Computergenerierte Bleistiftzeichnungen von 3D-Stadtmodellen

Daniel Müller

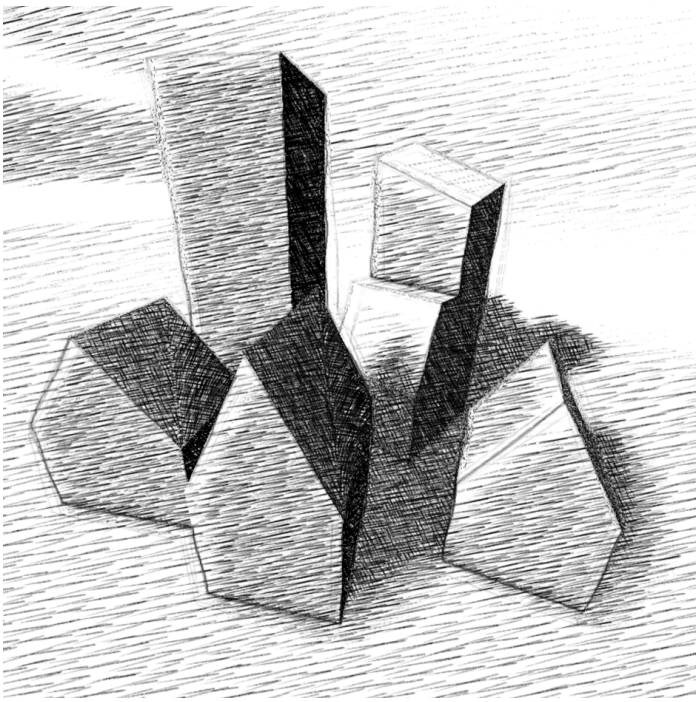


Abbildung 1: *Bauklotzstadt schraffiert.*

Zusammenfassung Diese Arbeit befasst sich mit der Erstellung von Bleistiftzeichnungen dreidimensionaler Modelle in Echtzeit und prüft die Anwendbarkeit auf virtuelle 3D-Stadtmodelle. Bisherige Verfahren gehen kaum auf stilprägende subjektive Einflüsse der Zeichner ein. So werden stilrelevante Maßnahmen wie weiche Schatten und Lichteinflüsse durch Umgebungsverdeckung häufig vernachlässigt. Nach einer kurzen Zusammenfassung bisheriger Ansätze zur Nachahmung von Bleistiftzeichnungen, werden effiziente Implementierung für Konturdarstellung und Krümmungsapproximation zur Schraffur vorgestellt. Da die meisten Berechnungen diskretisiert im Bildraum vorgenommen werden, entstehen beim Rendern komplexer 3D-Stadtmodelle störende, mit der Tiefe zunehmende Artefakte. Zur Abschwächung dieser Störfaktoren werden praktische Lösungen wie stilkonforme Tiefenfokussierung diskutiert.

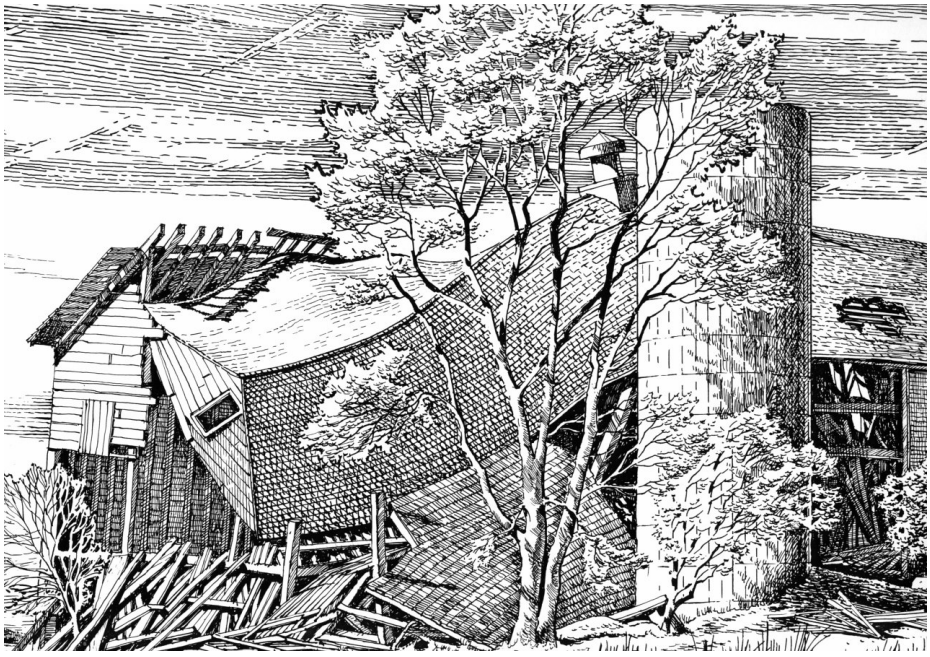


Abbildung 2: *Zeichnung im Pen&Ink-Stil.*

1 Einleitung

Der Bleistift ist eines der weltweit am häufigsten verwendeten Werkzeuge zum Zeichnen. Ob Skizzen, Notizen, Portraits, Landschaftsbilder, Mangas oder Comics - der Bleistift findet in nahezu allen Bereichen Verwendung. Da in technischen Bereichen häufiger mit Entwürfen, statt mit Visualisierungen vollständig ausmodelierter Modelle, gearbeitet wird und CAD-Konturen bzw -Drahtgittermodelle nicht sehr ästhetisch aussehen, lohnt es, sich im Rahmen der nicht-photorealistischen Bilderzeugung mit diesem Thema zu befassen. Konturzeichnungen helfen, einzelne Abschnitte bzw. Bestandteile komplexer 3D Objekte voneinander abzugrenzen und werden häufig für technische Zeichnungen verwendet (z.B. Illustrationen in Handbüchern, Patentskizzen). Während der Recherche ist mir bewusst geworden, wie viele verschiedene Ansätze und vor allem noch nicht umgesetzte Konzepte in diesem Bereich existieren. Den Erweiterungsmöglichkeiten sind kaum Grenzen gesetzt.

Beim Zeichnen mit „Feder und Tinte“ greift man nur auf einen Ton zurück. Eine Abstufung erfolgt durch Schraffuren oder verschiedenfarbige Tinte. Die Granularität der Abstufung hängt von dem verwendeten Stiftwerkzeug ab (Abbildung 2). Je breiter die Stiftspitze, desto weniger Abstufungen bei der Schraffierung und somit der Helligkeits und Kontrastverteilung.

Die Verwendung eines Bleistifts erweitert durch die Beschaffenheit der Bleistiftmine, welche aus Graphit, Wachs und Ton besteht, die Schattierung um

übergangslose Schraffierungen (Abbildung 3). Dabei können nur Töne die im Intensitätsbereich des Bleistifts liegen, abgedeckt werden. Die Menge an Graphit und Ton sind ausschlaggebend für die Härte der Mine. Umso mehr Graphit enthalten ist, desto weicher und dichter ist sie, desto stärker ist folglich der Abrieb und desto höher die erreichbare Intensität. Die bekannten Härtegrade B, HB, F und H entstammen dem Englischen: black, hard-black, firm und hard. Bisher gibt es jedoch keine weltweit standardisierte Härtegradunterteilung.

Abbildung 3: *Bleistiftzeichnung.*

2 Finden und Darstellen von Kanten

Fast jede Zeichnung beginnt mit einer groben Skizze. Dabei werden Konturen skizziert, welche am stärksten wahrgenommen werden und die charakteristischen Züge der Szenenobjekte abbilden. Diese anfänglich eher groben Konturen werden während des Zeichnens fortlaufend verfeinert und bleiben bis zum Schluss Bestandteil des Bildes (siehe Abbildung 4). Bei anderen Stilen überlagern die Konturen meist die Schraffuren. Eine klare und deutliche Darstellung wichtiger Konturen der 3D-Szenengeometrie kann folglich die geometrischen Eigenschaften verstärken und somit die Wahrnehmung erleichtern.

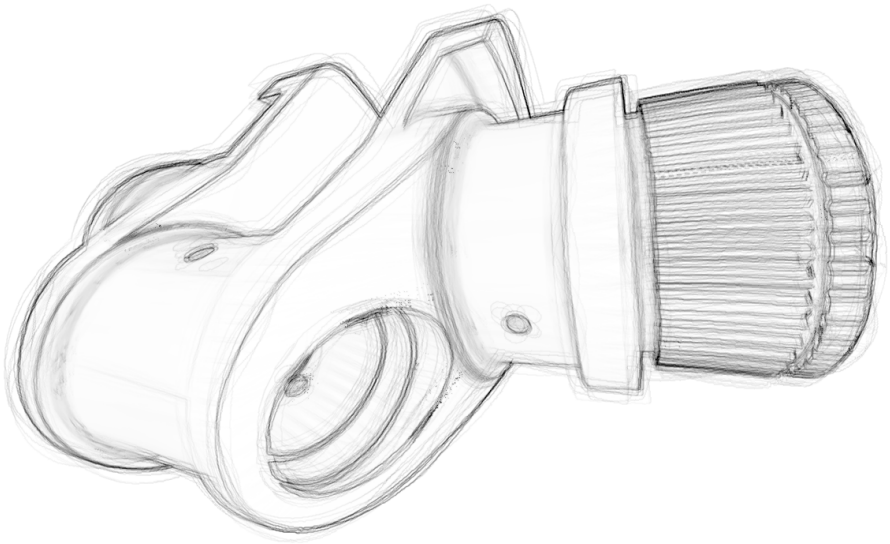


Abbildung 4: *Kurbelwelle als Konturzeichnung - Ergebnis des Pencil-Shader.*

2.1 Kantenklassifizierung

Nienhaus & Döllner (2003) zufolge weisen dreidimensionale Objekte, welche durch polygonale 3D Geometrien angenähert sind, folgende grundlegende Kantentypen auf:

- **Umrisskanten** oder auch Silhouettenkanten sind Kanten zweier einander angrenzender Dreiecke, wobei das eine zur Kamera (*front-facing*) und das andere in Kamerablickrichtung (*back-facing*) orientiert ist.
- **Randkanten** sind Kanten eines einzelnen Dreiecks.
- **Faltenkanten** sind Kanten zwischen einander angrenzender, zur Kamera orientierter Dreiecke.

Neben diesen haben DeCarlo et al. (2004) mit *suggestive contours* ein weiteren Kantentypus beschrieben. Derartige Kanten entstehen an starken Oberflächenkrümmungen. Da die Berechnung dieser Krümmungen, wie beispielhaft in Abschnitt 4 beschrieben, sehr aufwendig und nur bedingt echtzeitfähig ist, werden diese im Folgenden nicht weiter betrachtet.

2.2 Kantenfindung

Kanten durch Drahtgitter darzustellen ist unzureichend, da keine visuelle Unterscheidung zwischen wirklichen Kanten und Dreiecksgrenzen möglich ist. Sie kann sogar die Wahrnehmung erschweren. Zur Zeit existieren verschiedene Ansätze die entweder im Objektraum (Buchanan & Sousa, 2000; Hertzmann & Zorin, 2000), im Bildraum (Nienhaus & Döllner, 2003; Mitchell et al., 2002) oder in

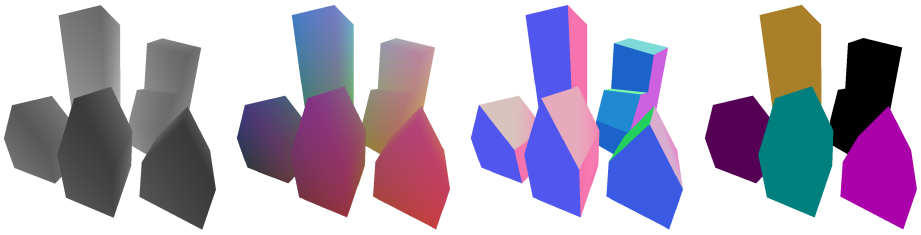
Beiden charakteristische Kanten der 3D-Geometrie aufspüren und verstärken. Objektraumbasierte Algorithmen iterieren über polygonale Kanten oder gleichwertigen geometrischen Annäherungen und erzeugen sichtbare Kanten durch gerenderte Geometrie. Hybride Lösungen greifen auf ähnliche Daten zurück, erzeugen die Kanten jedoch erst im Bildraum oder verbessern bestimmte Darstellungseigenschaften. Für eine ausführliche Beschreibung sei auf (Isenberg et al., 2003) verwiesen. Da alle Kantentypen nach (Nienhaus & Döllner, 2003) auch im Bildraum auftreten, habe ich unter Berücksichtigung von (Sousa & Buchanan, 1999; Loviscach, 2002; Nienhaus & Döllner, 2003, 2004; Lee et al., 2006) einen Fragment-Shader erstellt, der Konturen findet und für deren Darstellung verschiedene Zeichentechniken nachahmt - dazu mehr in Abschnitt 2.3.

Bildbasierte Algorithmen nutzen Unstetigkeiten in G-Buffern (Saito & Takahashi, 1990) welche während der Rasterisierung entstehen und extrahieren diese mit bildverarbeitenden Methoden. Somit können diese Verfahren leicht in Shadern auf die GPU ausgelagert werden und sind normalerweise um Größeneinheiten schneller als Methoden im Objektraum. Des Weiteren verhalten sie sich relativ robust gegenüber Geometriefehlern und sind unabhängig von der Komplexität der Szene. Nachteile sind zum einen, fehlende analytische Kantenbeschreibungen - es kann folglich keine weiter Kantenprozessierung, wie beispielsweise in (Döllner & Walther, 2003) vorgenommen werden - und zum anderen, je nach Kernel und Auflösung des Buffers, entstehenden Treppeneffekte (Aliasing) bei der Darstellung (Abbildung 6). Des Weiteren müssen die gefundenen Kanten keine deutlich erkennbaren Start- bzw. Endpunkte aufweisen und können in Intensität und Dicke nur sehr eingeschränkt angepasst werden.

G-Buffer und ihre Eigenschaften: Ein nahe liegender Weg Linien zu finden ist die Kantendetektion im Farb-Buffer. Dabei sollte darauf geachtet werden, dass eine aussagekräftige Farbgebung durch z.B. Texturen vorhanden ist und das Beleuchtung und Schattierung noch nicht angewandt wurden. Darin gefundene Kanten sind unabhängig von der Geometrie und können bei richtiger Verwendung zusätzliche, nicht ausmodellierte Details darstellen.

Den Tiefen- oder auch Z-Buffer genannten G-Buffer zu verwenden, wurde erstmals von Saito & Takahashi (1990) vorgeschlagen. Dieser ermöglicht das Auffinden von C^0-Unstetigkeiten in der Geometrie und liefert ausschließlich Objektkanten (vor allem Silhouetten). Die Z-Werte sollten dazu linearisiert sein. Der Geometrie-Buffer bildet, ähnlich dem Z-Buffer, die interpolierten Koordinaten der Geometrie z.B. in den RGB-Farbkanälen ab. Dabei kann zwischen Sichtraum (*Eyespace*) und Objektraum gewählt werden. Die Unstetigkeitsstellen unterscheiden sich aufgrund der gleichartigen Geometrieinformationen einander nicht, was dazu führt, dass Geometrie-Buffer für Kantendetektion kaum verwendet werden.

Hertzmann & Zorin (2000) haben die Idee des Tiefen-Buffers weitergeführt, indem sie die interpolierten Geometrie-Normalen in die einzelnen Farbkanälen kodierten, um auch C^1-Unstetigkeiten (somit auch Faltenkanten) abzubilden. Sehr nützlich können hier mit Normalmaps texturierte Geometrien sein - bisher hat davon, unverständlicher Weise, noch keiner Gebrauch gemacht. Da der Tiefen-



(a) Depth-Buffer **(b)** Geometry-Buffer **(c)** Normal-Buffer **(d)** ID-Buffer

Abbildung 5: *Je nach Art, Inhalt und Format der Buffer müssen Unstetigkeiten unterschiedlich stark gewichtet werden.*

Buffer nur einen Farbkanal belegt, bietet es sich an, diesen im vierten Farbkanal (Alpha) abzubilden. So kann man in nur einer RGBA-Textur Normalen und Tiefen speichern und benötigt folglich weniger Speicher und, viel wichtiger, nur die Hälfte an Texturzugriffen. Bei weitläufigeren Szenen ist auch dieser Ansatz unzureichend. Die Tiefenunterschiede können nicht mehr korrekt verarbeitet werden, da bei großen Tiefensprüngen entweder keine Kanten für relativ dicht beieinander liegenden Objekten gefunden werden können oder die großen Tiefensprünge zu sehr harten Kanten und somit starken Treppenbildung führen. Hier bietet es sich an, den einzelnen Objekten der Szene, eindeutig eine ID zuzuordnen und diese als Farbe in den Buffer zu kodieren. Obwohl im ID-Buffer aus jedem Farbsprung, unabhängig von dessen Stärke, eine harte Kante höchster Intensität entstehen muss, kann die Verwendung zur Kantenergänzung z.B. in 3D-Stadtmodellen hilfreich sein (siehe Abschnitt 5).

Bei Verwendung von Schatten-Buffern, kann man die Informationen - sofern es ein harter Schatten ist (z.B. Stencil-Shadow) - direkt im Normalen-Buffer abbilden, indem man an schattierten Pixeln die Normale invertiert (Mitchell et al., 2002). Der Nutzen dieses Buffers zur Kantendetektion ist fragwürdig und im Bereich der Bleistiftzeichnungen kaum zu gebrauchen. Zumal man mittlerweile auch im Echtzeitbereich auf weiche Schattierungen zurückgreift um harte, unglaubwürdige Schattenkanten zu vermeiden. Abbildung 5 zeigt häufig verwendete G-Buffer.

Kantendetektoren: Nienhaus & Döllner (2003) berechnen die Stärke I_N der Unstetigkeit eines Pixel im Normalen-Buffer mit

$$I_N = \frac{1}{2}\left(\boldsymbol{a}\cdot\boldsymbol{h} + \boldsymbol{c}\cdot\boldsymbol{f}\right), \tag{1}$$

wobei $\{\boldsymbol{a},\ \boldsymbol{h}\}$ und $\{\boldsymbol{c},\ \boldsymbol{f}\}$ die bezüglich des Pixel diagonalen, sich jeweils gegenüberliegenden Normalen sind. Für die Intensität I_Z der Tiefenunstetigkeiten werden zwei unterschiedliche Funktionen vorgeschlagen:

$$I_{Z1} = \left(1 - \frac{|\boldsymbol{a}-\boldsymbol{h}|}{2}\right)^2 \cdot \left(1 - \frac{|\boldsymbol{c}-\boldsymbol{f}|}{2}\right)^2 \tag{2}$$

$$
\begin{aligned}
I_{Z2} =& \frac{1}{8}\big(|\boldsymbol{a}-\boldsymbol{x}| + |\boldsymbol{c}-\boldsymbol{x}| + |\boldsymbol{f}-\boldsymbol{x}| + |\boldsymbol{h}-\boldsymbol{x}|\big) \\
&+ \frac{1}{4}\big(|\boldsymbol{b}-\boldsymbol{x}| + |\boldsymbol{d}-\boldsymbol{x}| + |\boldsymbol{e}-\boldsymbol{x}| + |\boldsymbol{g}-\boldsymbol{x}|\big)
\end{aligned} \tag{3}
$$

Formel 2 bezieht sich wie auch Formel 1 nur auf die vier, diagonal am Pixel anliegenden Punkte. Formel 3 entspricht Sobel's Gradienten eines Pixel $\boldsymbol{x}$, welcher alle umliegenden Punkte berücksichtigt, und wird von Saito & Takahashi (1990) empfohlen. Der Behauptung, I_{Z1} führe zu besseren Ergebnissen bzw. geringerem Aliasing muss ich widersprechen. Beide Kanten weisen zum Teil starke Aliasing Artefakte auf. I_{Z1} verstärkt lediglich die Randpixel der Kanten, was im Eindruck zu einer breiteren Kontur führt - jedoch nicht zu einer Glatteren. Des Weiteren führt die stärkere Gewichtung bereits kleiner Unstetigkeiten zu leichten Intensitätsverläufen (bzw. sehr hellen, breiten Kanten). Der G-Buffer sollte für optimale Ergebnisse die normalisierten Objektnormalen sowie die linearisierten Tiefenwerte im RGBA32F-Format enthalten. Wendet man vor der Kantendetektion einen einfachen Sharpen-Kernel mit den Faktor ≈ -0.1 auf den Normalen-Tiefen-Buffer an, erhält man weichere, gleichzeitig aber auch stärkere Konturen (Abbildung 6).

(a) Überblick **(b)** I_{Z2} **(c)** I_{Z1} **(d)** $I_{Z2_{blurred}}$

Abbildung 6: *Vergleich der zwei Ansätze (b) und (c) zur Berechnung von Unstetigkeits-Intensitäten.*

2.3 Kantendarstellung in Bleistiftzeichnungen

Loviscach (2002) stellen fest, dass beim Zeichnen einer Kontur mehrmals angesetzt wird und es somit zu einer feinen Überlagerung mehrerer Konturen kommt. Die Nachahmung dieses Phänomens erhöht die Qualität einer generierten Bleistiftzeichnung enorm. Zudem lässt sich diese Idee um zusätzliche, noch gröbere und hellere Konturen erweitern, was wie anfänglich erwähnt, der natürlichen Abfolge beim Zeichnen entspricht. Eine Umsetzung in Echtzeit wurde erstmalig von Lee et al. (2006) vorgenommen. Diese habe ich für aktuelle Shader (GLSL 1.20 - Shadermodel 4) stark optimiert und erweitert. Für den vollständigen Pencil-Shader sei auf Anhang B verwiesen.

Pencil-Shader: Für die Überlagerung mehrerer Konturen, benötigt man voneinander verschiedene Konturen. Die einfachste Möglichkeit ist die aktuell zu verarbeitende Texturkoordinate in jedem Durchlauf durch eine wellenartige Transformation (Verwackelung) zu variieren:

$$u = u + \lambda \cdot \sin\big(P(\omega_{min}, \omega_{max}) \cdot \sigma_y + P(0, 2\pi)\big) - P(-1, +1) \tag{4}$$

und

$$v = v + \lambda \cdot \cos\big(P(\omega_{min}, \omega_{max}) \cdot \sigma_x + P(0, 2\pi)\big) - P(-1, +1) \tag{5}$$

mit den ursprünglichen Texturkoordinaten u und v, λ als frei wählbare Amplitude, minimaler bzw. maximaler Wellenlänge ω der Abweichung, $P(min, max)$ als Funktion, welche einen zufälligen Wert zwischen min und max liefert, sowie die normalisierte Texturkoordinate σ. Die Intensität I ergibt sich schließlich aus $I_N \cdot (1 - I_{Z^2})$. Hier ist darauf zu achten, dass - unter Verwendung von I_N - der Buffer flächendeckend eine Normale zurück gibt, da sonst an den Rändern der Geometrie keine Unstetigkeiten auftreten und die Kanten folglich nicht darüber hinausgehen. In den meisten Fällen hilft eine grob tesselierte, invertierte Hüllsphere oder nochmaliges normalisieren der gelesenen Normalen. Damit die gröberen, stärker verwackelten Konturen geringfügig „älter“ wirken, werden diese mit Hilfe einer im Shader gebleichten Intensitätstextur (z.B. normalisierter und wiederholbarer Perlin Cloud Noise) verblendet[1]. Anschließend werden die groben mit den feinen Konturen verblendet[2]. Da die Konturen in ihrer Gruppe jeweils anteilig summiert werden, wird mit zunehmender Anzahl, einander überlagernder Konturen, die finale Kontur immer weicher. Dies führt schon bei wenigen Überlagerungen ≥ 3 zu geringeren Treppeneffekten. Je nach Amplitude führen 6 und mehr Überlagerungen zu sehr weichen und breiten Konturen, was je nach Zeichentechnik gewünscht bzw. unerwünscht sein kann. Für die Generierung der Zufallswerte diente eine eindimensionale RGBA-Textur mit zufälligen Werten in allen Kanälen. Bei der Wahl einer geeigneten Breite reichen schon sehr kleine Texturen aus. Wählt man als Breite ein Primzahl (ich habe meist 1021 verwendet), wiederholen sich die Zufallswerte bei regelmäßigen Offsets nicht. Folgende Vorteile ergeben sich im Vergleich zu (Lee et al., 2006) und (Nienhaus & Döllner, 2003) durch meine Implementierung:

- zur Laufzeit einstellbare Anzahl an jeweiligen Überlagerungen - Begrenzung nur durch Hardware
- bessere Auflösung, da die Konturen zur Überlagerung nicht verkleinert gerendert werden müssen
- schnellere Kantendetektion, da aufgrund des kombinierten G-Buffers weniger Texturzugriffe nötig sind
- höhere Gesamtperformance, da nur zwei Passes notwendig sind (Rendern und Verarbeiten des Buffers)
- höhere Anpassungsfähigkeit an verschiedene Zeichentechniken (konfigurierbare Wellenlängen, Amplituden, ...)

[1] Screen-Blending - siehe (ISO, 2008)
[2] Darken-Blending - siehe (ISO, 2008)

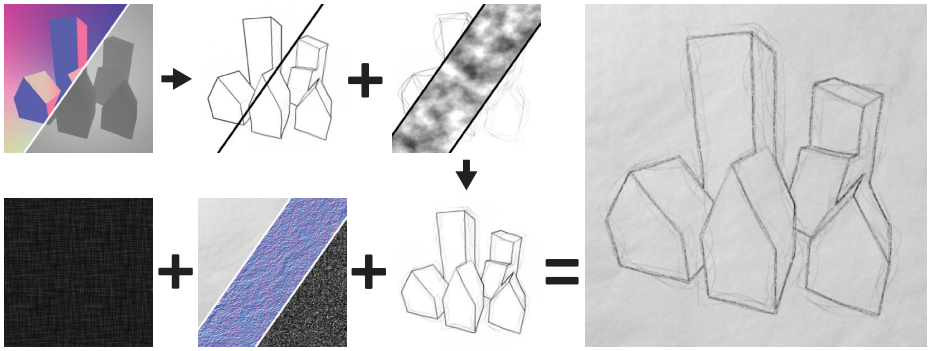


Abbildung 7: *Ablauf und Ressourcen Skizze der Konturdetektion und -darstellung mit groben Merkmalen einer Bleistiftzeichnung. Die groben Umrisse im Hintergrund sind hier zur besseren Sichtbarkeit unnatürlich groß gewählt.*

– wesentlich weniger Grafikkarten- und Arbeitsspeicher, da nur ein Buffer benötigt wird.

Von den in (Sousa & Buchanan, 1999) aufgeführten, physikalisch sowie technisch begründeten Eigenschaften, bildet meine Implementierung trotz allem nur einen Bruchteil ab. Zur weiteren Optimierung können die Texturzugriffe bei der Kantenfindung gepuffert (*cachen*) werden. Um die Verständlichkeit des Shader zu vereinfachen, habe ich in der im Anhang B abgebildeten Implementierung darauf verzichtet.

Vortäuschen von Papier- und Bleistift-Strukturen: Die entstandenen Konturen können nun mit einer generierten Bleistiftschraffur verblendet werden[3]. Anschließend werden diese auf fotografierte oder auch generierte (z.B. sehr feiner Perlin Noise) Papierstrukturen geblendet und in der Intensität beeinflusst. Dazu errechne ich aus der Papiertextur eine Normalmap und verstärke abhängig vom jeweiligen Winkel zur Bildschirmnormalen die Konturen. Somit entsteht der Eindruck eines unregelmäßigen Graphitabriebs des Bleistifts an der Papieroberfläche, ähnlich wie in (Lee et al., 2006) vorgeschlagen (Abbildung 8) . Diese Schritte können in einem einzigen weiteren Pass integriert werden. Bei der Bewegung im virtuellen 3D-Raum führen statische, zweidimensionale Papierstrukturen zu „Shower-Door"-Effekten. Hier haben wir (Stephan Richter und Ich) verschiedene Ansätze experimentell ausprobiert und vielversprechende Ergebnisse erhalten. Auf diese wird im Rahmen dieser Arbeit allerdings nicht eingegangen. Eine Übersicht über den beschrieben Ablauf ist in Abbildung 7 aufgeführt.

[3] Generierung von TAMs siehe Abschnitt 3

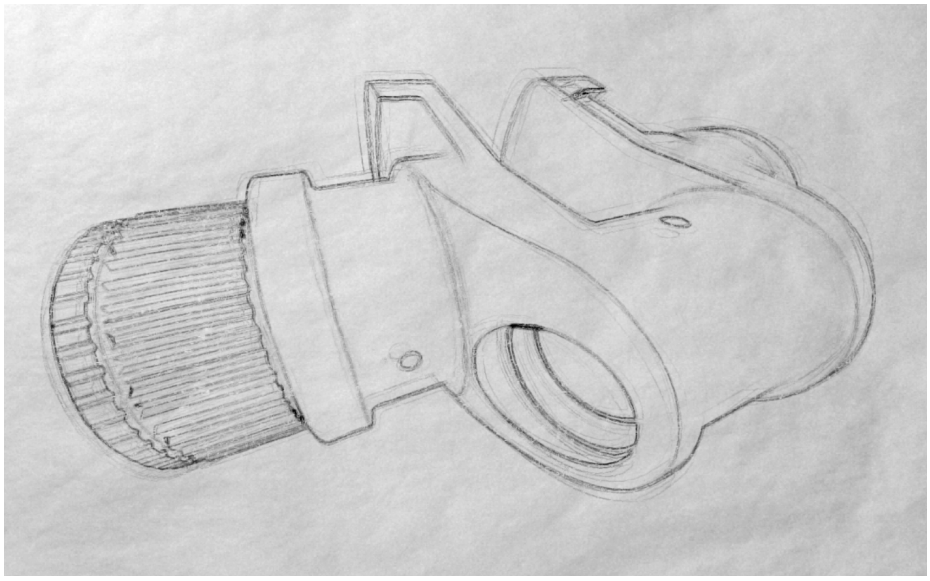


Abbildung 8: *Kurbelwelle als Konturzeichnung mit Bleistifttextur, Papierstruktur und Papierhintergrund.*

3 Erzeugung von Tonal-Art-Maps

In diesem Abschnitt wird eine vereinfachte Technik zur Erzeugung von Bleistifttexturen vorgestellt. Wie in Abschnitt 2.3 erwähnt, weisen Bleistifte bzw. Bleistiftzeichnungen eine Reihe unterschiedlicher Eigenschaften auf. Farbe, Richtung und Dicke eines Linienzuges, die Beziehung zum Papiermaterial und dessen Oberflächenstruktur, Drehung, Auflagewinkel, Position der führenden Hand und diverse Effekte der Graphitüberlagerungen wurden in (Sousa & Buchanan, 1999) ausgiebig untersucht, mathematisch beschrieben und teilweise simuliert. Eine Simulation all dieser Eigenschaften mit den Mitteln der Echtzeit-Computergrafik ist aus meiner Sicht nicht in absehbarer Zeit realisierbar. Ausgehend von (Praun et al., 2001; Webb et al., 2002; Lee et al., 2006) zeige ich einen einfachen und schnellen Weg, Texturen in einem Vorbereitungsschritt, zur Laufzeit zu generieren. Die Texturen sollen helfen, unterschiedliche Intensitäten der Schraffur und Schattierung von Bleistiftzeichnungen abzubilden. Realen Schraffuren angelehnt, sollten sich auch generierte Texturen aus Strichen zusammensetzen. Greift man, wie in den meisten Verfahren, nur auf einen einzigen Strich zurück, sollte dieser folgende Eigenschaften aufweisen:

- keine prägnanten Merkmale, da sich diese bei hundertfachen Überlagerungen in wiederholt auftretenden Mustern widerspiegeln
- geradliniger Verlauf
- gleichmäßige Intensitätsverteilung
- vergleichbare Start- und Endpunkte

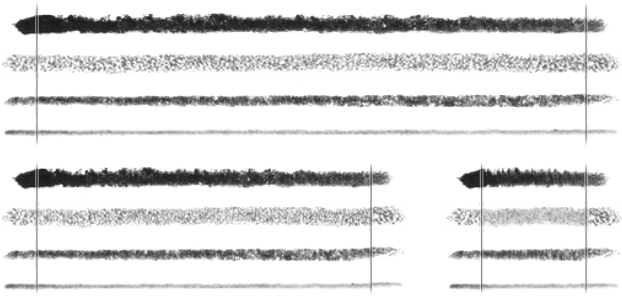
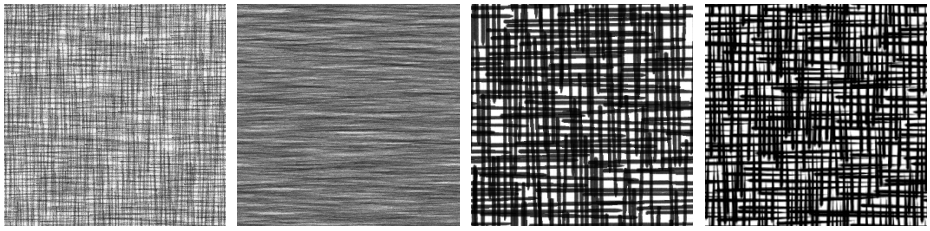


Abbildung 9: *Vier Aufnahmen von Strichen unterschiedlicher Stifte (oben), sowie Skalierungsbeispiele (unten). Stiftarten von oben beginnend: Aqua-Bunti, 3B Bleistift, Lyra-Color-Giant und HB Bleistift.*

Im Idealfall greift man auf eine große Menge ähnlicher Striche zurück. Sogenannte *Brushes*, wie sie in modernen Bildbearbeitungs- und Zeichenprogrammen vordefiniert sind, könnte man automatisiert an zufälligen Liniensegmenten entlang führen und somit große Mengen an individuellen Strichen generieren. Bis auf (Sousa & Buchanan, 1999) wurde stets auf analog oder digital von Hand erstellte Striche zurückgegriffen. So habe auch ich verschiedenste Striche auf Papier gezeichnet und diese hochauflösend eingescannt. Auf 512px×32px skaliert und mit Alphakanal versehen, lässt sich bereits eine große Bandbreite an Einsatzgebieten (nicht nur Bleistift) ausmachen. Wie auch in (Döllner & Walther, 2003) sind Start und Ende für alle Linien so angepasst, dass eine dynamische Skalierung des Zwischensegments möglich ist (siehe Abbildung 9). Für starke Verkleinerung ist diese Technik unzureichend. Um Skalierungen mit geringeren Strukturverlusten zu ermöglichen, können die Anforderungen erweitert werden, sodass jeder Strich aus mehreren Liniensegmenten besteht, welche, einzeln oder in Kombination mit anderen Segmenten, übergangslos mit Start- und Endpunkt verbunden werden können.

Es gibt zwei Arten von Schattierungen: Zum einen Schraffuren wie z.B. Kreuzschraffuren (Abbildung 10a) und zum anderen Gradationen, also übergangslose Tonabstufungen (Abbildung 10b). Obgleich die einzelnen Schraffurschichten in beliebige Richtungen möglich sind, werden in der Computergrafik haupsächlich einfache Kreuzschraffuren verwendet. Schraffuren zeichnen sich durch wenige Überlappungen und gleichgroße Abstände aus. Die Intensität wird durch die Anzahl der Striche und die Größe ihrer Abstände zueinander genau definiert. Gradationen legen durch flächendeckende Überlagerung der Striche ihren Intensitätsgrad fest. Eine grobe Ausrichtung der Striche ist meist vorhanden. Um diesen Effekt zu erreichen, passt man bei der Texturerzeugung die Transparenz der Striche an.



(a) Kreuzschraffur **(b)** weiche Tonabstufung **(c)** naive Platzierung **(d)** Praun et al. (2001) Platzierung

Abbildung 10: *Generierte Bleistifttexturen im Vergleich. Die Qualität der Platzierung im Naiven Ansatz (c) unterscheidet sich nur gering von der durch (d).*

3.1 Positionierung von Strichen

Eine willkürliche Positionierung einzelner Striche verursacht oft eine ungleichmäßige Intensitätsverteilung und kann zu störenden Effekten bei der Texturierung führen, da hier meist nur mehrere Texturausschnitte ineinander verblendet werden. Die Positionierung muss gleichverteilt und trotzdem unregelmäßig, zufällig sein. Als naiven, ersten Ansatz habe ich die zu Beginn leere Fläche in, von der angestrebten Intensität und der Strichdicke abhängige, gleichgroße Streifen eingeteilt und anschließend jeden Abschnitt mit einem Strich gefüllt. In Richtung der zum Streifen parallelen Achse dient die gesamte Texturbreite zur zufälligen Positionierung. In orthogonaler Richtung nur die Dicke des Streifens - um Überschneidungen einzelner Striche bei Schraffuren zu vermeiden sollte nicht der komplette Bereich zur Positionierung genutzt werden. Praun et al. (2001) schlagen dagegen vor, mehrere hundert Striche testweise zu platzieren und deren Auswirkung zu berwerten um anschießend den Strich, der eine zuvor festgelegte Intensität am besten annähert und zugleich den Gesamteindruck der Gleichverteilung am ehesten aufrecht erhält, auszuwählen. In beiden Fällen wird die Länge des Striches zufällig zwischen 30% bis 100% der Ausgangslänge variiert. Der Vorteil bei (Praun et al., 2001) ist, dass die Intensitätsunterteilung der gewünschten Texturen sehr genau bestimmbar ist. Die „Best-Fitting-Stroke“ Lösung unterscheidet sich in den Resultaten überraschend gering mit denen meines naiven Vorschlags (siehe Vergleich in Abbildung 10c und Abbildung 10d).

Im Rahmen des Bundeswettbewerb für Informatik habe ich einen Algorithmus entworfen, der unter zuvor festgelegten Bedingungen, mit Hilfe diverser Annäherungen eine bestmögliche, zufällige Raumeinteilung vorschlägt (jedoch mit völlig anderem Anwendungsfall). Dessen vielversprechende Zweckentfremdung zur Texturerzeugung steht leider noch aus.

3.2 Intensitätsabstufungen

Zur Anzahl der Abstufungen gibt es stark voneinander abweichende Empfehlungen. Webb et al. (2002) schlagen 64 Texturen je Mipmap Ebene vor. Wie

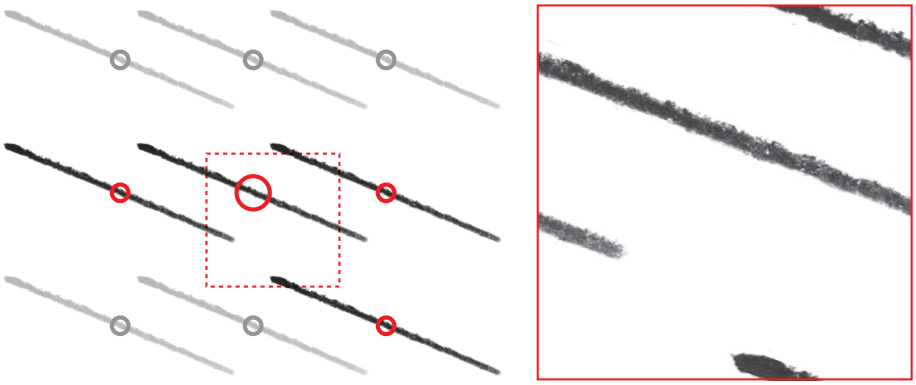


Abbildung 11: *Tileability - zusätzliche, auf einem Raster der Texturgröße, platzierte Striche sorgen für eine Widerholbarkeit der Textur.*

auch Lee et al. (2006) finde ich eine Intensitätsabstufung in nur 6 Schritten ausreichen. Bei Gradation können alle Intensitäten durch jeweils zwei Texturen linear interpoliert werden. Bei Schraffuren sind so viel Abstufungen ohnehin nicht möglich, da sich der Platz für zusätzliche Linien schnell verringert. Zudem spart die Nutzung weniger Texturen enorm viel Speicherplatz. Bei der Ablage der generierten Texturen in einer 3D-Textur sollte darauf geachtet werden, das bei Schraffuren nicht auf interpolierte Werte zurückgegriffen wird[4].

Bei der Untersuchung der von Sousa & Buchanan (1999) beschriebenen, mehrdimensionalen, sich gegenseitig beeinflussenden Überlagerungseigenschaften, fällt auf, dass diese, bei vielen sinnvollen Wertepaaren, ebenso durch lineare Funktionen approximiert werden können. Bei der Qualität aller bisher entstandenen Bleistift-Renderings lässt sich eine derart ausführliche und vergleichsweise umständliche Beschreibung kaum rechtfertigen. Generell ist zu beachten, dass bei mehrfachen Überlagern von Graphitstrichen, der Intensitätsanstieg, je nach Druck und Materialeigenschaften, meist abrupt abnimmt. Dem kann man beispielsweise durch entsprechende Blendingmodes gerecht werden.

Tileablity - also wiederholtes, übergangsloses Aneinanderlegen der selben Textur - der Strichtexturen kann durch einen simplen Trick gewährleistet werden. Bei der Platzierung eines Striches wird dieser einfach mehrfach, um die Texturbreite verschoben und somit ausserhalb der eigentlichen Textur, auf einem Raster, wie in Abbildung 11 beispielhaft dargestellt, positioniert. Je nach Verhältnis zwischen Textur- und Strichausdehnung müssen zusätzlich 8 (1-Ring), 24 (2-Ring) oder mehr Platzierungen vorgenommen werden.

Um unterschiedlichen Ansätzen zur Texturierung gerecht zu werden, habe ich (Praun et al., 2001) entsprechend auch Mipmaps generiert, die kohärente Übergänge erlauben. Eine Menge von zueinander koherenten, zur Schattierung von Bleistiftrenderings verwendeten Bilder und deren zugehörigen Mipmaps

[4] Dies lässt sich am einfachsten durch Nearest-Filtering in der Hardware erreichen.

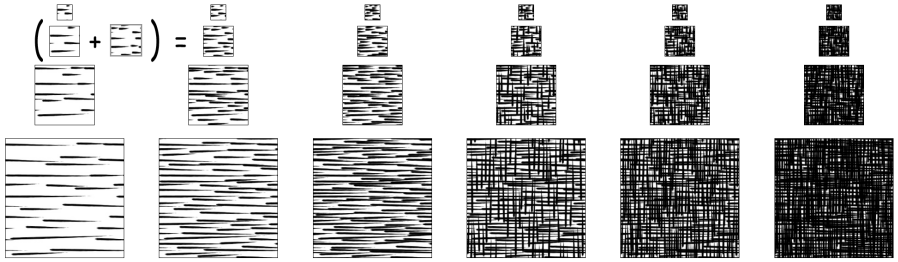


Abbildung 12: *Tonal Art Map nach (Praun et al., 2001). Striche eines Bildes befinden sich auch in allen intensiveren Bildern und allen niedriegeren Mipmapstufen. Abbildung ist (Praun et al., 2001) entnommen.*

wird als Tonal Art Map (TAM) bezeichnet. Für die Erzeugung der Bilder wird, wie in Abbildung 12 zu erkennen ist, bei dem kleinsten Mipmap der höchsten Helligkeit begonnen. Alle darin enthaltenen Striche tauchen an gleicher Stelle in allen höheren Intensitätsstufen sowie allen kleineren Mipmapstufen wieder auf. Wichtig ist dabei, dass die Dicke der Striche vom *level of detail* (LOD) unabhängig bleibt, sodass in der Texturierung die Strichstärken einheitlich sind. Vereinfachen lässt sich die Implementierung indem die Positionsbestimmung der Striche in einem von der Texturgröße unabhängigen Wertebereich (z.B. $[0; 1]$) abläuft und erst zum Zeitpunkt der individuellen Platzierung in entsprechende Pixelkoordinaten umgerechnet wird.

Neben Strichen können auch beliebige andere Formen verwenden werden (Webb et al., 2002). Zudem können TAMs, wie von Winkenbach & Salesin (1994) beschrieben, LOD-spezifische Materialinformationen enthalten oder spezielle Details der Szenengeometrie kapseln - ähnlich wie in (Döllner & Walther, 2003) könnten so beispielsweise Fassaden-Elemente wie Fenster besser in den gewünschten Stilen abgebildet werden.

4 Schraffur und Schattierung

In Bleistiftzeichnungen wird üblicherweise in eine formbildende bzw. formbetonende Richtung schattiert. Diese Richtung ist gezielt durch den Zeichner gewählt und reflektiert die Oberflächenform des gezeichneten Objekts. Es lässt sich beobachten, dass die Linien häufig abgeschwächt gekrümmt sind, die Oberflächenkrümmung somit nur angedeutet ist. Sehr kleine Details werden gar ausschließlich durch die Intensität erfasst. Im zeitlichen Ablauf des Zeichnens erfolgen die Schraffuren eher in Bereichen, die bereits durch grobe Konturen definiert wurden. Diese werden meist überzeichnet und anschließend durch Präzisere ersetzt oder durch stärkere Schraffuren oder indirekten Kanten - durch Ausnutzung des als *Closure* (McCloud, 1994) bekannten Effekt der Wahrnehmung - verstärkt.

Zur Schraffierung, also Texturierung der Szenengeometrie wird also eine flächendeckende Richtungsvorgabe benötigt. In früheren Echtzeit-Renderings

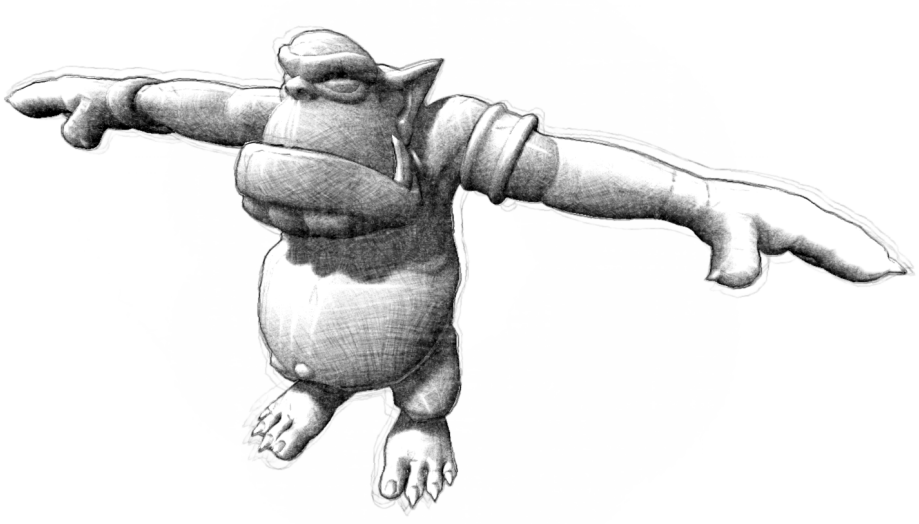


Abbildung 13: *Schraffierter Olaf mit Konturen, Schatten, Phong-Beleuchtung und Papierabrieb.*

wurden diese entweder vernachlässigt Lake et al. (2000) oder nachträglich, manuell beschrieben Salisbury et al. (1997). Neuere Verfahren versuchen diese automatisiert, entweder im Voraus aus den Geometriedefinitionen abzuleiten oder zur Laufzeit dem Bildraum zu entnehmen. In beiden Fällen wird die Richtung aus den lokalen Krümmungsverhalten der Geometrie geschlussfolgert. Krümmung ist ein Maß für die Richtungsänderung pro Längeneinheit oder anders: Die Krümmung an einem Punkt ist der Kehrwert des Radius einer Kugel die die Geometrie lokal am besten annähert. Es gibt eine Vielzahl von Möglichkeiten die Krümmung zu beschreiben. Evolute Flächen (*focal surfaces* in (Yu et al., 2007)) oder die 2. Ableitung seien als klassische Ansätze erwähnt. Krümmungsbeschreibungen werden weitestgehend mit *Principal Direction Fields* (*PDF*) bezeichnet, welche die Richtung der geringsten und die der stärksten Krümmung durch zweidimensionale Vektoren auf der Oberfläche angeben.

Um Vor- und Nachteile zwischen Vorberechnungen im Objektraum und echtzeitfähigen bildbasierten Annäherungen auszumachen, habe ich je einen aktuellen Vertreter ausgewählt und umgesetzt.

4.1 Kurvenapproximation im Objektraum

Für die Vorberechnung der *PDF* an der Geometrie habe ich die in Alliez et al. (2003) vorgestellte Methode implementiert. Nach Erzeugen einer entsprechenden Halbkantenstruktur - durch einen osg::Visitor, welcher alle Geometrien einer Node zusammenfasst und über ihre zugehörigen Dreiecke die Halbkantenstruktur aufbaut - wird für jeden Punkt die „geodesischen Scheibe“ mit konstanten Radius ermittelt und dessen durchschnittlicher Tensor errechnet. Mit diesem lassen

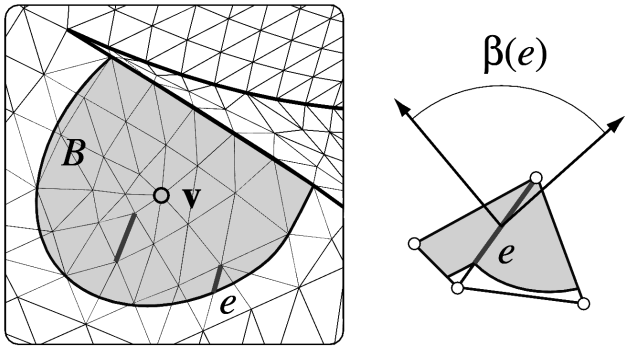


Abbildung 14: *Skizze zu Formel 6. (Alliez et al., 2003) entnommen.*

sich die Eigenvektoren und -werte bestimmen, welche anschließend mit ihrer Umgebung geglättet und als Multitextur-Koordinaten oder Vertex-Attribute an den Shader übermittelt werden. *Geodesic disc* eines Punktes, meint alle Vertices, welche innerhalb eines Radius liegen und der selben Oberfläche angehören. Die Punkte müssen also nahtlos über dazwischen befindliche Kanten erreichbar sein. Um charakteristische Kantenzüge der Geometrie zu erhalten können Punkte (ähnlich wie bei Smartedges) verworfen werden, sofern der Winkel zwischen den beiden Kanten einen Schwellwert überschreitet. Der Tensor $\Im(v)$ eines Vertex v berechnet sich dann über die folgende Formel:

$$\Im(v) = \frac{1}{|B|} \sum_{\text{edges } e} \beta(e) \cdot |e \cap B| \cdot \bar{e} \cdot \bar{e}^T \tag{6}$$

Dabei ist $|B|$ der Flächeninhalt um v, $\beta(e)$ der Winkel zwischen den Normalen der beiden, an dieser Kante anliegenden Dreiecke, $|e \cap B|$ die Länge der Kante innerhalb der Fläche B und $\bar{e}$ der normalisierte Kantenvektor zu e. Die Krümmungsvektoren der *PDF* ergeben sich schließlich aus den Eigenvektoren des geglätteten Tensors (Abbildung 14). Die Berechnung der Eigenwerte und -vektoren erfolgt unter Verwendung der Integrated Performance Primitive Bibliothek von Intel (IPP, 2008).

4.2 Kurvenapproximation im Bildraum

Für eine echtzeitfähige Approximation des *PDF* im Bildraum habe ich das kürzlich von Yu et al. (2007) bzw. Kim et al. (2008) vorgestellte Verfahren implementiert. Dabei werden, basierend auf den Werte im Normal- und Geometrie-Buffer, die Oberflächen Normalen als Normalenstrahlen parametrisiert. An jedem Punkt P der Geometrie wird der dazugehörige Strahl so transformiert, dass er der Normalen (in z-Richtung) der Tangentialebene ($z = 0$ Ebene) entspricht - also im Punkt $[0, 0, 0]$ nach $[0, 0, 1]$ ausgerichtet ist. Dies geschieht durch eine einfache Transformation $\boldsymbol{M_P}$ eines Liniensegments. Anschließend werden zwei angrenzende Normalen - dabei wird weder in (Yu et al., 2007) noch in (Kim et al., 2008)

beschrieben, welche der acht angrenzenden gewählt wurden - relativ zur aktuellen $z = 0$ Ebene ausgedrückt, also ebenfalls mit $\boldsymbol{M_P}$ transformiert. Stellt man sich weiterhin eine $z = 1$ Ebene vor, lassen sich die beiden Vektoren durch die Schnittpunkte $[u, v, 0]$ und $[s, t, 1]$ mit den zwei Ebenen durch $[\sigma, \tau, u, v]$ mit $\sigma = s - u$ und $\tau = t - v$ parametrisieren.

Yu et al. (2007) haben gezeigt, dass dicht bei einander liegende, derart parametrisierte Normalenstrahlen in zwei „Schlitzen“ fokussieren, deren Ausrichtung der minimalen und maximalen Krümmungsrichtung entsprechen. Die Berechnung dieser Ausrichtung ist recht umfangreich, weshalb für weitere Informationen hier auf (Kim et al., 2008) sowie den Shadercode in Anhang C verwiesen sei.

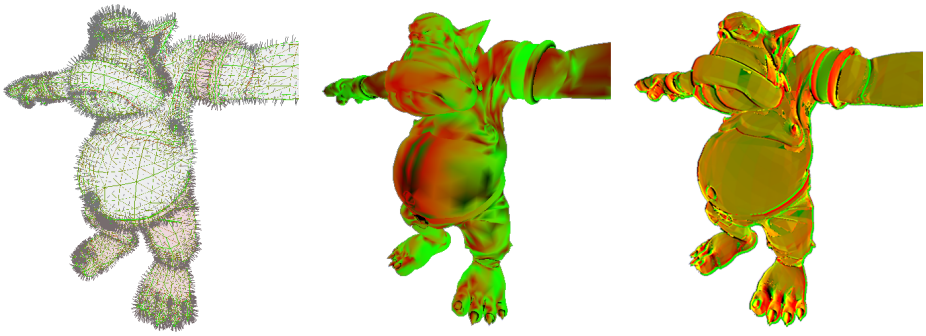
4.3 Vor- und Nachteile der Approximationen

Die Berechnung der Oberflächen-Krümmungsbeschreibung aus dem Objektraum heraus bietet den Vorteil, dass sie nur einmal pro Szenengeometrie berechnet werden muss, was je nach Geometrie sehr lange dauern kann. Im Vergleich zur Bildraum-Variante ist sie zudem stabiler aufgrund ihrer Berechnungsgenauigkeit und lässt sich besser glätten (Abbildung 15b). Zu den Nachteilen gehören zum einen, dass eine geeignete Struktur zur Iteration der Geometrieoberfläche vorhanden sein muss - in diesem Fall hatte ich die Halbkantenstruktur (Abbildung 15a) gewählt, welche jedoch ungenügend stabil für fehlerhafte Geometrien ist - und zum anderen bietet sie wie auch die bildbasierte Berechnung keine akzeptablen Vorschläge zur Bestimmung der Krümmung an Nabelpunkten und planaren Flächenstücken.

Die Approximation im Bildraum ist trotz ihrer mathematisch komplexeren Überlegungen relativ einfach zu implementieren. Es werden Normal-, Geometrie- sowie Tiefen-Buffer benötigt und die Berechnung ist in Echtzeit möglich. Zudem ist sie, da bildbasiert, unabhängig von der Szenenkomplexität und weniger problematisch an Nabelpunkten. Der einzige Nachteil ist die geringe Genauigkeit der Berechnung, da sie von sehr kleinen, teils nicht abbildbaren Unterschieden in der Fließkommadarstellung abhängt. Als Folge wird die Tesselierung der Geometrie, unabhängig von den Normalenglättung, schnell sichtbar (Abbildung 15c).

4.4 Texturierung

Von allen, mir bekannten Ansätzen zur Texturierung, habe ich den jüngst von Kim et al. (2008) vorgestellten verfolgt. Dabei wird im Fragmentshader der Winkel $\theta \in [0, \pi]$ zur Orientierung der Bleistifttextur dem *PDF* entnommen und zu θ_k quantisiert (mit k als Anzahl der Unterteilungen). Die Texturkoordinate wird nun um θ_k rotiert und der entsprechende Farbwert aus der Bleistifttextur gelesen. Um harte Kanten an den Winkelübergängen zu verringern, werden zwei weitere Farbwerte an den um θ_{k-1} und θ_{k+1} rotierten Texturkoordinaten entnommen und entsprechend mit $\frac{k \cdot |\theta - \theta_{k-1}|}{2\pi}$ bzw. $\frac{k \cdot |\theta - \theta_{k+1}|}{2\pi}$ ineinander verblendet. Wendet man auch hier zusätzlich den Trick des wiederholten, zufällig variierenden Überlagerns an, entstehen nicht sichtbare Übergänge. Um die korrekte Intensität des Pixels



(a) Halbkantenstruktur **(b)** Objektraum Ansatz **(c)** Bildraum Ansatz

Abbildung 15: *Ein direkter Vergleich darf Aufgrund der unterschiedlichen Farbbedeutungen in (b) und (c) nicht vorgenommen werden.*

abzubilden, wird bei den zuvor beschriebenen Texturzugriffen, einfach aus der als 3D-Textur vorliegenden TAM die entsprechende Ebene gewählt. Die Intensität kann dabei durch beliebige Faktoren wie Beleuchtung, vorherige Texturierung oder Schattenwurf beeinflusst werden. Abbildung 13 zeigt alle diese Einflüsse in einer gerenderten Bleistiftzeichnung.

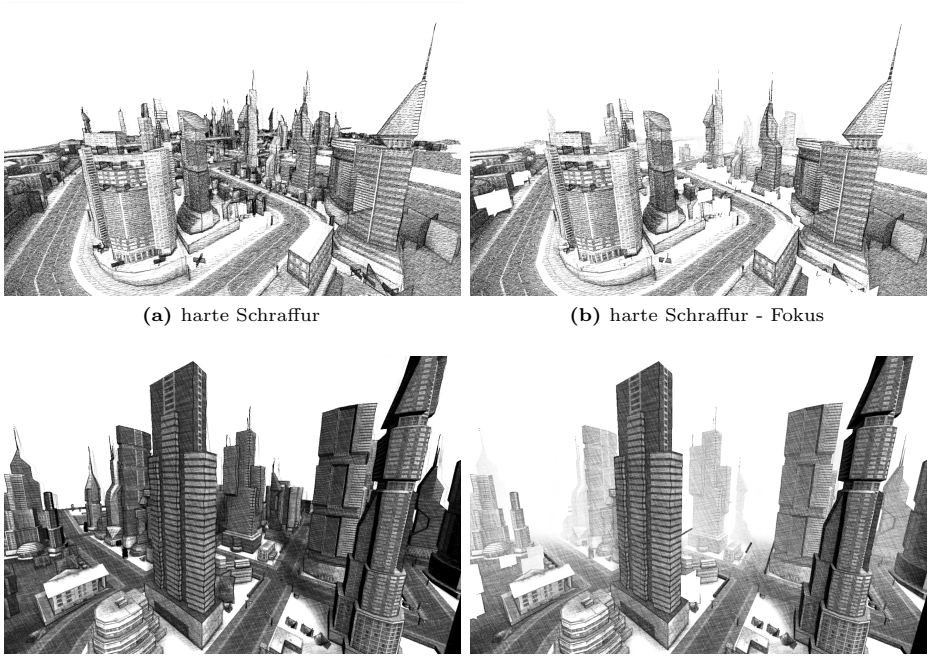
Durch den Benutzer muss lediglich der Offset zur *PDF* manuell angepasst werden, bis eine ihm angenehme Schraffur bzw. Schattierung vorliegt.

5 Anwendung auf Stadtmodellen

In weitläufigen 3D-Stadtmodellen führt eine Schattierung, wie u.a. in (Döllner & Walther, 2003) begründet, zur enormen Aufwertung der gerenderten Darstellung. Für die Umsetzung konnte dazu auf die *Shadow-Techniques* von OpenSceneGraph zurückgegriffen werden. Weiterhin bringt eine einfache Beleuchtung nach Phong eine Verbesserung mit sich - dabei sollte der spekulare Faktor relativ klein gewählt werden um einer, in der Realität normalerweise eher diffusen Ausleuchtung gerecht zu werden.

Für eine vernünftige Schraffur sind die angesprochenen Verfahren zur Krümmungsapproximation ungenügend, da planare Flächen keine Krümmung aufweisen. Alliez et al. (2003) schlägt vor, „einfach“ Werte aus der unmittelbaren Umgebung zu nehmen - was sich als sehr schwer erweist, wenn auch die Umgebung ungekrümmt ist. Kim et al. (2008) schlagen hingegen vor, die Richtung der Koordinaten-Achse mit dem kleinsten Winkel zur jeweiligen Fläche zu wählen und erzielt damit gute Ergebnisse.

Durch die begrenzte Auflösung der bildbasierten Ansätze kommt es - vor allem mit zunehmender Tiefe - schnell zu starken Aliasing-Artefakten. Diesem Problem kann man durch eine Fokussierung, z.B. mithilfe des Tiefen-Buffers, auf nahe liegende (bezüglich der Sichtebene) Stadtteile entgegenwirken (Abbildung 16).



(a) harte Schraffur **(b)** harte Schraffur - Fokus

(c) weiche Schraffur **(d)** weiche Schraffur - Fokus

Abbildung 16: *Megacity Bleistiftzeichnungen ohne (a), (c) und mit (b), (d) Tiefenfokussierung.*

Die Generierung von Bleistiftzeichnungen von Stadtmodellen ist folglich mit den in dieser Arbeit vorgestellten Techniken nicht oder nur eingeschränkt sinnvoll, sofern man nur einzelne Gebäude oder kleinere Gebäudegruppen visualisiert.

Für ein stilkonformes Rendering von Bäumen sei noch auf (Deussen & Strothotte, 2000) verwiesen.

6 Ausblick

Bleistiftskizzen könnte man mit zusätzlichen durch *Depthpeeling* gefundenen Kanten aufwerten. Anstelle der bildbasierten Kantendetektion bieten geometrische Kanten weitere, stilverändernde Anwendungen wie beispielsweise überzogene - also über einen angestrebten Punkt geradlinig, hinaus gezogene - Linien[5]. Die Linien könnten ausserdem geglättet gerendert und anschließend mit den hier vorgestellten Techniken weiterverarbeitet werden. Dies würde auch, die Auflösung betreffende Probleme bei der Darstellung von 3D-Stadtmodellen verringern, da geometrische Liniensegmente wesentlich feiner gezeichnet werden könnten. Für

[5] Für eine detaillierte Auswertung diesbezüglicher Möglichkeiten sei auf (Mölter, 2009) verwiesen.

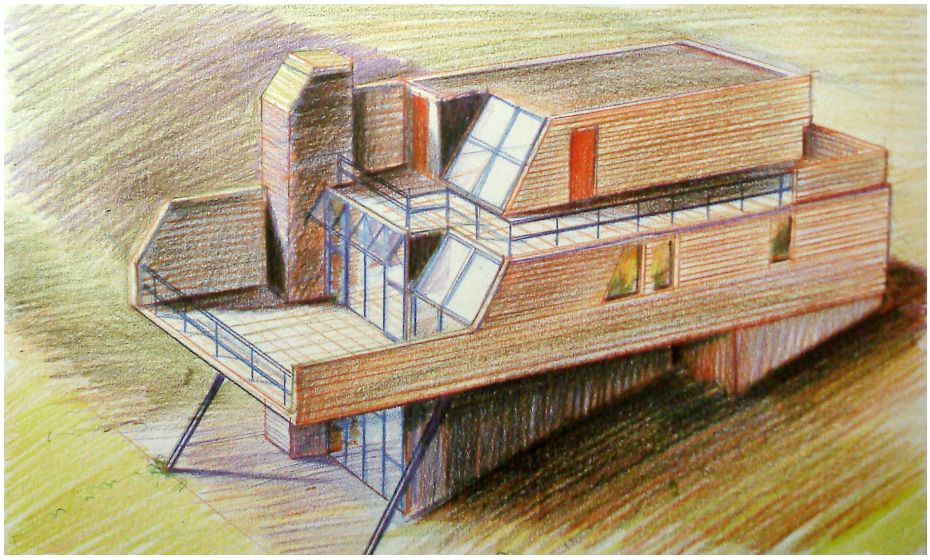


Abbildung 17: *Zeichnung aus „Architektur-Präsentation: Techniken der visuellen Darstellung“ von Eissen Koos, 1990.*

eine Bessere Abgrenzung der Szenenobjekte zueinander könnten sich stilisierte *Haloed Lines* als nützlich erweisen. Auch hat bisher niemand untersucht, wie sich über Konturen hinauslaufende Schraffuren auf den Gesamteindruck der synthetischen Bleistiftzeichnung auswirken.

Einen bisher völlig unangetasteten Bereich stellen Buntstiftzeichnungen dar. Hier stößt man auf viele Probleme bei der stilisierten Abbildungen der Farben durch Überlagerung farbiger Striche. Erstrebenswert wäre es allemal - für ein Beispiel siehe Abbildung 17.

Literaturverzeichnis

Alliez, P., Cohen-Steiner, D., Devillers, O., Lévy, B., & Desbrun, M. (2003). Anisotropic polygonal remeshing. *ACM Trans. Graph.*, 22(3), 485–493.

Buchanan, J. W. & Sousa, M. C. (2000). The edge buffer: a data structure for easy silhouette rendering. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering* (pp. 39–42). New York, NY, USA: ACM.

DeCarlo, D., Finkelstein, A., & Rusinkiewicz, S. (2004). Interactive rendering of suggestive contours with temporal coherence. In *Third International Symposium on Non-Photorealistic Animation and Rendering (NPAR)* (pp. 15–24).

Deussen, O. & Strothotte, T. (2000). Computer-generated pen-and-ink illustration of trees. In *SIGGRAPH '00: Proceedings of the 27th annual conference on*

Computer graphics and interactive techniques (pp. 13–18). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Döllner, J. & Walther, M. (2003). Real-time expressive rendering of city models. *Information Visualisation, International Conference on*, 0, 245.

Hertzmann, A. & Zorin, D. (2000). Illustrating smooth surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (pp. 517–526). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

IPP (2008). Intel® Integrated Performance Primitives (Intel® IPP) - Intel® Software Network. http://software.intel.com/en-us/intel-ipp/.

Isenberg, T., Freudenberg, B., Halper, N., Schlechtweg, S., & Strothotte, T. (2003). A developer's guide to silhouette algorithms for polygonal models. *IEEE Comput. Graph. Appl.*, 23(4), 28–37.

ISO (2008). *ISO 32000-1:2008*. International Organization for Standardization.

Kim, Y., Yu, J., Yu, X., & Lee, S. (2008). Line-art illustration of dynamic and specular surfaces. *ACM Transactions on Graphics (SIGGRAPH ASIA 2008)*, 27(5).

Lake, A., Marshall, C., Harris, M., & Blackstein, M. (2000). Stylized rendering techniques for scalable real-time 3d animation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering* (pp. 13–20). New York, NY, USA: ACM.

Lee, H., Kwon, S., & Lee, S. (2006). Real-time pencil rendering. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering* (pp. 37–45). New York, NY, USA: ACM.

Loviscach, J. (2002). Rendering Artistic Line Drawings Using Off-the-Shelf 3-D Software. In I. N. Alvaro & P. Slusallek (Eds.), *Eurographics 2002 Short Paper Proceedings* (pp. 125–130). Oxford, UK: Eurographics Association Blackwell Publishers.

McCloud, S. (1994). *Understanding Comics: The Invisible Art*. New York, NY, USA: Harpercollins.

Mitchell, J. L., Brennan, C., & Card, D. (2002). Real-time image-space outlining for non-photorealistic rendering. In *Siggraph 02 Conf. Abstracts and Applications, ACM Press* (pp. 239).

Mölter, S. (2009). GPU-basierte Qualitätsverbesserung für Konturen in 3D-Stadtmodellen.

Nienhaus, M. & Döllner, J. (2003). Edge-enhancement – an algorithm for real-time non-photorealistic rendering.

Nienhaus, M. & Döllner, J. (2004). Blueprints - illustrating architecture and technical parts using hardware-accelerated non-photorealistic rendering. In W. Heidrich & R. Balakrishnan (Eds.), *Proceedings of Graphics Interface 2004* (pp. 49–56). Canada: A K Peters.

Praun, E., Hoppe, H., Webb, M., & Finkelstein, A. (2001). Real-time hatching. In *Proceedings of ACM SIGGRAPH 2001* (pp. 579–584).

Saito, T. & Takahashi, T. (1990). Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4), 197–206.

Salisbury, M. P., Wong, M. T., Hughes, J. F., & Salesin, D. H. (1997). Orientable textures for image-based pen-and-ink illustration. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series (pp. 401–406).

Sousa, M. C. & Buchanan, J. W. (1999). Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models. 18(3), 195–207.

Webb, M., Praun, E., Finkelstein, A., & Hoppe, H. (2002). Fine tone control in hardware hatching. In *NPAR 2002: Second International Symposium on Non Photorealistic Rendering* (pp. 53–58).

Winkenbach, G. & Salesin, D. H. (1994). Computer-generated pen-and-ink illustration. *Computer Graphics*, 28(Annual Conference Series), 91–100.

Yu, J., Yin, X., Gu, X., McMillan, L., & Gortler, S. (2007). Focal surfaces of discrete geometry. In *SGP '07: Proceedings of the fifth Eurographics symposium on Geometry processing* (pp. 23–32). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

A Weitere Beispiele

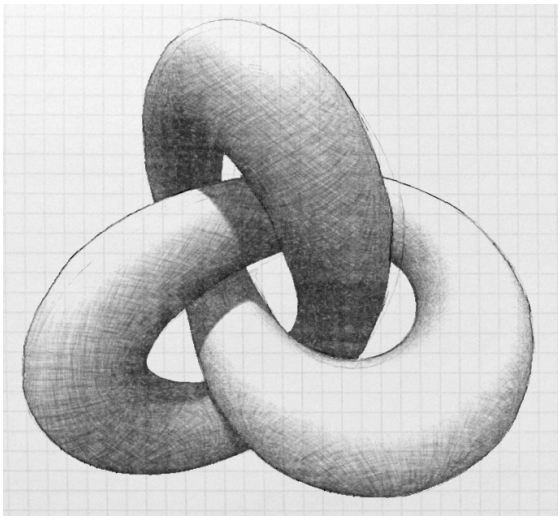
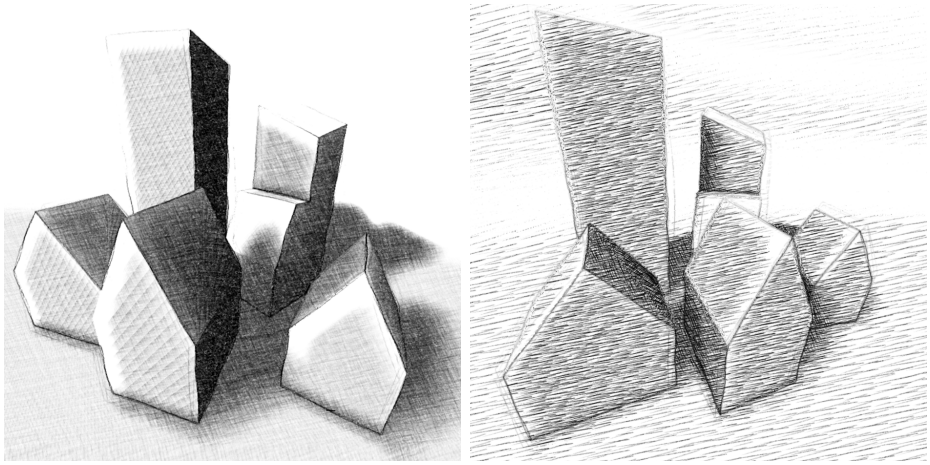


Abbildung 18: *Knoten von M. C. Escher als Bleistiftzeichnung - (3, 2)*-torus knot.



(a) weiche Tonabstufung **(b)** harte Schraffur

Abbildung 19: *Vergleich: weiche Tonabstufung mit harter Schraffur.*

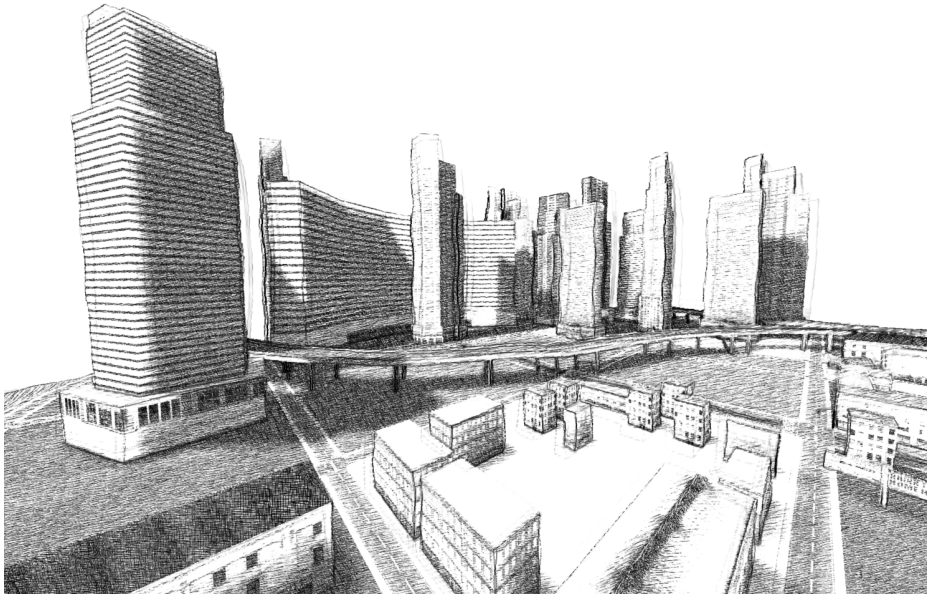


Abbildung 20: *Bleistiftschraffur ohne Papierstruktur.*

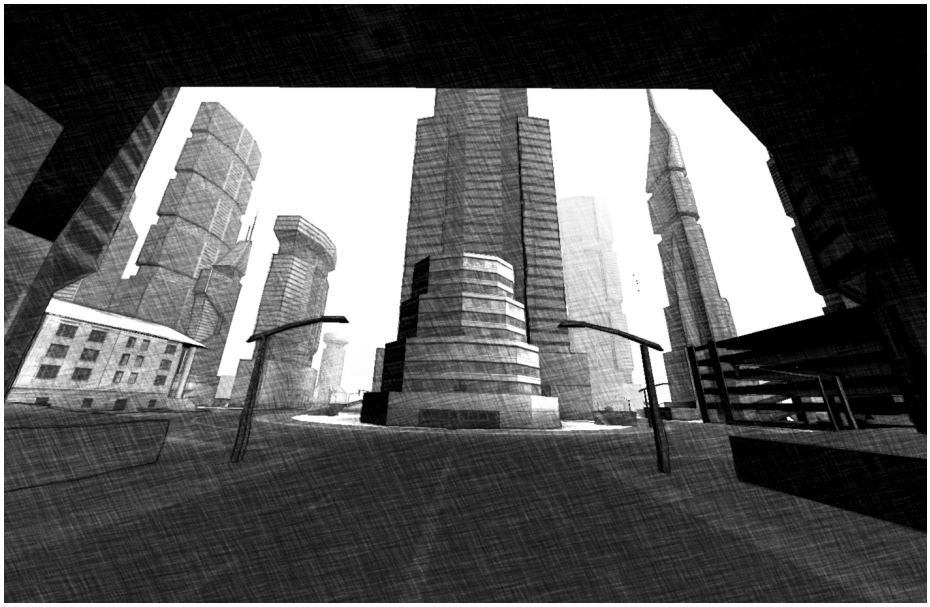


Abbildung 21: *Bleistiftzeichnung ohne Papierstruktur, mit weicher Tonabstufung und Tiefenfokussierung.*

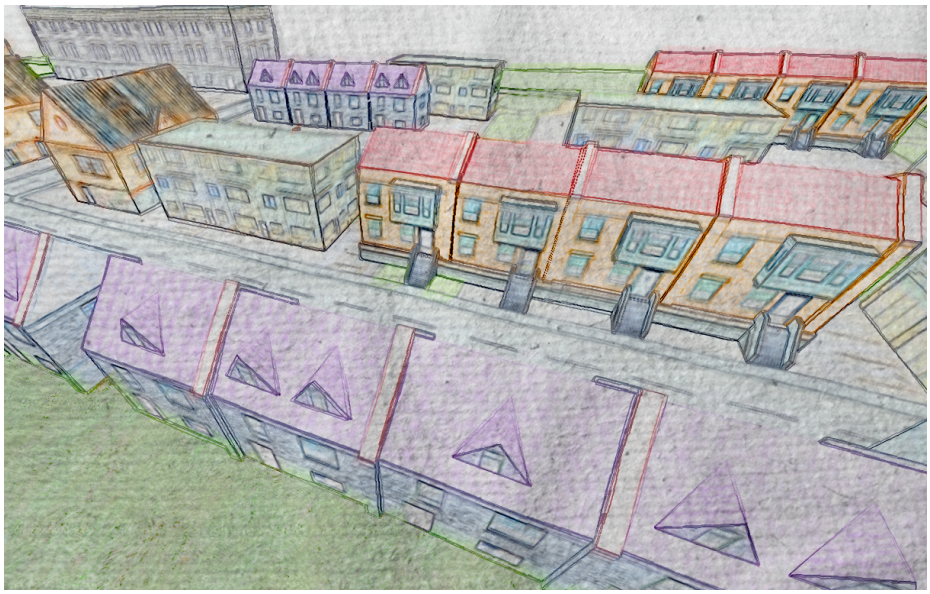


Abbildung 22: *Erster Versuch für Buntstiftkonturen.*

B Pencil-Fragmentshader

```
// author: Daniel Müller - dm@g4t3.de

#version 120
#extension GL_EXT_gpu_shader4 : enable


uniform sampler2D src; // src should contain Normals in rgb and Depth in a
uniform sampler2D perlin; // should contain cloud noise like texture

varying out vec4 dst;

uniform sampler2D rand;
int rand_offset = 0;

uniform vec2 amplitudes = vec2( 2f, 16f);
uniform ivec2 coverings = ivec2( 4, 3);

uniform vec4 wavelengths = vec4( 8f, 64f, 4f, 8f);

uniform float sketchIntensity = .6f;


const float f = 1f;

const vec2 uv_a = vec2(-f, -f);
const vec2 uv_b = vec2( 0, -f);
const vec2 uv_c = vec2(+f, -f);
const vec2 uv_d = vec2(-f,  0);
const vec2 uv_e = vec2(+f,  0);
const vec2 uv_f = vec2(-f, +f);
const vec2 uv_g = vec2( 0, +f);
const vec2 uv_h = vec2(+f, +f);

const float PI2 = 6.28f;

#define FETCH(__offset) \
texture2D(src, (uv + __offset) * src_size)

#define RAND(__min, __max) \
    (texture2D(rand, vec2(++rand_offset * rand_size, .5f))[rand_offset % 4] \
    * (__max - __min) + __min)

#define PERLIN \
    (1f - (texture2D(perlin, uv * perlin_size).r \
    * sketchIntensity + (1f - sketchIntensity)))


float edge_i(in vec2 uv, in vec2 src_size)
{
    vec4 a = FETCH(uv_a);
    vec4 b = FETCH(uv_b);
    vec4 c = FETCH(uv_c);
    vec4 d = FETCH(uv_d);
    vec4 x = FETCH(vec2(0));
    vec4 e = FETCH(uv_e);
    vec4 f = FETCH(uv_f);
    vec4 g = FETCH(uv_g);
    vec4 h = FETCH(uv_h);

    //float Iz1 = (pow(1f - .5f * abs(a.a - h.a), 8f))
    //          * (pow(1f - .5f * abs(c.a - f.a), 8f));

    float Iz2 =  1f -
                (.5f * (abs(a.a - x.a) + abs(c.a - x.a)
                     +  abs(f.a - x.a) + abs(h.a - x.a))
```

```
            +  1f * (abs(b.a - x.a) + abs(d.a - x.a)
                  +  abs(e.a - x.a) + abs(g.a - x.a)));

    // normalize should be used to retrieve
    // valid normals from background colors
    float In = .5f * (dot(normalize(a.rgb), normalize(h.rgb))
                  + dot(normalize(c.rgb), normalize(f.rgb)));

    return In * Iz2;
}


void main (void)
{
    vec2 src_size = 1f / vec2(textureSize2D(src, 0));
    float rand_size = 1f / textureSize2D(rand, 0).x;
    vec2 perlin_size = 1f / vec2(textureSize2D(perlin, 0));

    vec2 uv = vec2(gl_FragCoord.xy);
    vec2 uv_s = uv * src_size;

    vec2 intensity = vec2(0f);
    vec2 uv_modified;

    // lighter sketchy lines
    for(int i = 0; i < coverings[1]; ++i)
    {
        uv_modified = uv;
        uv_modified.x += amplitudes[1]
            * sin(RAND(wavelengths[2], wavelengths[3])
            * uv_s.y + RAND(0f, PI2)) - RAND(-1f,+1f);
        uv_modified.y += amplitudes[1]
            * cos(RAND(wavelengths[2], wavelengths[3])
            * uv_s.x + RAND(0f, PI2)) - RAND(-1f,+1f);
        intensity[1] += edge_i(uv_modified, src_size);
    }

    // darker final lines
    for(int i = 0; i < coverings[0]; ++i)
    {
        uv_modified = uv;
        uv_modified.x += amplitudes[0]
            * sin(RAND(wavelengths[0], wavelengths[1])
            * uv_s.y + RAND(0f, PI2)) - RAND(-1f,+1f);
        uv_modified.y += amplitudes[0]
            * cos(RAND(wavelengths[0], wavelengths[1])
            * uv_s.x + RAND(0f, PI2)) - RAND(-1f,+1f);
        intensity[0] += edge_i(uv_modified, src_size);
    }

    float final_i = min(

        1f - (1f - intensity[1] / coverings[1]) * PERLIN,
        intensity[0] / coverings[0]);

    dst = vec4(vec3(final_i), 0f);
}
```

C Curvature-Fragmentshader

```
// author: Daniel Müller - dm@g4t3.de

#version 120
#extension GL_EXT_gpu_shader4 : enable

// geometry, normals and mask are supposed to be equely sized
uniform sampler2D geometry;
```

```
uniform sampler2D normals;

//speeds up the calculation by discarding on black pixel
uniform sampler2D mask;

uniform vec2 offset1 = vec2(1f, 0f);
uniform vec2 offset2 = vec2(0f, 1f);

varying out vec4 dst;

struct ray
{
    vec3 dir;
    vec3 pos;
};

ray fetchRay(in vec2 src_size, in vec2 offset = vec2(0f, 0f))
{
    ray r;
    r.dir = normalize(
        texture2D(normals, (gl_FragCoord.xy + offset) * src_size).rgb);
    r.pos = texture2D(geometry, (gl_FragCoord.xy + offset) * src_size).rgb;
    return r;
}

void main (void)
{
    vec2 src_size = 1f / vec2(textureSize2D(geometry, 0));

    if(texture2D(mask, gl_FragCoord.xy * src_size).r == 0f)
        vec4(0f);

    ray r0 = fetchRay(src_size);
    ray r1 = fetchRay(src_size, offset1);
    ray r2 = fetchRay(src_size, offset2);

    // transformation to origin and rotation around y-axis onto x = 0 plane
    vec2 r0_xz = r0.dir.xz / length(r0.dir.xz);

    if(length(r0.dir.xz) == 0) // not needed in eyespace
        r0_xz = vec2(0f);

    mat4 t = mat4(1f, 0f, 0f, 0f, 0f, 1f, 0f, 0f, 0f, 0f, 1f, 0f,
        -r0.pos.x, -r0.pos.y, -r0.pos.z, 1f);

    mat4 ry = mat4(
        +r0_xz[1], 0f, r0_xz[0], 0f,
              0f, 1f,        0f, 0f,
        -r0_xz[0], 0f, r0_xz[1], 0f,
              0f, 0f,    0f, 1f) * t;

    // rotation around x-axis onto y = 0 plane
    vec4 r0_yz = ry * vec4(r0.pos + r0.dir, 1f); // / length(r0.dir);

    mat4 m = mat4(
         1f,     0f,        0, 0f,
         0f, r0_yz[2], r0_yz[1], 0f,
         0f,-r0_yz[1], r0_yz[2], 0f,
         0f,     0f,       0f, 1f) * ry;

    ray r00, r01, r02;

    // parameterization of the two adjacent rays
    r01.dir = (m * vec4(r1.pos + r1.dir, 1f)).xyz;
    r01.pos = (m * vec4(r1.pos, 1f)).xyz;
    r02.dir = (m * vec4(r2.pos + r2.dir, 1f)).xyz;
    r02.pos = (m * vec4(r2.pos, 1f)).xyz;
```

```
    // plane p0(uv) in z = 0 is dot((0,0,1),x) = 0
    // plane p1(st) in z = 1 is dot((0,0,1),x) - 1 = 0

    // intersect r1 and r2 with p0 and p1 to obtain s,t and u, v values

    // parameterizaion is [s - u, t - v, u, v]
    // parameterized ray0 is [0,0,0,0] in its local frame

    vec4 _r1;   // parameterized ray1
    _r1.zw = ((-r01.pos.z) / r01.dir.z * r01.dir + r01.pos).xy;
    _r1.xy = ((-r01.pos.z + 1) / r01.dir.z * r01.dir + r01.pos).xy - _r1.zw;
    vec4 _r2;   // parameterized ray2
    _r2.zw = ((r02.pos.z) / r02.dir.z * r02.dir + r02.pos).xy;
    _r2.xy = ((r02.pos.z + 1) / r02.dir.z * r02.dir + r02.pos).xy - _r2.zw;

    // assume that the area formed by three intersection points,
    // in the z = lambda plane, becomes zero, the corresponding
    // rays will focus at a slit
    float A = _r1.x * _r2.y - _r2.x * _r1.y;
    float B = _r1.x * _r2.w - _r2.x * _r1.w - _r1.y * _r2.z + _r2.y * _r1.z;
    float C = _r1.z * _r2.w - _r2.z * _r1.w;

    float lambda = sqrt(abs(B * B - 4 * A * C));

    // TODO: can still become undefined - e.g. on planar surfaces
    float lambda0 = -2 * C / (B - lambda);
    float lambda1 = -2 * C / (B + lambda);

    // TODO: approximate curvature for planar surfaces

    vec2 pd0 = vec2(_r1.z * lambda0 + _r1.x , _r1.w * lambda0 + _r1.y);
    // that is an assumption of mine, because
    // the supposed equ. in Kim08 seemed wrong
    vec2 pd1 = vec2(_r2.z * -lambda1 + _r2.x , _r2.w * -lambda1 + _r2.y);

    dst = vec4(pd0, pd1);
}
```