Optimierungsstrategien fuer grossskalige 3D-Visualisierungen: Eine empirische Evaluierung von Octree-Culling, Level-of-Detail Rendering und Cache-Optimization

Cabrell Valdice Teikeu Kana
772538

Hochschule Darmstadt
Fachbereich Informatik

Seminar: Computergraphik und Visualisierung
Betreuer: Prof. Dr. [Name]

Wintersemester 2025/2026

18. Januar 2026

Zusammenfassung

Diese Arbeit untersucht die praktische Implementierung und Evaluierung von drei etablierten Optimierungstechniken im Kontext der Echtzeitvisualisierung grossskaliger Szenen. Unter Verwendung eines realen Datensatzes von 29.722 Windkraftanlagen mit einer zeitlichen Ausdehnung von 35 Jahren werden Octree-basierte raeumliche Indizierung, Level-of-Detail Rendering und CPU-Cache Optimierung systematisch analysiert. Durch eine Kombination von theoretischem Verstaendnis und empirischer Validierung werden die Trade-offs zwischen Implementierungskomplexitaet, Memory-Overhead und Performance-Gewinn kritisch diskutiert. Die Erkenntnisse zeigen, dass unter CPU-Rendering-Constraints eine selektive Kombination dieser Techniken erfolgreich eingesetzt werden kann, wobei die groessten Optimierungspotentiale in der Cache-Effizienz und der Datenstruktur-Wahl liegen, nicht in der raeumlichen Indizierung allein.

Inhaltsverzeichnis

1. Einleitung

Die Echtzeit-Visualisierung grossskaliger geometrischer Datensaetze zaehlt zu den fundamentalen Herausforderungen der Computergraphik. Waehrend moderne GPU-Rasterisierer in der Lage sind, Millionen von Dreiecken pro Frame zu verarbeiten, wird die Gesamtperformance zunehmend durch die Praeprocessing-Pipeline limitiert: Das Zusammentragen von zu rendernden Objekten, Transformationen und Batch-Operationen muessen auf der CPU erfolgen, die fuer solche Aufgaben um Groessenordnungen langsamer ist. Wenn ein Datensatz 30.000 Objekte enthaelt, der sichtbare Frustum aber nur 300–500 davon enthaelt, wird die naive Verarbeitung aller Objekte zur Bremse.

Die zur Verfuegung stehende Zeit ist praezise. Bei einer angestrebten Renderfrequenz von 60 Hz verbleiben 16,7 Millisekunden pro Frame. Davon entfallen etwa 10–12 ms auf die GPU-Rasterisierung, die CPU-seitige Vorbereitung hat also nur 4–6 ms zur Verfuegung. Bei 29.722 Objekten bedeutet das durchschnittlich 0,15 Mikrosekunden pro Objekt fuer alle CPU-seitigen Operationen—eine unmoegliche Anforderung fuer komplexe Algorithmen.

Industrie-Loesungen wie die Unreal Engine oder Unity adressieren dieses Problem durch spezialisierte Optimierungstechniken, die ueber Jahrzehnte gewachsen sind. Diese Techniken umfassen raeumliche Indizierungsstrukturen fuer schnelle Abfragen, progressive Geometriereduktion fuer entfernte Objekte und Memory-Layout Optimierungen fuer effiziente Batch-Verarbeitung. Was in Game-Engines implizit ist, wird in akademischen Kontexten selten explizit behandelt.

Die vorliegende Arbeit adressiert diese Luecke durch eine systematische Implementierung und Evaluierung dreier klassischer Optimierungstechniken: Octree-basierte raeumliche Indizierung fuer Frustum-Culling, Level-of-Detail Rendering zur geometrischen Reduktion sichtbarer Komplexitaet, und CPU-Cache Optimierung durch Structure-of-Arrays Memory-Layout. Der Kontext ist ein reales Visualisierungsproblem: die interaktive 3D-Darstellung der deutschen Windkraftlandschaft zwischen 1990 und 2025, mit allen ihren Widerspruechen zwischen Theorie und Praxis.

1.1 Motivation und Problemstellung

Das Windkraft-Projekt war urspruenglich als Lehr-Visualisierung konzipiert, wurde aber schnell zu einer Demonstrator-Anwendung, an der sich empirische Optimierungsfragen untersuchen lassen. Der Datensatz enthaelt georeferenzierte Koordinaten, Baujahre und Leistungswerte von 29.722 installierten Windkraftanlagen. Eine naive Visualisierung wuerde alle 29.722 Turbinen in jedem Frame durchlaufen, ihre Projektionen in den sichtbaren Frustum berechnen und Rendering-Befehle generieren. Selbst mit optimierten GPU-Befehlen ist dies unter CPU-Bottleneck-Bedingungen nicht praktikabel.

Der induzierende Kontext ist dabei weniger das Rendering selbst—moderne GPUs rasterisieren Geometrie schnell genug—sondern die CPU-seitige Organisation dieser Renderauftraege. Das Problem ist klassisches Culling: Die unsichtbaren Objekte zu identifizieren und gar nicht erst zur GPU zu schicken. Dies ist nicht trivial: Ein Octree mit 29.722 Blatt-Knoten erfordert eine sorgfaeltige Implementierung, um den Overhead der hierarchischen Struktur nicht aufzubrauchen. Ein LOD-System muss Uebergaenge elegant handhaben, um visuelles Pop-In zu vermeiden. Und die Memory-Layouts der Datenstrukturen bestimmen, ob Batch-Operationen cache-effizient sind oder nicht.

1.2 Struktur und Forschungsfragen

Diese Arbeit verfolgt drei Forschungsfragen, die im Verlauf beantwortet werden:

RQ1: Welche Performance-Gewinne sind durch Octree-basierte raeumliche Indizierung und hierarchisches Frustum-Culling erreichbar, und bei welchen Szenen-Charakteristiken amortisiert sich der Aufbau-Overhead?

RQ2: Wie wirken sich Level-of-Detail Systeme auf die Polygon-Reduktion aus, und welche Artefakte entstehen durch die geometrische Vereinfachung?

RQ3: Welche Memory-Layout-Strategien (Array-of-Structures vs. Structure-of-Arrays) maximieren die Effizienz von CPU-seitigen Batch-Operationen und wie hoch sind die Gewinne durch NumPy-Vektorisierung?

Kapitel 2. praesentiert die theoretischen Grundlagen fuer alle drei Techniken. Kapitel 3. detailliert die konkrete Implementierung mit Code und Design-Entscheidungen. Kapitel 4. dokumentiert die experimentelle Validierung auf dem realen Windkraft-Datensatz. Kapitel 5. diskutiert kritisch die Limitationen und Trade-offs. Kapitel 6. zieht Schlussfolgerungen.

2. Theoretische Grundlagen

2.1 Raeumliche Indizierungsstrukturen und Octrees

Die fundamentale Herausforderung der raeumlichen Indizierung ist die Beantwortung von geometrischen Bereichsabfragen: Gegeben ein Abfrage-Volumen (wie ein Viewing Frustum) und eine Menge von Objekten im 3D-Raum, welche Objekte liegen im Abfrage-Volumen? Die naive Loesung iteriert ueber alle Objekte und prueft jedes einzelne—eine $O(n)$ Operation mit einem hohen konstanten Faktor aufgrund der Wiederholung von Checks.

Hierarchische Raumpartitionierungsmethoden reduzieren dies auf $O(\log n)$ durch Vorbau: Der 3D-Raum wird rekursiv in Unterraeume unterteilt, die in einer baumartigen Datenstruktur organisiert sind. Das Octree [1] ist ein klassischer Ansatz, bei dem der Raum rekursiv in 8 kongruente Oktanten (3D) oder 4 Quadranten (2D) unterteilt wird. Die Unterteilung setzt sich fort, bis die Blatt-Knoten klein genug sind oder nur wenige Objekte enthalten.

[PLACEHOLDER IMAGE:
$\text{octree}_structure.png - SchematischeDarstellungeinesOctrees$]

Abbildung 1: Schematische Darstellung eines Octrees mit rekursiver Raumpartitionierung. Der Root-Knoten wird in Quadranten unterteilt; Knoten mit vielen Objekten werden weiter unterteilt.

Die Komplexitaet eines Octree-basierten Range-Queries ergibt sich aus zwei Faktoren: erstens der Tiefe des Baums ($\log_8 n$ fuer n Objekte bei gleichmaessiger Verteilung), zweitens der Anzahl der Knoten, die mit der Abfragevolumen ueberlappen. Der Best-Case ist $O(\log n)$, der Worst-Case $O(n)$ wenn der Abfrage-Frustum die ganze Szene bedeckt. In der Praxis liegt die Performance zwischen diesen Grenzen und haengt stark von der raeumlichen Verteilung der Objekte ab.

Ein wichtiges Konzept ist die Komplexitaet der Abfrage-Operation selbst. Ein Frustum-Culling Check benoetigt einen stabilen Algorithmus zur Pruefung, ob eine Bounding Box (AABB) den Viewing Frustum schneidet. Akenine-Moeller et al. [2] haben mehrere optimierte Algorithmen beschrieben, die die AABB in lokalen Frustum-Koordinaten schnell pruefen. Diese Algorithmen sind nicht trivial: Eine naive Implementierung prueft alle 6 Frustum-Ebenen

gegen alle 8 Ecken der AABB, was zu 48 Operationen fuehrt. Optimierte Versionen reduzieren dies durch fruehe Aussteigs-Bedingungen.

Die Konstruktion eines Octrees hat eigene Kosten. Fuer n Objekte betraegt die Aufbau-Zeit typischerweise $O(n \log n)$, kann aber bei unguenstigen Verteilungen $O(n^2)$ sein. Dies ist ein wichtiger Trade-off: Der Octree muss sich amortisieren ueber mehrere Abfragen. In einer interaktiven Visualisierung mit 60 FPS verteilt sich dieser Aufbau-Overhead ueber 16,7 ms, was bei 4–6 ms CPU-Zeit knapp ist.

2.2 Level-of-Detail Rendering und Mesh Simplification

Das Kernproblem von LOD ist, dass die geometrische Komplexitaet einer Szene schneller mit der Objektanzahl waechst als die Rendering-Kapazitaet. Ein einzelnes Windkraftwerk kann aus 3.000–5.000 Dreiecken bestehen (Rotor, Gondel, Turm). Bei 30.000 Windkraftwerken wuerde dies zu 90–150 Millionen Dreiecken fuehren. Selbst GPU-Rasterisierer stoessen hier an Grenzen (nicht nur Durchsatz, sondern auch Memory-Bandbreite und Transformations-Overhead).

LOD adressiert dies durch die Beobachtung, dass Objekte, die weit entfernt sind, weniger geometrische Detail benoetigen. Ein Windkraftwerk, das mehrere Kilometer entfernt ist, kann durch eine vereinfachte Geometrie oder sogar ein Billboarding-Sprite dargestellt werden. Dies fuehrt zu einer klassischen Raum-Zeit-Kompromiss: Mehr CPU-Overhead fuer die LOD-Verwaltung, aber weniger Geometrie, die die GPU verarbeiten muss.

Das theoretische Fundament von LOD ist die Mesh-Simplification, ein Problem, das seit den 1990er Jahren untersucht wird. Hoppe [3] praegte das Konzept der Progressive Meshes: Eine Ausgangsgeometrie wird durch eine Sequenz von Edge-Collapse Operationen vereinfacht, wobei jeder Schritt den Fehler zwischen Original und vereinfachter Version minimiert. Die Fehlermetrik ist dabei zentral: Eine zu aggressive Simplification erzeugt sichtbare Artefakte, eine zu konservative bringt keinen Performance-Gewinn.

[PLACEHOLDER IMAGE: $\text{lod}_{c}omparison.png - LOD - LevelVergleich$]

Abbildung 2: Level-of-Detail Progression: Drei aufeinanderfolgende LOD-Level mit zunehmender Polygon-Reduktion.

Die Wahl der Uebergangspunkte zwischen LOD-Leveln ist kritisch. Ein zu schneller Uebergang erzeugt visuelles Pop-In, bei dem Objekte ploetzlich ihre Erscheinung veraendern. Ein zu langsamer Uebergang verhindert Performance-Gewinne, da die hoehere LOD zu lange beibehalten wird. Gaengige Strategien verwenden exponentielle Abstaende oder kalibrieren die Uebergaenge basierend auf Screen-Space-Groesse [4].

2.3 CPU-Cache Optimierungen und Memory Layouts

Der dritte Optimierungsvektor ist nicht algorithmen-spezifisch, sondern architektur-spezifisch: die CPU-Cache Hierarchie. Moderne CPUs haben mehrere Cache-Level (L1, L2, L3), deren Groesse und Latenz sich um Groessenordnungen unterscheiden. Ein L1-Cache-Hit kostet etwa 3–4 Zyklen, ein L3-Hit kostet 40–75 Zyklen, und ein Main-Memory-Hit kostet 200–300 Zyklen. Wenn die Datenstruktur unguenstig im Memory liegt, werden einfache Operationen von Memory-Latenz dominiert.

Das klassische Problem ist die Wahl zwischen Array-of-Structures (AoS) und Structure-of-Arrays (SoA) [5]. Ein AoS-Layout speichert Turbinenobject als Instanzen. Das Problem mit

AoS zeigt sich bei Operationen wie Culling, bei der nur das x,y,z-Attribut benoetigt wird. Jeder Zugriff auf x, y, z verursacht moeglicher weise einen Cache-Miss, weil die Daten ueber den Memory-Space verstreut sind. Der CPU-Prefetcher kann nicht vorhersagen, wo die naechsten Daten liegen.

Ein SoA-Layout gruppiert Attribute nach Typ. Nun liegt x dicht an x, was die Cache-Effizienz dramatisch verbessert. Eine Culling-Operation wird nicht nur vektorisierbar (SIMD), sondern auch cache-effizient. Der Speedup kann 100x–500x sein fuer Batch-Operationen [6], [7].

Das theoretische Fundament ist Cache-Oblivious Design [8], das besagt, dass Algorithmen, die rekursiv Divide-and-Conquer anwenden, automatisch cache-effizient sind, ohne explizite Cache-Parameter zu kennen. Die praktische Konsequenz ist, dass NumPy-Vektorisierung auf modernen CPUs nicht nur wegen SIMD-Parallelismus schnell ist, sondern auch wegen der cache-Lokalitaet der SoA-Daten.

3. Implementierung

3.1 Octree-System fuer raeumliche Indizierung

Das Octree-System wurde als eigenstaendiges Python-Modul implementiert, das in die bestehende TurbineManager-Klasse integriert wird. Die Kerndatenstruktur besteht aus drei Komponenten: eine BoundingBox-Klasse fuer geometrische Operationen, eine OctreeNode-Klasse fuer die hierarchische Struktur, und ein OctreeManager fuer die API.

Die BoundingBox-Klasse implementiert AABB-Schnitt-Tests mit zwei anderen Bounding-Boxen (ebenfalls ein AABB) und mit einem Frustum. Der AABB-zu-AABB-Test ist trivial (six separating axis test), aber der AABB-zu-Frustum-Test benoetigt sorgfaeltige Implementierung. Fuer jede der 6 Frustum-Ebenen wird geprueft, ob die gesamte AABB auf der Outside-Seite liegt. Falls ja, ist das AABB nicht sichtbar. Dies erfordert die Berechnung der extremalen Punkte der AABB relativ zu jeder Ebenen-Normale.

[PLACEHOLDER IMAGE: frustum$_culling.png - Frustum - CullingGeometrie$]

Abbildung 3: Frustum-Culling Geometrie: Das View-Frustum wird durch 6 Ebenen definiert. Objekte werden gegen diese Ebenen getestet.

Der OctreeNode ist rekursiv definiert. Ein Knoten kann ein Blatt sein (max. 8 Turbinen) oder ein innerer Knoten mit 8 Kindern. Jeder Knoten speichert seine BoundingBox und eine Liste von Objektindizes. Die Tiefe wird begrenzt durch einen min_size-Parameter: Wenn ein Knoten weniger als 8 Objekte enthaelt, wird nicht weiter unterteilt.

Die Konstruktion folgt einem Top-Down-Ansatz: Beginnend mit dem Root-Knoten ueber alle 29.722 Turbinen wird rekursiv unterteilt. Jedes Objekt wird in einen der 8 Kind-Knoten sortiert basierend auf seiner Position relativ zum Octree-Zentrum. Diese Konstruktion ist $O(n \log n)$ im besten Fall, wenn die Objekte gleichmaessig verteilt sind. Die gemessene Build-Time fuer 29.722 Turbinen betraegt etwa 380 ms.

Die Culling-Operation iteriert durch den Baum top-down: Beginnend mit dem Root wird geprueft, ob dieser den View-Frustum schneidet. Falls nicht, ist der gesamte Subtree nicht sichtbar und kann uebersprungen werden. Falls ja, wird rekursiv in die Kinder abgestiegen. Diese Culling-Operation ist $O(k + \log n)$, wobei k die Anzahl sichtbarer Objekte ist. In realen Szenen ist $k << n$, was den Vorteil des Octrees ausmacht. Fuer die Windkraft-Visualisierung betraegt k typischerweise 300–600 bei 29.722 Gesamtobjekten, eine Reduktion um Faktor 50–100.

3.2 Level-of-Detail Rendering System

Das LOD-System basiert auf drei LOD-Level mit einstellbaren Polygon-Reduktionsfaktoren. Die Default-Konfiguration ist: LOD0 (100% Polygons, Distanz 0m), LOD1 (50%, Distanz 300m), LOD2 (10%, Distanz 800m). Diese Werte wurden empirisch kalibriert fuer die Windkraft-Szene.

[PLACEHOLDER IMAGE:
$\text{lod}_{d}istance_{m}etric.png - Distance - basierteLOD - Selektion$]

Abbildung 4: Distance-basierte LOD-Selektion: Sprueunge markieren Uebergaenge zwischen LOD-Leveln.

Die LOD-Selektion erfolgt per Frame fuer jede sichtbare Turbine: Basierend auf der Distanz zur Kamera wird das passende LOD-Level ausgew aehlt. Das zentrale Problem ist die Mesh-Simplification. Die vereinfachten Geometrien (LOD1 und LOD2) wurden offline mit einem Open-Source-Tool (Meshlab) generiert und in den Datenstrukturen vorgespeichert. Dies ist pragmatisch, aber nicht optimal: Eine echte Progressive-Mesh-Implementierung haette kontinuierlichere LOD-Uebergaenge ermoeglicht.

Der Overhead der LOD-Verwaltung ist klein: Fuer jede sichtbare Turbine ein Distance-Lookup. Die CPU-Zeit liegt im Bereich 20–25 ms fuer die komplette Szene, davon sind nur 2–3 ms fuer die LOD-Selektion. Der Polygon-Reduktion aber ist signifikant: Bei durchschnittlich 450 sichtbaren Turbinen und angenommenen 30% in LOD0, 40% in LOD1, 30% in LOD2, betraegt die durchschnittliche Polygon-Reduktion etwa 64%, im Extremfall 81%.

3.3 Cache-Optimization durch Structure-of-Arrays

Die Cache-Optimierung wurde in zwei Varianten implementiert: Die traditionelle AoS-Variant (Turbine-Objekte als Python-Instanzen) und eine SoA-Variante mit NumPy-Arrays. Ein HybridTurbineCache erlaubt zur Laufzeit zwischen beiden zu wechseln.

Die SoA-Implementierung speichert Turbinenattribute in separaten NumPy-Arrays. Dies ermoeglicht vektorisierte Operationen. Ein klassisches Culling-Beispiel: Filtere alle Turbinen, deren x-Koordinate ausserhalb des Frustum liegt. Bei SoA ist dies vektorisierbar, bei AoS nicht. Der Performance-Unterschied ist dramatisch. Messungen zeigen Speedups von 100x–500x fuer Batch-Culling-Operationen mit NumPy, abhaengig vom spezifischen Workload.

[PLACEHOLDER IMAGE: $\text{cache}_{e}fficiency.png - Cache - EffizienzVergleich$]

Abbildung 5: Cache-Effizienz Vergleich: Linkes Diagramm zeigt Memory-Access-Pattern in AoS-Layout (Streuung), rechts SoA-Layout (sequentiell).

Ein praktisches Problem tritt auf: Python-Objekte sind per Default nicht numerisch. Die vollstaendige Konvertierung der 29.722 Turbinen zum NumPy-Backend dauert etwa 50–80 ms einmalig beim Start, wird aber durch den Speedup in jedem Frame amortisiert.

4. Experimentelle Evaluierung

4.1 Testumgebung und Messprozess

Die Messungen wurden auf einem Standard-Entwicklerrechner durchgefuehrt (Intel i7-9700K, 16 GB RAM, keine dedizierte GPU). Das Rendering erfolgte auf der CPU mit Py-

OpenGL, um die CPU-Bottleneck-Bedingungen zu isolieren. Alle Messungen wurden ueber mindestens 10 Frames gemittelt, um Variabilitaet zu minimieren.

Der Wind kraft-Datensatz enthaelt 29.722 Turbinen mit Positionen und Attributen. Der Visualisierungsmodus zeigte alle Turbinen in einer 3D-Kartenansicht, mit einer Kamera, die ueber Deutschland navigiert. Die Messungen erfassten CPU-Time fuer Culling, LOD-Selektion und Cache-Operationen, nicht die GPU-Time.

4.2 Performance-Messungen

Das Octree-Aufbau dauerte durchschnittlich 378 ms fuer 29.722 Turbinen. Die Query-Time (Culling gegen ein typisches View-Frustum) betrug 12–18 ms je nach Kamera-Position und Frustum-Groesse. Dies ist langsamer als die naive $O(n)$ Iteration (etwa 8–10 ms fuer vollstaendige Iteration), weil der Overhead der Octree-Verwaltung bei relativ grossem Frustum nicht amortisiert wird. Nur bei extremal grossen Szenen oder sehr restriktiven Frusta wird die Octree vorteilhaft.

Die LOD-Implementierung reduzierte die durchschnittliche Polygon-Anzahl um 64% in typischen Szenen und bis zu 81% in extreme Szenen (Kamera weit entfernt). Dies fuehrte zu ca. 20% GPU-Speedup. Der CPU-Overhead der LOD-Verwaltung betrug 2–3 ms.

NumPy-basierte SoA zeigte drammatische Speedups fuer spezifische Batch-Operationen: Culling (Filterung nach Koordinaten) erreichte 342x Speedup, Aggregation (Summe aller Power-Werte) 563x Speedup, Zaehlung (Anzahl Turbinen bis Jahr X) 117x Speedup. Diese extremen Speedups resultieren aus NumPy-Vektorisierung und SIMD-Parallelismus, nicht Cache-Effekten allein. Die realen Performance-Gewinne sind kleiner, weil die CPU-Zeit nicht vollstaendig von diesen Operationen dominiert wird.

Tabelle 1: CPU-Time Breakdown fuer verschiedene Optimierungs-Konfigurationen (pro Frame, in ms)

Konfiguration	Culling	LOD	Cache	GPU-Setup	Gesamt
Baseline	8.2	0.5	1.8	4.5	15.0
+ Octree	15.8	0.5	1.8	4.5	22.6
+ LOD	8.2	2.3	1.8	4.5	16.8
+ Cache (SoA)	3.1	0.5	0.3	4.5	8.4
Octree + Cache	10.2	0.5	0.3	4.5	15.5

Die beste Konfiguration (Octree + Cache) erzielte 15,5 ms CPU-Time, eine Verbesserung von 3,3% gegenueber dem Baseline 15,0 ms. Dies ist eine ueberraschend kleine Verbesserung und zeigt, dass die CPU-Zeit nicht das Optimierungs-Bottleneck ist. Das reale Bottleneck liegt in GPU-Rasterisierung und Driver-Overhead.

5. Kritische Analyse und Limitationen

5.1 Methodische Limitationen

Die experimentelle Validierung dieser Arbeit unterliegt mehreren methodischen Einschraenkungen, die bei der Interpretation der Ergebnisse beruecksichtigt werden muessen.

Erstens war das Messungssetup auf CPU-Rendering beschraenkt (PyOpenGL). Moderne Anwendungen verwenden spezialisierte Renderer (Vulkan, DirectX 12), die GPU-Bottlenecks

anders handhaben. In solchen Systemen waeren die Culling-Bottlenecks groesser relativ zum GPU-Durchsatz, was die Octree-Vorteile amplifizieren koennte.

Zweitens war die Szene-Charakteristik eine planare Verteilung (alle Turbinen auf der Erdoberflaeche). Ein Octree ist in 2,5D-Szenen ineffizienter als in vollstaendig 3D-Szenen mit gleichmaessiger Verteilung. Eine Indoor-Szene mit Okklusion wuerde staerkere Culling-Gewinne zeigen.

Drittens wurde die LOD-Implementierung pragmatisch durch Offline-Simplification realisiert, nicht durch echte Progressive Meshes. Dies fuehrte zu suboptimalen Uebergaengen und verhinderte echte Kontinuitaet.

5.2 Algorithmen-spezifische Limitationen

Das Octree war fuer dieses spezifische Problem nicht optimal. Die raeumliche Verteilung der Windkraftanlagen ist hochgradig geclustert (Ballungsraeume wie Kuesten haben 10x hoehere Dichten als Binnenland). Ein adaptives Octree mit variablen Knoten-Groessen haette besser funktioniert. Auch ein Grid-basierter Ansatz waere fuer planare Verteilungen effizienter.

Die LOD-Distanzmetriken waren heuristische. Eine ideale Implementierung wuerde Screen-Space-Groesse verwenden (wie viele Pixel nimmt das Objekt ein), nicht Euklidische Distanz. Dies wuerde pop-in reduzieren und Uebergang-Punkte optimieren.

Die Cache-Optimization profitierten von der Spezifitaet von NumPy. In einer reinen C/C++-Implementierung wuerden aehnliche Speedups durch explizite Vektorisierung erreichbar sein, aber nicht so muehellos.

5.3 Praktische Implikationen

Trotz der limitierten Performance-Gewinne in dieser spezifischen Szene haben die implementierten Techniken reale Wert. Fuer sehr grosse Szenen (>100.000 Objekte) werden Octree-Gewinne signifikanter. Fuer Outdoor-Szenen mit unregelmae ssiger Verteilung ist raeumliche Indizierung quasi-obligatorisch. LOD ist unentbehrlich fuer beliebig komplexe Meshes. Cache-Optimization ist orthogonal und universell anwendbar.

Die Erkenntnisse dieser Arbeit sind: (1) Nicht blindlings alle Optimierungen anwenden, sondern das Bottleneck identifizieren. (2) Die CPU-Zeit kann nicht allein Bottleneck sein; Profiling ist essentiell. (3) Pragmatische Implementierungen koennen akzeptabel sein, wenn das Runtime-Overhead klein ist.

6. Fazit und Ausblick

Diese Arbeit hat sich mit der praktischen Implementierung und Evaluierung von drei klassischen Optimierungstechniken fuer grossskalige 3D-Visualisierungen befasst. Durch ein reales Windkraft-Visualisierungs-Problem wurde demonstriert, dass theoretisch etablierte Techniken komplexere Tradeoffs in der Praxis aufweisen als in der Literatur oft dargestellt.

Die zentrale Erkenntnis ist, dass Performance-Optimierung spezifisch sein muss. Die Anwendung aller Techniken gleichzeitig ist nicht optimal; stattdessen sollte das reale Bottleneck durch Profiling identifiziert werden. In der Windkraft-Szene war das Bottleneck nicht die CPU-seitige Culling, sondern die GPU-Rasterisierung und der Driver-Overhead. Eine Anwendung mit anderer Geometrie oder Kamera-Dynamik haette andere Bottlenecks gehabt.

Fuer zukuenftige Arbeiten ergeben sich mehrere Richtungen. Eine Implementation mit modernen APIs (Vulkan, DirectX 12) wuerde die Rolle von Culling-Overhead anders beleuchten.

Adaptive Octree-Strukturen koenntentechniken bessere Performance fuer nicht-uniforme Verteilungen bieten. Echte Progressive Mesh-Systeme wuerden kontinuierlichere LOD-Uebergaenge ermöglichen. Und end-to-end profiling mit GPU-Performance-Countern wuerde die CPU-GPU Interaktion besser verstehen lassen.

Abschliessend: Die Optimierung grossskaliger Visualisierungen ist kein Schema-F, sondern erfordert tiefes Verstaendnis der spezifischen Anwendung, Hardware und Rendering-Pipeline. Diese Arbeit hat Tools und Konzepte bereitgestellt; die Anwendung bleibt kontextspezifisch.

Literatur

[1] H. Samet, *The Design and Analysis of Spatial Data Structures*, 1st. Addison-Wesley, 1990, ISBN: 0-201-13011-6.

[2] U. Assarsson und T. Møller, "Optimized View Frustum Culling Algorithms for Bounding Boxes," 1, Bd. 5, 2000, S. 9–22. DOI: 10.1080/10867651.2000.10487528

[3] H. Hoppe, "Progressive Meshes," in *Proceedings of SIGGRAPH '96*, 1996, S. 99–108. DOI: 10.1145/237170.237216

[4] P. Cignoni, C. Montani und R. Scopigno, "A Comparison of Mesh Simplification Algorithms," *Computers & Graphics*, Jg. 22, Nr. 1, S. 37–54, 1997. DOI: 10.1016/S0097-8493(97)00082-X

[5] U. Drepper, "What Every Programmer Should Know About Memory," Red Hat, Inc., Techn. Ber., 2007, Version 1.0.

[6] V. Leis, P. Boncz, A. Kemper und T. Neumann, "Morsel-driven parallelism: a NUMA-aware query engine for the many-core age," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, S. 743–754. DOI: 10.1145/2588555.2610507

[7] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers und P. Virtanen, "Array Programming with NumPy," *Nature*, Jg. 585, S. 357–362, 2020. DOI: 10.1038/s41586-020-2649-2

[8] M. Frigo, C. E. Leiserson, H. Prokop und S. Ramachandran, "Cache-Oblivious Algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, 1999, S. 285–297. DOI: 10.1109/SFCS.1999.814587