Nodges - Technische Dokumentation

Einleitung

Willkommen zur technischen Dokumentation für **Nodges**, einer modularen 3D-Netzwerkvisualisierungs-Anwendung. Dieses Dokument dient als umfassendes Nachschlagewerk für Entwickler, Architekten und technisch Interessierte, die die Funktionsweise, Architektur und die internen Prozesse der Anwendung verstehen möchten.

Die Dokumentation ist in mehrere Hauptkapitel unterteilt:

1. **Architektur-Übersicht**: Ein deklaratives Kapitel, das die grundlegende, modulare Architektur der Anwendung beschreibt und die Interaktion der Kernkomponenten visualisiert.
2. **Datenfluss & Rendering-Pipeline**: Detaillierte Analyse des Prozesses vom Laden einer JSON-Datei bis zur Darstellung der 3D-Objekte auf dem Bildschirm.
3. **Kernkomponenten im Detail**:
    - StateManager
    - InteractionManager
    - LayoutManager
    - HighlightManager
    - UIManager
4. **Konzept für die Zukunft: Versioniertes Datenmodell**: Siehe Entwurf
5. **Anhang: Diagramm-Quelldateien**:
    - Architektur-Diagramm
    - Datenfluss-Diagramm

(Hinweis: Die ursprüngliche Datenformat-Spezifikation aus `doc/` wird hier nicht erneut dupliziert, sondern die Konzepte werden in den relevanten Kapiteln erläutert.)

---

1. Architektur-Übersicht

Die Nodges-Anwendung basiert auf einer klaren, modularen Architektur, die auf dem Prinzip der **Trennung der Zuständigkeiten (Separation of Concerns)** beruht. Das Herzstück ist eine zentrale `NodgesApp`-Klasse, die als Orchestrator für eine Reihe von spezialisierten **Managern** fungiert. Jeder Manager ist für einen spezifischen Aspekt der Anwendung verantwortlich.

Dieses Design bietet mehrere Vorteile:

- **Wartbarkeit**: Änderungen in einem Bereich (z.B. der UI) haben minimale Auswirkungen auf andere Bereiche (z.B. die Layout-Berechnung).
- **Erweiterbarkeit**: Neue Funktionen, wie z.B. ein neuer Layout-Algorithmus oder ein neuer visueller Effekt, können durch das Hinzufügen oder Erweitern eines Managers einfach integriert werden.
- **Testbarkeit**: Jeder Manager kann isoliert getestet werden.

Die Kernkomponenten

Die Architektur lässt sich am besten durch das Zusammenspiel der folgenden Hauptkomponenten verstehen:

- **NodgesApp**: Die Hauptklasse, die alles initialisiert und den globalen Anwendungskontext hält.
- **StateManager**: Das zentrale Nervensystem. Er hält den gesamten Anwendungszustand (z.B. welches Objekt ist ausgewählt) und informiert andere Komponenten über Änderungen mittels eines Publish/Subscribe-Musters.
- **CentralEventManager**: Fängt alle rohen Browser-Events (Mausklicks, Bewegungen, Tastatureingaben) ab und leitet sie als semantische Events (z.B. "Objekt geklickt") weiter.
- **InteractionManager**: Die "Hände" der Anwendung. Er lauscht auf die semantischen Events und entscheidet, was zu tun ist (z.B. ein Objekt auswählen). Er ändert den Zustand, indem er den `StateManager` aktualisiert.
- **LayoutManager**: Der "Architekt". Er berechnet die 3D-Positionen der Knoten basierend auf verschiedenen Graphen-Layout-Algorithmen.
- **UIManager**: Verantwortlich für die Steuerung und Darstellung aller HTML-basierten UI-Elemente wie Info-Panels und Buttons.
- **HighlightManager**: Der "Maler". Er reagiert auf Zustandsänderungen im `StateManager` und wendet visuelle Effekte (Farben, Glühen) auf die 3D-Objekte an.
- **Daten-Objekte (`Node`, `Edge`, `NodeObjects`):** Klassen, die die Datenstruktur und die Erzeugung der visuellen 3D-Objekte (Meshes) definieren.

Architektur-Diagramm

Das folgende Diagramm visualisiert das Zusammenspiel dieser Komponenten. Es zeigt, wie die Manager um den zentralen `StateManager` herum organisiert sind und wie die Daten und Events fließen.

Siehe Architektur-Übersicht

Meine Meinung & Verbesserungsideen zur Architektur

Die aktuelle Architektur ist bereits sehr robust und gut durchdacht. Die Trennung in Manager ist vorbildlich. Dennoch gibt es Potenzial für Verfeinerungen:

1. **Striktere Entkopplung vom DOM:** Der `InteractionManager` und teilweise auch die `NodgesApp` greifen noch direkt auf DOM-Elemente zu (z.B. für das Info-Panel). Nach meinem jüngsten Refactoring ist die UI-Logik zwar im `UIManager` zentralisiert, aber für eine noch sauberere Architektur könnte der `UIManager` eine API bereitstellen (z.B. `uiManager.showNodeDetails(nodeData)`), die von anderen Managern aufgerufen wird, anstatt dass diese die DOM-Manipulation selbst anstoßen.
2. **Service Locator / Dependency Injection:** Aktuell werden die Manager in der `NodgesApp` manuell instanziiert und teilweise untereinander referenziert. Die Einführung eines einfachen Service Locators oder eines Dependency Injection Containers könnte die Erstellung und den Zugriff auf die Manager weiter vereinfachen und Abhängigkeiten expliziter machen.
3. **Event-Typisierung:** Die Events im `CentralEventManager` werden als Strings übergeben. Die Verwendung von Enums oder Konstanten für Event-Namen (`Events.HOVER_START`) würde Tippfehler verhindern und die Code-Intelligenz in der IDE verbessern.

Nodges - Architektur-Diagramm

Dieses Diagramm zeigt die high-level Architektur der Nodges-Anwendung und das Zusammenspiel der Kernkomponenten.

Siehe Architektur-Diagramm

Interpretation des Diagramms

1. **User Input & Event Handling (Gelb -> Rot):**
    - Der Benutzer interagiert mit der Anwendung (`User Input`).
    - Der `CentralEventManager` fängt diese rohen DOM-Events ab und übersetzt sie in bedeutungsvolle, anwendungsinterne Events.
2. **Interaktion & Zustandsverwaltung (Rot -> Dunkelviolett):**
    - Der `InteractionManager` verarbeitet diese anwendungsinternen Events.
    - Anstatt die Szene direkt zu manipulieren, ändert er den zentralen Zustand, indem er den `StateManager` aktualisiert. Der `StateManager` ist das Herzstück der Anwendung (dicker Rand).
3. **Reaktion auf Zustandsänderungen (Dunkelviolett -> Violett/Türkis):**
    - Der `StateManager` benachrichtigt alle seine "Subscriber" über die Zustandsänderung.
    - Der `HighlightManager` (der "Maler") reagiert, indem er visuelle Effekte auf die 3D-Objekte in der Szene anwendet.
    - Der `UIManager` reagiert, indem er die HTML-Panels mit neuen Informationen aktualisiert.
4. **Daten und Initialisierung (Grün -> Grau -> Blau):**
    - Der Prozess beginnt damit, dass die `NodgesApp` (der Orchestrator) eine `JSON`-Datei lädt.
    - Mit diesen Daten ruft sie den `LayoutManager` auf, um die Positionen zu berechnen, und erstellt dann die eigentlichen 3D-Objekte (`Node Meshes`, `Edge Meshes`), die zur Szene hinzugefügt werden.

Nodges - Datenfluss und Rendering-Pipeline

Dieses Dokument beschreibt den detaillierten Prozess, wie aus einer rohen JSON-Datei ein interaktives 3D-Netzwerk auf dem Bildschirm wird.

Prozess-Diagramm

Das folgende Diagramm zeigt den sequenziellen Ablauf von Funktionsaufrufen und Datentransformationen, die beim Laden eines neuen Netzwerks durchlaufen werden.

Siehe Datenfluss-Diagramm

Phasen der Pipeline

1. **Auslösung (User -> UIManager -> NodgesApp):**

- Der Prozess beginnt mit einer Benutzeraktion, z.B. dem Klick auf einen Button im `FilePanel`.
- Der `UIManager`, der für die Panels zuständig ist, fängt diesen Klick ab und ruft die zentrale `loadData(url)`-Methode in der `NodgesApp`-Klasse auf.

2. **Datenbeschaffung und -bereinigung (NodgesApp):**

   - `loadData` holt die angeforderte JSON-Datei über einen `fetch`-Aufruf.
   - Unmittelbar danach wird `clearScene()` aufgerufen, um alle existierenden 3D-Objekte (Knoten und Kanten) aus der vorherigen Visualisierung aus der Three.js-Szene zu entfernen und deren Speicher freizugeben.

3. **Daten-Normalisierung (NodgesApp & FutureDataParser):**

   - Der Code prüft das Format der geladenen JSON-Datei.
   - **Zukunftsformat:** Wenn die Daten eine Struktur wie `data.entities` haben, wird der `FutureDataParser` verwendet, um diese in das interne `nodes`/`edges`-Standardformat zu konvertieren.
   - **Standardformat:** Andernfalls wird das `nodes`/`edges`-Array direkt verwendet. Hier findet eine wichtige Normalisierung statt: Kanten, die im `source`/`target`-Format mit String-IDs vorliegen, werden in das intern verwendete `start`/`end`-Format mit numerischen Indizes umgewandelt.

4. **Layout-Berechnung (LayoutManager):**

   - Die Anwendung prüft, ob die Knoten in den geladenen Daten bereits Positionsinformationen (`x`, `y`, `z`) besitzen.
   - Falls **nicht**, wird der `LayoutManager` aufgerufen, um die Positionen zu berechnen. Standardmäßig wird hier ein Force-Directed-Algorithmus verwendet. Das Ergebnis ist, dass jeder Knoten im `currentNodes`-Array nun über `x`, `y`, und `z` Koordinaten verfügt.

5. **3D-Objekt-Erstellung (Rendering):**

   - **Knoten (`createNodes`):** Die `NodgesApp` iteriert über das `currentNodes`-Array. Für jeden Datensatz wird die `NodeObjects`-Factory aufgerufen, um das entsprechende `THREE.Mesh`-Objekt (Würfel, Kugel etc.) zu erstellen. Jedes erstellte Mesh wird direkt der Three.js-Szene hinzugefügt.
   - **Kanten (`createEdges`):** Danach iteriert die App über das `currentEdges`-Array. Für jede Kante wird eine neue Instanz der `Edge`-Klasse erstellt. Der Konstruktor der `Edge`-Klasse erzeugt direkt die gebogene 3D-Röhre (`tube`) als `THREE.Mesh` und gibt sie zurück. Auch diese wird sofort der Szene hinzugefügt.

6. **UI-Aktualisierung (NodgesApp -> UIManager):**

   - Nachdem die 3D-Szene aufgebaut ist, ruft die `NodgesApp` die `updateFileInfo`-Methode des `UIManager` auf.
   - Der `UIManager` aktualisiert daraufhin die HTML-Elemente in den Info-Panels, um die Metadaten des neuen Netzwerks (Dateiname, Anzahl der Knoten/Kanten etc.) anzuzeigen.

Komponente im Detail: StateManager

Rolle in der Architektur

Der `StateManager` ist die **zentrale Wahrheit** ("Single Source of Truth") der Nodges-Anwendung. Er agiert als zentrales Nervensystem und entkoppelt die verschiedenen Anwendungsmodule voneinander. Anstatt dass die Manager direkt miteinander kommunizieren, ändern sie den Zustand im `StateManager` oder reagieren auf dessen Änderungen.

Hauptverantwortlichkeiten:

1. **Zustandsspeicherung:** Er hält alle ephemeren (kurzlebigen) Zustände der Anwendung in einem einzigen, zentralen `state`-Objekt.
2. **Zustandsaktualisierung:** Er bietet eine `update`-Methode, um Teile des Zustands sicher zu ändern.
3. **Benachrichtigungssystem (Publish/Subscribe):** Er benachrichtigt alle angemeldeten "Subscriber" (andere Manager), wenn sich der Zustand geändert hat, sodass diese darauf reagieren können.

Dieses Muster reduziert die Komplexität, verhindert "Spaghetti-Code" und macht das Verhalten der Anwendung vorhersagbar und nachvollziehbar.

Verwaltete State Properties

Das zentrale `state`-Objekt enthält unter anderem folgende Schlüssel:

- `hoveredObject`: Das 3D-Objekt (`THREE.Mesh`), über dem sich der Mauszeiger gerade befindet. `null`, wenn kein Objekt überfahren wird.
- `selectedObject`: Das aktuell vom Benutzer ausgewählte 3D-Objekt.
- `highlightedObjects`: Ein `Set` von Objekten, die aus verschiedenen Gründen (z.B. Pfadsuche) hervorgehoben werden.
- `glowIntensity`: Ein numerischer Wert (0 bis 1), der die Intensität des "Glüh"-Effekts für das ausgewählte Objekt steuert. Wird von der internen `animate`-Schleife des Managers gesteuert, um einen pulsierenden Effekt zu erzeugen.
- `highlightEffectsEnabled`: Ein Boolean, der steuert, ob visuelle Effekte wie das Hervorheben überhaupt aktiv sind.
- `isInteractionEnabled`: Ein Boolean, um alle Benutzerinteraktionen global zu de-/aktivieren.

Funktionsweise des Publish/Subscribe-Systems

1. **`subscribe(callback, category)`:**
    - Ein Manager (z.B. der `HighlightManager`) meldet sich mit einer `callback`-Funktion beim `StateManager` an.
    - Optional kann eine `category` (z.B. 'highlight') angegeben werden, um gezielte Benachrichtigungen zu ermöglichen.
    - Der `StateManager` speichert den Callback in einer internen `subscribers`-Map.
2. **`update(partialState)`:**
    - Ein anderer Manager (z.B. der `InteractionManager`) ruft `update({ selectedObject: newObject })` auf.
    - Der `StateManager` aktualisiert sein internes `state`-Objekt mit den neuen Daten.

3. **`notifySubscribers()`:**
    - Nach einer erfolgreichen Aktualisierung durchläuft der `StateManager` alle registrierten `callback`-Funktionen und ruft sie mit dem neuen, vollständigen `state`-Objekt auf.
    - Jeder Subscriber kann dann auf die für ihn relevanten Änderungen im Zustand reagieren (z.B. der `HighlightManager` prüft, ob sich `selectedObject` geändert hat und wendet einen neuen visuellen Effekt an).

Wichtige Methoden

- **`update(partialState)`**: Die primäre Methode, um den Zustand zu ändern.
- **`batchUpdate(updates)`**: Eine optimierte Version von `update`, die mehrere Änderungen auf einmal annimmt, um die Anzahl der Benachrichtigungen zu reduzieren und die Performance zu verbessern.
- **`setHoveredObject(object)`** / **`setSelectedObject(object)`**: Spezifische Hilfsmethoden für die häufigsten Zustandsänderungen, die intern `update` aufrufen.
- **`animate()`**: Eine interne Animationsschleife, die per `requestAnimationFrame` läuft und kontinuierliche Zustandsänderungen wie die `glowIntensity` steuert.

Meine Meinung & Verbesserungsideen

Der `StateManager` ist bereits sehr gut implementiert und bildet ein solides Fundament.

1. **Selektive Benachrichtigungen:** Aktuell wird bei jeder Änderung das gesamte `state`-Objekt an alle Subscriber gesendet. Bei sehr häufigen Updates (z.B. Mausbewegungen) könnte dies zu unnötiger Last führen. Eine Verbesserung wäre, den Callbacks nur die *geänderten* Teile des Zustands mitzugeben (`callback(changedState, fullState)`). Die `batchUpdate`-Methode ist bereits ein Schritt in diese Richtung, könnte aber noch konsequenter genutzt werden.

2. **Zustandspersistenz (optional):** Für zukünftige Features wie das Speichern eines "Arbeitsstandes" könnte der `StateManager` um Methoden zum Serialisieren (`toJSON()`) und Deserialisieren (`fromJSON()`) seines Zustands erweitert werden. Damit ließe sich der gesamte Zustand der Visualisierung (selektiertes Objekt, Kamera-Position, etc.) speichern und wiederherstellen.

3. **Middleware/Redux-Pattern:** Für noch komplexere Anwendungen könnte man über die Einführung eines strengeren State-Management-Patterns wie Redux nachdenken. Anstatt `update` direkt aufzurufen, würde man "Actions" (z.B. `{ type: 'SELECT_OBJECT', payload: object }`) an den StateManager senden, die dann von "Reducern" verarbeitet werden. Dies würde die Nachverfolgung von Zustandsänderungen noch transparenter machen (Stichwort: "Time-Travel Debugging"), ist aber für den aktuellen Umfang der Anwendung wahrscheinlich übertrieben.

Komponente im Detail: InteractionManager

Rolle in der Architektur

Der `InteractionManager` ist die "Hand" der Anwendung. Er ist die zentrale Instanz, die rohe Benutzereingaben in konkrete, logische Aktionen innerhalb der 3D-Szene übersetzt. Er lauscht nicht direkt

auf DOM-Events, sondern auf die bereits vorverarbeiteten, semantischen Events des `CentralEventManager` (z.B. 'click', 'hover_start').

Seine Kernaufgabe ist es, zu entscheiden, **was** als Reaktion auf eine Benutzerinteraktion passieren soll, und diese Aktion dann zu delegieren.

Hauptverantwortlichkeiten:

1. **Event-Verarbeitung:** Reagiert auf semantische Events wie `click`, `doubleclick`, `hover_start`, `hover_end` und `keydown`.
2. **Aktions-Delegation:** Leitet Aktionen an die zuständigen Manager weiter. Anstatt die Szene direkt zu verändern, weist er andere an, dies zu tun:
    - **Zustandsänderung:** Aktualisiert den `StateManager` (z.B. `stateManager.setSelectedObject(clickedObject)`).
    - **Visuelle Effekte:** Beauftragt den `HighlightManager`, Objekte hervorzuheben oder die Hervorhebung zu entfernen.
    - **UI-Aktualisierung:** Beauftragt den `UIManager` (indirekt über den `StateManager`), das Info-Panel anzuzeigen.
3. **Unterscheidung von Interaktionen:** Er enthält Logik, um zwischen einem Klick und einer Drag-Bewegung zu unterscheiden, um unerwünschte Klick-Events während des Verschiebens der Szene zu verhindern.
4. **Kamera-Steuerung:** Führt Aktionen wie das Fokussieren der Kamera auf ein ausgewähltes Objekt aus (`focusOnObject`).

Wichtige Methoden und Abläufe

- **`initializeEventSubscriptions()`**: Hier werden alle relevanten Events des `CentralEventManager` abonniert und mit den entsprechenden Handler-Methoden verknüpft.
- **`handleClick(data)`**:
    - Wird bei einem Klick-Event ausgelöst.
    - `data` enthält das geklickte 3D-Objekt (`clickedObject`).
    - Wenn ein Objekt geklickt wurde, ruft es `this.selectObject(clickedObject)` auf.
    - Wenn ins Leere geklickt wurde, ruft es `this.deselectAll()` auf.
- **`handleHoverStart(data)` / `handleHoverEnd(data)`**:
    - Reagieren auf das Überfahren eines Objekts mit der Maus.
    - Rufen den `HighlightManager` auf, um den Hover-Effekt zu aktivieren (`highlightHoveredObject`) oder zu deaktivieren (`removeHighlight`).
- **`selectObject(object)`**:
    - Deselektiert das zuvor ausgewählte Objekt (falls vorhanden), indem es den `HighlightManager` anweist, dessen Hervorhebung zu entfernen.
    - Setzt das neue Objekt im `StateManager` als `selectedObject`.
    - Beauftragt den `HighlightManager`, den Selektions-Effekt auf das neue Objekt anzuwenden.
- **`focusOnObject(object)`**:

    - Eine besonders nützliche Funktion für die Navigation.
    - Berechnet die Position und Größe des Objekts.
    - Bewegt die Kamera (`this.camera`) und das Ziel der Orbit-Steuerung (`this.controls.target`) sanft zu diesem Objekt, sodass es im Mittelpunkt steht.

- **`handleKeyDown(data)`**:
    - Implementiert Tastaturkürzel.
    - **`Escape`**: Ruft `deselectAll()` auf, um die aktuelle Auswahl aufzuheben.
    - **`Delete`**: Ruft `deleteSelected()` auf, um das selektierte Objekt aus der Szene zu entfernen.
    - **`f`**: Ruft `focusOnObject()` für das aktuell selektierte Objekt auf.

Meine Meinung & Verbesserungsideen

Der `InteractionManager` erfüllt seine Rolle als zentraler Interaktions-Handler sehr gut. Die Trennung zwischen Event-Empfang und Delegieren der Aktionen ist ein sauberes Muster.

1. **Zustandsbasierte Interaktionsmodi:** Die Klasse hat bereits eine `modes`-Variable (`SELECT`, `DRAG` etc.), die aber noch nicht konsequent genutzt wird. Man könnte ein mächtiges System implementieren, bei dem der `currentMode` das Verhalten von Klicks und Mausbewegungen komplett ändert.
    - **Beispiel `CONNECT_MODE`:** Im "Verbindungsmodus" würde ein Klick auf zwei Knoten nicht auswählen, sondern stattdessen eine neue Kante zwischen ihnen erstellen.
    - **Beispiel `DELETE_MODE`:** Im "Löschmodus" würde jeder Klick auf ein Objekt dieses sofort löschen. Dies würde die `handle...`-Methoden mit `switch(this.currentMode)`-Anweisungen erweitern und die Anwendung sehr viel interaktiver machen.
2. **Vollständige Entkopplung von UI-Panels:** Nach meinem Refactoring ist schon viel UI-Logik im `UIManager`. Der `InteractionManager` hat jedoch immer noch direkte Referenzen auf die Panel-Elemente in `initializeInfoPanel()` und ruft `showInfoPanel()` mit generiertem HTML auf. Ein noch saubereres Design wäre, wenn der `InteractionManager` nur noch `this.uiManager.showDetailsForNode(nodeData)` aufrufen würde. Der `UIManager` wäre dann allein dafür verantwortlich, wie und wo diese Informationen dargestellt werden.
3. **Feingranulare Kamera-Steuerung in `focusOnObject`:** Die `focusOnObject` Methode ist gut, könnte aber noch verfeinert werden. Der "Zoom"-Abstand wird pauschal berechnet. Für eine bessere User Experience könnte man den Abstand abhängig von der Kamera-Perspektive und der relativen Größe des Objekts auf dem Bildschirm berechnen, um zu verhindern, dass man bei sehr großen Objekten "im" Objekt landet.

Komponente im Detail: LayoutManager

Rolle in der Architektur

Der `LayoutManager` ist der "Architekt" der Nodges-Anwendung. Seine alleinige Aufgabe ist es, die 3D-Positionen (`x`, `y`, `z`) für alle Knoten in einem Netzwerk zu berechnen. Eine gute Anordnung ist entscheidend für die Lesbarkeit und das Verständnis des Graphen.

Hauptverantwortlichkeiten:

1. **Algorithmen-Verwaltung:** Er registriert und verwaltet eine Bibliothek von verschiedenen Graphen-Layout-Algorithmen (z.B. Force-Directed, Circular, Grid).
2. **Berechnungs-Ausführung:** Er wendet den ausgewählten Algorithmus auf einen Satz von Knoten und Kanten an.
3. **Performance-Optimierung durch Web Worker:** Für rechenintensive Algorithmen (wie Force-Directed) lagert er die Berechnung in einen separaten JavaScript-Worker-Thread aus. Dies ist ein entscheidendes Architekturmerkmal, da es verhindert, dass die Benutzeroberfläche (UI) während der Sekunden andauernden Berechnung einfriert.
4. **Normalisierung:** Nach jeder Berechnung stellt er sicher, dass das resultierende Layout in einen vordefinierten visuellen Rahmen passt und nicht unkontrolliert "explodiert".

Funktionsweise und wichtige Methoden

- `registerLayout(id, layout)`: Diese Methode wird verwendet, um einen neuen Layout-Algorithmus zur internen Bibliothek hinzuzufügen.
- `applyLayout(layoutId, nodes, edges, options)`: Dies ist die zentrale öffentliche Methode.
  - Sie prüft, ob der angeforderte `layoutId` existiert.
  - **Worker-Delegation:** Wenn es sich um einen rechenintensiven Algorithmus handelt (z.B. `force-directed`), packt sie die Knoten- und Kantendaten zusammen und sendet sie an den `layout-worker.js`.
  - **Asynchronität:** Die Methode gibt ein `Promise` zurück. Der Hauptthread kann weiterarbeiten, während der Worker rechnet. Wenn der Worker fertig ist, sendet er die berechneten Positionen zurück.
  - Im `onmessage`-Handler des Workers werden die Positionsdaten aus dem Ergebnis extrahiert und auf die ursprünglichen `nodes`-Objekte im Hauptthread angewendet.
  - **Synchrone Ausführung:** Bei einfachen, schnellen Algorithmen (z.B. `circular`, `grid`) wird die Berechnung direkt im Hauptthread ausgeführt.
- `applyForceLayout(...)` **& Co.**: Private Methoden, die die eigentliche mathematische Logik des jeweiligen Algorithmus implementieren, falls er nicht in einen Worker ausgelagert ist.
- `normalizeNodePositions(nodes, maxExtent)`: Eine sehr wichtige Hilfsfunktion, die nach jeder Layout-Berechnung aufgerufen wird. Sie analysiert die resultierenden Positionen, findet deren Bounding Box (minimale und maximale Ausdehnung) und skaliert das gesamte Layout so, dass es in einen Würfel mit der Kantenlänge `maxExtent` passt und um den Ursprung (0,0,0) zentriert ist.

Der Layout-Worker (`layout-worker.js`)

Der Worker ist eine separate JavaScript-Datei, die im Hintergrund läuft.

1. Er erhält eine Nachricht mit den Knoten, Kanten, dem gewünschten Algorithmus und den Optionen.
2. Er importiert oder enthält die Logik für die rechenintensiven Algorithmen.
3. Er führt die Berechnung durch, ohne den Hauptthread zu blockieren.
4. Nach Abschluss sendet er eine Nachricht mit den neuen Positionen zurück an den `LayoutManager`.

Meine Meinung & Verbesserungsideen

Der `LayoutManager` ist durch den Einsatz von Web Workern bereits sehr performant und gut strukturiert.

1. **Animiertes Layout:** Aktuell "springen" die Knoten von ihrer alten Position zur neuen, vom Layout berechneten Position. Eine deutliche visuelle Verbesserung wäre, die Knoten sanft zu ihrer neuen Position zu animieren. Dies könnte mit der `TWEEN.js`-Bibliothek umgesetzt werden, die bereits im Projekt vorhanden ist. Der `LayoutManager` würde nicht nur die finale Position berechnen, sondern könnte auch Zwischenschritte liefern oder die Animation in der `NodgesApp` würde die Transition über einen bestimmten Zeitraum durchführen.

2. **Konfigurierbare Layout-Parameter:** Die Optionen für die Algorithmen (z.B. `repulsionStrength`, `attractionStrength`) sind aktuell im Code festgeschrieben. Ein großer Mehrwert wäre, diese Parameter über die Benutzeroberfläche (z.B. in der `LayoutGUI`) für den Benutzer einstellbar zu machen. Der Benutzer könnte so das Layout interaktiv an seine Bedürfnisse anpassen. Der `LayoutGUI` müsste dazu erweitert werden, um für jedes Layout die entsprechenden Einstellungs-Slider oder -Inputfelder dynamisch zu generieren.

3. **Inkrementelles Layouts:** Aktuell wird das Layout immer von Grund auf neu berechnet. Für Force-Directed-Layouts wäre es performanter, ein "inkrementelles" Layout zu ermöglichen. Wenn nur ein paar Knoten hinzugefügt oder entfernt werden, könnte der Algorithmus vom bestehenden Layout ausgehen und nur die betroffenen Bereiche neu justieren, anstatt das gesamte System neu zu berechnen. Dies würde die Reaktionszeit bei dynamischen Änderungen am Graphen erheblich verkürzen.

Komponente im Detail: HighlightManager

Rolle in der Architektur

Der `HighlightManager` ist der "visuelle Künstler" oder "Maler" der Anwendung. Er ist ausschließlich dafür verantwortlich, auf Zustandsänderungen im `StateManager` zu reagieren und entsprechende visuelle Effekte auf den 3D-Objekten zu erzeugen. Er entkoppelt die Logik der Interaktion (`InteractionManager`) von der Logik der visuellen Darstellung.

Hauptverantwortlichkeiten:

1. **Zustands-Abonnent:** Er abonniert den `StateManager` und wird bei jeder Zustandsänderung benachrichtigt.
2. **Effekt-Anwendung:** Er analysiert den neuen Zustand und entscheidet, welches Objekt wie hervorgehoben werden muss (z.B. Hover-Effekt für `hoveredObject`, Selektions-Effekt für `selectedObject`).
3. **Material-Management:** Er verwaltet die Materialien der 3D-Objekte. Das ist eine seiner kritischsten Aufgaben. Er erstellt Sicherungskopien der ursprünglichen Materialien, bevor er sie für einen Effekt verändert, und stellt sie danach wieder her.
4. **Effekt-Delegation:** Für komplexe Effekte wie das "Glühen" delegiert er die Aufgabe an spezialisierte Klassen wie den `GlowEffect`.
5. **Effekt-Bereinigung:** Er stellt sicher, dass Hervorhebungen korrekt entfernt werden, wenn ein Objekt nicht mehr überfahren oder ausgewählt ist.

Funktionsweise und wichtige Methoden

- **`handleStateChange(state)`**: Dies ist die zentrale Methode, die vom `StateManager` aufgerufen wird. Sie erhält das neue `state`-Objekt und dient als Ausgangspunkt für alle Aktionen des Managers.
- **`applyHighlight(object, type, options)`**: Die universelle Methode zum Anwenden eines Highlights.
    - Sie prüft zuerst, ob Highlights global aktiviert sind (`state.highlightEffectsEnabled`).
    - **`backupMaterial(object)`**: Bevor das Material des Objekts geändert wird, wird eine Sicherungskopie erstellt. Ein entscheidendes Detail hier ist die `isMaterialShared`-Prüfung: Wenn mehrere Objekte dasselbe Material verwenden (was bei optimierten Szenen üblich ist), wird das Material **geklont**, bevor es verändert wird. Das verhindert, dass die Hervorhebung eines Objekts versehentlich andere Objekte mit demselben Material mit hervorhebt.
    - **`applyVisualHighlight(...)`**: Wendet den spezifischen Effekt basierend on `type` an (z.B. `applyHoverEffect`).
- **`clearHighlight(object)`**: Die universelle Methode zum Entfernen eines Highlights.
    - **`restoreMaterial(object, backup)`**: Stellt das ursprüngliche Material aus der Sicherungskopie wieder her.
    - Beauftragt den `GlowEffect`, alle Glüheffekte vom Objekt zu entfernen.
    - Entfernt bei Kanten den speziellen `addEdgeOutline`-Effekt.
- **`applyHoverEffect(object)`**: Implementiert die spezifische Optik für den Hover-Zustand. Bei Knoten wird die Helligkeit der Farbe erhöht. Bei Kanten wird die Farbe zu einem leuchtenden Blau geändert und `addEdgeOutline` wird aufgerufen.
- **`addEdgeOutline(edge)`** / **`removeEdgeOutline(edge)`**: Diese Methoden erzeugen einen markanten Umriss-Effekt für Kanten, indem sie eine zweite, etwas dickere `TubeGeometry` mit einer helleren Farbe um die ursprüngliche Kante legen. Dies macht die Hover-Interaktion bei Kanten sehr deutlich sichtbar.

Meine Meinung & Verbesserungsideen

Der `HighlightManager` ist durch sein robustes Material-Management und die klare Trennung der Effekt-Logik bereits sehr leistungsfähig.

1. **Konfigurierbare Highlight-Stile:** Aktuell sind die Farben und Effekte für Hover und Selektion fest im Code verankert. Ein großer Fortschritt wäre, diese Stile konfigurierbar zu machen, ähnlich wie es in `definition_json.txt` für die Basisfarben der Fall ist. Man könnte im `metadata`-Block eines Knotens oder in einer globalen Konfigurationsdatei eigene Highlight-Stile definieren:

```
"metadata": {
  "highlightStyle": {
    "hover": { "color": "#FFD700", "glowIntensity": 0.8 },
    "selection": { "color": "#00FF00", "outline": true }
  }
}
```

Der `HighlightManager` würde diese Konfiguration lesen und die Effekte entsprechend anpassen.

2. **Highlight-Hierarchien:** Was passiert, wenn ein Objekt gleichzeitig Teil einer Pfadsuche ist UND vom Benutzer ausgewählt wird? Aktuell würde der zuletzt angewendete Effekt den vorherigen überschreiben. Ein fortschrittlicheres System könnte eine Prioritätenliste für Highlights definieren (`Selektion > Pfad > Hover`). Der Manager würde dann den Effekt mit der höchsten Priorität anwenden oder sogar Effekte kombinieren (z.B. die Pfad-Farbe mit dem Selektions-Glühen).

3. **Performance-Optimierung für Massen-Highlights:** Bei der Hervorhebung von ganzen Gruppen (`highlightGroup`) wird über jedes Objekt iteriert. Für sehr große Gruppen könnte man die `THREE.InstancedMesh`-Klasse in Betracht ziehen. Anstatt hunderte einzelne Objekte zu verändern, könnte man die Farbe für eine Instanz direkt ändern, was deutlich performanter ist. Dies würde jedoch größere Umbauten an der `createNodes`-Methode erfordern.

Komponente im Detail: UIManager

Rolle in der Architektur

Der `UIManager` ist die alleinige Kontrollinstanz für alle HTML-basierten Benutzeroberflächen-Elemente in Nodges. Nach einem umfassenden Refactoring bündelt er nun die gesamte Logik für die Anzeige, Positionierung und Interaktion mit den Info-Panels, die zuvor auf `index.html` und `main.js` verteilt war.

Hauptverantwortlichkeiten:

1. **UI-Initialisierung:** Erfasst alle relevanten DOM-Elemente (Panels, Buttons) und initialisiert deren Funktionalität.
2. **Panel-Management:** Steuert das Verhalten der Panels, insbesondere das Ein- und Ausklappen (`Toggling`).
3. **Dynamische Positionierung:** Sorgt dafür, dass sich die Panels intelligent auf dem Bildschirm anordnen und auf Änderungen der Fenstergröße oder anderer Panel-Größen reagieren, um Überlappungen zu vermeiden.
4. **Daten-Präsentation:** Stellt Methoden bereit, um anwendungsspezifische Daten in den Panels darzustellen (z.B. `updateFileInfo`, `showInfoPanelFor`).
5. **Event-Handling für UI-Elemente:** Verwaltet Klick-Events für UI-Elemente wie die Dateiliste oder den "Highlight-Toggle"-Schalter.

Funktionsweise und wichtige Methoden

- **`constructor(app)`**: Erhält die Haupt-App-Instanz, um auf andere Manager (wie den `StateManager`) und Anwendungsdaten zugreifen zu können.

- **`init()`**: Die zentrale Initialisierungsroutine. Sie ruft alle anderen `init...`-Methoden auf, um die UI in einen funktionsfähigen Zustand zu versetzen.

- **`initPanelToggling()`**: Fügt allen "Toggle"-Buttons die notwendigen `click`-Event-Listener hinzu. Bei einem Klick wird die CSS-Klasse `.collapsed` auf dem entsprechenden Panel umgeschaltet und das Pfeil-Icon (`>` oder `v`) aktualisiert.

- **`initPanelPositioning()`**:
    - Ruft einmalig `updateAllPanelPositions()` auf, um eine korrekte Start-Anordnung sicherzustellen.
    - Registriert `ResizeObserver`, um auf Größenänderungen einzelner Panels (z.B. wenn das `layoutPanel` erscheint oder das `fileInfoPanel` Inhalt bekommt) zu reagieren und die Positionen neu zu berechnen.
    - Registriert einen `resize`-Listener auf dem Browserfenster.
- **`updateAllPanelPositions()`**: Die Kernlogik für die dynamische Anordnung. Sie berechnet die Position jedes Panels basierend auf der Position und Größe des Panels darüber. Die Reihenfolge ist dabei entscheidend: `fileInfo` -> `file` -> `layout` (falls vorhanden) -> `info` -> `dev`.
- **`loadAvailableFiles()`** / **`createFileButtons(filenames)`**: Diese Methoden sind dafür verantwortlich, die Liste der verfügbaren JSON-Dateien zu holen (aktuell eine statische Liste) und dynamisch als klickbare `<div>`-Elemente im `filePanel` zu rendern. Der `onclick`-Handler ruft dann die `loadData`-Methode der `NodgesApp` auf.
- **`updateFileInfo(...)`** / **`updateFps(fps)`**: Öffentliche API-Methoden, die von der `NodgesApp` aufgerufen werden, um die entsprechenden Text-Elemente im `fileInfoPanel` zu aktualisieren.
- **`showInfoPanelFor(object)`**: Wird aufgerufen, wenn sich `state.selectedObject` ändert. Die Methode generiert den passenden HTML-Inhalt für das ausgewählte Objekt und zeigt das `infoPanel` an.

Meine Meinung & Verbesserungsideen

Durch das Refactoring ist der `UIManager` nun ein zentraler und logischer Baustein der Architektur.

1. **Web Components / Framework-Nutzung:** Der aktuelle Ansatz mit direkten DOM-Manipulationen und manuell angehängten Event-Listenern ist funktional, aber nicht ideal für komplexe UIs. Ein großer Schritt nach vorn wäre die Kapselung jedes Panels in eine eigenständige **Web Component**.
    - **Vorteile:** Jedes Panel (`<file-panel>`, `<info-panel>`) wäre eine wiederverwendbare, gekapselte Komponente mit eigenem HTML-Template und eigener Logik. Der `UIManager` würde diese Komponenten nur noch instanziieren und mit Daten versorgen, anstatt ihre interne DOM-Struktur zu kennen.
    - Alternativ könnte man ein leichtgewichtige UI-Bibliothek wie Preact oder Svelte in Betracht ziehen, um die UI deklarativ zu beschreiben, was die Komplexität weiter reduzieren würde.
2. **Zustands-gesteuerte CSS-Klassen:** Anstatt die `style`-Attribute direkt per JavaScript zu manipulieren (z.B. `panel.style.top = ...`), sollte dies bevorzugt über CSS-Klassen geschehen, die vom `UIManager` umgeschaltet werden. JavaScript setzt die Klasse, CSS kümmert sich um die Darstellung. Dies verbessert die Trennung von Verhalten und Aussehen. Die `.collapsed`-Klasse ist bereits ein gutes Beispiel für dieses Prinzip.
3. **Flexbox/Grid für Panel-Layout:** Die aktuelle Positionierung basiert auf der Berechnung der `bottom`-Position des vorherigen Elements. Ein modernerer Ansatz wäre, die Panels in einen Container zu legen und sie mit CSS Flexbox (`flex-direction: column`) oder CSS Grid anzuordnen. Dies würde die

JavaScript-basierte Positionsberechnung überflüssig machen und das Layout robuster und einfacher zu verwalten sein.

Konzept: Versioniertes & animierbares Datenmodell

Dieses Dokument beschreibt ein Entwurf für ein zukünftiges, erweitertes JSON-Format für Nodges. Das Hauptziel ist es, über die statische Darstellung hinauszugehen und **Zustandsänderungen über die Zeit** zu ermöglichen, beispielsweise um Bewegungen, Wertänderungen oder das Entstehen und Vergehen von Knoten/Kanten zu animieren.

Das hier entworfene Konzept ist für eine **spätere Implementierung** gedacht und soll sicherstellen, dass die zukünftige Datenstruktur flexibel und leistungsfähig genug ist.

Kernanforderungen

1. **Versionierung/Staging**: Knoten und Kanten sollen nicht nur einen einzigen Zustand haben, sondern eine Serie von Zuständen oder "Versionen" über eine Zeitachse hinweg.
2. **Effizienz**: Die Struktur soll vermeiden, für jeden Zeitpunkt den gesamten Graphen neu zu speichern. Nur die Änderungen (`Deltas`) sollen erfasst werden.
3. **Flexibilität**: Das Format muss weiterhin beliebige Metadaten zulassen.
4. **Kompatibilität**: Es sollte klar erkennbar sein, damit der Parser in `NodgesApp` dieses Format von den bestehenden Formaten unterscheiden kann.

Entwurf des Datenformats

Ich schlage eine Struktur vor, die eine Basis-Definition des Graphen von einer oder mehreren Zeitachsen mit Zustandsänderungen trennt.

```
{
  "format": "nodges_timeline_v1",
  "metadata": {
    "title": "Dynamisches System Beispiel",
    "description": "Demonstriert die Bewegung und Zustandsänderung von
Knoten über die Zeit.",
    "duration": 100, // Gesamtdauer der Timeline in Ticks oder Sekunden
    "version": "1.0"
  },

  "graph": {
    "nodes": [
      {
        "id": "node_01",
        "name": "Server A",
        "physicalGroup": "rack_1",
        "base": {
          "size": 1.5,
          "color": "#C70039",
          "geometry": "cube"
```

```
        }
      },
      {
        "id": "node_02",
        "name": "Database",
        "physicalGroup": "rack_1",
        "base": {
          "size": 1.2,
          "color": "#FFC300",
          "geometry": "cylinder"
        }
      }
    ],
    "edges": [
      {
        "id": "edge_01",
        "name": "Main Connection",
        "source": "node_01",
        "target": "node_02",
        "siblings": ["edge_02"],
        "physicalGroup": "rack_1",
        "base": {
          "color": "#00A896"
        }
      }
    ]
  },

  "timeline": [
    {
      "at": 0,
      "updates": [
        { "targetId": "node_01", "properties": { "position": {"x": -5, "y":
0, "z": 0} } },
        { "targetId": "node_02", "properties": { "position": {"x": 5, "y":
0, "z": 0} } }
      ]
    },
    {
      "at": 50,
      "updates": [
        { "targetId": "node_01", "properties": { "color": "#FF5733" } },
        { "targetId": "edge_01", "properties": { "color": "#FFC300" } }
      ]
    },
    {
      "at": 100,
      "updates": [
        { "targetId": "node_01", "properties": { "position": {"x": -5, "y":
5, "z": 0} } }
      ]
    }
  ]
}
```

Erklärung der Struktur

1. `format: nodges_timeline_v1`
    - Eine explizite Kennzeichnung, damit der Parser sofort weiß, wie diese Datei zu interpretieren ist.
2. `graph`:
    - Definiert die **grundlegenden, statischen Eigenschaften** aller Elemente.
    - `nodes`: Enthält die Basis-Definition jedes Knotens.
        - `id`, `name`: Deine geforderten "must-have" Parameter.
        - `physicalGroup`: Deine geforderte Eigenschaft. Ich interpretiere sie als Gruppierung für Physik- oder Layout-Zwecke. Knoten in derselben Gruppe könnten sich z.B. bei Layout-Änderungen wie ein starrer Körper verhalten.
        - `base`: Ein Objekt für alle initialen visuellen Eigenschaften (`size`, `color`, `geometry`). Dadurch trennen wir die veränderlichen von den statischen Eigenschaften.
    - `edges`: Enthält die Basis-Definition jeder Kante.
        - `id`, `name`, `source`, `target`: Deine geforderten Parameter (`start_pos`/`end_pos` als `source`/`target` interpretiert).
        - `siblings`: Deine geforderte Eigenschaft. Ein Array von Edge-IDs, das Kanten gruppiert, die eine ähnliche Funktion haben. Dies könnte genutzt werden, um sie visuell zusammenzufassen oder gemeinsam zu animieren.
3. `timeline`:
    - Ein Array von **Zustands-Snapshots** (Keyframes).
    - `at`: Definiert den Zeitpunkt (Tick, Frame, Sekunde) für diesen Snapshot.
    - `updates`: Ein Array von Änderungen, die zu diesem Zeitpunkt eintreten.
        - `targetId`: Die `id` des Knotens oder der Kante, die geändert wird.
        - `properties`: Ein Objekt, das **nur die geänderten Eigenschaften** enthält. In diesem Beispiel werden `position` und `color` geändert. Andere Eigenschaften wie `size` bleiben unverändert.

Wie würde die Anwendung dies verarbeiten?

1. **Parsing:** Ein neuer oder erweiterter `FutureDataParser` würde das `nodges_timeline_v1`-Format erkennen.
2. **Initialisierung:** Er würde den Graphen wie gewohnt aus dem `graph`-Objekt aufbauen und die `base`-Eigenschaften für die initiale Darstellung nutzen.
3. **`TimelineManager`:** Eine neue Klasse, der `TimelineManager`, würde das `timeline`-Array erhalten.
4. **Animation/Scrubbing:** Der `TimelineManager` würde eine Abspiel-Logik bereitstellen. Wenn die Zeit voranschreitet (z.B. über einen Play-Button oder einen Scrubber), würde der Manager die `updates` aus der Timeline interpolieren.
    - Bei `at: 0` setzt er die Startpositionen.
    - Zwischen `at: 0` und `at: 100` würde er die Position von `node_01` sanft von `(x: -5, y: 0)` zu `(x: -5, y: 5)` animieren (z.B. mittels TWEEN.js).

    - Bei `at: 50` würde er die Farben von `node_01` und `edge_01` abrupt ändern oder ebenfalls sanft überblenden.

Meine Meinung & weitere Ideen

- **Trennung von `position`:** Die Eigenschaft `position` (dein `xpos`, `ypos`, `zpos`) sollte bevorzugt in der `timeline` stehen, da sie die am häufigsten animierte Eigenschaft ist. Das `base`-Objekt sollte sie nur enthalten, wenn ein Objekt eine feste Position hat.
- **Komplexere Animationen:** Für sanfte Übergänge (Easing) könnte das `update`-Objekt erweitert werden:

```
{ "targetId": "node_01", "properties": { "position": {...} },
"transition": { "duration": 20, "ease": "easeInOutQuad" } }
```

- **Events in der Timeline:** Man könnte auch `events`-Objekte in die Timeline integrieren, die bei einem bestimmten Zeitpunkt ausgelöst werden, z.B. um einen Sound abzuspielen oder ein UI-Element anzuzeigen.

```
 { "at": 75, "events": [{ "type": "showAnnotation", "targetId":
"node_01", "text": "Status Critical!" }] }
```

- **Alternative zu `siblings`:** Anstatt `siblings` manuell zu pflegen, könnte der `physicalGroup`-Bezeichner für Kanten denselben Zweck erfüllen und wäre konsistenter mit dem Knoten-Schema.