Chapter 2: The Game Core
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

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

- MMO-Server: Managing the game state / no visualization necessary
- MMO-Client: Parts of the game state / but need for I/O and visualization. Supports the implementation of different clients for the same game (different quality of graphics)
Game State

All data representing the current state of the game
• object, attribute, relationship, ...
  (compare ER or UML models)
• models all alterable 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
• environmental 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 are not accessible 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 system rules
- game Core organizes their Creation via:
  - player (user input)
  - control of NPCs (AI-control)
  - environmental model

Example:

- a ball is placed on a slope
- environmental model decides that the ball will roll assisted by the physics engine
- action, that changes position and state of the ball (acceleration), is 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 a one player in chess)
  - delaying (processing an action as soon as it is valid)

=> solutions strongly depend on the game principle
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
- realized by loops, state machine, ...
- no concurrency possible
- **examples:**
  - Chess
  - Civilization
  - Settlers of Catan
  - Turn-based RPGs

**Disadvantages:**
- player attendance is needed
- game principle 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 straightforward

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 (serialisation)
Tick-Systems (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 strongly synchronized

- all actions in one tick count as concurrent.
  (no serialization)
- the next game state is created by a cumulated view of all actions.
  (no isolation)
- model used in rendering because it requires fixed frame rates and concurrent changes.
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)
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:
    - distribution of game state to the persistence layer
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

- permanence: the game state has fixed transaction results and they are (at least partially) handed to the persistence layer.

**Furthermore:**
Transitions have to be consistent to rules of the game. (maintaining integrity)

- static: game state is rule-consistent
- dynamic: transition 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 the same amount of possible actions per time unit

• concurrent actions should be theoretically possible
  (Simulating reality)

• a limit to processing time is necessary for smooth game play

• possible elimination of actions for exceeding the time limit

• 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
  - handling user input(=> P layer actions)
  - calls to NPC AI (=> NPC-actions)
  - calls to the environmental model
  - graphics and sound rendering
  - saving certain game aspects to secondary storage
  - transmitting data to the network
  - update supporting data structures (spatial index structures, graphics-buffer, …)
  - ...

create actions
Implementing a game loop

• one game loop for all tasks:
  • no overhead due to synchronization => efficient
  • poor layering of the architecture: a change in one aspects 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)
    e.g. 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
e.g. 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
    (order of operations ≠ order of actions)
    => **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
  - **disadvantages:**
    - redundant code for the same mechanics
    - Inconsistencies are possible

- Encapsulate parts of action processing in modules ans 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 programmed on the same basis
  - advantages: some designs require changing the scripting engine

**example**: LUA
(http://http://lua-users.org/wiki/ClassesAndMethodsExample)

```lua
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("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, simplifications: 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
- high tick-rates use on the client side because display is available anyway
- on server side mechanics 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 Servers

- 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 a NPC?

![Diagram of Attack-Range and Hit-Box/Range](image-url)
Spatial Queries (1)

Spatial queries (here w.r.t to euclidian distance $IR^2$)

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

- Box-Query
  \[ BQ(x, y) = \{ v \in GS \mid x_1 \leq v_1 \leq y_1 \land x_2 \leq v_2 \leq y_2 \} \]
Spatial Queries (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

\[ NN(q) = \left\{ v \in GS \mid \forall x \in GS : \sqrt{(q_1 - v_1)^2 + (q_2 - v_2)^2} \leq \sqrt{(q_1 - x_1)^2 + (q_2 - x_2)^2} \right\} \]
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 24,000,000 distance computations per second

- **conclusion:** the cost for spatial query processing strongly increases with the size of the game state
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
  - $\varepsilon$-range Join (simultaneously compute all Aols)