01 Einführung und Projektvision

01.1 Projektidentität

Was ist Nodges?

Nodges (eine Wortschöpfung aus **NO**des und e**DGES**) ist eine fortschrittliche Softwarelösung zur Visualisierung komplexer Netzwerke im dreidimensionalen Raum. Anders als herkömmliche 2D-Graphen nutzt Nodges die dritte Dimension, um dichte Informationsstrukturen zu entflechten und intuitiv erfassbar zu machen.

Das Projekt versteht sich nicht nur als Viewer, sondern als interaktives Analysewerkzeug. Es wurde entwickelt, um die Lücke zwischen abstrakten Datenstrukturen und menschlicher Wahrnehmung zu schließen. Durch den Einsatz modernster Web-Technologien (**Three.js**, **WebGL**) läuft Nodges performant direkt im Browser, ohne dass zusätzliche Software installiert werden muss.

Kernziele

Die Entwicklung von Nodges folgt drei primären Leitlinien:

1. **Räumliche Übersichtlichkeit**: Nutzung des 3D-Raums, um Überlappungen ("Hairball-Effekt"), die in 2D-Graphen häufig auftreten, zu minimieren.
2. **Performance**: Flüssige Darstellung auch bei tausenden von Knoten und Kanten durch Einsatz von Hardware-Instancing (`InstancedMesh`).
3. **Interaktivität**: Jedes Element ist anfassbar. Benutzer sollen den Graphen nicht nur betrachten, sondern ihn durch Selektion, Highlighting und Filterung aktiv explorieren.

Abgrenzung zu 2D-Lösungen

Während 2D-Lösungen (wie D3.js oder Cytoscape.js) hervorragend für hierarchische oder kleinere Netze geeignet sind, stoßen sie bei hochgradig vernetzten Daten schnell an Grenzen. Nodges positioniert sich hier als spezialisierte Lösung für Szenarien, in denen die Topologie komplex ist und die räumliche Anordnung (Layout) eine entscheidende Rolle für das Verständnis spielt.

01.2 Features und Funktionsumfang

Nodges bietet eine reichhaltige Palette an Funktionen, die in fünf Kernbereiche unterteilt sind:

Interaktive 3D-Welt

- **Orbit-Navigation**: Freie Rotation, Zoom und Pan-Bewegungen ermöglichen die Betrachtung aus jedem Winkel.
- **Raycasting-Selektion**: Präzise Auswahl von Objekten im 3D-Raum per Mausklick.
- **Fokus-System**: Automatische Kamerafahrten zu ausgewählten Knoten.

Visuelle Feedback-Systeme

- **Highlighting**: Ein ausgeklügeltes System mit 5 Modi (`HOVER`, `SELECTION`, `SEARCH`, `PATH`, `GROUP`) sorgt für sofortiges visuelles Feedback.
- **Glow-Effekte**: Animierte, pulsierende Emissionen lenken die Aufmerksamkeit des Benutzers auf relevante Bereiche.
- **Dynamische Materialien**: Farben und Transparenzen passen sich dem Status (z.B. aktiv/inaktiv) an.

Layout-Automatisierung

Eine integrierte Layout-Engine berechnet Positionen basierend auf Algorithmen:

- **Physik-Simulationen** (Force-Directed) für organische Strukturen.
- **Geometrische Anordnungen** (Kreis, Raster, Sphäre, Helix) für strukturierte Daten.
- **Hierarchische Layouts** für Baumstrukturen.

Daten-Flexibilität

- Unterstützung für das klassische **Legacy-Format** (einfache Node/Edge-Listen).
- Umfassendes **Future-Format** mit Metadaten, Typisierung und visuellen Mappings.
- Echtzeit-Validierung beim Import.

01.3 Technologie-Stack und Voraussetzungen

Nodges basiert auf einem modernen, typ-sicheren Tech-Stack:

Core Technologies

- **Language**: **TypeScript** (v5.3.3) - Für robuste, wartbare Codebasis und Typsicherheit.
- **Rendering**: **Three.js** (v0.161.0) - Der Industriestandard für WebGL. Nodges nutzt fortgeschrittene Features wie `BufferGeometry`, `InstancedMesh` und Custom Shader Materials.
- **Build Tool**: **Vite** (v5.1.4) - Für extrem schnelle Entwicklungszyklen (HMR) und optimierte Production-Builds.

Datenintegrität & Validierung

- **Zod** (v3.22.4): Fungiert als "Gatekeeper". Jede importierte Datei wird gegen ein striktes Schema validiert. Dies verhindert Laufzeitfehler (`undefined is not an object`) und garantiert, dass die internen Datenstrukturen immer konsistent sind. Zod generiert zudem automatisch die TypeScript-Typen für die Datenmodelle.

User Interface (UI)

- **HTML/CSS**: Das Overlay-UI (Panels, Buttons) ist in sauberem, performantem Vanilla HTML/CSS gehalten, um den Rendering-Thread der 3D-Engine nicht zu blockieren.
- **lil-gui** (v0.19.1): Für Entwickler-Tools und Parameter-Steuerung (z.B. Layout-Kräfte justieren) wird diese leichtgewichtige Library eingesetzt.

01.4 Anwendungsbereiche

Nodges ist branchenneutral konzipiert, eignet sich aber besonders für:

- **IT-Sicherheit & Netzwerkanalyse**: Visualisierung von Server-Kommunikation, Angriffspfaden oder Botnet-Strukturen.
- **Wissensmanagement**: Darstellung von Knowledge Graphs, Ontologien und semantischen Beziehungen.
- **Software-Architektur**: Visualisierung von Microservices, Abhängigkeiten zwischen Modulen oder Datenbank-Schemata.
- **Bioinformatik**: Darstellung von Protein-Interaktions-Netzwerken oder metabolischen Pfaden.
- **Social Network Analysis**: Untersuchung von Communities, Influencern und Informationsflüssen in sozialen Gruppen.

Ende Kapitel 01

02 Systemarchitektur und Design-Prinzipien

Dieses Kapitel widmet sich dem technischen Fundament von Nodges. Die Architektur folgt strikten Prinzipien der Modularisierung und Entkopplung, um Wartbarkeit und Erweiterbarkeit zu gewährleisten.

02.1 Architektur-Überblick

Nodges implementiert eine Architektur, die stark auf dem **Manager-Pattern** basiert. Anstatt einer monolithischen Applikationslogik werden Verantwortlichkeiten in spezialisierte Manager-Klassen ausgelagert. Die Klasse `App` fungiert hierbei als Orchestrator und Einstiegspunkt, enthält aber selbst nur minimale Geschäftslogik.

Das Manager-Ecosystem

Jeder Aspekt der Anwendung wird von einem dedizierten Manager verwaltet:

- `NodeObjectsManager` / `EdgeObjectsManager`: Verwalten die 3D-Repräsentation.
- `LayoutManager`: Berechnet Positionen.
- `HighlightManager`: Steuert visuelle Effekte.
- `UIManager`: Verwaltet das HTML-Overlay.

Dependency Injection & Kopplung

Die `App`-Klasse initialisiert alle Manager und injiziert notwendige Abhängigkeiten (wie `scene`, `camera` oder andere Manager) in deren Konstruktoren. Dies ermöglicht eine klare Hierarchie. Um eine zu enge Kopplung ("Spaghetti-Code") zu vermeiden, kommunizieren Manager untereinander primär über zwei Mechanismen:

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

02.2 Zentrales Zustandsmanagement (`StateManager`)

Der `StateManager` ist das Herzstück der Reaktivität in Nodges. Er folgt dem **Single Source of Truth** Prinzip.

Reaktives State-Design

Der gesamte Anwendungszustand (Selektion, Hover, UI-Sichtbarkeit, Einstellungen) wird in einem zentralen `State`-Objekt gehalten. Manager halten keinen redundanten Zustand, sondern lesen diesen aus dem `StateManager` oder abonnieren Änderungen.

Observer-Pattern (Subscribe/Notify)

Der `StateManager` implementiert ein klassisches Observer-Pattern:

```
// Beispiel: Ein Manager abonniert Änderungen
stateManager.subscribe((state) => {
    if (state.selectedObject) {
       // Reagiere auf Selektion
```

```
    }
}, 'categoryName');
```

Änderungen am State werden über `update()` oder `batchUpdate()` ausgelöst, was automatisch alle Subscriber benachrichtigt. Dies entkoppelt den Auslöser einer Änderung (z.B. User klickt) von der Reaktion (z.B. Info-Panel öffnet sich).

Batch-Updates

Um unnötige Render-Zyklen oder Rechenoperationen zu vermeiden, unterstützt der `StateManager` atomare Batch-Updates. Mehrere Änderungen (z.B. "Selektiere Objekt X" UND "Öffne Info-Panel" UND "Aktiviere Glow") werden gesammelt und lösen nur *eine* Benachrichtigung an die Subscriber aus.

02.3 Event-Driven Architecture (`CentralEventManager`)

Der `CentralEventManager` (CEM) abstrahiert die rohen Browser-Events und bietet eine einheitliche Schnittstelle für Interaktionen.

Entkopplung von Input und Logik

Anstatt dass jeder Manager eigene `addEventListener` am DOM registriert, laufen alle Inputs (Maus, Tastatur, Touch) über den CEM. Dieser normalisiert die Events und reichert sie mit Kontext an (z.B. "Welches 3D-Objekt wurde getroffen?").

Event-Bus System

Der CEM fungiert auch als globaler Event-Bus. Komponenten können Custom Events publishen (`publish('PROJECT_LOADED', data)`) und subscriben. Dies ist besonders nützlich für lose gekoppelte Systemteile, die nicht direkten Zugriff aufeinander haben sollen.

Interaktions-Pipeline

1. **Raw Input**: Browser feuert `mousemove` oder `click`.
2. **Normalization**: CEM berechnet relative Koordinaten.
3. **Contextualization**: Raycaster ermittelt getroffenens 3D-Objekt.
4. **Distribution**: CEM benachrichtigt StateManager oder feuert spezifische Events.
5. **Debouncing**: Hover-Events werden gedrosselt (z.B. 100ms), um Performance-Spikes bei schnellen Mausbewegungen zu verhindern.

02.4 Datenfluss-Diagramme

Der Datenfluss in Nodges ist unidirektional konzipiert, um Seiteneffekte zu minimieren.

Flow: Daten-Import bis Rendering

```
graph LR
    JSON[JSON Data] -->|Load| DP[DataParser]
    DP -->|Validate & Normalize| GD[GraphData]
    GD -->|Inject| NOM[NodeObjectsManager] & EOM[EdgeObjectsManager]
```

```
    NOM -->|Generate Geometry| IM_N[InstancedMesh (Nodes)]
    EOM -->|Generate Geometry| IM_E[InstancedMesh (Edges)]
    IM_N & IM_E -->|Add to| Scene[THREE.Scene]
    Scene -->|Render| Canvas
```

Flow: User-Interaktion (Selektion)

```
graph TD
    User[User Click] --> CEM[CentralEventManager]
    CEM -->|Raycast| RC[RaycastManager]
    RC -->|Hit Object| CEM
    CEM -->|Set Selection| SM[StateManager]

    SM -->|Notify| HM[HighlightManager]
    SM -->|Notify| UI[UIManager]
    SM -->|Notify| GE[GlowEffect]

    HM -->|Update Visualization| Scene
    UI -->|Show Details| DOM[InfoPanel]
    GE -->|Animate Intensity| Scene
```

Ende Kapitel 02

03 Datenmanagement und Validierung

Ein robustes Datenmanagement ist essentiell für Nodges, da die Anwendung unterschiedlichste Graph-Strukturen verarbeiten muss. Dieses Kapitel erläutert die Strategie zur Datenhaltung, Validierung und Transformation.

03.1 Das Datenmodell: Legacy vs. Future

Nodges unterstützt historisch bedingt zwei verschiedene JSON-Datenformate. Die Architektur ist so ausgelegt, dass sie beide Formate nahtlos verarbeiten kann, wobei intern alles auf das moderne Format normalisiert wird.

Das Legacy-Format

Dies ist das ursprüngliche Format, das auf Einfachheit ausgelegt war. Es eignet sich hervorragend für schnelle Prototypen oder einfache Graphen.

Struktur:

- **nodes**: Ein flaches Array von Objekten mit `id`, `x`, `y`, `z` Koordinaten und optionalen Attributen.
- **edges**: Ein flaches Array von Verbindungen, definiert durch `start` und `end` (die auf Node-IDs verweisen).

```
{
  "nodes": [
    { "id": 1, "x": 0, "y": 0, "z": 0, "label": "Start" },
    { "id": 2, "x": 10, "y": 5, "z": -2, "label": "End" }
  ],
  "edges": [
    { "start": 1, "end": 2, "type": "connects_to" }
  ]
}
```

Das Future-Format

Das neue Format ist semantisch reicher und flexibler. Es trennt Strukturdaten von Metadaten und visuellen Definitionen. Es orientiert sich an modernen Graph-Datenbank-Exporten und Standards wie GEXF oder GraphML.

Struktur:

1. **data**: Enthält `entities` (Knoten) und `relationships` (Kanten).
2. **metadata**: Autor, Version, Beschreibung, Erstellungsdatum.
3. **dataModel**: Beschreibt das Schema der Eigenschaften (z.B. "weight ist eine Zahl zwischen 0 und 1").
4. **visualMappings**: Definiert Regeln, wie Daten auf visuelle Eigenschaften abgebildet werden (z.B. "Map 'weight' auf 'edge-thickness'").

```
{
  "system": "Network V2",
  "metadata": { "version": "2.0", "author": "User" },
  "data": {
    "entities": [
      { "id": "n1", "type": "server", "position": { "x": 0, "y": 0, "z": 0 } }
    ],
    "relationships": [
      { "source": "n1", "target": "n2", "type": "http_request" }
    ]
  }
}
```

03.2 Schema-Validierung mit Zod

Um die Integrität der importierten Daten zu gewährleisten, setzt Nodges auf **Zod**. Zod ist eine TypeScript-First Schema-Deklarations- und Validierungsbibliothek.

Warum Zod?

In JavaScript/TypeScript sind Daten, die von "außen" kommen (wie ein JSON-Import aus einer Datei), zur Laufzeit typ-unsicher (`any`). Ein einfacher Cast (`as GraphData`) täuscht Sicherheit vor, die nicht existiert. Zod löst dieses Problem durch strikte Laufzeit-Überprüfungen.

Schema-Definitionen (`types.ts`)

In `types.ts` werden Zod-Schemas definiert, die exakt beschreiben, wie valide Daten auszusehen haben.

Beispiel (vereinfacht):

```
const NodeSchema = z.object({
  id: z.union([z.string(), z.number()]),
  x: z.number(),
  y: z.number(),
  z: z.number(),
  // Optionale Felder
  label: z.string().optional()
});

const GraphSchema = z.object({
  nodes: z.array(NodeSchema),
  edges: z.array(EdgeSchema)
});
```

Automatische Typ-Generierung

Ein großer Vorteil von Zod ist, dass TypeScript-Interfaces direkt aus den Schemas abgeleitet werden können:

```
export type NodeData = z.infer<typeof NodeSchema>;
```

Dies garantiert, dass die Validierungslogik und die verwendeten TypeScript-Typen niemals auseinanderlaufen (**Single Source of Truth**).

Error-Handling

Wenn eine importierte Datei nicht dem Schema entspricht (z.B. fehlende Koordinaten, falsche Datentypen), wirft Zod detaillierte Fehler. Nodges fängt diese ab und zeigt dem Benutzer präzise Fehlermeldungen an (z.B. *"Fehler in Datei: In 'nodes[5]' fehlt das Feld 'z'"*), anstatt dass die Anwendung abstürzt oder undefiniertes Verhalten zeigt.

03.3 Der `DataParser`

Der `DataParser` (`src/core/DataParser.ts`) ist die zentrale Komponente, die Rohdaten in das interne Format der Applikation überführt.

Format-Erkennung

Beim Laden einer Datei analysiert der Parser zunächst die Struktur, um das Format zu bestimmen:

- Hat das Objekt `img` und `nodes` Properties? -> Legacy Format.
- Hat es `data.entities` Properties? -> Future Format.

Normalisierungs-Pipeline

Unabhängig vom Eingabeformat ist das Ziel der `App`, immer mit einheitlichen Datenstrukturen zu arbeiten.

1. **Legacy-Input**: Wird in temporäre Future-Strukturen konvertiert.
    - `nodes` -> `entities` (Typ wird auf 'default' gesetzt).
    - `edges` -> `relationships`.
2. **Future-Input**: Wird validiert und ggf. mit Default-Werten angereichert.
3. **ID-Management**: Da IDs Strings oder Numbers sein können, werden sie intern konsistent zu Strings normalisiert, um Lookup-Maps (`Map<string, Node>`) effizient nutzen zu können.

Konvertierung für visuelle Komponenten

Nach der Daten-Normalisierung bereitet der Parser die Daten für die `ObjectManager` auf. Er extrahiert Positionsdaten und Attribute, die für das Rendering relevant sind, und stellt sicher, dass keine "Dangling Edges" existieren (Kanten, die auf nicht existierende Knoten verweisen).

Ende Kapitel 03

04 3D-Rendering und Szenen-Management

Die grafische Darstellung ist das Kernstück von Nodges. Dieses Kapitel behandelt die Implementierung der 3D-Engine mittels Three.js und die verwendeten Strategien zur effizienten Darstellung grosser Graphen.

04.1 Three.js Integration

Nodges nutzt **Three.js** als WebGL-Abstraktionslayer. Der Renderer wird in der `App`-Klasse initialisiert und verwaltet.

Szenen-Setup (`App.initThreeJS`)

- **Scene**: Der Container für alle 3D-Objekte.
- **Camera**: Eine `PerspectiveCamera` simuliert das menschliche Sehfeld (FOV 75°). Sie wird so positioniert, dass der Graph initial vollständig sichtbar ist.
- **Renderer**: Der `WebGLRenderer` zeichnet die Szene in ein HTML5-Canvas. Anti-Aliasing ist aktiviert, um glatte Kanten zu gewährleisten.
- **Controls**: Die `OrbitControls` ermöglichen dem Benutzer, die Kamera um den Mittelpunkt des Graphen zu drehen, zu zoomen und zu verschieben.

Beleuchtungskonzept

Um Tiefe und Dreidimensionalität zu erzeugen, wird ein kombiniertes Beleuchtungssetup verwendet:

1. **AmbientLight**: Sorgt für eine Grundhelligkeit, damit Schattenbereiche nicht komplett schwarz sind.
2. **DirectionalLight**: Simuliert Sonnenlicht und erzeugt Schatten und Glanzlichter (Specular Highlights) auf den Knotenoberflächen, was deren Form (z.B. Kugelrundung) betont.
3. **PointLights**: Werden dynamisch eingesetzt, z.B. bei Glow-Effekten (optional).

Post-Processing

Die Architektur ist für Post-Processing vorbereitet (z.B. Bloom-Effekte für stärkeres Leuchten), aktuell werden visuelle Effekte jedoch primär über Material-Eigenschaften (`emissive`) gelöst, um die Performance hoch zu halten.

04.2 Knoten-Visualisierung (`NodeObjectsManager`)

Der `NodeObjectsManager` ist für die Erstellung und Verwaltung der Knoten (Nodes) verantwortlich. Da ein Graph tausende Knoten enthalten kann, ist Performance hier der kritische Faktor.

InstancedMesh für maximale Performance

Anstatt für jeden Knoten ein eigenes `THREE.Mesh`-Objekt zu erstellen (was tausende Draw-Calls verursachen würde), nutzt Nodges **InstancedMesh**.

- **Prinzip**: Eine Geometrie (z.B. eine Kugel) und ein Material werden nur *einmal* an die GPU gesendet.
- **Instancing**: Die Position, Skalierung und Rotation für tausende Kopien werden in einem einzelnen Buffer übergeben.

- **Vorteil**: Die Grafikkarte kann 10.000+ Knoten in einem einzigen Draw-Call zeichnen, was die Framerate auch bei grossen Netzwerken stabil bei 60 FPS hält.
- **Herausforderung**: Individuelle Eigenschaften (wie Farbe) müssen über Instanz-Attribute oder Texture-Maps verwaltet werden, nicht über einfache Material-Änderungen.

Geometrie-Typen

Nodges unterstützt verschiedene Formen zur Repräsentation von Knoten-Typen:

- **Sphere**: Standard für generische Knoten.
- **Box**: Oft für Infrastruktur-Komponenten (Server, Datenbanken).
- **Icosahedron / Octahedron**: Für spezielle Entitäten.

Der Manager verfügt über einen Cache für Geometrien und Materialien, um Speicher zu sparen.

04.3 Kanten-Visualisierung (`EdgeObjectsManager`)

Die Darstellung von Verbindungen (Edges) ist komplexer als die von Knoten, da sie zwei Punkte im Raum verbinden und unterschiedliche Formen annehmen können.

Rendering-Strategien

Der `EdgeObjectsManager` entscheidet intelligent, welche Geometrie verwendet wird:

1. **InstancedMesh (Cylinders)**:
    - **Einsatz**: Für einfache, gerade Verbindungen zwischen zwei Knoten.
    - **Technik**: Ein Einheits-Zylinder wird so skaliert und rotiert (mittels Quaternions), dass er von Punkt A nach Punkt B reicht. Auch hier wird Instancing für Performance genutzt.
2. **TubeGeometry (Curved Lines)**:
    - **Einsatz**: Wenn *mehrere* Kanten zwischen denselben zwei Knoten existieren (Multi-Edges).
    - **Problem**: Gerade Linien würden sich exakt überlagern und wären nicht unterscheidbar.
    - **Lösung**: Es werden Bezier-Kurven (`QuadraticBezierCurve3`) berechnet. Jede zusätzliche Kante erhält einen stärkeren "Bauch" (Offset), sodass alle Verbindungen wie Kabelstränge nebeneinander sichtbar sind.
    - **Performance**: Da Tubes komplexe Geometrien sind, werden sie individuell erstellt (kein Instancing), was bei sehr vielen Multi-Edges rechenintensiver ist.
3. **LineSegments (Debugging/Legacy)**:
    - Einfache Linien (1px breit) werden primär für Debugging oder als Fallback auf schwacher Hardware genutzt.

Dynamische Updates

Wenn sich Knoten bewegen (z.B. durch Layout-Algorithmen), müssen die Kanten nachgezogen werden. Der Manager optimiert dies:

- **Gerade Kanten**: Nur die Matrix-Transformation der Instanzen wird aktualisiert (sehr schnell).

- **Kurvige Kanten**: Die Geometrie-Punkte müssen neu berechnet werden.

Ende Kapitel 04

05 Visuelle Effekte und Feedback-Systeme

Ein intuitives visuelles Feedback ist entscheidend, um dem Benutzer Orientierung in der komplexen 3D-Welt zu geben. Nodges implementiert hierfür ein mehrschichtiges System aus Highlights und Animationen.

05.1 Das Highlight-System (`HighlightManager`)

Der `HighlightManager` ist die zentrale Instanz für alle visuellen Hervorhebungen. Er verwaltet, welches Objekt gerade warum hervorgehoben wird und stellt sicher, dass sich Effekte nicht gegenseitig stören.

Architektur der Highlight-Verwaltung

Das System basiert auf einer **Highlight-Registry** (`Map<Object3D, HighlightData>`). Jedes aktive Highlight wird dort registriert, zusammen mit Metadaten (Typ, Zeitstempel, Original-Material).

- **Vermeidung von Konflikten**: Wenn ein Objekt bereits markiert ist (z.B. durch eine Selektion), verhindert die Registry, dass ein niedriger priorisierter Effekt (z.B. ein Hover-Effekt einer anderen Komponente) den Status überschreibt.
- **Material-Sicherheit**: Bevor ein Objekt visuell verändert wird, legt der Manager ein Backup des originalen Materials an. Beim Entfernen des Highlights wird dieser Originalzustand exakt wiederhergestellt. Dies verhindert "Geister-Effekte", bei denen Objekte versehentlich die Highlight-Farbe behalten.
- **Cleanup**: Eine automatische Bereinigung entfernt Highlights, die nicht mehr gültig sind (z.B. wenn ein Objekt gelöscht wird).

Priorisierung

Effekte haben implizite Prioritäten. Ein `SELECTION`-Effekt ist "stärker" als ein `HOVER`-Effekt. Der Manager sorgt dafür, dass ein selektiertes Objekt seinen Status behält, auch wenn die Maus darüber hin- und herbewegt wird.

05.2 Highlight-Modi im Detail

Das System unterscheidet fünf spezifische Modi, die unterschiedliche semantische Bedeutungen haben und visuell unterscheidbar sind.

1. HOVER-Modus (Temporäre Interaktion)

- **Zweck**: Signalisiert "Interaktivität". Zeigt an, welches Element bei einem Klick ausgewählt würde.
- **Aktivierung**: Automatisch durch Raycasting bei Mausbewegung.
- **Visuell (Nodes)**: Helligkeit +20%, Cyan-transparenter Halo-Umriss, Opacity 0.3.
- **Visuell (Edges)**: Cyan-blauer Umriss (TubeGeometry mit größerem Radius), Opacity 0.8.

2. SELECTION-Modus (Fokus)

- **Zweck**: Dauerhafter Fokus auf ein Element zur Detailansicht. Triggert das Info-Panel.
- **Aktivierung**: Linksklick auf ein Objekt.
- **Visuell**: Grüner Glow (RGB 0,1,0), erhöhte Intensität (0.4-1.0).

- **Besonderheit**: Nutzt eine **animierte Pulsation**, um die Aufmerksamkeit dauerhaft zu binden (siehe 05.3).

3. SEARCH-Modus (Finden)

- **Zweck**: Hervorhebung von Ergebnissen einer Suchanfrage (z.B. "Finde Node 'Server-01'").
- **Aktivierung**: Programmatisch durch Suchfunktion.
- **Visuell**: Hochkontrastierendes Gelb (0xFFFF00) mit Cyan-Glow. Bleibt aktiv, bis die Suche gelöscht wird.

4. PATH-Modus (Analyse)

- **Zweck**: Visualisierung von Verbindungen, z.B. der kürzeste Weg zwischen zwei Knoten.
- **Aktivierung**: Algorithmen (Dijkstra, BFS).
- **Visuell**: Cyan (0x00FFFF) für alle Elemente entlang des Pfades. Ermöglicht das Verfolgen von Linien durch das "Dickicht" des Graphen.

5. GROUP-Modus (Clustering)

- **Zweck**: Kennzeichnung von Communities oder Kategorien.
- **Aktivierung**: Cluster-Algorithmen oder manuelle Gruppierung.
- **Visuell**: Benutzerdefinierte Farbe (Standard: Magenta), die konsistent auf alle Mitglieder der Gruppe angewendet wird.

05.3 Der Glow-Effekt (`GlowEffect`)

Nodges nutzt keine teuren Post-Processing Shader (wie Unreal Bloom) für den Standard-Glow, um Performance auf mobilen Geräten zu sparen. Stattdessen wird ein geometrischer und material-basierter Ansatz verfolgt.

Technik: Emissive Materials & Halos

- **Emissive Property**: Three.js Materialien besitzen eine `emissive` Eigenschaft (Selbstleuchten). Diese wird genutzt, um die Grundfarbe des Objekts zu überstrahlen.
- **Halo-Meshes**: Für den "Schein" um ein Objekt herum (Outline) wird ein separates, transparentes Mesh erzeugt, das etwas größer ist als das Originalobjekt ("Shell"-Technik).

Animierte Pulsation

Aktivierte Objekte (besonders im Selection-Modus) "atmen". Diese Animation wird vom `StateManager` gesteuert.

Funktionsweise:

- Eine `animate()`-Schleife läuft mit 60 FPS.
- Ein Oszillator berechnet die Intensität basierend auf einer Sinus-Kurve oder einem Ping-Pong-Algorithmus (0.0 -> 1.0 -> 0.0).
- Formel: `newIntensity = currentIntensity + deltaTime * π * 0.2 * frequency * direction`.
- Dieser Wert steuert direkt die `emissiveIntensity` des Materials.

Dies erzeugt einen organischen, lebendigen Eindruck, der den Benutzer subtil daran erinnert, welches Objekt gerade aktiv ist.

Ende Kapitel 05

06 Interaktions-Design und Input-Processing

Eine nahtlose Interaktion ist für User Experience in 3D-Anwendungen entscheidend. Dieses Kapitel beschreibt, wie Nodges Benutzereingaben verarbeitet und in Aktionen innerhalb der 3D-Welt übersetzt.

06.1 Raycasting und Picking

Das größte technische Problem bei Interaktionen in 3D ist das "Picking": Wie stellt man fest, welches 3D-Objekt sich unter dem Mauszeiger befindet, der sich nur auf einem 2D-Bildschirm bewegt? Nodges nutzt hierfür **Raycasting**.

Das Prinzip

1. **Mauskoordinaten**: Die 2D-Pixel-Koordinaten (x, y) der Maus werden auf den normalisierten Gerätekoordinatenraum (NDC) von -1 bis +1 umgerechnet.
2. **Strahl aussenden**: Von der Kameraposition wird ein unsichtbarer Strahl (Ray) durch diese Koordinate in die 3D-Szene "geschossen".
3. **Schnittmenge**: Three.js prüft mathematisch, welche Objekte dieser Strahl durchschneidet.
4. **Sortierung**: Die getroffenen Objekte werden nach Entfernung zur Kamera sortiert. Das erste Objekt ist das "Gesehene".

Optimierungen

Da Raycasting rechenintensiv ist (Prüfung gegen tausende Meshes), wendet Nodges Optimierungen an:

- **Bounding Spheres**: Zuerst wird geprüft, ob der Strahl überhaupt die grobe Umhüllung eines Objekts trifft, bevor die exakte Geometrie geprüft wird.
- **Layering**: Raycasting wird nur auf relevante Layer (Knoten, Kanten) angewendet, Dekorations-Elemente werden ignoriert.
- **Schwellenwerte (Thresholds)**: Bei dünnen Linien (Lines) wird ein Toleranzbereich (Threshold) definiert, damit der Benutzer nicht pixelgenau klicken muss.

06.2 Input-Handler (`CentralEventManager`)

Der `CentralEventManager` fängt alle nativen Browser-Events ab und leitet sie in die Logik der Anwendung weiter.

Maus-Interaktionen

- **MouseMove**: Löst kontinuierlich Raycasting aus, um Hover-Zustände zu aktualisieren. Wird "throttled" (gedrosselt, z.B. alle 100ms), um die CPU-Last zu begrenzen.
- **Click**: Differenziert zwischen einem echten "Klick" und dem Ende eines "Drag"-Vorgangs. Wenn sich die Maus zwischen `mousedown` und `mouseup` signifikant bewegt hat, wird *kein* Klick-Event gefeuert (da der User vermutlich die Kamera drehen wollte).
- **ContextMenu**: Rechtsklicks können kontextsensitive Menüs öffnen (aktuell reserviert für zukünftige Features).
- **Wheel**: Steuert den Zoom der Kamera.

Keyboard-Shortcuts

Zur Power-User-Steuerung werden Tastatureingaben überwacht:

- `Leertaste`: Pausiert/Startet die Layout-Simulation.
- `R`: Setzt die Kamera zurück (Reset).
- `H`: Blendet UI-Elemente ein/aus.
- `Esc`: Hebt Selektionen auf oder schließt Panels.

06.3 Kamera-Steuerung

Die Kamera ist das Auge des Benutzers. Eine schlechte Kamerasteuerung führt sofort zu Orientierungsverlust ("Motion Sickness").

OrbitControls Integration

Nodges verwendet standardisierte `OrbitControls`. Diese erlauben eine intuitive Steuerung ähnlich wie in CAD-Software:

- **Orbit (Linke Maus + Drag)**: Drehung um den Mittelpunkt.
- **Pan (Rechte Maus + Drag)**: Verschieben der Kameraebene.
- **Zoom (Mausrad)**: Annäherung an den Fokuspunkt.

Automatische Kamera-Fahrten (Auto-Focus)

Wenn ein Benutzer einen Knoten markiert (z.B. über die Suche), führt die Kamera oft eine sanfte Animation durch:

1. **Ziel-Ermittlung**: Berechnung der Position des Zielknotens.
2. **Interpolation**: Die Kamera "fliegt" über einen Zeitraum von z.B. 1000ms von der aktuellen Position zur neuen, wobei Position und Blickrichtung (Target) interpoliert werden (mithilfe der *Tween.js* Bibliothek).
3. **Easing**: Die Bewegung startet langsam, beschleunigt und bremst am Ende sanft ab (Quadratic Ease-In-Out), um abrupte Bewegungen zu vermeiden.

Ende Kapitel 06

07 Algorithmen und Layout-Engine

Die Art und Weise, wie Knoten angeordnet sind, bestimmt massgeblich die Lesbarkeit eines Graphen. Nodges bietet eine leistungsfähige Layout-Engine, die verschiedene Algorithmen unterstützt und rechenintensive Aufgaben effizient parallelisiert.

07.1 Layout-Manager Architektur

Der `LayoutManager` ist für die Berechnung der Positionen (x, y, z) aller Knoten verantwortlich. Er folgt dem **Strategy Pattern**: Unterschiedliche Algorithmen (Strategien) können zur Laufzeit ausgetauscht werden, ohne dass der Rest der Anwendung angepasst werden muss.

Synchron vs. Asynchron

- **Synchrone Algorithmen**: Einfache geometrische Anordnungen (wie Grid oder Circle) werden direkt im Haupt-Thread berechnet, da sie mathematisch trivial und extrem schnell sind (< 10ms).
- **Asynchrone Algorithmen**: Komplexe Simulationen (wie Force-Directed Layouts), die hunderte Iterationen benötigen, laufen asynchron im Hintergrund, um das UI nicht einzufrieren.

07.2 Web Worker Integration

Physik-Simulationen sind teuer ($O(n^2)$ oder $O(n \log n)$). Wenn sie im Haupt-Thread (Main Thread) laufen würden, würde die Benutzeroberfläche und das Rendering ruckeln oder komplett blockieren.

Multithreading

Nodges lagert diese Berechnungen in **Web Workers** aus. Ein Web Worker ist ein Skript, das in einem separaten Thread läuft.

1. **Bootstrapping**: Beim Start initialisiert der LayoutManager einen Worker.
2. **Daten-Transfer**: Die Knoten- und Kanten-Daten werden an den Worker gesendet (mittels `postMessage`).
3. **Iteration**: Der Worker berechnet einen Simulations-Schritt (Tick).
4. **Synchronisation**: Nach jedem Schritt sendet der Worker die neuen Positionen zurück an den Main Thread.
5. **Rendering**: Der `NodeObjectsManager` aktualisiert die 3D-Szene live. Der Benutzer sieht, wie sich der Graph "entfaltet".

07.3 Implementierte Algorithmen

Nodges stellt eine breite Palette an Algorithmen bereit, um unterschiedliche Datenstrukturen optimal darzustellen.

Force-Directed Layouts

Diese Algorithmen simulieren physikalische Kräfte: Knoten stossen sich ab (wie Magnete), Kanten ziehen verbundene Knoten zusammen (wie Federn).

- **ForceAtlas2 (ähnlich)**: Eignet sich hervorragend für Cluster-Bildung.
- **Fruchterman-Reingold**: Ein Klassiker, der ästhetisch ansprechende, symmetrische Graphen erzeugt.
- **Spring-Embedder**: Einfaches Feder-Masse-Modell.

Vorteil: Deckt organische Strukturen und Cluster automatisch auf.

Strukturelle Layouts

Diese Algorithmen ordnen Knoten deterministisch nach festen Regeln an.

- **Grid (Raster)**: Ordnet Knoten in einem 3D-Würfelraster an. Ideal, um Mengenverhältnisse abzuschätzen.
- **Circular (Kreis)**: Alle Knoten auf einem Ring. Gut für kleine Netzwerke.
- **Spherical (Kugel)**: Verteilt Knoten auf der Oberfläche einer Kugel (Fibonacci Sphere).
- **Helix**: Anordnung in einer Spirale (z.B. für Zeitreihen).

Hierarchische Layouts (Tree)

- Für Daten mit klarer Eltern-Kind-Beziehung (Bäume).
- **3D-Tree**: Nutzt die Z-Achse für die Tiefe oder ordnet Ebenen radial an (Cone Tree).

Ende Kapitel 07

08 Benutzeroberfläche (UI) und UX

Nodges setzt auf ein minimalistisches, funktionales Overlay-Interface, das die 3D-Visualisierung nicht verdeckt, sondern ergänzt.

08.1 UI-Architektur (`UIManager`)

Die UI ist strikt vom 3D-Rendering getrennt. Während Three.js in einem `<canvas>` Element im Hintergrund läuft, liegt die Benutzeroberfläche als HTML/CSS-Layer darüber (`z-index`).

Trennung von Belangen

- **Canvas**: Zuständig für die 3D-Welt.
- **DOM Overlay**: Zuständig für Texte, Buttons, Listen und Formulare. Diese Trennung garantiert, dass UI-Interaktionen (wie Scrollen in einer Liste) performant sind und nicht die Framerate der 3D-Engine beeinflussen, und umgekehrt.

Manager-Verantwortung

Der `UIManager` ist die Brücke zwischen Logik und Darstellung. Er abonniert den `StateManager` und manipuliert das DOM entsprechend:

- `state.selectedObject` ändert sich -> `UIManager` füllt und öffnet das Info-Panel.
- `state.isLoading` ist true -> `UIManager` zeigt den Lade-Spinner.

08.2 Komponenten im Detail

Die Benutzeroberfläche besteht aus modularen Panels.

Info-Panel (Rechts)

Das wichtigste Werkzeug für die Datenanalyse.

- **Dynamischer Content**: Wenn ein Knoten ausgewählt wird, generiert das Panel automatisch eine Tabelle mit allen verfügbaren Attributen (ID, Typ, Metadaten).
- **Kontext-Sensitiv**: Zeigt unterschiedliche Daten für Knoten und Kanten.
- **Persistent**: Bleibt geöffnet, damit Benutzer Daten vergleichen können, kann aber eingeklappt werden.

File-Browser (Links)

Ermöglicht das Laden verschiedener Datensätze.

- **Dateiliste**: Listet JSON-Dateien aus dem `/data` Verzeichnis.
- **Drag & Drop**: (Geplant) Benutzer können eigene Dateien direkt in das Fenster ziehen.

Settings & Controls (lil-gui)

Für technische Einstellungen, die oft während der Entwicklung oder Analyse angepasst werden müssen, wird die Bibliothek `lil-gui` verwendet.

- **Layout-Parameter**: Feder-Stärke, Gravitation, Iterationen.
- **Visuelle Einstellungen**: Glow-Intensität, Kantendicke, Farbschemata.
- **Debug-Tools**: Anzeigen von Hit-Boxen oder Performance-Metriken.

08.3 Visuelles Feedback im UI

Gutes UX bedeutet, den Benutzer nie im Unklaren über den Systemzustand zu lassen.

Lade-Indikatoren

Da das Parsen grosser JSON-Dateien und das Berechnen des Layouts Zeit beanspruchen kann, wird ein prominenter Lade-Indikator angezeigt, der den aktuellen Fortschritt widerspiegelt (z.B. "Loading Geometry...", "Calculating Layout...").

Fehler-Meldungen

Nodges fängt Fehler (z.B. fehlerhaftes JSON) ab und zeigt sie in gut lesbaren "Toast"-Benachrichtigungen oder modalen Fenstern an, anstatt Fehlermeldungen nur in der Browser-Konsole zu verstecken. Dies macht die Anwendung auch für Nicht-Entwickler bedienbar.

Ende Kapitel 08

09 Utilities und Hilfssysteme

Hinter den Kulissen von Nodges arbeiten zahlreiche kleine Hilfssysteme, die den Hauptkomponenten Arbeit abnehmen und für Stabilität sorgen.

09.1 Mathematische Helfer

Three.js bietet bereits starke mathematische Klassen (`Vector3`, `Matrix4`), aber Nodges benötigt zusätzliche Spezialfunktionen für Graphen.

Geometrie-Berechnungen

- **Abstandsmessung**: Effiziente Berechnung von Distanzen (Squared Distance), um Kollisionen im Layout zu vermeiden, ohne teure Wurzelziehen-Operationen (`Math.sqrt`).
- **Kurven-Interpolation**: Für "Multi-Edges" werden Bezier-Kontrollpunkte berechnet, die einen ästhetischen Bogen erzeugen, abhängig von der Entfernung der Knoten.

Farbraum-Transformationen

Visuelles Feedback benötigt oft Farbanpassungen (z.B. "Mach diesen Knoten 20% heller").

- **RGB zu HSL**: Nodges arbeitet intern oft mit HSL (Hue, Saturation, Lightness), da sich Farben so intuitiver manipulieren lassen (z.B. "Sättigung verringern für inaktive Elemente").
- **Farb-Interpolation**: Das Überblenden zwischen Farben (z.B. von Rot nach Grün je nach Knotengewicht) wird durch Helper-Klassen gesteuert.

09.2 Performance-Tools

Performanz ist ein Feature. Nodges überwacht sich selbst.

FPS-Monitoring

Ein integriertes FPS-Meter (Frames Per Second) warnt, wenn die Leistung einbricht. Dies hilft Entwicklern, Performance-Flaschenhälse (wie zu viele transparente Objekte) zu identifizieren.

Caching und Object Pooling

Um den Garbage Collector (GC) von JavaScript zu entlasten, werden Objekte wiederverwendet:

- **Material Cache**: Wenn 500 Knoten rot sind, teilen sie sich *eine* Material-Instanz, anstatt 500 Kopien zu erzeugen.
- **Vector3 Recycling**: Temporäre Vektoren für Berechnungen werden einmal erstellt und immer wieder überschrieben, statt in jedem Frame neu alloziert zu werden.

09.3 Export und Import

Aktuell liegt der Fokus auf dem Import von Daten, aber rudimentäre Export-Funktionen existieren:

- **Screenshot-Generator**: Rendert den aktuellen Canvas-Inhalt in ein hochauflösendes Bild (unter Berücksichtigung des aktuellen Zooms).
- **Positions-Export**: Wenn ein Layout automatisch berechnet wurde, können die finalen Koordinaten exportiert werden, um beim nächsten Laden Rechenzeit zu sparen (Caching).

Ende Kapitel 09

10 Entwicklungs-Guide und Deployment

Dieses Kapitel richtet sich an Entwickler, die Nodges lokal aufsetzen, weiterentwickeln oder deployen möchten.

10.1 Setup und Installation

Voraussetzungen

- **Node.js**: Version 18 oder höher wird empfohlen.
- **npm**: Wird automatisch mit Node.js installiert.

Installation

1. **Repository klonen**:

   ```
   git clone https://github.com/IhrRepo/Nodges.git
   cd Nodges
   ```

2. **Abhängigkeiten installieren**:

   ```
   npm install
   ```

   Dies installiert alle in `package.json` definierten Pakete (Three.js, Typescript, Vite, Zod, etc.).

10.2 Build-Pipeline mit Vite

Nodges nutzt **Vite** als Build-Tool, was signifikante Vorteile gegenüber älteren Tools wie Webpack bietet.

Development Server

```
npm run dev
```

- Startet einen lokalen Server (meist `http://localhost:5173`).
- **HMR (Hot Module Replacement)**: Änderungen am Code werden sofort im Browser reflektiert, ohne die Seite neu laden zu müssen. Der State (z.B. Kameraposition) bleibt dabei oft erhalten.

Production Build

```
npm run build
```

- Kompiliert TypeScript zu JavaScript.
- Bundled und minifiziert den Code.
- Erstellt optimierte Assets im `/dist` Verzeichnis.
- Splitted Code in Chunks für schnelleres Laden (Lazy Loading).

Vorschau des Builds

```
npm run preview
```

Startet einen lokalen Server, der das `/dist` Verzeichnis ausliefert, um den Produktions-Build vor dem Deployment zu testen.

10.3 Code-Style und Best Practices

TypeScript

Nodges verwendet striktes TypeScript (`"strict": true` in `tsconfig.json`).

- **Kein** `any`: Explizite Typen für alle Variablen und Funktionsparameter.
- **Interfaces**: Definition von Datenstrukturen über Interfaces (in `types.ts`).

Modul-Struktur

- **ES Modules**: Nutzung von `import` / `export` Syntax.
- **Dateinamen**: PascalCase für Klassen (`HighlightManager.js`), camelCase für Instanzen und Funktionen.

Linter & Formatter

Es wird empfohlen, **ESLint** und **Prettier** zu verwenden, um einen einheitlichen Code-Stil (Einrückung, Semikolons) sicherzustellen.

Ende Kapitel 10

Future Format & Types – Technische Dokumentation

1. Einleitung und Kontext

Diese Dokumentation analysiert die Implementierung der Typsicherheit in `types.ts` und beschreibt den strategischen Übergang des Nodges-Projekts zu einem semantisch reichhaltigen "Future Format". Sie dient als Leitfaden für Entwickler, um die aktuellen Datenstrukturen zu verstehen, die Rolle der Validierung (Zod) einzuordnen und zukünftige Erweiterungen wie Multi-Target-Edges und Ports zu planen.

---

2. Analyse der `types.ts`

Die Datei `types.ts` ist in logische Sektionen unterteilt, die den hybriden Zustand der Anwendung widerspiegeln.

2.1 Legacy Typen (`NodeData`, `EdgeData`)

Dies ist das Format, das aktuell von der Rendering-Engine (`NodeObjectsManager`, `EdgeObjectsManager`) erwartet wird.

```
export interface NodeData {
    id: string | number;
    x: number; y: number; z: number; // Explizite Koordinaten
    [key: string]: any;              // Erlaubt beliebige Zusatzdaten
}
```

- **Merkmal:** Stark auf die 3D-Darstellung fokussiert (`x, y, z` sind Pflicht).
- **Flexibilität:** Durch `[key: string]: any` extrem flexibel, aber unsicher. Es gibt keine Garantie, welche Eigenschaften existieren.

2.2 Future Format Types (`DataModel`, `VisualMappings`)

Hier beginnt die Abstraktion. Das Future Format trennt **Daten** strikt von der **Darstellung**.

Data Model Schema (`PropertySchema`, `EntityTypeSchema`)

Anstatt Daten einfach als JSON-Objekt zu laden, definiert das Future Format ein Schema für diese Daten.

```
export interface PropertySchema {
    type: 'continuous' | 'categorical' | 'vector' | 'spatial' | 'temporal';
    range?: [number, number];
    unit?: string;
    // ...
}
```

- **Idee:** Das System "weiß", was ein Wert bedeutet. Ein Wert ist nicht einfach eine Zahl `0.5`, sondern z.B. ein 'continuous' Wert zwischen 0 und 1, der eine Intensität darstellt.
- **Vorteil:** Dies ermöglicht eine automatisierte UI (z.B. Slider für `continuous`, Dropdowns für `categorical`).

Entity & Relationship Data

```
export interface EntityData {
    id: string;
    type: string;
    // Position ist hier optional! Das Framework könnte es berechnen.
    position?: { x: number; y: number; z: number };
    [key: string]: any;
}
```

- **Unterschied zu Legacy:** Ein `Entity` ist ein abstraktes Objekt. Es wird erst durch ein `VisualMapping` zu einem sichtbaren 3D-Objekt.

2.3 Visual Mappings

Dies ist der mächtigste Teil des Future Formats. Er definiert, wie Datenattribute in visuelle Eigenschaften übersetzt werden.

```
export interface VisualMapping {
    source: string;              // Welches Feld? z.B. "traffic_load"
    function: MappingFunction;   // Wie mappen? z.B. "heatmap"
    range?: [number, number];    // Zielbereich?
}
```

- **Anwendung:** Wenn `traffic_load` hoch ist, wird die Farbe Rot (via Heatmap-Funktion) oder die Größe skaliert. Im Code (`FutureDataParser.js`) sind Funktionen wie `sphereComplexity`, `pulse` oder `geographic` bereits angedacht.

---

3. Das Future Format: Nutzung, Vor- & Nachteile

Nutzungskonzept

Das Future Format zielt auf ein **"Connect Interface"** ab. Der Nutzer lädt rohe Daten (Entities/Relationships) und definiert *separat*, wie diese aussehen sollen.

1. **Daten laden:** JSON enthält reine Fakten (z.B. "Server A verbunden mit Server B, Latenz: 50ms").
2. **Mapping definieren:** "Latenz" entscheidet über die Dicke der Kante. "Server-Typ" entscheidet über die Farbe der Node.

Vor- und Nachteile

Aspekt	Legacy Format	Future Format
Simplicity	Hoch. Direkte `x,y,z` Werte, einfach zu parsen.	Niedrig. Erfordert Parsing- und Mapping-Schritte.
Semantik	Keine. Daten sind nur "Props".	Hoch. Datentypen sind bekannt (`spatial`, `temporal`).
Flexibilität	Begrenzt. Neue Visualisierungen erfordern Code-Änderungen.	Extrem hoch. Visualisierung ist Konfiguration, kein Code.
Trennung	Vermischt (Daten & View in einem Objekt).	Sauber getrennt (Datenmodell vs. VisualMappings).

Alternativen

- **Graphology:** Eine Standard-Bibliothek für Graphen in JS. Vorteil: Standardisiert. Nachteil: Weniger Fokus auf 3D-Visualisierungs-Mappings.
- **GEXF (Graph Exchange XML Format):** Ein XML-Standard. Nachteil: XML ist veraltet und mühsam in Web-Apps zu parsen. Das JSON-basierte Future Format ist moderner.

---

4. Exkurs: Zod - Sinn und Zweck

Im Projekt ist **Zod** als Dependency vorhanden, wird jedoch im aktuellen Kern-Code noch nicht voll ausgeschöpft.

Was ist Zod?

Zod ist eine TypeScript-First Schema-Deklarations- und Validierungsbibliothek.

Warum Zod im Future Format essentiell ist

Aktuell (siehe `core/DataParser.ts`) wird "Duck Typing" verwendet:

```
// Manuelle Prüfung - Fehleranfällig und verbos
if (data.nodes !== undefined || data.edges !== undefined) { ... }
```

Mit Zod definiert man das Schema einmal und erhält Validierung **und** Typen automatisch:

```
import { z } from "zod";

// Definition des Schemas
const EntitySchema = z.object({
  id: z.string(),
  type: z.string(),
  position: z.object({
    x: z.number(),
    y: z.number(),
```

```
    z: z.number()
  }).optional(),
}).passthrough(); // WICHTIG: Erlaubt unbekannte Eigenschaften!

// Automatische Validierung zur Laufzeit
const result = EntitySchema.safeParse(loadedJson);
if (!result.success) {
  console.error("Invalid format:", result.error);
}
```

Vorteile von Zod

1. **Runtime Safety:** TypeScript Typen verschwinden beim Kompilieren. Zod prüft das *tatsächliche* JSON zur Laufzeit.
2. **Single Source of Truth:** Man leitet den TypeScript-Typ aus dem Zod-Schema ab (`type Entity = z.infer<typeof EntitySchema>`).
3. **Parsen von unbekannten Strukturen:** Zod kann unbekannte Keys strippen oder behalten (`passthrough`), was für das dynamische Datenmodell von Nodges entscheidend ist.

---

5. Einbettung des JSON-Ladens im Code

Der aktuelle Datenfluss im Projekt ist wie folgt implementiert:

1. **Initialisierung (`App.ts`):** `loadDefaultData` oder `loadData(url)` ruft `fetch` auf und übergibt das Ergebnis an den Parser.
2. **Parsing (`core/DataParser.ts`):** Der Parser normalisiert Legacy- und Future-Daten in eine interne `GraphData` Struktur.
3. **Konvertierung für Rendering (`App.ts`):** Die sauberen `EntityData` Objekte werden temporär zurück in flache Legacy-Objekte (`x, y, z` auf Top-Level) konvertiert, damit `NodeObjectsManager` sie verarbeiten kann.
4. **Verarbeitung (`FutureDataParser.js`):** Enthält Logik für semantisches Mapping (`continuous` -> `heatmap`), die sukzessive in den Kern integriert werden soll.

---

6. Machbarkeitsanalyse: Unbekannte Eigenschaften

Inwiefern können eine unbekannte Anzahl Eigenschaften von Edges und Nodes gespeichert werden?

Das Problem

TypeScript mag statische Typen, JSON-Daten von Nutzern sind dynamisch (`customField_1`, `server_status`, etc.).

Die Lösung

1. **TypeScript:** Nutzung von Index Signatures in Interfaces: `[key: string]: any;`.
2. **Zod (Laufzeit):** Nutzung von `.passthrough()`. Das Schema validiert die *bekannten* Felder (id, type) streng, lässt aber alle zusätzlichen Felder passieren, ohne einen Fehler zu werfen.

Fazit: Es ist technisch absolut machbar. Der Konflikt wird durch die bewusste Konfiguration des Parsers (`passthrough`) gelöst.

---

7. Zukunftsszenarien: Erweiterung des Modells

7.1 One-to-Many Edges (Hyperedges)

Eine Edge könnte eine Liste von Targets haben (Broadcast/Multicast).

Erforderliche Änderungen in `types.ts`:

```
interface HyperEdgeData {
    id: string;
    source: string;
    targets: string[]; // Array statt einzelner String
    type: 'broadcast' | 'multicast';
}
```

Eine solche Änderung erfordert ein Update im `EdgeObjectsManager`, um solche Edges entweder als mehrere Linien oder als verzweigte Geometrie darzustellen.

7.2 Ports (Node -> Port -> Edge -> Port -> Node)

Nodes sind Komponenten mit Anschlüssen. Eine Kante geht von "Port A an Node 1" zu "Port B an Node 2".

Erforderliche Änderungen in `types.ts`:

```
interface Port {
    id: string;
    nodeId: string;
    localPosition: { x: number; y: number; z: number }; // Relativ zur Node
}

interface PortConnection extends RelationshipData {
    sourcePort: string; // Referenz auf Port ID statt nur Node ID
    targetPort: string;
}
```

Dies erfordert Anpassungen im Datenmodell (Ports als Sub-Entities) und im Rendering (Berechnung der Weltkoordinaten der Ports vor dem Zeichnen der Kanten).