeitig)
- Konfliktlösung bei gleichzeitigen und widersprüchlichen Aktionen
- Zeitliche Reihenfolge
(alle Aktionen, die in einem Tick ankommen werden als gleichzeitig betrachtet)

16

Zeit-Modellierung für Aktionen

weitere wichtige Aspekte im Tick-System:

- Berechnungszeit eines Ticks hängt von vielen Einflüssen ab:
 - Hardware
 - Größe des Game States
 - Aktionen
 - Komplexität der Aktionen
 - Durchführung der Synchronisation und der Aufgaben anderer Subsysteme:
 - Verteilung eines Game States an das Persistenz-System

17

Aktionen und Transaktionen

Züge/Handlungen erinnern stark an Transaktionen in DBS

- **Atomarität:** Zug/Handlung wird ganz oder gar nicht durchgeführt
Beispiel: Spieler A zieht, Schachuhr für A wird angehalten, Schachuhr für B läuft weiter
- **Consistency:** Überführen eines gültigen Game States in einen anderen gültigen Game State
- **Dauerhaftigkeit:** Ergebnisse von Transaktionen stehen fest im Game State und werden (zumindest partiell) an das Persistenz-System übergeben.

Außerdem:

Übergänge müssen den Spielregeln entsprechen (Integritätserhalt)

- statisch: Game State ist regelkonform
- dynamisch: Übergang ist regelkonform

18

Unterschiede zu Transaktionen

Zeitliche Verarbeitung der Handlungen spielt eine wichtige Rolle

- Durchführung von Handlungen sollte möglichst fair sein
 - keine Verzögerung der Handlungen eines Spielers
 - Anzahl der max. Handlungen pro Zeit für alle Spieler gleich
- Gleichzeitiges Handeln (Simulation der Realität) sollte prinzipiell möglich sein
- max. Bearbeitungsdauer ist für ein flüssiges Spiel notwendig evtl. wegfallen von Handlungen bei Zeit-Überschreitungen
- Zeitsynchronisation zwischen Spielzeit und Echtzeit sollte möglich sein

19

Unterschiede zu Transaktionen

kein zwingender logischer Einbenutzerbetrieb (Isolation)

- Bei gleichzeitigen Handlungen müssen Aktionen in Abhängigkeit berechnet werden. (keine Serialisierbarkeit)
- *Beispiel:*
 - Charakter A hat 100/100 HP (=Hit Points)
 - Zum Zeitpunkt t_j bekommt A 100 HP Schaden von Charakter B
 - Zum Zeitpunkt t_j bekommt A 100 HP Heilung von Charakter C

Ergebnis unter Isolation:

- Erst Heilung (Überheilung) und dann Schaden => A hat 0 HP und stirbt
- Erst 100 Schaden => A stirbt und Heilung ist nicht mehr möglich

Ergebnis unter gleichzeitigen Handlungen:

- A bekommt 100 Schaden und 100 Heilung: Verrechnet heben sich beide Effekte auf.

20

Umsetzung im Game Loop

- Endlosschleife, in der Aktionen auf den aktuellen Game State angewendet werden und diesen so konsistent verändern (Handlungsverarbeitung)
- Zeitmodell ist für den Start der nächsten Iteration verantwortlich
- weitere Funktionalitäten, die vom Game Loop abhängen
 - Lesen und Verarbeiten von Benutzereingaben (=> Benutzerhandlungen)
 - Aufruf der KI von NPCs (=> NPC Handlungen)
 - Aufruf Umgebungsmodell
 - Grafik und Sound Rendering
 - Speichern bestimmter Spielinhalte auf dem Sekundärspeicher
 - Übermittlung von Daten an das Netzwerk
 - Update unterstützender Datenstrukturen (räumliche Indexstrukturen, Graphik-Buffer, ...)
 - ...

} erzeugen Handlungen

21

Realisierung von Game Loops

- Ein Game Loop für alle Aufgaben:
 - kein Overhead durch Synchronisation => effizient
 - schlechte Schichtung der Architektur: bei Änderung eines Aspektes muss auch der Game Core überarbeitet werden
- Unterschiedliche Game Loops für unterschiedliche Subsysteme (Bsp.: AI-Loop, Netzwerk-Loop, Rendering Loop, ...)
 - Gute Schichtung des Systems
 - Subsysteme können bei Client-Server Einteilung ausgeschaltet werden
 - Client braucht keine eigene NPC Steuerung
 - Server braucht keine Rendering Loops
 - Synchronisation der Game Loops

22

Kommunikation mit dem Game Loop

- Game Loop ruft andere Module auf
 - Lösung für Systeme, die im Takt oder langsamer als Game Loop laufen
 - Schlecht geeignet für Multi-Threading
 - Beispiel: Persistenz-System, Netzwerk, Sound Rendering, ...
- Game Loop schickt Messages an Subsystem
 - Erlaubt Multi-Threading
 - Aufrufhäufigkeit ist ein Vielfaches vom Takt der Game Loop
 - **Beispiel:** NPC-Steuerung, Synchronisation mit Clients, Sound-Rendering, ...
- Synchronisation über lesenden Zugriff auf Game State
 - Bei schneller getakteten Systemen benötigt das Subsystem eine eigene Loop
 - Multi-Threading mit umfassendem Zugriff auf den Game State
 - Gelesene Daten müssen konsistent sein (noch nicht geändert)
z.B. Grafik-Rendering, Persistenz-System, ...

23

Verarbeiten von Aktionen

- Aufgabe der Aktionsverarbeitung: Umsetzung von Spiel-Aktionen (laufen, schießen, springen, ...) in Änderungen des Game State
- Aktionsverarbeitung ist dabei Umsetzung der Spielmechanik
- Berechnungsvorschriften für erlaubte Aktionen
- Leseoperation
- Schreiboperation
- Verwendung von Subsystemen möglich
z.B. Spatial Management Modul oder Physics Engine

24

Konsistenzerhalt bei der Aktionsverarbeitung

- Im Tick-System: gleichzeitige Aktionen möglich
- Reihenfolgeunabhängigkeit bzgl. der Aktionen in einem Tick
- Problem: Lesen von bereits geänderten Daten
- Lösungen:
 - Schattenspeicherkonzept:
 - Es gibt 2 Game States G1 und G2
 - G1 enthält den letzten konsistenten Stand (aktiv)
 - G2 wird in aktueller Iteration geändert (inaktiv)
 - Bei Abschluss des Ticks wird G2 auf aktiv und G1 auf inaktiv gesetzt
 - Feste Reihenfolge von Lese und Schreibeoperationen
 - Erfordert das Zerlegen und Neuordnen der Aktionen
 - Alle Aktionen werden gleichzeitig bearbeitet

25

Konflikte bei Gleichzeitigkeit

- Bei Gleichzeitigkeit können Konflikte entstehen (z.B. gleichzeitiges aufheben einer Goldmünze)
- Problem: das Ergebnis der Handlung kann nicht in Isolation berechnet werden. (Wenn A die Münze bekommt, kann B sie nicht auch bekommen.)
- Konfliktbehandlung:
 - Streichen beider Handlungen (Undo both)
=> Konflikterkennung und evtl. rücksetzen der Daten
 - Zufällige Auswahl einer der Aktionen und löschen (random)
=> Konflikt muss erkannt und evtl. rücksetzen der Daten
 - Erste Aktion bekommt recht (natural order)
=> Lösung nicht unbedingt fair
(Ausführungsreihenfolge \neq Handlungsreihenfolge)
=> **aber**: Einteilung in Ticks kann die Handlungsreihenfolge ohnehin beeinflussen

26

Modellierung von Aktionen

Wie werden die erlaubten Aktionen eines Spiels umgesetzt?

- Direkte Implementierung in der Host-Sprache
 - **Vorteil:** hohe Effizienz
 - **Nachteile:**
 - Doppelte Implementierung der gleichen Effekte
 - Redundanter Code
 - Inkonsistenzen sind schwer überprüfbar
- Verwendung von Packages und Subsystemen, die bestimmte Teile der Aktionsverarbeitung kapseln:
 - Physics Engine (Kollisionstests, Beschleunigung, Abprallen, ...)
 - Spatial Management Module (nächste Nachbarn, Sichtbereiche, ...)
 - AI Engine (Wegewahl, Schwarmverhalten, ...)

27

Modellierung von Aktionen

- Scripting Engine
 - Stellt eigene Programmiersprache zur Verfügung
 - Befehle können direkten Zugriff auf Game State verhindern (unterstützt Konsistenzerhalt)
 - Entitäten und ihr Verhalten werden einheitlich modelliert
 - Nachteil: nicht alle Designoptionen sind ohne Änderungen der Skriptsprache umsetzbar
 - **Beispiel:** LUA (<http://lua-users.org/wiki/ClassesAndMethodsExample>)

<pre>require("INC_Class.lua") ===== cAnimal=setclass("Animal") function cAnimal. methods:init(action, cutename) self.superaction = action self.supercutename = cutename end</pre>	<pre>----- 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</pre>
--	---

28

Physics Engines

- Umsetzung von Festkörperphysik/klassische Mechanik
- Game Entität muss alle benötigten Parameter abbilden
 - Räumliche Ausdehnung (Polygonmesh, Vereinfachungen: Zylinder, MUR)
 - Geschwindigkeitsvektor
 - Masse
 - ...
- Umsetzung meist über Differentialgleichungssysteme
- Für realistische Effekte: hohe Tick-Raten und detaillierte Modellierung
- hoher Berechnungsaufwand macht effiziente numerische Näherungsalgorithmen notwendig

29

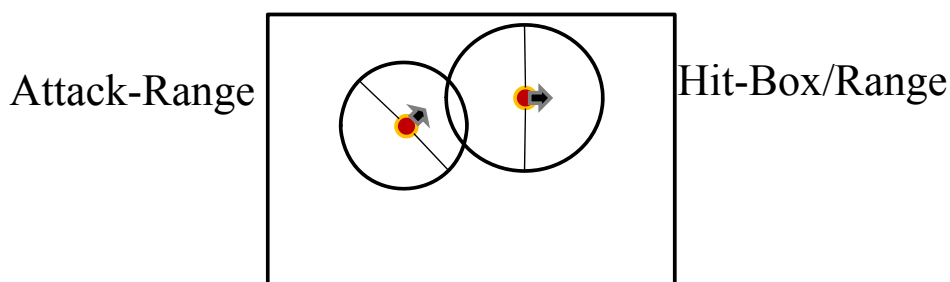
Physics Engines und MMO-Server

- Viele der Ergebnisse einer klassischen Physics Engine werden nur zur realistischeren Darstellung benötigt
 - z.B. Partikelfilter, Rag-Doll Animation, ...
- Gekoppelte Berechnung mit Darstellungs-Layer ist häufig sinnvoll, da Ergebnisse für Änderung der Darstellung und des Game States notwendig sind (Bewegung)
- Hohe Tick-Raten
 - ⇒ Client-seitige Verwendung
 - ⇒ Auf Serverseite meist zu hoher Aufwand
 - ⇒ Für Umsetzung des Designs reichen häufig grobe Vereinfachungen
 - ⇒ Verwendung einer Physics Engine kann auch zur Parameterbestimmung von direkten Implementierungen verwendet werden

30

Spatial Management in Game Servern

- Großteil der Spiele besitzt eine räumliche Komponente z.B. 2D/3D Karten, Spielwelt, ...
- Aktionsverarbeitung, NPC-Steuerung und Netzwerk-Layer beinhalten räumliche Anfragen:
 - Welche anderen Game Entitäten kann ich beeinflussen, können mich beeinflussen oder kann ich überhaupt sehen? (AoI = Area of Interest)
 - Unterstützung von Kollisionsanfragen (vgl. Physics Engine) und Bereichsschnittmengen
 - Welcher Spieler steht mir am nächsten?
 - Betritt ein Spieler die Angriffsumgebung (Aggro-Range) eines NPCs?



31

Spatial Management in Game Servern

- Bei kleinen Spielwelten wenige räumliche Objekte (Game State als Liste organisiert)
- Anfragebearbeitung über sequentielle Suche
- Bei wiederholter Anfragebearbeitung und großer Anzahl beweglicher Entitäten entsteht erheblicher Suchaufwand
Beispiel: 1000 Game Entities in einer Zone, 24 Ticks/s
naive AoI Berechnung: 24 Mio. Distanzvergleiche pro Sekunde
- **Folge:** bei wachsenden Game States fällt sehr viel Berechnungsaufwand für räumliche Anfragen an.

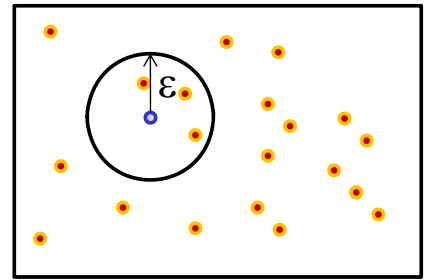
32

räumliche Anfragen (1)

Räumliche Anfragen (hier mit euklidischer Distanz im IR^2)

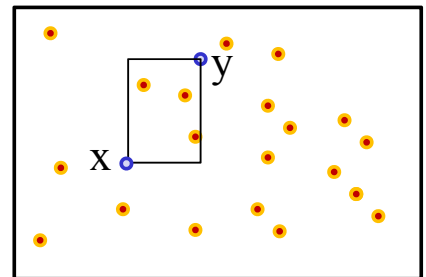
- Range-Query

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



- Box-Query

$$BQ(x, y) = \{v \in GS \mid x_1 \leq v_1 \leq y_1 \wedge x_2 \leq v_2 \leq y_2\}$$



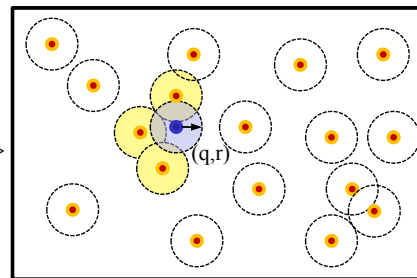
33

räumliche Anfragen (2)

- Intersection Query

$$SIQ(q, r) =$$

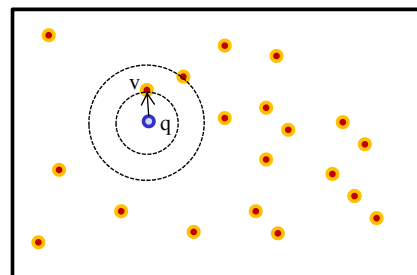
$$\left\{ (v, s) \in GS \times IR \mid \sqrt{(q_1 - v_1)^2 + (q_2 - v_2)^2} \leq r + s \right\}$$



- NN-Query

$$NN(q) =$$

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



34

Effizienzsteigerung für räumliche Anfragen

- Methoden zur Reduktion der betrachteten Objekte (Pruning)
 - Aufteilung der Spielwelt (Zoning, Instanziierung, Sharding, ...)
 - Index-Strukturen (BSP-Tree, KD-Tree, R-Tree, Ball-Tree)
- Reduktion der Aufrufe von räumlichen Anfragen
 - Reduktion der Anfrage-Ticks
 - Spatial-Publish-Subscribe
- Effiziente Anfragealgorithmen
 - Nearest-Neighbor Anfragen
 - ϵ -Range Join (gleichzeitiges Bestimmen aller Areas of Interest)

35

Sharding und Instanziierung

- Kopieren einer räumlichen Region für eine bestimmte Gruppe
- Dabei existiert die gleiche Region der Karte beliebig häufig
- Instanzen und Shards wurden primär zur Umsetzung des Game Designs geschaffen
(Begrenzung der Spieler zum Lösen einer Aufgabe)
- Aber: Je mehr Spieler sich in einer Instanz aufhalten, desto weniger Performanzprobleme entstehen in der offenen Welt.

Problem:

- Löst das Problem nicht (keine zusammenhängende MMO Welt)
- Speicherung des lokalen Gamestates, auch wenn kein Spieler mehr in der Instanz ist
 - => Management der Instanzen kann Aufwand verursachen
(Worst Case: 1000 parallele Game States für 1000 Spieler)

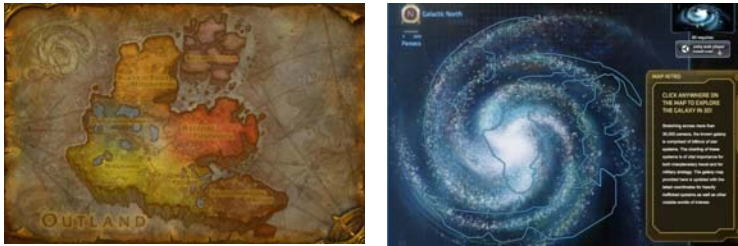
36

Zoning

- Aufteilen der offenen Spielwelt in mehrere feste Teillandschaften
- Anfragen müssen nur Objekte in der aktuellen Zone berücksichtigen
- Teilt nicht nur den Raum, sondern auch den Game State auf
- Erleichtert Verteilung der Spielwelt auf mehrere Rechner

Probleme:

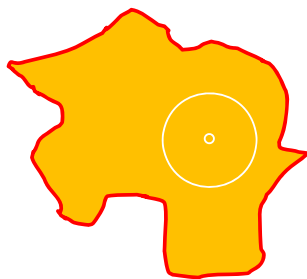
- Bei Randbereichen müssen evtl. Objekte in mehreren Zonen berücksichtigt werden
- Ungleichmäßige Verteilung der Spieler



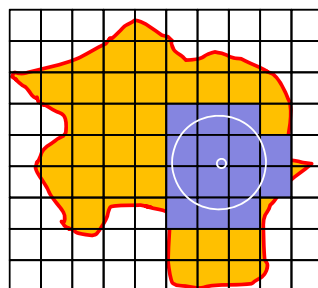
37

Micro-Zoning

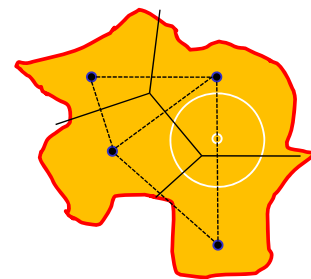
- Spielwelt wird in viele kleine Zonen (Micro-Zonen) aufgeteilt
- Game Entities werden in ihrer aktuellen Micro-Zone verwaltet
- Nur Zonen, die die AoI schneiden sind relevant
- Sequentielle Suchen innerhalb der Region
- Zonen können durch verschiedene Methoden erzeugt werden (Grids, Voronoizellen, ...)



Zoning



Micro-Zoning
(Grid-basiert)



Micro-Zoning
(Voronoi-basiert)

38

Spatial Publish-Subscribe

- Micro-Zoning in Kombination mit Subscriber-System
- Game Entities werden in ihre aktuelle Micro-Zone eingetragen (publish)
- Game Entities abonnieren die Informationen aller Micro-Zonen, die ihre AoI schneiden (subscribe)
- Liste aller Game Entities in AoI über Vereinigung der Einträge der abonnierten Micro-Zonen

Vorteile:

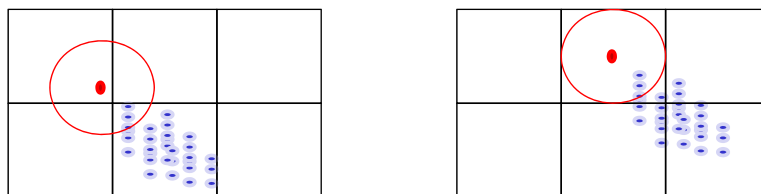
- Effiziente Bestimmung nahe gelegener Objekte
- Bei Änderung können neue Informationen an Subscriber durchgereicht werden (keine regelmäßige Anfrage notwendig)

39

Micro Zoning und Spatial Publish-Subscribe

Nachteile:

- Auch Micro-Zonen können überfüllt sein
=> je kleiner die Fläche, desto weniger wahrscheinlich
- Bei zu kleinen Zonen steigt der Overhead für einen Zonenwechsel
=> je kleiner die Zonen, je häufiger der Wechsel
- Lage des Zonengrenzen kann zu starken Schwankungen bei den betrachteten Objekten führen.
- Bei sehr hohen Änderungsraten steigt der Overhead von Zonen-Wechsel extrem an.
=> sehr viele Subscribe- und Unsubscribe-Vorgänge bremsen das System aus



40

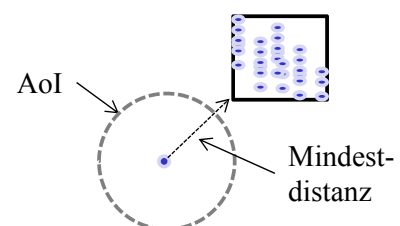
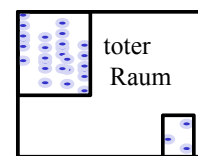
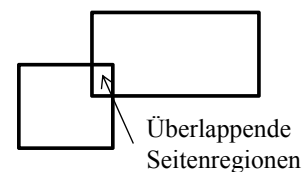
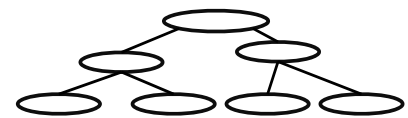
Klassische Indexstrukturen

- Verwaltung von räumlichen Objekten kann auch über räumliche Suchbäume erfolgen
- Suchbäume passen ihre Seitenregionen (Zonen) der Datenverteilung an
 - ⇒ Garantie über maximale Füllung einer Seitenregion/Zone
 - ⇒ Bessere Suchperformanz durch Reduktion der betrachteten Objekte
 - ⇒ Es entsteht ein Aufwand bei der Anpassung des Suchbaumes
- Anpassung durch rekursive Aufteilung des Raumes (Quad-Tree, BSP-Trees)
- Anpassung durch Aufteilen der Daten auf minimal umgebende Seitenregionen

41

Wichtige Merkmale von räumlichen Suchbäumen

- *Seitenregion*: umgebende Approximation mehrerer Objekte
- *Balanzierung*: Unterschiedlichkeit der Pfadlängen von der Wurzel zu den Blattknoten
- *Seitenkapazität*: Anzahl der Objekte, die mindestens/höchstens in der Seitenregion liegen
- *Überlappung*: Schnitt zwischen Seitenregionen ist erlaubt
- *toter Raum*: Raum, in dem keine Seitenregionen/Objekte liegen
- *Pruning*: Ausschluss aller Objekte in einer Seitenregion durch den Test auf Seitenregionen



42

Anforderungen an MMO Server

- i.d.R. liegt der ganze Baum im Hauptspeicher
- hohe Änderungsrate, jede Positionsänderung eines Game Entities
 - Je nach Spiel eine Änderung pro Tick
 - Struktur des Baums kann sich drastisch verändern
- hohe Anfragerate
- Unterstützung mehrerer Anfrage in einem Tick
- Dimensionalität der Objekte ist 2D bzw. 3D
- Objekte haben räumliche Ausdehnungen (Kollision, Hitbox, ...)

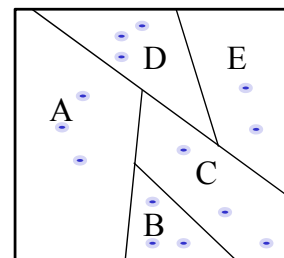
Folgerungen:

- ⇒ Datenstrukturen, die primär Seitenzugriffe optimieren sind ungeeignet (Baum ist im Hauptspeicher)
- ⇒ Overhead für Index-Aufbau/-Anpassung muss durch die Anfragebearbeitung aufgewogen werden.

43

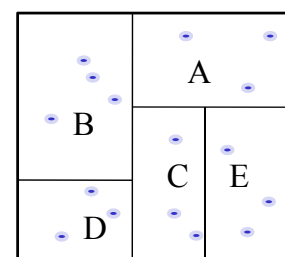
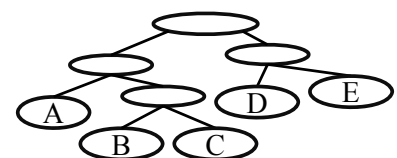
Binary Space Partitioning Trees (BSP-Tree)

- Wurzel enthält gesamten Datenraum
- Jeder innere Knoten hat 2 Söhne
- Datenobjekte in den Blättern



Bekannteste Variante: *kD-Tree*

- max. Seitenkapazität sind M Einträge
- min. Seitenkapazität sind $M/2$ Einträge
- bei Überlauf achsenparalleler Split
- nach Löschen Vereinigung von Geschwisterknoten
- Split-Achse wechselt nach jedem Split
- 50%-50% Aufteilung der Daten



44

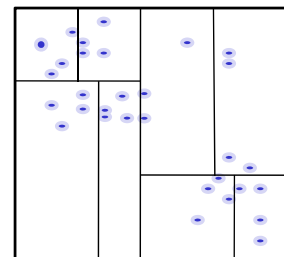
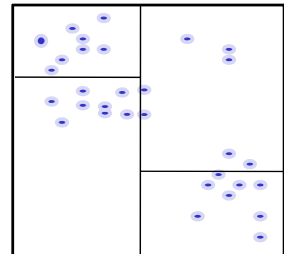
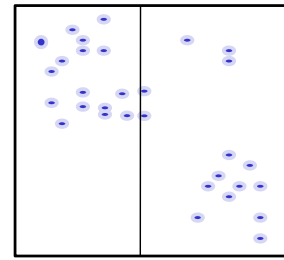
Binary Space Partitioning Trees (BSP-Tree)

Problem bei dynamischen Verhalten:

- keine Balancierung (Degeneration des Baums)
- Korrektur der Balancierung möglich aber aufwendig
=> Hohe Update-Komplexität

Bulk-Load

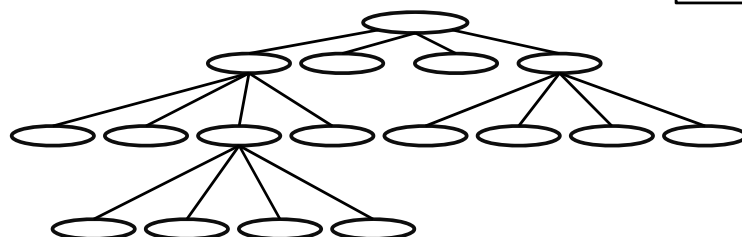
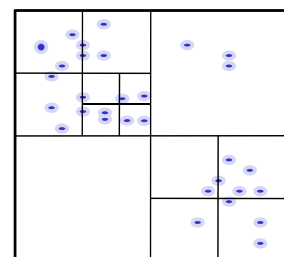
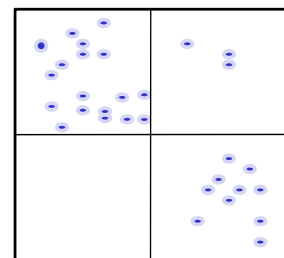
- Annahme: Kenntnis aller Datenobjekte
- Aufbau: durch rekursive 50-50 Aufteilung der Objekte bis jedes Blatt weniger als M Objekte enthält
- Bulk-Load erzeugt immer einen balancierten Baum
- Datenseite eine Baums mit n Objekten und Höhe h enthält mindestens $\lfloor \frac{n}{2^h} \rfloor$ Objekte und höchstens $\lfloor \frac{n}{2^h} \rfloor + 1$



45

Quad-Tree

- Wurzel stellt ganzen Datenraum dar
- Jeder innere Knoten hat 4 Nachfolger
- Geschwisterknoten teilen den Raum ihres Elternknotens in 4 gleich große Teile ein
- Quad-Trees sind i.d.R. nicht balanciert
- Seiten haben einen max. Füllungsgrad M , aber keine Mindestfüllung
- Datenobjekte in den Blättern

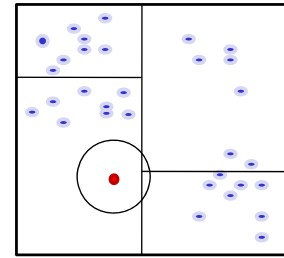


46

Datenpartitionierende Index-Strukturen

Raumpartitionierende Verfahren:

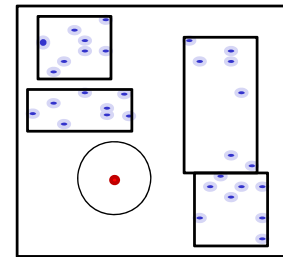
- Aufteilung des gesamten Datenraums durch Splits in den Dimensionen
- Seitenregionen enthalten toten Raum
 - => evtl. schlechtere Suchperformanz bei räumlichen Anfragen



Range-Query in BSP-Tree

Datenpartitionierende Verfahren:

- Beschreibung der Seiten-Region durch minimal umgebende Regionen (z.B. Rechtecke)
 - => Bessere Pruning Leistung
- Seitenregionen können überlappen
 - => Degeneration bzgl. Überlappung
- Split- und Einfüge-Algorithmen minimieren:
 - Überlappung der Seitenregionen
 - Toter Raum in den Seiten
 - Balancierung bzgl. des Füllungsgrades



Range-Query in R-Tree

47

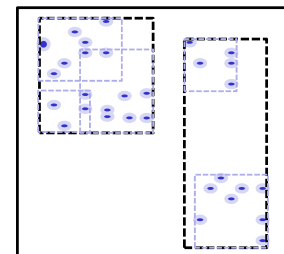
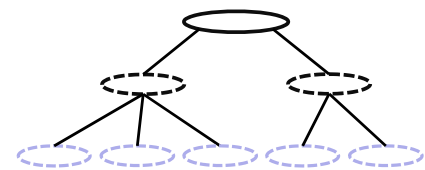
R-Baum

Struktur eines R-Baums:

- Wurzel umgibt den gesamten Datenraum und hat maximal M Einträge
- Seitenregionen werden durch minimal umgebende Rechtecke (MUR) modelliert
- innere Knoten im R-Baum haben zwischen m und M Nachfolger (wobei $m \leq M/2$)
- Das MUR eines Nachfolgers ist vollständig im MUR des Vorgängers enthalten
- Alle Blätter sind auf dem gleichen Level
- Datenobjekte werden in den Blättern gespeichert

Mögliche Datenobjekte:

- Punkte
- Rechtecke



48

Einfügen in den R-Baum

Das Objekt x ist in einen R-Baum einzufügen

Durch Überlappung können 3 Fälle auftreten

- Fall 1: x fällt vollständig in genau ein Directory-Rechteck D
⇒ Einfügen in Teilbaum von D
- Fall 2: x fällt vollständig in mehrere Directory-Rechtecke D_1, \dots, D_n
⇒ Einfügen in Teilbaum von D_p , das die geringste Fläche aufweist
- Fall 3: x fällt vollständig in kein Directory-Rechteck
⇒ Einfügen in Teilbaum von D , das den geringsten Flächenzuwachs erfährt
(in Zweifelsfällen: \dots , das die geringste Fläche hat)
⇒ D muss entsprechend vergrößert werden

49

Split-Algorithmus im R-Baum

(im Folgenden wird von inneren Knoten ausgegangen: Objekte sind MURs)

Der Knoten K läuft mit $|K| = M+1$ über:

⇒ Aufteilung auf zwei Knoten K_1 und K_2 , so dass $|K_1| \geq m$ und $|K_2| \geq m$

Quadratischer Algorithmus

- Wähle das Paar von Rechtecken R_1 und R_2 mit dem größten Wert für den „toten Raum“ im MUR, falls R_1 und R_2 in denselben Knoten K_i kämen.

$$d(R_1, R_2) := \text{Fläche}(\text{MUR}(R_1 \cup R_2)) - \text{Fläche}(R_1) - \text{Fläche}(R_2)$$

Setze $K_1 := \{R_1\}$ und $K_2 := \{R_2\}$

- Wiederhole den folgenden Schritt bis zu STOP:
 - wenn alle R_i zugeteilt sind: STOP
 - wenn alle restlichen R_i benötigt werden, um den kleineren Knoten minimal zu füllen: teile sie alle zu und STOP
 - sonst: wähle das nächste R_i und teile es dem Knoten zu, dessen MUR den kleineren Flächenzuwachs erfährt. Im Zweifelsfall bevorzuge den K_i mit kleinerer Fläche des MUR bzw. mit weniger Einträgen

50

schnellere Splitstrategie für R-Baum (1)

Linearer Algorithmus

- Der lineare Algorithmus ist identisch mit dem quadratischen Algorithmus bis auf die Auswahl des initialen Paares (R_1, R_2) .
- Wähle das Paar von Rechtecken R_1 und R_2 mit dem „größten Abstand“, genauer:
 - Suche für jede Dimension das Rechteck mit dem kleinsten Maximalwert und das Rechteck mit dem größten Minimalwert (*maximaler Abstand*).
 - Normalisiere den *maximalen Abstand* jeder Dimension, indem er durch die Summe der Ausdehnungen der R_i in der Dimension dividiert wird (*setze den maximalen Abstand der Rechtecke ins Verhältnis zur ihrer Ausdehnung*).
 - Wähle das Paar von Rechtecken mit dem größten normalisierten Abstand bzgl. aller Dimensionen. Setze $K_1 := \{R_1\}$ und $K_2 := \{R_2\}$.
- Dieser Algorithmus ist linear in der Zahl der Rechtecke $(2m+1)$ und in der Zahl der Dimensionen d .

51

Splitalgorithmus im R*-Baum

Idee der R*-Baum Splitstrategie

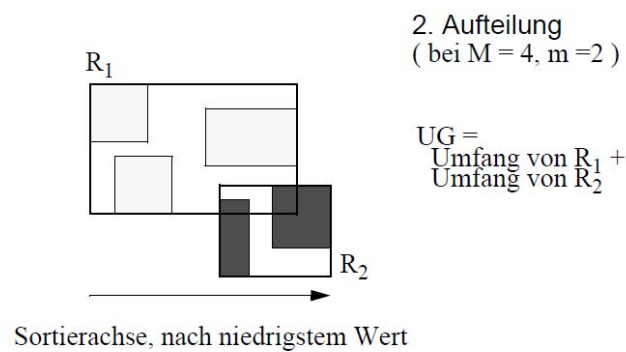
- sortiere die Rechtecke in jeder Dimension nach beiden Eckpunkten und betrachte nur Teilmengen nach dieser Ordnung benachbarter Rechtecke
- Laufzeitkomplexität ist $O(d \cdot M \cdot \log M)$ für d Dimensionen und M Rechtecke

Bestimmung der Splitdimension

- Sortiere für jede Dimension die Rechtecke gemäß beider Extremwerte
 - Für jede Dimension:
 - Für jede der beiden Sortierungen werden $M-2m+2$ Aufteilungen der $M+1$ Rechtecke bestimmt, so dass die 1. Gruppe der j -ten Aufteilung die ersten $m-1+j$ Rechtecke und die 2. Gruppe die übrigen Rechtecke enthält
 - U_G sei die Summe aus dem Umfang der beiden MURs R_1 und R_2 um die Rechtecke der beiden Gruppen
 - U_S sei die Summe der U_G aller berechneten Aufteilungen
- ⇒ Es wird die Dimension mit dem geringsten U_S als Splitdimension gewählt.

52

Splitalgorithmus im R*-Baum



Bestimmung der Aufteilung

- Es wird die Aufteilung der gewählten Splitdimension genommen, bei der R_1 und R_2 die geringste Überlappung haben.
 - In Zweifelsfällen wird die Aufteilung genommen, bei der R_1 und R_2 die geringste Überdeckung von totem Raum besitzen.
- ⇒ Die besten Resultate hat bei Experimenten $m = 0,4 \cdot M$ ergeben.

53

Bulk-Loads im R-Baum

- **Vorteile:**
 - schneller im Aufbau
 - Struktur ermöglicht i.d.R. schnellere Anfragebearbeitung
- **Optimierungskriterien:**
 - möglichst hoher Füllungsgrad der Seiten (geringe Höhe)
 - geringe Überlappung
 - wenig toter Raum

Sort-Tile-Recursive:

- Bottom-up Aufbau des R-Baums
- keine Überlappung auf Blattebene bei Punktobjekten
- Zeitkomplexität: $O(n \log(n))$

54

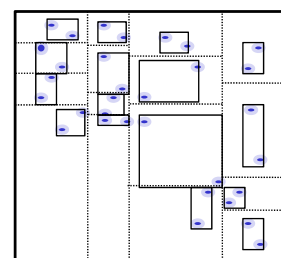
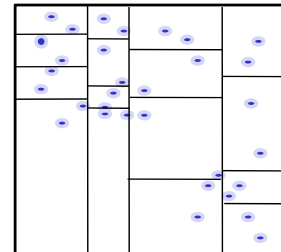
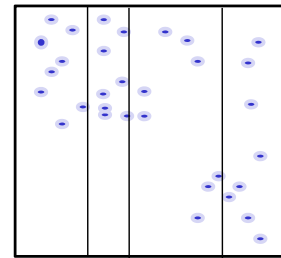
Sort-Tile Recursive

Algorithmus:

1. Setze DB gleich der Objektmenge P mit $|P| = n$
2. Berechne Anzahl der Quantile: $q = \left\lceil \sqrt{\frac{n}{M}} \right\rceil$
3. Sortiere Datenelemente in Dimension 1
4. Bilde Quantile nach jeweils $q \cdot M$ Objekten in Dim. 1
5. Sortiere Objekte in jedem Quantil nach Dimension 2
6. Bilde Quantile nach jeweils M Objekten in Dim. 2
7. Bilde eine MUR um die Punkte in jeder Zelle
8. Starte den Algorithmus mit der Menge abgeleiteten MURs oder stoppe falls $q < 2$
(alle verbleibenden MURs fallen in die Wurzel)

Anmerkung:

- bei Punkten entstehen überlappungsfreie MUR
- bei MURs kann es zu Überlappungen kommen
- bei MURs ist die Einteilung in Quantile nach Minimalwerten, Maximalwerten oder komplizierteren Heuristiken möglich



55

Löschen im R-Baum

Das Objekt x ist aus dem R-Baum zu löschen

Löschen :

- Teste ob Seite S nach entfernen von x unterfüllt ist: $|S| < m$
- Falls nicht, entferne x und STOP
- Falls ja bestimme, welche Vorgängerknoten ebenfalls unterfüllt sind
- Für jede unterfüllten Knoten:
 - Lösche die unterfüllte Seite aus dem Vorgängerknoten
 - Füge die restlichen Elemente der Seite in den R-Baum ein
 - Falls Wurzel nur noch 1 Kind enthält wird Kind zur neuen Wurzel (Höhe verringert sich)

Bemerkungen:

- Löschen ist mit diesem Algorithmus nicht auf einen Pfad beschränkt
- Erfordert das Einfügen eines Teilbaums auf Ebene l in den R-Baum
- Im Worst Case sehr teuer

56

Suchalgorithmen auf Bäumen

Bereichsanfragen:

```
FUNCTION List RQ( $q, \varepsilon$ ):  
List C // Kandidatenliste (MURs/Objekte)  
List Result // Liste aller Objekte in  $\varepsilon$ -Umgebung von  $q$   
C.insert(Wurzel)  
WHILE(not C.isEmpty())  
    E := C.removeFirstElement()  
    IF E.isMUR()  
        FOREACH F  $\in$  E.children()  
            IF minDist(F,q) <  $\varepsilon$   
                C.insert(F)  
    ELSE  
        Result.insert(E)  
RETURN Result
```

Bemerkung: BOX und Intersection-Queries funktionieren nach demselben Prinzip.

57

Nächste-Nachbar-Anfragen

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

```
FUNCTION Object NNQuers( $q$ ):  
PriorityQueue Q // noch zu untersuchende Objekte/Seiten nach  
                //mindist sortiert  
Q.insert(0,Wurzel)  
WHILE(not Q.isEmpty())  
    E := Q.removeFirstElement()  
    IF E.isMUR()  
        FOREACH F  $\in$  E.children()  
            Q.insert(mindist(F,q), F)  
    ELSE  
        RETURN E
```

Bemerkung:

- $mindist(R,P)$ ist die minimale Distanz zwischen 2 Punkten in R und P . Sind R und P Punkte gilt: $mindist = dist$
- *PriorityQueue* sind i.d.R über *Heap-Strukturen* implementiert(vgl. Heapsort)

58

Spatial Joins

Idee: Definiere Join-Anfragen über räumliche Prädikate

Vorteil: Parallele Verarbeitung vieler Anfragen mit Durchlauf

Beispiel: ε -Range-Join

Seien G und S Mengen räumlicher Objekte mit $G, S \subseteq D$,
 $dist: D \times D \rightarrow IR$ ein Distanzfunktion und $\varepsilon \in IR$.

Dann heißt

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

ε -Range-Join von G und S .

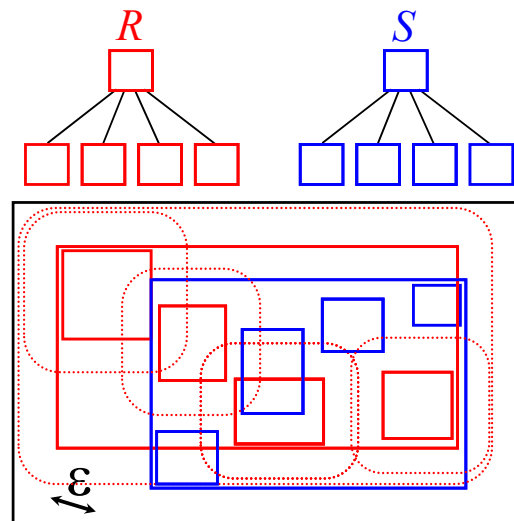
Anwendung: Bestimme AoI für alle Spielerentitäten in einem Tick.

59

R-tree Spatial Join (RSJ)

Algorithmus:

```
FUNCTION rTreeSimJoin ( $R, S, result, \varepsilon$ )
  IF  $R.isDirectoryPage()$  and  $S.isDirectoryPage()$ 
    FOREACH  $r \in R.children()$ 
      FOREACH  $s \in S.children()$ 
        IF  $\minDist(r,s) \leq \varepsilon$ 
          rTreeSimJoin( $r,s,result,\varepsilon$ )
  ELSE //assume  $R,S$  are both DataPoints
    FOREACH  $p \in R.points$ 
      FOREACH  $q \in S.points$ 
        IF  $\text{dist}(p,q) \leq \varepsilon$ 
          result.insertPair( $p,q$ )
  RETURN result
```



60

Probleme durch Datenvolatilität

Probleme bei räumlicher Fortbewegung aller Objekte:

In Spielen bewegt sich die Mehrzahl der Objekte mehrmals pro Sekunde.

- Positionsänderungen durch Löschen und Einfügen
 - dynamische Änderungen können die Struktur negativ beeinflussen (Verlust der Balance, Erhöhung von Überlappungen, Überfüllung einer Micro-Zone)
 - Änderung hat hohen Overhead (Suche nach Objekt, Folge-Einfügungen, Underflow- und Overflow-Behandlungen)
- Positionsänderung durch spezielle Änderungsoperationen
 - Ausdehnen von Seitenregionen: Overlap der Seiten kann extrem ansteigen (*nur bei Datenpartitionierung möglich*)
 - Wechsel der Seitenregionen:
 - Balance kann negativ beeinflusst werden
 - Unterfüllung und Überfüllung von Seiten

Fazit: Dynamische Änderungen sind entweder aufwendig in der Berechnung oder können die Organisation der Daten ungünstig für Anfragealgorithmen beeinträchtigen.

61

Throw-Away Indices

Idee:

- Bei sehr volatilen Daten ist die Änderung vorhandener Datenstrukturen teurer als ein Neuaufbau mittels Bulk-Load.
- Ähnlich wie beim Game-State existieren immer 2 Indexstrukturen:
 - Index I1 organisiert Positionen des letzten konsistenten Ticks und wird zur Anfragebearbeitung genutzt
 - Index I2 wird parallel zu I1 aufgebaut:
 - Aufbau per Bulkload: geringere Nebenläufigkeit, aber schneller Aufbau, gute Struktur
 - Dynamischer Aufbau: mehr Rechenaufwand und evtl. schlechtere Struktur, aber Aufbau bei jeder neuen Position möglich
 - bei Beginn des neuen Ticks wird auf I2 angefragt und I1 gelöscht und anschließend neu aufgebaut

Fazit: Entscheidend ist, dass der Tick mit Aufbau des Baums schneller verarbeitet wird als ohne unterstützende Datenstruktur.

(Abhängigkeit vom Game-Design)

62

Game Design

Die räumliche Problematik hängt stark vom Spiel-Design ab:

- Anzahl und Verteilung der räumlichen Objekte
- Anzahl und Verteilung der Spieler
- Umgebungsmodell Felder, 2D oder 3D
(3D Umgebung macht 3D-Indexing nicht unbedingt notwendig)
- Bewegungsgeschwindigkeit und Bewegungsart der Objekte

63

Lernziele

- Game State und Game Entities
- Aktionen und Zeitsteuerung
- Game Loop und Synchronisation mit anderen Subsystemen
- typische Verarbeitungsschritte in einer Iteration
- Zusammenhang Scripting-Engine, Physics Engine und Spatial Management
- Zoning, Sharding und Instanziierung
- Micro-Zoning und Spatial-Publish-Subscribe
- BSP-Tree, KD-Tree, Quad-Tree und R-Tree
- Einfügen, Löschen, Bulk-Load
- Anfragebearbeitung: Range-Query, *NN*-Query und Range-Join
- Probleme bei hoch volatilen Daten

64

Literatur und 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.