02 Systemarchitektur und Design-Prinzipien

Dieses Kapitel widmet sich dem technischen Fundament von Nodges. Die Architektur folgt strikten Prinzipien der **Modularisierung** (`Separation of Concerns`) und **Entkopplung**, um Wartbarkeit, Testbarkeit und Erweiterbarkeit über Jahre hinweg zu gewährleisten.

02.1 Architektur-Überblick: Das "Manager-Orchestrator-Pattern"

Nodges vermeidet monolithischen Code ("God Classes"). Stattdessen implementiert es eine Architektur, die wir als **Manager-Orchestrator-Pattern** bezeichnen.

Die Rolle der `App`-Klasse

Die Klasse `App.ts` ist der **Orchestrator**. Sie enthält kaum eigene Geschäftslogik. Ihre Aufgaben sind:

1. **Bootstrapping**: Initialisierung der 3D-Szene (Three.js) und des DOM.
2. **Dependency Injection**: Instanziierung aller Manager und Injektion der Abhängigkeiten (z.B. bekommt der `RaycastManager` Zugriff auf die `Camera`).
3. **Loop-Management**: Steuerung des zentralen Render-Loops (`requestAnimationFrame`).

Das Manager-Ecosystem

Jeder funktionale Aspekt der Anwendung wird in eine spezialisierte Manager-Klasse gekapselt. Ein Manager kümmert sich um *genau eine* Domäne:

Manager	Verantwortlichkeit
`NodeManager`	Erstellung, Update und Rendering der Knoten (Spheres, InstancedMeshes).
`EdgeObjectsManager`	Verwaltung der Kanten (Lines, Tubes), Handling von Kurvengeometrien.
`LayoutManager`	Berechnung von Positionen (x, y, z), Steuerung von Physik-Simulationen.
`HighlightManager`	Visuelle Effekte (Selektion, Hover, Glow), Verwaltung von Materialzuständen.
`UIManager`	Steuerung des HTML-Overlays (Panels, Ladebalken), Brücke zum DOM.
`RaycastManager`	Mathematische Berechnung von Schnittpunkten Maus ↔ 3D-Welt.
`InteractionManager`	Interpretation von User-Inputs (Klick, Drag) in Aktionen.

Vermeidung von "Spaghetti-Code"

Um eine zu enge Kopplung zu vermeiden, dürfen Manager (mit wenigen Ausnahmen) nicht direkt aufeinander zugreifen. Wenn der `LayoutManager` fertig ist, ruft er nicht direkt `NodeManager.update()` auf. Stattdessen nutzen sie zwei Kommunikationswege:

1. **Shared State** (via `StateManager`)
2. **Events** (via `CentralEventManager`)

02.2 Zentrales Zustandsmanagement (`StateManager`)

Der `StateManager` ist das Herzstück der Reaktivität in Nodges ("Single Source of Truth").

Reaktives State-Design

Ähnlich wie Redux oder Vuex hält der `StateManager` den kompletten relevanten Anwendungszustand in einem zentralen Objekt. Dazu gehören:

- `selectedObject`: Welcher Knoten ist gerade aktiv?
- `hoveredObject`: Wo ist die Maus?
- `graphData`: Die aktuellen Rohdaten.
- `config`: Visuelle Einstellungen (Farben, Größen).

Observer-Pattern

Der Manager implementiert ein **Publish/Subscribe** System. Komponenten können sich auf spezifische State-Änderungen abonnieren.

```
// Beispiel: Der UIManager lauscht auf Selektions-Änderungen
stateManager.subscribe('selection', (newState) => {
    if (newState.selectedObject) {
       uiManager.openPanel(newState.selectedObject);
    } else {
       uiManager.closePanel();
    }
});
```

Batch-Updates & Performance

Jede State-Änderung könnte potenziell teure Render-Updates auslösen. Der `StateManager` unterstützt daher **Batch-Updates**. Mehrere logische Änderungen (z.B. "Selektiere Node A" UND "Setze Kamera-Fokus auf A" UND "Ändere UI-Text") werden gesammelt und lösen nur *ein* Benachrichtigungs-Event ("Tick") aus. Dies verhindert unnötige Zyklen ("Data Thrashing").

02.3 Event-Driven Architecture (`CentralEventManager`)

Der `CentralEventManager` (CEM) ist die Abstraktionsschicht zwischen dem Browser (DOM) und der Applikationslogik.

Abstraktion von Raw Inputs

Der CEM fängt rohe Browser-Events (`mousemove`, `pointerdown`, `keydown`) ab. Er normalisiert diese – z.B. rechnet er Pixel-Koordinaten in relative Screen-Koordinaten um – und reichert sie mit Kontext an. Er fungiert als "Pförtner": Kein anderer Manager sollte direkt `document.addEventListener` aufrufen.

Der globale Event-Bus

Über Input-Events hinaus dient der CEM als systemweiter Event-Bus für entkoppelte Kommunikation ("Fire and Forget").

- Module A feuert: `CEM.emit('DATA_LOADED', { nodes: 500 })`
- Module B (z.B. ein Logging-Service) hört zu: `CEM.on('DATA_LOADED', ...)`

So kann man neue Funktionen hinzufügen (z.B. Analytics), ohne den bestehenden Code (den Loading-Prozess) ändern zu müssen.

02.4 Datenfluss-Diagramme

Der Datenfluss in Nodges ist streng **unidirektional** konzipiert, um Seiteneffekte zu minimieren und den Status deterministisch zu halten.

Flow 1: Daten-Import bis Rendering

Dieser Prozess beschreibt, wie aus einer JSON-Datei ein 3D-Bild wird.

Siehe Diagramm: mermaid_01.mmd

Flow 2: User-Interaktion (Loop of Interaction)

Wie ein Mausklick verarbeitet wird.

Siehe Diagramm: mermaid_02.mmd

02.5 Design-Prinzipien im Detail

1. "Prefer Immutability where possible"

Auch wenn JavaScript Objekte per Reference übergibt, versuchen wir im State-Management, Daten nicht "in place" zu mutieren, sondern neue State-Objekte zu erzeugen. Dies erleichtert das Debugging (Time-Travel) und verhindert "Spooky Action at a Distance".

2. "Graceful Degradation"

Wenn ein Feature auf dem System des Nutzers nicht verfügbar ist (z.B. WebGL 2.0 Features oder Web Workers), sollte die App nicht abstürzen, sondern auf einen einfacheren Modus zurückfallen (z.B. einfacheres Rendering, synchrone Berechnung).

3. "Configuration over Code"

Das System ist hochgradig konfigurierbar. Farben, Größenverhältnisse, Physik-Parameter und Rendering-Optionen sind nicht hardcoded, sondern in Config-Objekten zentralisiert. Dies ermöglicht schnelle Anpassungen (auch durch Designer) ohne Code-Eingriffe.

Ende Kapitel 02