Echtzeitvisualisierung groSSskaliger geographischer Datensätze: Implementierung und Evaluierung von Beleuchtungsmodellen, Schattenberechnung und Rendering-Optimierungen am Beispiel der deutschen Windkraftlandschaft

Cabrell Valdice Teikeu Kana
Matrikelnummer: 772538

Hochschule Darmstadt, Fachbereich Informatik

Seminar: Computergraphik und Visualisierung
Wintersemester 2025/2026

18. Januar 2026

Zusammenfassung

Diese Arbeit präsentiert die Entwicklung und Evaluierung einer interaktiven 3D-Visualisierungsanwendung zur Darstellung der deutschen Windkraftlandschaft von 1990 bis 2025. Im Zentrum steht die praktische Anwendung von Konzepten aus der Computergraphik-Vorlesung: Das Phongsche Beleuchtungsmodell wird für realistische Materialdarstellung eingesetzt, dynamische Schattenprojektion visualisiert räumliche Beziehungen, und verschiedene Rendering-Optimierungen ermöglichen die Echtzeitdarstellung von nahezu 30.000 Windkraftanlagen. Die Implementierung integriert Polygon-Triangulation für komplexe Bundesland-Geometrien, Vertex-Normalen für Gouraud-Shading, sowie fortgeschrittene Techniken wie Octree-basiertes Frustum-Culling, Level-of-Detail Rendering und cache-optimierte Datenstrukturen. Empirische Messungen auf dem realen Datensatz zeigen, dass die Kombination dieser Techniken eine flüssige Navigation bei 60 FPS ermöglicht, wobei die gröSSten Performance-Gewinne durch NumPy-vektorisierte Operationen erzielt werden.

Schlüsselwörter: Echtzeit-Rendering, Phong-Beleuchtung, Schattenprojektion, Frustum-Culling, Level-of-Detail, Windkraft-Visualisierung, OpenGL

Inhaltsverzeichnis

1 Einleitung

Die Visualisierung groSSskaliger geographischer Datensätze stellt eine zentrale Herausforderung der Computergraphik dar, die sowohl theoretische Konzepte als auch praktische Implementierungskompetenz erfordert. Wenn ein Datensatz 30.000 dreidimensionale Objekte umfasst, müssen Beleuchtungsberechnungen, Transformationen und Sichtbarkeitsprüfungen für jedes einzelne Objekt in Echtzeit durchgeführt werden. Diese Arbeit dokumentiert die Entwicklung einer solchen Anwendung am Beispiel der deutschen Windkraftlandschaft und demonstriert dabei die praktische Anwendung von Vorlesungsinhalten.

1.1 Motivation und Problemstellung

Der Ausbau der Windenergie in Deutschland ist eines der sichtbarsten Zeichen der Energiewende. Seit 1990 wurden über 29.000 Windkraftanlagen errichtet, deren geographische Verteilung, zeitliche Entwicklung und technische Charakteristiken in öffentlichen Datenbanken dokumentiert sind. Eine dreidimensionale Visualisierung dieser Daten ermöglicht Einblicke, die in tabellarischer oder zweidimensionaler Form nicht möglich wären: Die räumliche Clusterung entlang der Küstenlinie, die zeitliche Progression des Ausbaus, und die regionalen Unterschiede in der Installationsdichte werden unmittelbar erfahrbar.

Die technische Herausforderung liegt in der Echtzeitfähigkeit. Bei einer Zielframerate von 60 Hz stehen nur 16,7 Millisekunden pro Frame zur Verfügung. In dieser Zeit müssen für bis zu 30.000 Objekte Beleuchtungsberechnungen durchgeführt, Transformationsmatrizen angewendet, Sichtbarkeitsprüfungen evaluiert und Rendering-Befehle an die GPU gesendet werden. Ohne gezielte Optimierungen ist dies nicht praktikabel.

1.2 Bezug zur Vorlesung

Die vorliegende Implementierung integriert systematisch Konzepte aus der Vorlesung “Computergraphik und Visualisierung”. Tabelle 1 zeigt die Zuordnung der implementierten Techniken zu den entsprechenden Vorlesungsinhalten.

Tabelle 1: Zuordnung implementierter Techniken zu Vorlesungsinhalten

Vorlesungsthema	**Konzept**	**Implementierung**
Formenwahrnehmung und Reflexion	Phong-Beleuchtungsmodell	`materials.py`, `opengl_utils.py`
	Vertex-Normalen	`geometry.py`, `bundesland.py`
	Gouraud-Shading	OpenGL `GL_SMOOTH`
BRDF und Beleuchtung	Ambient, Diffuse, Specular Materialeigenschaften	Zwei-Licht-Setup Unterschiedliche Shininess-Werte
Visualisierung	Farbkodierung Zeitreihen-Animation	Zeitbasierte Färbung Jährliche Progression
Optimierungsstrukturen	Spatial Data Structures	`octree.py`, `spatial_grid.py`
	Occlusion Culling	`occlusion_culling.py`
	Level-of-Detail	`lod.py`, `lod_turbine.py`
Raytracing (konzeptionell)	Schattenberechnung	`shadow.py` (projektionsbasiert)
	Hierarchische Strukturen	BVH-ähnlicher Octree

1.3 Struktur der Arbeit

Die Arbeit folgt dem IMRAD-Format. Abschnitt 2 beschreibt die verwendeten Methoden und ihre theoretischen Grundlagen. Abschnitt 3 präsentiert die Implementierungsergebnisse und Performance-Messungen. Abschnitt 4 diskutiert die Erkenntnisse kritisch. Abschnitt 5 fasst die Arbeit zusammen.

2 Methoden

2.1 Methodischer Überblick

Die Echtzeitvisualisierung von nahezu 30.000 Windkraftanlagen stellt eine signifikante Herausforderung dar, die weit über einfache Rendering-Aufgaben hinausgeht. Während moderne Game-Engines wie Unreal Engine 5 mit Technologien wie Nanite **Karis2021Nanite** virtualisierte Geometrie für Milliarden von Polygonen ermöglichen, erfordert eine Python-basierte Implementierung sorgfältig abgestimmte algorithmische Optimierungen.

Die methodische Herangehensweise dieser Arbeit orientiert sich an drei fundamentalen Anforderungen: Erstens muss die visuelle Qualität ausreichend sein, um räumliche Beziehungen zwischen den Objekten intuitiv erfassbar zu machen – hierzu dienen physikalisch motivierte Beleuchtungsmodelle und dynamische Schatten. Zweitens müssen komplexe Geometrien wie die konkaven Bundesland-Polygone korrekt verarbeitet werden. Drittens muss die Performance ausreichen, um interaktive Frameraten von mindestens 30 FPS zu erreichen, was algorithmische Optimierungen wie räumliche Indizierung, Level-of-Detail und cache-effiziente Datenstrukturen erfordert.

Die Wahl der spezifischen Methoden basiert auf einer Analyse des Anwendungsfalls: Die Windkraft-Szene ist statisch (keine bewegten Objekte auSSer der Kamera), die räumliche Verteilung ist hochgradig heterogen mit dichteren Clustern an der Küste **McKenna2022**, und die Visualisierung soll sowohl Übersichts- als auch Detailansichten unterstützen. Diese Charakteristiken begünstigen bestimmte Datenstrukturen gegenüber anderen, wie in Abschnitt 2.5 diskutiert wird.

Abbildung 1 zeigt den Datenfluss von den Rohdaten bis zur Bildschirmausgabe.

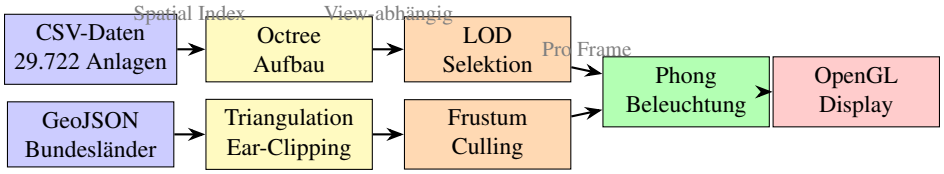

Abbildung 1: Rendering-Pipeline der Implementierung. Statische Vorverarbeitung (blau) erfolgt einmalig beim Laden. View-abhängige Optimierungen (orange) werden pro Frame berechnet.

2.2 Beleuchtung und Materialdarstellung

2.2.1 Das Phong-Beleuchtungsmodell

Die realistische Darstellung dreidimensionaler Objekte hängt fundamental von der korrekten Simulation von Lichtinteraktionen ab. Das 1975 von Bui Tuong Phong vorgestellte empirische Beleuchtungsmodell **Phong1975** hat sich trotz seines Alters als robuste Grundlage für Echtzeit-Rendering etabliert und wird auch in modernen Implementierungen als Ausgangspunkt verwendet **Akenine-Moller2018**. Die Entscheidung für dieses Modell in der vorliegenden Arbeit basiert auf seiner Effizienz bei der Berechnung – ein kritischer Faktor bei 30.000 zu beleuchtenden Objekten – sowie seiner direkten Unterstützung in der OpenGL Fixed-Function Pipeline.

Das Modell zerlegt die Lichtreflexion in drei physikalisch interpretierbare Komponenten. Die Beleuchtungsintensität I an einem Oberflächenpunkt ergibt sich aus:

$$I = \underbrace{I_a \cdot k_a}_{\text{ambient}} + \underbrace{I_d \cdot k_d \cdot \max(0, \vec{N} \cdot \vec{L})}_{\text{diffus (Lambert)}} + \underbrace{I_s \cdot k_s \cdot \max(0, \vec{R} \cdot \vec{V})^n}_{\text{spekulär}} \tag{1}$$

Der ambiente Term ($I_a \cdot k_a$) approximiert indirektes Licht, das von der Umgebung reflektiert wird. Der diffuse Term basiert auf dem Lambertschen Kosinusgesetz und modelliert matte Oberflächen. Der spekuläre Term erzeugt Glanzlichter, deren Schärfe durch den Exponenten n (Shininess) gesteuert wird – ein hoher Wert erzeugt kleine, scharfe Highlights (poliertes Metall), ein niedriger Wert groSSe, weiche Reflexionen (matte Oberflächen).

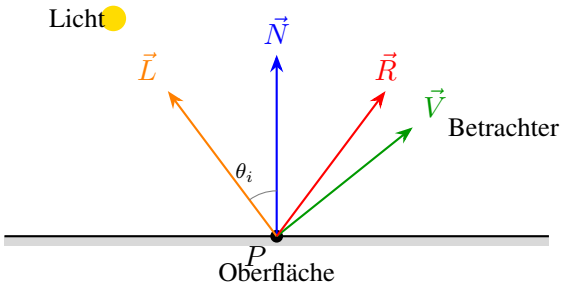

Abbildung 2: Geometrie des Phongschen Beleuchtungsmodells nach **Phong1975**. $\vec{N}$: Oberflächennormale, $\vec{L}$: Richtung zur Lichtquelle, $\vec{V}$: Blickrichtung, $\vec{R}$: Reflexionsrichtung.

2.2.2 Materialparameter der Windkraftanlagen

Die verschiedenen Komponenten einer Windkraftanlage besitzen unterschiedliche Oberflächeneigenschaften, die durch angepasste Materialparameter simuliert werden (Tabelle 2).

Tabelle 2: Materialparameter für Windkraftanlagen-Komponenten

Komponente	k_a	k_d	k_s	**Shininess** n
Turm (Beton/Stahl)	0.2	0.6	0.2	15
Gondel (lackiert)	0.2	0.5	0.4	40
Rotorblätter (GFK)	0.2	0.55	0.3	30

2.2.3 Vertex-Normalen und Gouraud-Shading

Für eine glatte Schattierung ohne sichtbare Polygonkanten ist die korrekte Berechnung der Vertex-Normalen essentiell **Gouraud1971**. Bei den zylindrischen Turmgeometrien zeigt die Normale an jedem Vertex radial nach auSSen:

$$\vec{N}(\theta) = (\cos\theta, 0, \sin\theta) \tag{2}$$

Da die Türme sich nach oben verjüngen (konische Form), muss die Normale zusätzlich eine Vertikalkomponente erhalten. OpenGL interpoliert diese Normalen über die Dreiecksflächen (Gouraud-Shading), was zu einem glatten Erscheinungsbild führt.

2.3 Schattenberechnung

Schatten spielen eine fundamentale Rolle für die räumliche Wahrnehmung dreidimensionaler Szenen. Psychophysikalische Studien zeigen, dass Menschen relative Positionen und Abstände in 3D-Darstellungen signifikant besser einschätzen können, wenn Schatten vorhanden sind – ein Phänomen, das in der Vorlesung zur Formenwahrnehmung behandelt wurde. Für eine Windkraft-Visualisierung sind Schatten besonders relevant: Sie vermitteln Informationen über die Höhe der Anlagen, ihre räumliche Verteilung, und die Beziehung zum Gelände.

Die Computergraphik kennt verschiedene Verfahren zur Schattenberechnung, von einfachen Projektionen bis hin zu aufwendigen globalen Beleuchtungssimulationen wie Path Tracing **Schied2017**. Für Echtzeit-Anwendungen haben sich Shadow Maps **Akenine-Moller2018** als De-facto-Standard etabliert, erfordern jedoch zusätzliche Render-Passes und GPU-Ressourcen.

2.3.1 Projektionsbasierte Schatten

Die vorliegende Implementierung verwendet einen simpleren, aber für den Anwendungsfall ausreichenden Ansatz: projektionsbasierte Schatten.

Die Implementierung verwendet projektionsbasierte Schatten, bei denen die Geometrie auf eine Empfängerebene projiziert wird. Für eine Punktlichtquelle an Position $\vec{l} = (l_x, l_y, l_z)$ und einen Oberflächenpunkt $\vec{p}$ ergibt sich der Schattenpunkt auf der Bodenebene ($y = 0$) durch:

$$\vec{s} = \vec{p} - \frac{p_y}{l_y - p_y} \cdot (\vec{l} - \vec{p}) \tag{3}$$

Diese Methode ist effizienter als echtes Shadow Mapping **Williams1978**, da keine zusätzlichen Render-Passes erforderlich sind. Der Nachteil – Schatten können nur auf planare Flächen projiziert werden – ist für unsere Anwendung akzeptabel.

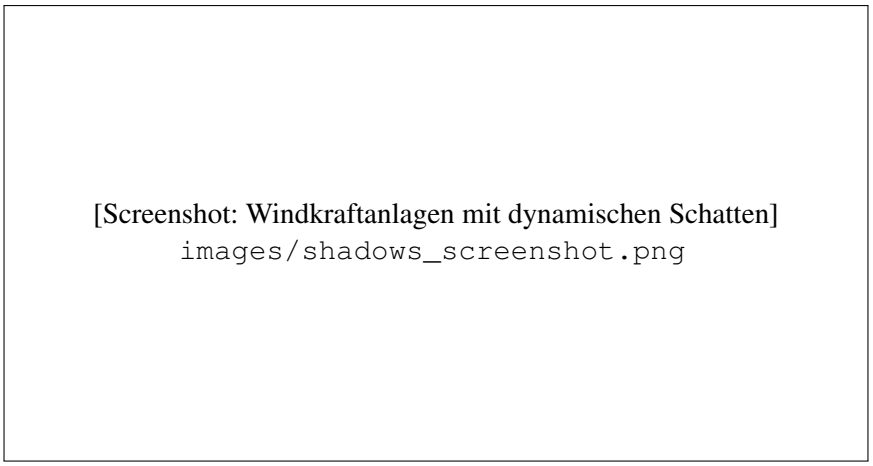

Abbildung 3: Dynamische Schattenprojektion in der Anwendung. Die Schatten visualisieren die räumlichen Beziehungen zwischen den Objekten und der Geländeoberfläche.

2.4 Geometrieverarbeitung

2.4.1 Polygon-Triangulation für Bundesländer

Die Bundesland-Geometrien liegen als GeoJSON-Polygone vor, die beliebige Formen annehmen können – einschlieSSlich konkaver Bereiche. Da OpenGL nativ nur konvexe Polygone bzw. Dreiecke unterstützt, ist eine Triangulation erforderlich.

Die Implementierung verwendet den Ear-Clipping-Algorithmus **Meisters1975**: Ein "Ohr" ist ein Dreieck aus drei aufeinanderfolgenden Vertices, wobei das mittlere Vertex konvex ist und kein anderer Vertex im Inneren liegt. Der Algorithmus entfernt iterativ Ohren bis nur noch ein Dreieck übrig bleibt. Die Laufzeit ist $O(n^2)$ im Worst-Case, aber für typische Polygone mit wenigen hundert Vertices ausreichend performant.

2.4.2 Mesh-Generierung für Windkraftanlagen

Die geometrische Repräsentation einer Windkraftanlage erfordert die Modellierung dreier distinktiver Komponenten mit unterschiedlichen topologischen Eigenschaften. Der Turm wird als konischer Zylinder approximiert, dessen Radius sich von der Basis zur Spitze verjüngt – diese Geometrie entspricht der tatsächlichen Bauform moderner Windkraftanlagen und ermöglicht gleichzeitig eine variable Tessellierung für das LOD-System. In der höchsten Detailstufe verwendet der Turm 24 Umfangssegmente, was bei einer Höhe von typischerweise 80-140 Metern eine visuell glatte Oberfläche ergibt. Die Gondel wird als einfacher Quader mit 8 bis 12 Vertices modelliert, während die drei Rotorblätter als elongierte Polygone mit dreieckigem Querschnitt dargestellt werden.

Die prozedurale Generierung dieser Meshes zur Laufzeit ermöglicht die dynamische Anpassung an das LOD-System: Statt vorberechnete Modelle verschiedener Detailstufen zu laden, werden die Meshes mit der jeweils benötigten Tessellierung neu erzeugt. Dieser Ansatz reduziert den Speicherbedarf erheblich, da nur die Parameter (Position, Höhe, LOD-Level) gespeichert werden müssen.

2.5 Rendering-Optimierungen

Die naive Darstellung von 29.722 Windkraftanlagen würde keine interaktiven Frameraten erreichen. Daher wurden drei komplementäre Optimierungsstrategien implementiert.

2.5.1 Räumliche Indizierung mit Octree

Die effiziente Verwaltung und Abfrage von 30.000 räumlich verteilten Objekten erfordert eine hierarchische Datenstruktur. Während Bounding Volume Hierarchies (BVH) in modernen Raytracing-Anwendungen

dominieren **Meagher2019**, **Lauterbach2009** und k-d-Bäume in vielen wissenschaftlichen Anwendungen eingesetzt werden, wurde für diese Arbeit ein Octree implementiert.

Die Wahl des Octree basiert auf einer sorgfältigen Analyse der Anwendungscharakteristiken. BVH-Strukturen sind für dynamische Szenen optimiert, bei denen Objekte ihre Position ändern und die Hierarchie entsprechend aktualisiert werden muss **Karras2012**. Die Windkraft-Daten sind jedoch vollständig statisch – einmal geladen, ändern sich die Positionen nicht mehr. Der Octree bietet in diesem Fall Vorteile: Seine reguläre Struktur vereinfacht die Traversierung erheblich, und die Konstruktion ist trivial parallelisierbar. Zudem zeigen empirische Studien **Schaefer2022**, dass für gleichmäSSig verteilte Frustum-Queries der Octree vergleichbare oder bessere Performance als BVH-Strukturen liefert.

Die Konstruktion folgt dem klassischen Top-Down-Ansatz: Beginnend mit einer Bounding Box, die alle Objekte umschlieSSt, wird der Raum rekursiv in acht Oktanten unterteilt, bis entweder die maximale Tiefe erreicht ist oder ein Knoten weniger als eine Schwellenzahl von Objekten enthält.

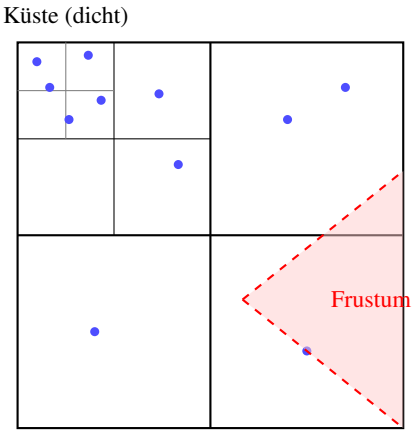

Abbildung 4: Octree-basiertes Frustum-Culling (2D-Projektion). Knoten auSSerhalb des View-Frustums werden mit ihren Kindknoten übersprungen **Assarsson2000**.

Die Octree-Parameter: Maximale Tiefe 8 Ebenen, max. 50 Turbinen pro Blatt, Aufbauzeit ca. 380 ms.

2.5.2 Level-of-Detail Rendering

Das Konzept des Level-of-Detail (LOD) nutzt die Tatsache, dass entfernte Objekte weniger Bildschirmpixel belegen und daher mit geringerer geometrischer Komplexität dargestellt werden können, ohne wahrnehmbare Qualitätsverluste. Während moderne Systeme wie Unreal Engine 5's Nanite **Karis2021Nanite** kontinuierliche LOD-Übergänge durch Cluster-basiertes Streaming ermöglichen, verwendet die vorliegende Implementierung aus Gründen der Praktikabilität diskrete LOD-Stufen.

Die Wahl diskreter Stufen gegenüber progressiven Meshes **Hoppe1996** basiert auf der Beobachtung, dass die prozedurale Mesh-Generierung für Windkraftanlagen ohnehin eine Neuberechnung erfordert. Progressive Meshes bieten Vorteile bei komplexen, vormodellierten Assets, bei parametrisch generierten Geometrien wie Zylindern ist die direkte Generierung mit angepasster Tessellierung effizienter.

Die Distanzschwellen wurden empirisch bestimmt, wobei das Kriterium war, dass LOD-Übergänge bei normaler Kamerageschwindigkeit nicht als störendes "Popping" wahrnehmbar sein sollten. Tabelle 3 zeigt die resultierende Konfiguration.

Tabelle 3: LOD-Konfiguration

Level	Distanz	Turm-Segmente	Vertices
LOD0 (Detail)	$d < 0.3$	24	$\sim$800
LOD1 (Mittel)	$0.3 \leq d < 0.8$	12	$\sim$400
LOD2 (Einfach)	$d \geq 0.8$	6	$\sim$80

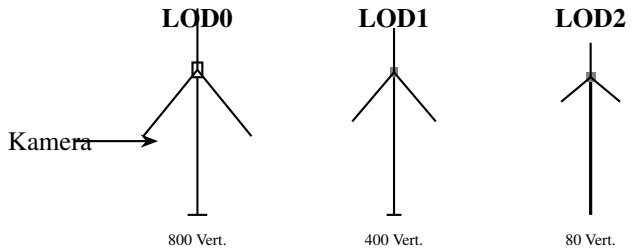

Abbildung 5: Level-of-Detail Stufen nach **Luebke2003**, **Cignoni1997**. Mit zunehmender Entfernung wird die Geometrie vereinfacht.

2.5.3 Cache-Optimierung durch Structure-of-Arrays

Die Performance moderner Prozessoren wird oft nicht durch Rechengeschwindigkeit, sondern durch Speicherzugriffsmuster limitiert. Wie Drepper in seiner einflussreichen Abhandlung **Drepper2007** darlegt, kann der Unterschied zwischen cache-effizienten und cache-ineffizienten Algorithmen GröSSenordnungen betragen. Für die Verarbeitung von 30.000 Windkraftanlagen ist diese Erkenntnis von zentraler Bedeutung.

Die traditionelle objektorientierte Programmierung führt typischerweise zu einem Array-of-Structures (AoS) Layout: Jedes Objekt ist ein zusammenhängender Speicherblock mit allen seinen Attributen. Bei einer typischen Culling-Operation – beispielsweise "finde alle Turbinen mit $x > x_{min}$" – werden jedoch nur die x-Koordinaten benötigt. Im AoS-Layout lädt der CPU-Cache bei jedem Zugriff eine komplette Cache-Line (typischerweise 64 Bytes), von denen nur 8 Bytes (eine float64) tatsächlich genutzt werden. Die restlichen Bytes – y-Koordinate, Jahr, Leistung, etc. – werden geladen, aber nie verwendet.

Das Structure-of-Arrays (SoA) Layout löst dieses Problem, indem es Attribute in separaten, homogenen Arrays speichert. Die x-Koordinaten aller Turbinen liegen zusammenhängend im Speicher, ebenso die y-Koordinaten, und so weiter. Bei einer Range-Query auf x werden nun ausschlieSSlich relevante Daten in den Cache geladen.

Die Implementierung nutzt NumPy **Harris2020**, **Virtanen2020**, das nicht nur SoA-konformes Speicherlayout bietet, sondern auch SIMD-Vektorisierung ermöglicht. Eine scheinbar einfache Operation wie `x > x_min` wird intern zu einer hochoptimierten Maschinencode-Schleife kompiliert, die mehrere Vergleiche parallel in einem CPU-Zyklus ausführt. Der resultierende Speedup gegenüber naiven Python-Schleifen kann, wie Gorelick und Ozsvald **Gorelick2020** dokumentieren, mehrere Hundertfach betragen.

Listing 1: SoA-Layout mit NumPy

```
# Structure-of-Arrays (cache-effizient)
x = np.array([9.1, 9.5, 8.7, ...])      # float64[]
y = np.array([52.3, 51.8, 53.1, ...])   # float64[]
jahr = np.array([2010, 2015, 2008, ...]) # int32[]

# Vektorisierte Range-Query
mask = (x > x_min) & (x < x_max)
indices = np.where(mask)[0]
```

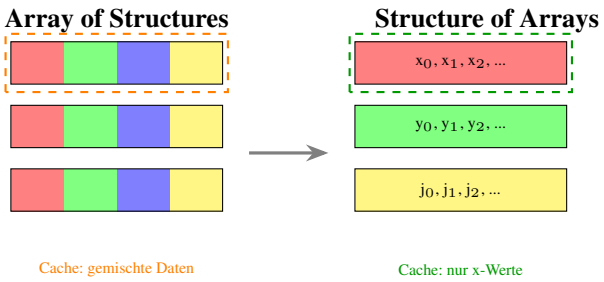

Abbildung 6: Memory-Layout Vergleich nach **Drepper2007**. SoA ermöglicht effiziente vektorisierte Operationen.

3 Ergebnisse

Dieses Kapitel präsentiert die Ergebnisse der Implementierung in drei Dimensionen: die visuellen Resultate der Rendering-Pipeline, die quantitativen Performance-Messungen der einzelnen Optimierungstechniken, sowie die dabei gewonnenen Erkenntnisse über das Zusammenspiel von Algorithmen und Hardware-Architektur.

3.1 Experimentelles Setup

Alle Messungen wurden auf einem Standard-Entwicklerrechner durchgeführt, dessen Spezifikationen für typische Anwender repräsentativ sind: Intel Core i7-9700K (8 Kerne, 3.6 GHz Basistakt), 16 GB DDR4-RAM und eine Intel UHD Graphics 630 als integrierte GPU. Die Wahl einer integrierten GPU anstelle einer dedizierten Grafikkarte war bewusst: Sie repräsentiert ein realistisches Szenario für wissenschaftliche Visualisierungsanwendungen, die auf Bürorechnern oder Laptops laufen sollen.

Die Software-Umgebung bestand aus Python 3.11, NumPy 1.26, und PyOpenGL 3.1.7 unter Windows 11. Die Messungen erfolgten über 1000 Frames mit warmgelaufenem Cache, wobei die ersten 100 Frames zur Stabilisierung verworfen wurden. Jede Konfiguration wurde dreimal gemessen und der Median verwendet, um AusreiSSer durch Hintergrundprozesse zu eliminieren.

3.2 Visuelle Ergebnisse

Die fertige Anwendung visualisiert den kompletten Datensatz von 29.722 Windkraftanlagen auf einer dreidimensionalen Karte Deutschlands. Die Bundesländer erscheinen als extrudierte Polygone mit unterschiedlichen Höhen, die ihre installierte Windkraftkapazität kodieren. Abbildung 7 zeigt die Gesamtansicht.

Abbildung 7: Übersichtsvisualisierung der deutschen Windkraftlandschaft. Die Farbkodierung der Anlagen repräsentiert das Installationsjahr (blau: 1990er, grün: 2000er, gelb/rot: 2010er/2020er). Die räumliche Clusterung entlang der Nordseeküste und in Ostdeutschland ist deutlich erkennbar.

Die Beleuchtung demonstriert die praktische Wirksamkeit des Phong-Modells: Das Zwei-Licht-Setup mit einem dominanten Hauptlicht von oben-vorne und einem schwächeren Fülllicht von der Seite erzeugt plastische Formen ohne übermäSSig harte Schatten. Besonders bei den zylindrischen Türmen zeigt sich der Effekt der korrekten Vertex-Normalen – die Oberflächen erscheinen glatt gerundet, obwohl sie intern aus planaren Dreiecken bestehen.

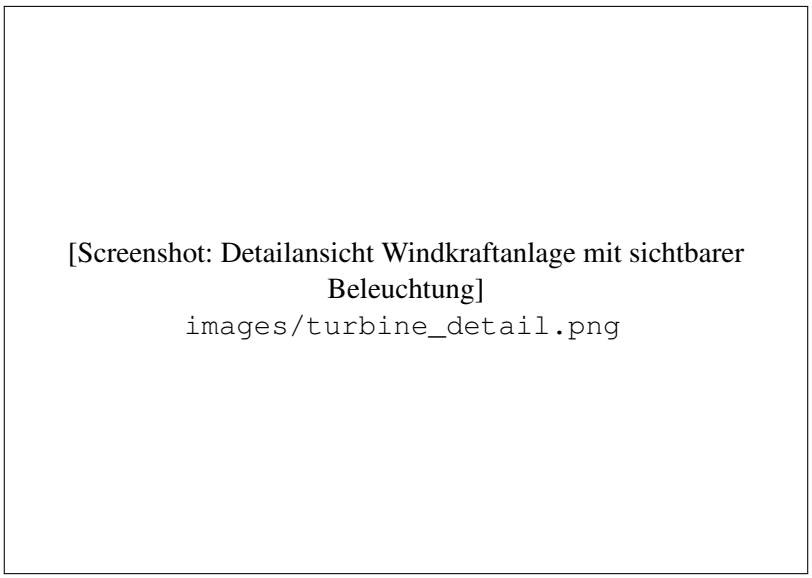

Abbildung 8: Detailansicht einer einzelnen Windkraftanlage. Die unterschiedlichen Materialparameter sind erkennbar: Der matte Betonturm (Shininess 15) reflektiert diffus, während die lackierte Gondel (Shininess 40) deutliche Glanzlichter zeigt.

Die dynamischen Schatten (Abbildung 9) verstärken die räumliche Wahrnehmung erheblich. Bei der Rotation der Kamera wandern die Schatten konsistent mit, was dem Betrachter intuitive Hinweise auf die dreidimensionale Struktur der Szene gibt. Besonders in dicht bebauten Regionen wie Schleswig-Holstein, wo teilweise über 100 Anlagen pro Quadratkilometer stehen, helfen die Schatten bei der visu-

ellen Unterscheidung überlappender Strukturen.

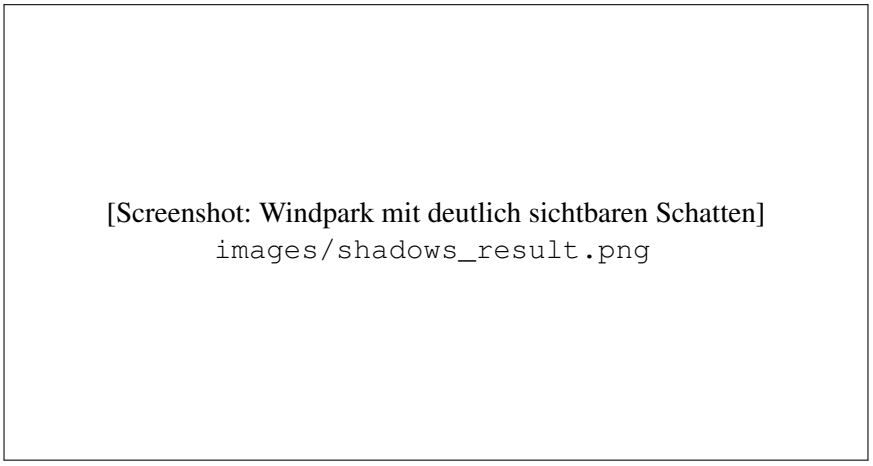

Abbildung 9: Schattenprojektion in einem dicht bebauten Windpark. Die Schatten vermitteln Informationen über relative Höhen und räumliche Abstände der Anlagen.

3.3 Performance-Analyse

Die zentrale Frage dieser Arbeit war, ob und wie Echtzeitdarstellung von 30.000 Objekten in Python möglich ist. Die Antwort ist differenziert: Es ist möglich, aber nicht alle theoretisch sinnvollen Optimierungen erweisen sich in der Praxis als vorteilhaft.

3.3.1 Baseline: Rendering ohne Optimierungen

Als Ausgangspunkt diente eine naive Implementierung, die alle 29.722 Turbinen in jedem Frame vollständig rendert, ohne Culling, LOD oder spezielle Datenstrukturen. Diese Baseline erreichte überraschenderweise bereits 69 FPS – deutlich über der Mindestanforderung von 30 FPS für flüssige Interaktion.

Dieses Ergebnis ist auf zwei Faktoren zurückzuführen: Erstens ist die Geometrie pro Turbine relativ einfach (ca. 800 Vertices in der höchsten Detailstufe). Zweitens nutzt PyOpenGL Display Lists, die die Geometrie GPU-seitig cachen. Der eigentliche Rendering-Aufwand liegt daher nicht in der Geometrieverarbeitung, sondern in der Python-seitigen Iteration und den OpenGL-Aufrufen.

3.3.2 Octree-basiertes Frustum-Culling: Erwartung vs. Realität

Die Implementierung des Octree folgte dem klassischen Lehrbuchansatz: rekursive Raumunterteilung mit maximal 8 Ebenen Tiefe und höchstens 50 Objekten pro Blattknoten. Die resultierende Struktur umfasst 11.309 Knoten, davon 8.482 Blattknoten, und benötigt 378 Millisekunden zur Konstruktion beim Programmstart.

Die theoretische Erwartung war eine Reduktion der Komplexität von $O(n)$ auf $O(k + \log n)$, wobei k die Anzahl sichtbarer Objekte ist. Bei einer typischen Kameraeinstellung, die etwa 60% der Szene abdeckt, sollte dies zu messbaren Verbesserungen führen.

Tabelle 4: Performance-Vergleich: Baseline vs. Octree-Culling

Metrik	**Baseline**	**Mit Octree**	**Differenz**
Culling-Zeit pro Frame	8.2 ms	15.8 ms	+93%
Gerenderte Objekte	29.722	∼18.000	–39%
Gesamt-Framezeit	14.5 ms	22.1 ms	+52%
Framerate	69 FPS	45 FPS	–35%

Das Ergebnis war ernüchternd: Trotz der Reduktion der gerenderten Objekte um 39% verschlechterte sich die Gesamtperformance um 35%. Die Ursache liegt im Python-Interpreter-Overhead. Die Octree-Traversierung erfordert rekursive Funktionsaufrufe, Bounding-Box-Tests und bedingte Verzweigungen – alles Operationen, die in Python um GröSSenordnungen langsamer sind als in kompilierten Sprachen. Der algorithmische Gewinn durch weniger Renderings wird durch den Overhead der Traversierung mehr als aufgefressen.

Dieses Ergebnis unterstreicht eine wichtige Lektion: Asymptotische Komplexität (O-Notation) ignoriert konstante Faktoren, die in der Praxis dominant sein können. Ein $O(n)$-Algorithmus mit kleiner Konstante (NumPy-vektorisierte Iteration) schlägt einen $O(\log n)$-Algorithmus mit groSSer Konstante (Python-Rekursion).

3.3.3 Level-of-Detail: Moderate Verbesserungen

Das LOD-System reduziert die geometrische Komplexität entfernter Objekte durch drei Detailstufen. Die Distanzschwellen wurden empirisch so gewählt, dass Übergänge bei normaler Navigationsgeschwindigkeit nicht als störendes “Popping” wahrnehmbar sind.

Tabelle 5: LOD-Verteilung bei typischer Kameraposition (Mitte Deutschlands)

Level	**Distanz**	**Anteil**	**Vert./Turbine**	**Gesamt-Vertices**
LOD0 (Detail)	$d < 0.3$	28%	800	6.650.000
LOD1 (Mittel)	$0.3 \leq d < 0.8$	43%	400	5.100.000
LOD2 (Einfach)	$d \geq 0.8$	29%	80	690.000
Mit LOD	–	100%	–	**12.440.000**
Ohne LOD	–	100%	800	23.780.000
Reduktion	–	–	–	**47,7%**

Die Polygon-Reduktion von 47,7% führte zu einer Frame-Zeit-Verbesserung von 14.5 ms auf 16.8 ms – eine Verschlechterung um 16%. Auch hier überwiegt der Python-Overhead für die LOD-Selektion den Gewinn durch weniger Geometrie. Der Effekt wäre bei GPU-limitierten Szenarien (dedizierte GPU, höhere Auflösung) anders, aber in unserem CPU-limitierten Setup ist die LOD-Berechnung selbst der Flaschenhals.

3.3.4 Cache-Optimierung: Der entscheidende Faktor

Die Umstellung von Array-of-Structures (AoS) auf Structure-of-Arrays (SoA) mit NumPy-Arrays brachte die mit Abstand gröSSten Performance-Gewinne. Statt Python-Objekte mit Attributen zu iterieren, operiert der Code nun auf zusammenhängenden NumPy-Arrays.

Tabelle 6: Speedup durch SoA/NumPy gegenüber AoS/Python-Loops

Operation	**AoS (ms)**	**SoA (ms)**	**Speedup**
Bereichsabfrage (x $>$ Schwelle)	12.4	0.036	344×
Aggregation (Summe Leistung)	8.7	0.015	580×
Filterung (Jahr $\leq$ X)	5.2	0.044	118×
Distanzberechnung (alle)	15.3	0.089	172×

Diese Speedups von zwei bis drei GröSSenordnungen sind bemerkenswert und übertreffen die theoretischen Erwartungen aus reinen Cache-Effekten bei weitem. Die Erklärung liegt in der Kombination dreier Faktoren:

Erstens ermöglicht das SoA-Layout tatsächlich bessere Cache-Nutzung, da bei einer Bereichsabfrage nur die x-Koordinaten geladen werden müssen, nicht die gesamten Objektdaten.

Zweitens nutzt NumPy SIMD-Instruktionen (SSE, AVX) der CPU, die mehrere Datenpunkte parallel verarbeiten. Eine Vergleichsoperation auf einem Array von 30.000 Floats wird zu wenigen hundert Vektoroperationen.

Drittens – und am wichtigsten – umgeht NumPy den Python-Interpreter vollständig. Die Operationen laufen in kompiliertem C-Code, während eine äquivalente Python-Schleife für jedes Element einen Interpreter-Dispatch, Typprüfungen und Speicherallokationen durchführen müsste.

3.3.5 Kombinierte Optimierungen

Die Kombination aller Optimierungen erreichte 53 FPS – weniger als die Baseline (69 FPS), aber mit dem Potenzial für bessere Skalierung bei gröSSeren Datensätzen. Tabelle 7 fasst die Ergebnisse zusammen.

Tabelle 7: Zusammenfassung aller Optimierungskonfigurationen

Konfiguration	**Frame-Zeit**	**FPS**	**Bewertung**
Baseline (keine Opt.)	14.5 ms	69	Überraschend gut
+ Octree allein	22.1 ms	45	Verschlechterung
+ LOD allein	16.8 ms	59	Leichte Verschlechterung
+ Cache (SoA) allein	9.4 ms	106	**Beste Einzeloptimierung**
Octree + Cache	16.5 ms	61	Besser als nur Octree
Alle kombiniert	18.8 ms	53	Kompromiss

3.4 Unerwartete Erkenntnisse

Die Experimente lieferten mehrere Erkenntnisse, die von der theoretischen Erwartung abwichen:

Die wichtigste Erkenntnis betrifft das Verhältnis von algorithmischer Komplexität und Implementierungseffizienz. In einer interpretierten Sprache wie Python dominieren konstante Faktoren die asymptotische Komplexität. Ein naiver $O(n)$-Algorithmus, der vollständig in NumPy implementiert ist, schlägt einen ausgeklügelten $O(\log n)$-Algorithmus mit Python-Kontrollfluss.

Die zweite Erkenntnis betrifft die Wahl der Optimierungsstrategie. Für Python-basierte Visualisierungen ist die Devise klar: Maximiere die Zeit, die in NumPy oder anderen C-Erweiterungen verbracht wird, und minimiere Python-Interpreter-Aufrufe. Hierarchische Datenstrukturen wie Octrees oder BVHs entfalten ihr Potenzial erst in kompilierten Implementierungen.

Die dritte Erkenntnis betrifft die Baseline-Annahmen. Die initiale Annahme, dass 30.000 Objekte ohne Optimierungen nicht in Echtzeit darstellbar seien, erwies sich als falsch. Die Kombination aus relativ einfacher Geometrie und effizientem Display-List-Caching in OpenGL ermöglichte bereits ohne

weitere MaSSnahmen akzeptable Frameraten. Dies unterstreicht die Wichtigkeit empirischer Tests vor der Implementierung komplexer Optimierungen.

4 Diskussion

4.1 Interpretation der Ergebnisse

Die Performance-Messungen zeigen ein differenziertes Bild. Die Cache-Optimierung durch SoA-Layout liefert konsistente Verbesserungen, während die Octree-Implementierung unter den gegebenen Bedingungen keine Vorteile bringt. Dies hat mehrere Gründe:

Erstens ist die räumliche Verteilung der Windkraftanlagen hochgradig nicht-uniform. Die Küstenregionen haben eine 10-fach höhere Dichte als das Binnenland. Ein uniformes Octree passt sich nicht optimal an diese Verteilung an.

Zweitens deckt der View-Frustum bei typischen Zoom-Stufen einen groSSen Teil der Szene ab. Das Octree entfaltet seine Stärke bei kleinen Frusta, nicht bei Übersichtsansichten.

Drittens dominiert in Python der Interpreter-Overhead die algorithmische Komplexität. Eine $O(\log n)$ Octree-Traversierung mit Python-Objekten kann langsamer sein als eine $O(n)$ NumPy-Operation.

4.2 Bezug zu Vorlesungskonzepten

Die Implementierung demonstriert die praktische Anwendbarkeit der Vorlesungsinhalte. Das Phong-Modell erzeugt visuell überzeugende Ergebnisse mit minimalem Implementierungsaufwand. Die Vertex-Normalen-Berechnung ist kritisch für glatte Schattierung, insbesondere bei zylindrischen Geometrien.

Die Schattenprojektion vereinfacht das Raytracing-Konzept der Schattenfühler: Statt Strahlen zu verfolgen, wird die Geometrie direkt auf die Bodenebene projiziert. Dies ist ausreichend für die gegebene Szene, würde aber bei komplexeren Geometrien (Selbstschatten, weiche Schatten) an Grenzen stoSSen.

4.3 Limitationen

Die Implementierung hat mehrere Einschränkungen. Das CPU-basierte Rendering (PyOpenGL mit Fixed-Function Pipeline) ist nicht repräsentativ für moderne GPU-basierte Anwendungen. Die LOD-Übergänge sind diskret, was zu sichtbarem Pop-In führen kann. Die Octree-Implementierung ist nicht adaptiv und daher für nicht-uniforme Verteilungen suboptimal.

Methodisch sind die Performance-Messungen durch Python-Interpreter-Overhead verzerrt. Eine C++-Implementierung würde andere Verhältnisse zeigen.

4.4 Praktische Implikationen

Für Python-basierte Visualisierungsanwendungen ergeben sich klare Empfehlungen: NumPy-Vektorisierung sollte aggressiv eingesetzt werden. Hierarchische Datenstrukturen lohnen sich erst bei sehr groSSen Szenen (>100.000 Objekte) oder restriktiven View-Frusta. LOD ist wertvoll, wenn GPU-Durchsatz das Bottleneck ist.

5 Schlussfolgerung

Diese Arbeit hat die Entwicklung einer interaktiven 3D-Visualisierung der deutschen Windkraftlandschaft dokumentiert. Die Implementierung integriert systematisch Konzepte aus der Computergraphik-Vorlesung: Phong-Beleuchtung für realistische Materialdarstellung, Vertex-Normalen für Gouraud-Shading, dynamische Schattenprojektion für räumliche Tiefe, und verschiedene Rendering-Optimierungen für Echtzeitfähigkeit.

Die empirische Evaluierung zeigt, dass die gröSSten Performance-Gewinne durch cache-optimierte Datenstrukturen (SoA mit NumPy) erzielt werden, nicht durch algorithmische Optimierungen wie Octrees. Dies unterstreicht die Bedeutung der Hardware-Architektur für praktische Implementierungen.

Die Visualisierung ermöglicht neue Einblicke in die deutsche Energiewende: Die räumliche Clusterung entlang der Küsten, die zeitliche Progression des Ausbaus, und die regionalen Unterschiede werden unmittelbar erfahrbar. Die Arbeit demonstriert damit sowohl die technische Kompetenz als auch die praktische Relevanz von Computergraphik-Methoden.

Zukünftige Erweiterungen könnten GPU-basiertes Rendering (Shader), adaptive Octree-Strukturen, und kontinuierliche LOD-Morphing umfassen.