Bericht: Node und Edge Erstellung als Mesh in Nodges

Übersicht

Die Erstellung von 3D-Meshes für Nodes und Edges in Nodges folgt einem modularen Design mit klarer Trennung der Verantwortlichkeiten. Der Prozess verwendet Performance-optimierte Techniken wie InstancedMesh für Nodes und spezialisierte Geometrien für Edges.

Beteiligte Dateien

Kern-Komponenten

1. `src/core/NodeManager.ts` - Verwaltung aller Node-Meshes
2. `src/core/EdgeObjectsManager.ts` - Verwaltung aller Edge-Meshes
3. `src/core/VisualMappingEngine.ts` - Visuelle Mapping-Logik
4. `src/objects/Edge.js` - Spezielle Edge-Geometrie-Erstellung

Unterstützende Komponenten

5. `src/types.ts` - Typdefinitionen für Datenstrukturen
6. `src/App.ts` - Orchestrator und Koordination
7. `src/core/DataParser.ts` - Datenverarbeitung und Konvertierung

Ablauf der Node-Erstellung

1. Datengruppierung nach Geometrie-Typ

```
// NodeManager.updateNodes()
const entitiesByType = new Map<string, { entity: EntityData, visual: any }[]>();
```

2. Visuelle Mapping-Anwendung

- `VisualMappingEngine.applyToEntity()` berechnet visuelle Eigenschaften
- Unterstützte Geometrien: Sphere, Cube, Cylinder, Cone, Torus
- Mapping-Funktionen: Linear, Exponential, Logarithmic, Heatmap, Bipolar

3. InstancedMesh-Erstellung (Performance-Optimierung)

```
const mesh = new THREE.InstancedMesh(geometry, material, count);
```

Vorteile von InstancedMesh:

- Ein Geometrie-Objekt für tausende von Instanzen
- Minimale Draw Calls
- GPU-optimiertes Rendering

4. Matrix-Berechnung für jede Instanz

```
dummy.position.set(x, y, z);
dummy.scale.set(visualScale, visualScale, visualScale);
dummy.updateMatrix();
mesh.setMatrixAt(index, dummy.matrix);
```

5. Farbzuweisung per Instanz

```
mesh.setColorAt(index, color);
mesh.instanceColor.needsUpdate = true;
```

Ablauf der Edge-Erstellung

1. Kanten-Klassifizierung

```
// EdgeObjectsManager.updateEdges()
connectionMap.forEach((group) => {
    if (group.length === 1) {
        straightEdges.push(group[0]);  // Einzelkante
    } else {
        // Mehrfachkanten → gebogen
        group.forEach((edge, i) => {
            curvedEdgesData.push({ edge, index: i, total: group.length });
        });
    }
});
```

2. Gerade Kanten (InstancedMesh)

```
this.cylinderMesh = new THREE.InstancedMesh(geometry, material,
straightEdges.length);
```

Transformation:

- Positionierung an Startpunkt
- Ausrichtung zum Endpunkt (`lookAt()`)
- Skalierung auf Kantenlänge

3. Gebogene Kanten (Spezial-Objekte)

```
const edgeObj = new Edge(startPos, endPos, 1, 1, options);
```

4. Curve-Berechnung in Edge.js

```
// QuadraticBezierCurve3 für Bögen
this.curve = new THREE.QuadraticBezierCurve3(
    startPos,
    controlPoint,
    endPos
);

// TubeGeometry entlang der Curve
new THREE.TubeGeometry(this.curve, 20, 0.4, 8, false);
```

Multi-Edge-Logik:

- Winkelverteilung um Verbindungsachse
- Index-basierte Rotation
- Abstandsvergrößerung für Klarheit

Performance-Optimierungen

1. Caching-Systeme

```
// Geometrie-Caching
private geometryCache: Map<string, THREE.BufferGeometry>;

// Material-Caching
private materialCache: Map<string, THREE.Material>;
```

2. InstancedMesh-Strategie

- **Nodes: Gruppierung nach Geometrie-Typ**
- **Edges: Gruppierung nach Gerade/Gebogen**
- Reduzierung von Draw Calls um 90%+

3. Dynamic Draw Usage

```
mesh.instanceMatrix.setUsage(THREE.DynamicDrawUsage);
```

Visual Mapping Pipeline

1. Entity-Visualisierung

```
const visual = this.visualMappingEngine.applyToEntity(entity);
// Ergebnis: { size, color, geometry, glow, animation }
```

2. Relationship-Visualisierung

```
const visual = this.visualMappingEngine.applyToRelationship(relationship);
// Ergebnis: { thickness, color, curvature, opacity, animation }
```

3. Mapping-Funktionen

- **Linear**: Direkte Skalierung
- **Exponential**: Potenzierung
- **Logarithmic**: Logarithmische Skalierung
- **Heatmap**: Farbverläufe
- **Bipolar**: Negative/Positive Werte

Datenfluss-Diagramm

```
JSON-Daten → DataParser → EntityData/RelationshipData
    ↓
VisualMappingEngine → VisualProperties
    ↓
┌──────────────────┬──────────────────┐
│   NodeManager    │ EdgeObjectsManager │
│                  │                  │
│ InstancedMesh    │ InstancedMesh    │
│ (pro Typ)        │ (gerade)         │
│                  │                  │
│                  │ Edge.js          │
│                  │ (gebogen)        │
└──────────────────┴──────────────────┘
    ↓
THREE.Scene → WebGL Renderer
```

Speicherverwaltung

1. Automatische Dispose-Methoden

```
public dispose() {
    this.meshes.forEach(mesh => {
        this.scene.remove(mesh);
        mesh.dispose();
    });
    this.geometryCache.forEach(geo => geo.dispose());
```

```
        this.materialCache.forEach(mat => mat.dispose());
    }
```

2. Update-Optimierungen

- **Positions-Updates**: Nur Matrix-Änderungen
- **Farb-Updates**: Nur Instanz-Farben
- **Neuberechnung**: Bei Geometrie-Wechseln

Interaktions-Support

1. Raycasting-Optimierung

- InstancedMesh mit `instanceId` mapping
- Direkte Zuordnung zu EntityData
- Optimiert für tausende Objekte

2. userData-Struktur

```
// Node userData
mesh.userData = { type: 'node_instanced', geometryType: type };

// Edge userData
tube.userData = {
    type: 'edge',
    edge: edgeData,
    curve: this.curve,
    connectionKey: connectionKey
};
```

Zusammenfassung

Die Mesh-Erstellung in Nodges ist hochoptimiert und modular:

Stärken:

- Performance durch InstancedMesh
- Flexibilität durch Visual Mapping
- Wartbarkeit durch klare Trennung
- Skalierbarkeit für große Graphen

Architektur-Prinzipien:

- Single Responsibility pro Manager
- Caching für Performance
- Typensicherheit durch TypeScript
- OpenGL-optimierte Rendering-Pipelines

Das System ermöglicht die Darstellung von tausenden von Nodes und Edges bei gleichzeitig flüssigen Interaktionsraten.

Erstellt am: 2025-12-17 Version: Nodges 0.96.3 Autor: System-Analyse