---

1. Spatial Management

Introduction

Spatial management is crucial for making games scale well. Most games take place in spatial environments (2D/3D maps), requiring efficient handling of spatial queries and data structures.

Spatial Management in Games

Key Requirements

- **Transition models, agent control, and transmission protocols** require spatial query processing
- Main spatial queries needed:
  - Which game entities are within interaction range? (Area of Interest - AoI)
  - Support collision detection and area intersections (pre-filtering)
  - Which game entity is closest?
  - Does a player enter the aggro-range of an opponent?

Performance Challenge Example

For small game worlds with limited entities, spatial positions can be organized in a list with sequential scan processing. However, this becomes expensive with high query frequencies and large numbers of moving objects.

Example calculation:

- 1000 game entities in one zone
- 24 ticks/second
- Naive AoI computation requires: (1000 × 999)/2 × 24 = **11,988,000 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 quadratically).

Types of Spatial Queries

1. ε-Range Queries

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

Returns all objects within Euclidean distance ε from query point q.

2. Box Query

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

Returns all objects within a rectangular region.

3. Intersection Query

$$IQ(q,r) = \{(v,s) \in GS \times \mathbb{R} \mid \sqrt{((q_1-v_1)^2 + (q_2-v_2)^2)} \leq r + s\}$$

Finds all objects whose regions intersect with the query region.

4. Nearest Neighbor Query

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

Finds the closest object to the query point.

Efficiency Tuning Methods

1. Methods to Reduce Number of Considered Objects (Pruning)

- Distribute the game world (zoning, instancing, sharding)
- Index structures (BSP-Tree, KD-Tree, R-Tree, Ball-Tree)

2. Reduce Number of Spatial Queries

- Reduce query ticks
- Spatial publish-subscribe

3. Efficient Query Processing

- Nearest-neighbor queries
- ε-range Join (simultaneously compute all AoIs)

Spatial Management Techniques

Sharding and Instantiation

- **Concept:** Copying a region for a specific group
- **Benefits:**

- Any number of the same region can exist
- Part of game design (e.g., limiting players for a quest)
- Reduces performance issues in open world

- **Complications:**
  - Doesn't solve underlying problem (no connected game world)
  - Instance management can cause additional expenses
  - Worst case: 1000 parallel game states for 1000 players

Zoning

- **Concept:** Splitting open world into several fixed areas
- **Benefits:**
  - Only objects in current zone need consideration
  - Partitions both space and game state
  - Easier distribution across multiple computers
- **Problems:**
  - Objects of adjacent zones need consideration
  - Heterogeneous distribution of players over zones

Micro-Zoning

- **Concept:** Game world partitioned into small areas (micro zones)
- **Process:**
  - Only entities within actual micro zone are managed
  - Only micro zones intersecting AoI are relevant
  - Sequential search within a region
  - Zones created via grids, Voronoi-cells, etc.

Spatial Publish-Subscribe

- **Concept:** Combination of micro-zoning and subscriber systems
- **Process:**
  1. Game entities register in current micro zone (publish)
  2. Entities subscribe to all micro zones with intersecting AoIs
  3. List of entities within AoI created by merging subscribed zones
- **Advantages:**
  - Close-by objects determined efficiently
  - Changes actively sent to subscribers (no regular queries needed)

- **Disadvantages:**
  - Micro zones can become overcrowded
  - Zone changing overhead increases if zones too small
  - Zone border placement may cause extreme fluctuations
  - High change rates increase overhead significantly

Index Structures

Classic Index Structures Overview

- **Space Partitioning:** Partition data space via dimensional splits
  - Examples: Quad-Tree, BSP-Trees
  - May include dead space
  - Potentially bad search performance for spatial queries
- **Data Partitioning:** Page regions defined by minimum bounding regions
  - Example: R-Trees
  - Better pruning performance
  - Page regions may overlap
  - Risk of degeneration

Important Features of Search Trees

- **Page region:** Surrounding approximation of several objects
- **Balancing:** Addresses variance of path lengths or objects per region
- **Page capacity:** Min/max number of objects within page region
- **Overlap:** Intersections between page regions
- **Dead space:** Space without page regions
- **Pruning:** Exclusion of all objects within a page region

Requirements for Games

- Whole tree stored in main memory
- High volatility (position changes every tick)
- Trees might degenerate requiring costly balancing
- Many queries per time unit
- Support for multiple queries during one tick
- Objects have 2-3 dimensions with volume (hitbox)

Key conclusion: Runtime increase for query processing must compensate for time spent on index creation/update.

Binary Space Partitioning (BSP) Trees

kD-tree (most popular BSP variant)

- Root contains whole data space
- Every inner node has two successors
- Data objects stored in leaf nodes
- Properties:
  - Max page capacity: M entries
  - Min page capacity: M/2 entries
  - At overflow: splitting w.r.t. an axis
  - Axis changes after every split
  - Data distributed 50%-50%
  - At deletion: merge sibling nodes

Problems with Dynamic Behavior

- No balancing (tree might degenerate)
- Rebalancing possible but expensive
- High update complexity

Bulk-Load for BSP Trees

- Assumption: All data objects known
- Creation: Recursively distribute objects with 50/50 split
- Always creates balanced tree
- Data page of tree height h with n objects contains:
  - At least $\lfloor n/2^h \rfloor$ objects
  - At most $\lfloor n/2^h \rfloor + 1$ objects

Quad-Trees

- Root represents whole data space
- Every inner node has four successors
- Sibling nodes split parent space in four equal parts
- Usually not balanced
- Pages have maximum filling ratio M, no minimum

- Leaves contain data objects

R-Trees

Structure

- Root encompasses complete data space (max M entries)
- Page regions modeled by Minimal Bounding Rectangles (MBR)
- Inner nodes have m to M successors ($m \leq M/2$)
- MBR of successor completely contained within predecessor's MBR
- All leaves at same height
- Leaves contain data objects (points or rectangles)

Insertion Algorithm

Three cases when inserting object x:

1. **x contained in directory rectangle D:** Insert into subtree of D
2. **x contained in several rectangles $D_1 \ldots D_n$:** Insert into subtree with smallest area
3. **x not contained in any rectangle:** Insert into subtree with smallest area increase

Split Algorithm (Basic $O(n^2)$)

When node K overflows ($|K| = M+1$):

1. Choose pair (R_1,R_2) with largest "dead space":

   $$d(R_1,R_2) = \text{area}(MBR(R_1 \cup R_2)) - \text{area}(R_1) - \text{area}(R_2)$$

2. Set $K_1 = \{R_1\}$ and $K_2 = \{R_2\}$
3. Repeat until all R_i assigned:
   - If all remaining needed to fill smaller node: assign all and stop
   - Otherwise: allocate next R_i to node with smallest MBR area increase

Linear Split Algorithm O(n)

- Same as basic but different initial pair selection
- Choose pair with "greatest distance":
  1. Find rectangles with lowest max and highest min values per dimension
  2. Normalize maximum distance by dividing by total expansion
  3. Choose pair with greatest normalized distance
- Linear complexity: $O(2m+1 \times d)$ where d = dimensions

Bulk-Load and R-Trees

- **Advantages:** Faster creation, better query processing structure
- **Optimization criteria:**
  - Greatest possible filling ratio (low height)
  - Little overlap
  - Small dead space

Sort-Tile-Recursive Algorithm

1. Consider n points/rectangles to store
2. Calculate quantile: $q = \sqrt{(n/M)}$
3. Sort data in dimension 1
4. Generate quantile after $q \cdot M$ objects
5. Sort objects of every quantile in dimension 2
6. Generate quantile after M objects
7. Create MBR around points in each cell
8. Restart with derived MBRs or stop if $q < 2$

- **Time complexity:** $O(n \log n)$
- **Note:** Creates non-overlapping MBRs for points

Deletion in R-Trees

- Test for underflow after deletion: $|S| < m$
- If no underflow: delete and stop
- If underflow:
  1. Delete underflowed page from predecessor
  2. Insert remaining elements into R-Tree
  3. If root has single child, child becomes new root

Search Algorithms

Range Query

```
FUNCTION List RQ(q,ε):
  List C // candidates (MBRs/Objects)
  List Result
  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
```

Nearest Neighbor Query (Top-Down Best-First-Search)

```
FUNCTION Object NNQuery(q):
  PriorityQueue Q // 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
```

Spatial Joins

ε-Range Join Definition

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

Allows determining objects within AoI for each entity with single operation.

R-tree Spatial Join 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)
  ELSE // R,S are data pages
    FOREACH p ∈ R.points
      FOREACH q ∈ S.points
        IF dist(p,q) ≤ ε
          result.insertPair(p,q)
  RETURN result
```

Problems of Data Volatility

Issues with Spatial Movement

- In games, majority of objects move several times per second
- **Changing position by delete/insert:**
  - Dynamic changes negatively influence structures
  - Causes overhead (queries, inserts, under/overflow handling)
- **Changing position via dedicated operations:**
  - Page region expansion increases overlap
  - Moving objects between regions affects balance

Conclusion: Dynamic calculation either has huge computational overhead or degenerates data structures.

Throw-Away Indices

Concept

For highly volatile data, rebuilding with bulk load is more efficient than updating existing structures.

Implementation

- Use 2 index structures:
  - **I_1:** Represents positions from last consistent tick (used for queries)
  - **I_2:** Created simultaneously
    - Via Bulk-Load: Less concurrency, fast creation, good structure
    - Dynamic creation: Higher effort, potentially worse structure
- At new tick start: I_2 used for queries, I_1 deleted and rebuilt

Key principle: Use tree only if creation + query processing time < brute force query time.

Design Considerations

Spatial problems depend heavily on game design:

- Number and distribution of spatial objects
- Number and distribution of players
- Environmental model (fields, 2D, 3D)
  - Note: 3D environment doesn't necessitate 3D indexing
- Movement type and speed of objects

---

2. Virtual Environments

Overview

This section covers:

- Structure of Games
- Game States and Game Entities
- Transitions and Temporal Models
- Games and Simulations
- Games as Stochastic Processes

Building Blocks in Game Architecture

Architecture Layers (top to bottom)

1. **Game Content**
2. **Game Engine**
   - Sound Manager
   - Animation Engine
   - Rendering Engine
   - Graphics Engine
   - Game Core
   - Persistence Layer
   - Script Language
   - User Interface
   - AI Engine

- Physics Engine
- Network Module

3. **Hardware Abstraction Layer** (DirectX, OpenGL, etc.)
4. **Hardware** (Sound Card, Graphics Card, Input devices, Network)

AI-Relevant Components

Directly involved:

- AI Engine: computes actions
- Game Core: represents the environment
- Physics Engine (sometimes)

Relevant infrastructure:

- Network layer: aspects of distributed environments
- Spatial management (in core or physics engine)
- Persistence layer: store data to analyze and learn from
- Scalability: required for massive multiplayer environments or fast forward simulations

Game States

Definition

All data representing the current state of the game:

- Objects, attributes, relationships (think ER or UML models)
- Contains all dynamic information
- Lists all game entities
- Contains all attributes of game entities
- Information concerning the whole game

Not necessarily in game state:

- Static information
- Environmental models/maps
- Preset attributes of game entities

Good Design Pattern

Model-View-Controller (MVC) Pattern:

- **Server:** Managing game state / no visualization (model and modification logic)

- **Client:** Only parts of game state but requires I/O and visualization (view and controller)

Example: Chess Game State

- **Game information:**
  - Players and assigned colors (black/white)
  - Game mode (with/without chess clock)
- **Game state:**
  - Positions of all figures / field occupation
  - Next player to move
  - Time left for both players (if using chess clock)

Transitions

What triggers game state changes?

- Player actions (e.g., chess move)
- Agent actions (AI move)
- Transition models (e.g., falling objects)

Need for synchronization:

- Model describing game time
- Organize temporal order of changes
- Restrict actions per time interval
- Ensure fair opportunities for all players
- Synchronize with wall-clock time

Consistency and Transitions

- Given consistent game state
- Transitions change valid state to new valid state
- Implement the rules of the game

Model-based Transitions Example

Ball on slope:

1. Environmental model decides ball rolls down
2. Transitions change position and state (acceleration)

Modeling Transitions

- Validity depends on game rules
- Distinguish between:
  - Computing an action (AI)
  - Environment's reaction (transition model)
- Transition models either:
  - Prevent prohibited actions
  - Compute valid environmental reaction

Examples:

- **Chess:** Only allow possible moves
- **Racing game:** Collisions, acceleration, deformation, disentanglement (physics engine)

Physics Engines

Implementation

- Solid state physics and classical mechanics
- Game entity provides parameters:
  - Spatial extension (polygon mesh, simplified: cylinders, MBRs)
  - Movement vectors
  - Mass
  - Other physical properties

Computational Requirements

- Uses differential equations
- Realistic effects require:
  - Large tick rates
  - Detailed models
- Large computational effort:
  - Precomputation
  - Numerical approximations

Detail Level vs. Performance Trade-off

- Transition models often use rough approximations
- More detail = less extended environment possible
  - Fewer entities
  - Lower framerates

- Detail level depends on game concept

Example contrast:

- **FPS:** Models physics of flying bullets
- **RPG:** Entire process modeled as random process

Temporal Models

Requirements

- Proper timing for transitions
- Time intervals for physics calculations
- Turn enforcement for board games
- Handle simultaneous actions
- Organize and synchronize transitions

Three Main Temporal Models

1. Turn-based Model

- **Process:**
  - Agents queried in fixed order
  - Actions sequential or parallel
  - Transitions computed in fixed order
  - Triggered when agents complete moves
- **Characteristics:**
  - Game state fixed during decision
  - Agents can block progression
  - Action order/amount determined by model
- **Pros:**
  - Clear, understandable temporal management
  - Time progresses based on slowest agent
- **Cons:**
  - Not suitable for real-time
  - No advantage for faster agents

Example: Round-based RPG

1. Environment randomly decides attack order

2. Agents choose actions sequentially
3. Environment computes state after each action
4. Environment progresses transition model

2. Transaction Model

- **Process:**
  - Actions queued and processed FIFO
  - Each transition processed sequentially
  - Game state = result of last transition
  - Agents submit actions at any speed
- **Essentially:** ACID database with sequential transactions
- **Pros:**
  - Guarantees valid game state
  - Faster agents can submit more actions
- **Cons:**
  - No sync with wall-clock time
  - No simultaneous actions

3. Tick System (Soft Real-Time)

- **Process:**
  - Transitions at fixed time intervals
  - Ticks represent game time intervals
  - All actions within tick treated as simultaneous
  - Joint state transition computation
- **Characteristics:**
  - Real-time w.r.t. tick length
  - Action delayed at most one tick
  - "Soft" = processing can exceed wall-clock interval
- **Pros:**
  - Synchronizes game and wall-clock time
  - Fair action rules per time unit
  - Concurrency possible
- **Cons:**
  - Handling lags

- Conflict resolution needed
- Chronological order issues

Actions vs. Transactions

Concurrent actions must be computed interdependently (not serializable)

Example scenario:

- Character A has 100/100 HP
- At time t:
  - A suffers 100 HP damage from B
  - A receives 100 HP healing from C

Outcomes:

- **Isolation (healing first):** A dies (overheal → damage)
- **Isolation (damage first):** A dies (can't heal dead)
- **Concurrent:** Effects cancel, A survives

Conflicts During Concurrency

Example: Two players pick up gold coin simultaneously

Conflict Resolution Options:

1. **Delete both actions:** Conflict detection + possible reset
2. **Random pick:** One succeeds, other fails
3. **First action executes:** Natural order (may not be fair)

Consistency During Action Handling

Problem: Reading already changed data

Solutions:

1. **Shadow Memory:**
   - Two game states: G_1 (active/consistent) and G_2 (inactive/changing)
   - All reads from G_1, all writes to G_2
   - On tick completion: swap active/inactive
2. **Fixed sequence of read/write operations:**
   - Break down action components
   - Rearrange operations

- Handle all actions simultaneously

Time Model Extensions

Turn-based can extend to tick-system by:

- Collecting actions from all agents each turn
- Setting time-out for acting
- Computing transitions regardless of submissions
- Providing game time per tick to environment model

Comparison to Scientific Simulations

Similarities

- Usually use tick systems
- Often no actors, just physics engine
- Agent-based simulations use AI
- Need tick systems for fixed real-world time interpretation

Differences

- Simulations don't require wall-clock sync
- Technically similar but transition models target maximal realism

Event-based vs. Tick-based Simulations

Event-based applicable if:

- Can schedule when each agent acts next
- Can compute future environment state without agent actions

Process:

1. Determine next tick for each agent
2. Organize in priority queue
3. Progress environment to top tick
4. Collect all actions for that tick

- **Benefit:** Skip eventless ticks for efficiency

Games as Stochastic Processes

Rationale

- Mathematical model helpful for analysis
- Games involve non-deterministic elements:
  - Environmental random components
  - Player action variation
- Otherwise all matches identical
- Match = stochastic process

Discrete Time Homogeneous Markov Process

Definition requires:

- Set of states S
- Stochastic transition function T: P(s'|s) for all s,s' ∈ S

Optional:

- Start distribution B: P(s'|-) for first state
- Terminal states S_terminal ending process

Example calculation:

$$\begin{aligned} P(ACBB) &= P(A|-) \cdot P(C|A) \cdot P(B|C) \cdot P(B|B) \cdot P(-|B) \\ &= 0.3 \cdot 0.15 \cdot 0.7 \cdot 0.4 \cdot 0.1 \end{aligned}$$

Properties of Markov Processes

1. **First-order Markov Property:**
   - Process is memoryless
   - s_{t+1} depends only on s_t
2. **Discrete time:**
   - Process runs in discrete steps
   - Corresponds to turn/tick-based time
3. **Continuous time Markov chains:**
   - Real-time transactions
   - Need to predict next transition time

Homogeneous vs. Inhomogeneous Processes

Homogeneous:

- Same transition function each time step
- Example: PageRank algorithm

Inhomogeneous:

- Transition function varies over time
- $p(s_{t+1}=A|s_t=B) \neq p(s_{t+i+1}=A|s_{t+i}=B)$ for $i \geq 1$
- Example: More mobs spawn at night

Markov Processes and Games

Games usually fulfill Markov Property:

- Game state contains all necessary information
- Player actions subsumed into transition function
- Without randomness/unknown actions → deterministic

Characteristics:

- Discrete time = tick/turn-based time
- Often inhomogeneous (player behavior changes)
- Extract agent actions later (Markov Decision Processes)
- Environment often homogeneous over time

Summary

Key Concepts

1. **Spatial Management:**
   - Critical for game scalability
   - Various techniques: zoning, indexing, publish-subscribe
   - Trade-off: update/query processing time
2. **Game Architecture:**
   - Separation of concerns (MVC pattern)
   - AI integrates with core systems
   - Physics and spatial management crucial
3. **Temporal Models:**
   - Turn-based, transaction, tick systems
   - Each has specific use cases
   - Synchronization challenges
4. **Stochastic Modeling:**
   - Games as Markov processes
   - Basis for data modeling

- Transition functions describe mechanics and behavior

5. **Performance Considerations:**
   - Spatial queries grow quadratically
   - Index structures must justify overhead
   - Throw-away indices for volatile data

These concepts form the foundation for understanding and implementing AI systems in modern games, balancing performance, scalability, and gameplay requirements.