Bachelorarbeit im Rahmen des Studiengangs
Scientific Programming

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 – Medizintechnik und Technomathematik

Entwicklung und Optimierung eines Software-Renderers für die GR3-Grafikbibliothek

Jülich, den 26. August 2019

Jonas Ritz

Eigenhändigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

(Ort, Datum)	(Unterschrift)

Diese Arbeit wurde betreut von:

1. Prüfer:	Prof. Dr.-Ing. U. Stegelmann	(FH Aachen, Campus Jülich)
2. Prüfer:	Florian Rhiem	(FZ Jülich, PGI/JCNS-TA)

Sie wurde angefertigt in der Forschungszentrum Jülich GmbH im Peter Grünberg Institut / Jülich Centre for Neutron Science.

Wissenschaftler am Peter Grünberg Institut/Jülich Centre for Neutron Science untersuchen in Experimenten und Simulationen Form und Dynamik von Materialien wie Polymeren, Zusammenlagerungen großer Moleküle und biologischen Zellen sowie die elektronischen Eigenschaften von Festkörpern. Für die Präsentation der in diesem Zusammenhang anfallenden Forschungsergebnisse in Vorträgen und Veröffentlichungen müssen häufig dreidimensionale Strukturen in Echtzeit dargestellt werden.

Bei der Darstellung besagter Strukturen wird bislang in GR3 auf OpenGL, die Spezifikation einer Programmierschnittstelle zur hardwarebeschleunigten Erzeugung von 3D-Grafiken, zurückgegriffen. Die zur Nutzung von OpenGL notwendigen Hardwarekomponenten und Bibliotheken sind allerdings in Umgebungen wie Docker-Containern oder Servern ohne grafische Ausgabe oft nur eingeschränkt oder gar nicht verfügbar. Um dennoch eine performante dreidimensionale Visualisierung in besagten Umgebungen zu ermöglichen, soll im Rahmen dieser Bachelorarbeit der Software-Renderer aus [Rit19], der bislang nur bivariate Funktionen als Oberflächen visualisieren kann, in seiner Funktionalität erweitert und optimiert werden, um verschiedene in Dreiecke zerlegte dreidimensionale Strukturen in angemessener Zeit darstellen zu können. Die dabei erzeugten Grafiken sollen zu der Ausgabe der bisher verwendeten, hardwarebeschleunigten Variante des GR3 nahezu identisch sein. Von besonderer Relevanz ist hierbei die Minimierung der Laufzeit, welche sich durch verschiedene Techniken an die durch die hardwarebeschleunigte Variante erzielte annähern soll. So wird in Zukunft auf Systemen ohne ausreichende Grafikhardware automatisch auf den Software-Renderer zurückgegriffen, ohne dass dies zu erkennbaren optischen Unterschieden oder groben Differenzen in der Ausführungszeit führt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Häufig dienen Grafiken in wissenschaftlichen Veröffentlichungen der besseren Verständlichkeit und Überschaubarkeit der Erkenntnisse für den Leser. Zusätzlich helfen sie den Wissenschaftlern selbst einen Überblick über unter Umständen große aus Experimenten oder Simulationen resultierenden Datenmengen zu gewinnen. Die Erzeugung einer Grafik sollte möglichst performant sein, um hochauflösende Darstellungen in angemessener Zeit mit verschiedenen Parametern generieren zu können. Zudem sollte die Funktionalität völlig unabhängig vom verwendeten System ermöglicht werden, um Portabilität und Benutzerfreundlichkeit zu steigern. GR3 erfüllt die besagten Anforderungen bezüglich dreidimensionaler Grafiken. Zum Rendern wird intern OpenGL verwendet, das auf dem verwendeten System in Form von Systembibliotheken oder Grafikkartentreibern implementiert sein muss. Auf bestimmten Systemen und in Docker-Containern ist jedoch häufig keine OpenGL Implementierung verfügbar. Sie kann zwar durch Alternativen wie Mesa, worin der Software-Renderer llvmpipe [VMw] enthalten ist, ersetzt werden, jedoch erfordert dies große Abhängigkeiten die in GR3 nicht zuletzt wegen des Speicherbedarfs und der angestrebten Plattformunabhängigkeit vermieden werden sollen.

Daher soll im Rahmen dieser Bachelorarbeit ein Software-Renderer entwickelt werden, der die gesamte Verwendung von OpenGL in GR3 ersetzen kann. Wie eingangs erwähnt ist die benötigte Laufzeit relevant, sodass die Funktionalität, die der Software-Renderer umfasst, spezifisch auf GR3 abgestimmt und möglichst effizient implementiert werden soll. Im Wesentlichen besteht der Rendering-Prozess aus der Transformation Eckpunkten, die zu triangulierten Oberflächen gehören, und der möglichst effizienten Rasterisierung der einzelnen daraus resultierenden Dreiecke. Die bislang von OpenGL beziehungsweise dessen Implementierung erledigte Rasterisierung zerfällt zum einen in das Durchlaufen der zur Darstellung des Dreiecks nötigen Pixel und zum anderen in die Farbberechnung jedes Pixels unter Berücksichtigung der Beleuchtung und Farben der Eckpunkte. Durch Abstimmung der Implementierung auf die Struktur des GR3 und zusätzlicher Verwendung verschiedener Techniken zur Parallelisierung soll die Laufzeit zur Erstellung und korrekten Füllung eines Farbpuffers, der auch aus dem Renderingvorgang mit OpenGL hervorgeht, möglichst minimiert werden.

2. Entwicklung und Erweiterung

Der bislang entwickelte Software-Renderer aus [Rit19] unterstützt lediglich die Visualisierung bivariater Funktionen als Oberflächen. Die vom Rasterisierer verursachten Fehler werden im Folgenden analysiert und deren Ursache festgestellt, anschließend sollen sie durch die Implementierung einer alternativen Rasterisierungsmethode entfernt werden. Weiterhin soll die Funktionalität auf alle durch das GR3 darstellbaren Szenen erweitert werden.

Im Zuge dessen wird das Prinzip des Zeichnens einzelner Dreiecksgitter aufgegriffen und die die Berechnung einer Beleuchtung mit punktueller Lichtquelle implementiert. Vom Software-Renderer soll ebenfalls Kantenglättung unterstützt werden, die analog zur hardwarebeschleunigten Variante funktioniert.

2.1. Ansatz des bisherigen Rasterisierers

Um darzustellen, wieso ein alternatives Verfahren zur Rasterisierung implementiert wird, wird hier zuerst kurz die Funktionsweise des bisherigen Verfahrens eingegangen. Dadurch werden die Nachteile und Komplikationen aufgedeckt, welche Ursache für die Neuimplementierung sind.

Die ältere Implementierung arbeitet mit dem Bresenham-Algorithmus, der zum Zeichnen von Geraden auf Rasteranzeigen im Jahre 1962 entwickelt wurde. Dieser ist sehr effizient und minimiert Rundungsfehler, da intern ausschließlich diskrete Koordinaten verwendet werden [Bre62]. In der vorherigen Implementierung baut die Rasterisierung eines Dreiecks auf der Rasterisierung der Kanten mit dem Bresenham-Algorithmus auf. Zur Füllung des Dreiecks werden dazu alternierend die gegenüberliegenden Kanten als Linien gezeichnet und zwischen den dadurch entstehenden Begrenzungen alle Pixel eingefärbt. Somit füllt sich das Dreieck zeilenweise von unten nach oben. Die durch den Linienzug entstandenen Grenzen sind in der folgenden Abbildung 2.1 blau markiert, die dazwischen eingefärbten Pixel dunkelgrau.

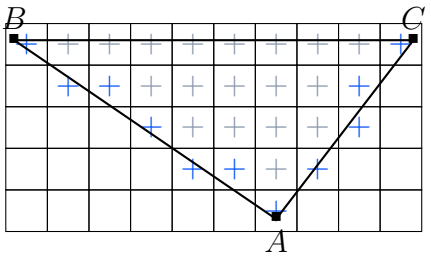


Abbildung 2.1.: Rasterisierungsvorgang mit dem Bresenham-Algorithmus

2.1.1. Kritik an der Rasterisierung des Bresenham-Algorithmus

Für den Bresenham-Algorithmus im Zusammenhang mit der Darstellung von Dreiecken auf Rasteranzeigen werden in dieser Implementierung die Eckpunkte zunächst ihrer y-Koordinate nach aufsteigend sortiert, damit das Dreieck von unten nach oben rasterisiert werden kann. Dies führt zu einem Anstieg der benötigten Rechenzeit.

Des Weiteren ist zu berücksichtigen, dass die Veröffentlichung des Bresenham-Algorithmus aus dem Jahre 1962 stammt. Moderne Rechnerarchitekturen unterstützen Gleitkommaoperationen mit einer höheren Genauigkeit, als für die Berechnung benötigt wird, was damals nicht der Fall war. Außerdem sind die Rechenoperationen für Gleitkommazahlen üblicherweise genauso schnell, wie die für Ganzzahlen. Die Vorteile, die den Bresenham-Algorithmus als Linienzugalgorithmus auszeichnen, entfallen also wegen des technischen Fortschrittes.

Das eigentliche Kernproblem und damit auch Hauptgrund für die Implementierung eines neuen Algorithmus sind jedoch falsche Ergebnisse, die in dem Linienzugalgorithmus begründet sind. Das Kriterium bei der Rasterisierung eines Dreiecks, ob ein Pixel dazugehört beziehungsweise eingefärbt werden soll, besteht darin, ob dessen Mittelpunkt innerhalb des Dreiecks liegt. Beim Linienzug durch den Bresenham-Algorithmus ist im Gegensatz dazu das Kriterium, wie groß der Abstand der aus dem Pixel austretenden Linie von den beiden nächsten potentiell einzufärbenden Pixeln ist. In der folgenden Abbildung 2.2 sind die Pixel rot eingefärbt, die der Bresenham-Algorithmus zwar als Einzufärbende markiert, aber letztendlich nicht eingefärbt werden, da ihr Pixelmittelpunkt sich nicht innerhalb des Dreiecks befindet.

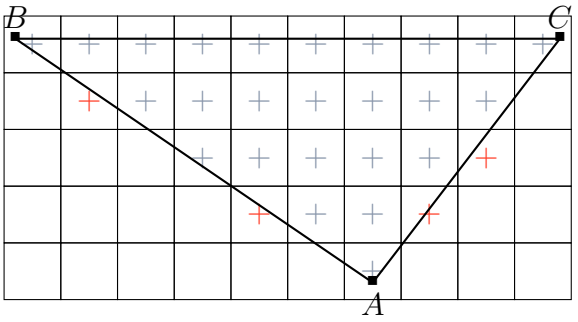


Abbildung 2.2.: Fehler des Rasterisierungsvorgangs mit dem Bresenham-Algorithmus

Die Abbildung untermauert, dass die Wahl dieses Algorithmus nicht optimal ist. Die rot markierten Pixel werden in der Implementierung nicht eingefärbt, da dort mindestens eine der baryzentrischen Koordinaten (vgl. Abschnitt 3.2) negativ ist [Rit19]. Werden diese trotzdem eingefärbt, resultiert beispielsweise in nicht passend dargestellten Rundungen, wie rechts in Abbildung 2.3 zu sehen. Links ist die korrekt durch OpenGL rasterisierte Variante zu sehen.

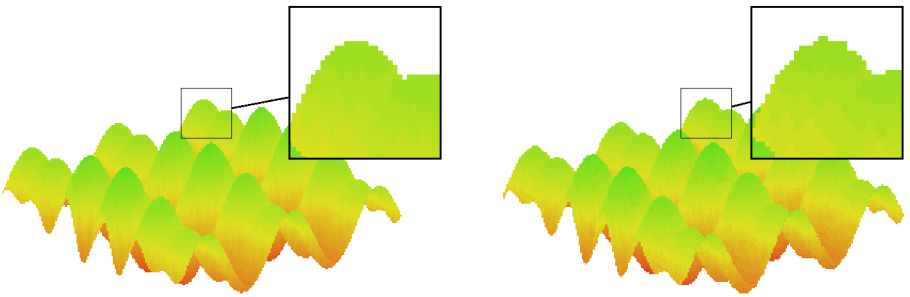


Abbildung 2.3.: Darstellung einer bivariaten Funktion als Oberfläche

Gleichzeitig können auch Lücken in der Zeichnung auftreten, falls ein Dreieck aus nur einem Pixel besteht. Dies könnte zum Beispiel wie in Abbildung 2.4 aussehen, wobei M der Mittelpunkt des Pixels und A, B und C die Eckpunkte des Dreiecks sind:

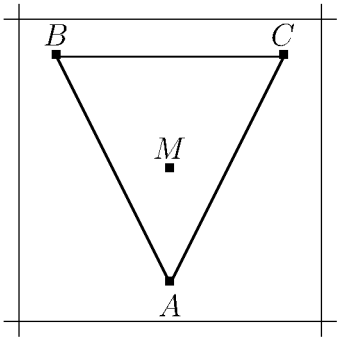


Abbildung 2.4.: Lücke im Rasterisierungsvorgang mit dem Bresenham-Algorithmus

Da der Mittelpunkt des Pixels innerhalb des Dreiecks liegt, muss dieser Pixel entsprechend der Farbe des Dreiecks eingefärbt werden. Der Bresenham-Algorithmus vollzieht jedoch keinen Schritt des Linienzugs, sodass es schlussendlich nicht zu einer Einfärbung kommt.

Außerdem wurden in der bisherigen Implementierung des Rasterisierers die Koordinaten jedes Eckpunktes als Ganzzahl hinterlegt. Jeder Eckpunkt wurde dabei zu Beginn in diese Darstellung ohne Nachkommastellen überführt. Diese Speicherung ist überflüssig, weil alle Berechnungen in Fließkomma-Arithmetik stattfinden und zur diskreten Indizierung der Farbpuffer in Ganzzahlen umgewandelt werden. Zudem wird dadurch viel Speicherplatz belegt.

2.2. Rasterisierung mit umgebendem Viereck

Die im vorherigen Abschnitt beschriebene Rasterisierungsmethode soll durch eine effiziente und korrekte Alternative ersetzt werden. Eine mögliche Methode, um eine

korrekte Rasterisierung zu gewährleisten, besteht darin, bei jedem Dreieck für alle vorhanden Pixel zu prüfen, ob ihr Mittelpunkt innerhalb des Dreiecks fällt. Diese Prüfung kann mittels baryzentrischer Koordinaten für jeden Pixel erfolgen, welche mindestens einen negativen Wert aufweisen, falls jener Pixel nicht innerhalb des Dreiecks liegt.

Um den Rechenaufwand zu reduzieren, wird die zu prüfende Fläche zunächst auf das Rechteck begrenzt, welches das Dreieck exakt umgibt und alle Pixel enthält, die zur Einfärbung in Frage kommen [Hen+11]. Bei allen anderen Pixel ist es unmöglich, dass deren Mittelpunkte innerhalb des Dreiecks liegen. Aus diesem Grund ist deren Prüfung und die damit zusammenhängenden, aufwendigen Rechenoperationen nicht von Nöten. Es gilt: Je kleiner das Dreieck, desto größer die erzielte Ersparnis. Beispielsweise erzielt ein Dreieck, dass die Hälfte des Bildschirmes einnimmt keine Ersparnis. Im Gegensatz dazu bezweckt diese Methode bei einem Dreieck, welches lediglich einen Pixel bedeckt, eine Ersparnis von allen Pixeln, die bei der Prüfung ohnehin negative baryzentrische Koordinaten liefern würde und den Pixel nicht einfärben würde.

Die obere Kante des besagten umgebenden Rechtecks (sog. `Bounding Box`) befindet sich auf der Höhe des höchsten Eckpunktes des Dreiecks und die untere Kante auf der des tiefsten Eckpunktes. Analog dazu werden die seitlichen Begrenzungen durch die minimalen und maximalen x-Werte der Eckpunkte ermittelt. So ein Rechteck könnte zum Beispiel wie Abbildung 2.5 zeigt aussehen:

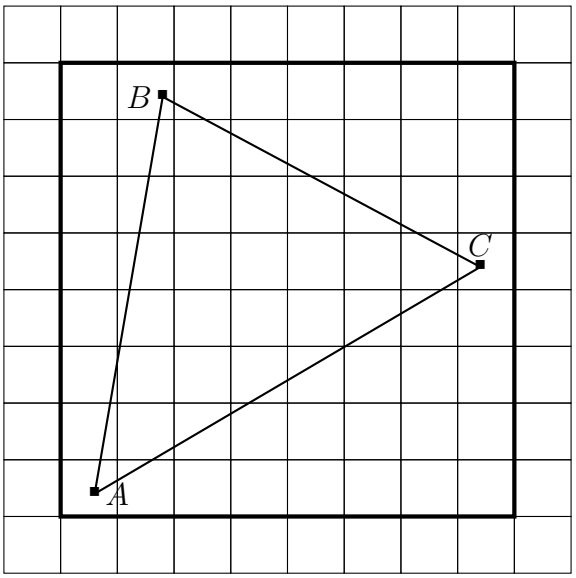


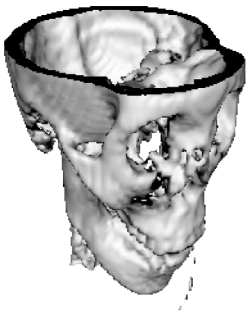
Abbildung 2.5.: Das kleinste ein Dreieck umgebende Rechteck, in dem alle potentiell einzufärbenden Pixel liegen

Durch diese erste, intuitive Optimierung wird die Berechnung der baryzentrischen Koordinaten aller restlichen Pixel bezüglich dieses Dreiecks gespart. In dem Rechteck befinden sich meist doppelt so viele Pixel wie in dem Dreieck, also ist die Anzahl der

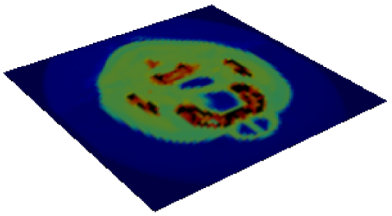
Überprüfungen doppelt so groß wie die Anzahl der Pixel im Dreieck. Für die Pixel im Dreieck müssen ohnehin Koordinaten zur Interpolation berechnet werden, wichtig ist die Reduzierung der Anzahl der restlichen, negativ geprüften Pixel.

2.3. Zeichnen eines Dreiecksgitters

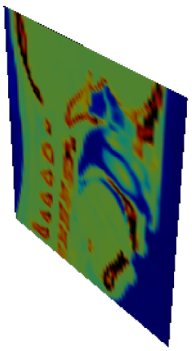
Der Software-Renderer des GR3 war bisher nur in der Lage bivariate Funktionen als Oberflächen zu visualisieren, welche vom Nutzer mit Hilfe des Aufrufs von `gr3_surface` erzeugt werden. Es können aber auch andere dreidimensionale Strukturen wie zum Beispiel Molekülgitter, Pendel oder ähnliches hardwarebeschleunigt visualisiert werden. Im Rahmen der Bachelorarbeit soll diese Funktionalität ebenfalls durch den Software-Renderer unter möglichst geringem Verlust der Performance unterstützt werden.



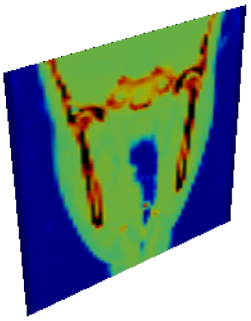
(a) Erstes Dreiecksgitter in der Szene



(b) Zweites Dreiecksgitter in der Szene



(c) Drittes Dreiecksgitter in der Szene



(d) Viertes Dreiecksgitter in der Szene

Abbildung 2.6.: Darstellung der einzelnen Dreiecksgitter

Eine dreidimensionale Grafik kann in GR3 aus mehreren Dreiecksgittern bestehen [Rhi12], welche von GR3 beim Aufruf verschiedener Methoden aus den Ausgangsdaten erzeugt werden. Ein Dreiecksgitter enthält ein Array mit Eckpunkten, die sich

auf der darzustellenden Struktur befinden, und Informationen darüber, welche dieser Eckpunkte zusammen ein Dreieck bilden. Die Information der Zusammengehörigkeit der Dreiecke ist in der Regel durch die Reihenfolge gegeben, in der die Eckpunkte hinterlegt sind. Sie kann aber auch optional in einem Indexpuffer gespeichert sein, der beliebige Eckpunkte referenzieren kann und daher eine mehrfache Nutzung gleicher Eckpunkte ermöglicht. Ein Dreiecksgitter bildet häufig ein in der Szene vorhandenes Objekt, wie Abbildung 2.6 verdeutlichen soll.

Zudem sind in jedem Dreiecksgitter die zu den Eckpunkten gehörenden Normalen und Farben hinterlegt, welche im Weiteren zur Darstellung beitragen, indem sie die Beleuchtung und die Farbe der Dreiecke und damit auch der Pixel beeinflussen. In einem Dreiecksgitter sind drei Fließkommawerte hinterlegt, mit denen die Intensität der Farbkanäle rot, grün und blau `RGB` jedes Pixels reguliert werden kann. Dadurch kann beispielsweise nur ein Farbkanal hervorgehoben oder die gesamte Szene verdunkelt werden.

Der Software-Renderer erhält einen mit einer Hintergrundfarbe gefüllten Farbpuffer und eine Liste mit Dreiecksgittern, mit deren Hilfe dieser Farbpuffer so gefüllt wird, dass ihre Visualisierung in der erwünschten Grafik resultiert. Dazu wird über die Dreiecksgitter iteriert und jedes nacheinander ohne Hardwarebeschleunigung in besagten Farbpuffer gezeichnet, sodass nach jedem abgearbeiteten Dreiecksgitter Informationen in jenem hinzukommen. In der resultierenden Grafik sind die Dreiecksgitter so gemischt, das bei jedem Pixel die Farbinformation des Dreiecksgitters hinterlegt ist, das sich am weitesten vorne befindet. In diesem Fall sieht die Grafik (Abbildung 2.7) wie folgt aus.

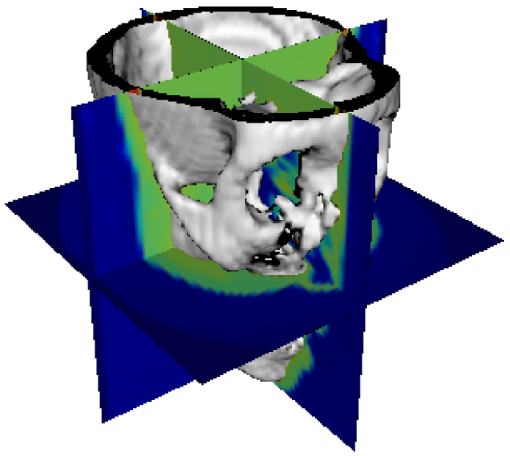


Abbildung 2.7.: Alle Dreiecksgitter der Szene vereint

Innerhalb einer Szene können verschiedene Transformationsmatrizen auf die Eckpunkte verschiedener zu dieser Szene gehörenden Dreiecksgitter angewandt werden. Die Lichtquelle hat immer denselben Ursprung, da schließlich die gesamte Szene aus genau einer Position beleuchtet wird.

2.4. Automatische Auswahl des Software-Renderers

Der Zweck des Software-Renderers ist, wie eingangs erwähnt, die Unterstützung der durch die GR3-Grafikbibliothek möglichen Visualisierungen in Umgebungen, in denen nicht auf die dazu nötige Hardware zugegriffen oder jene nicht angemessen ersetzt werden kann. Wünschenswert ist also, dass die GR3-Grafikbibliothek selber anhand der Umgebung, in der sie verwendet wird, entscheidet, ob mit dem Software-Renderer oder mit OpenGL gezeichnet wird.

GR3 entscheidet bislang abhängig vom Betriebssystem durch Präprozessor-Makros zur Übersetzungszeit, wie OpenGL initialisiert werden soll. Die Abfrage des verwendeten Betriebssystems geschieht über jeweils auf solchen definierten Makros. Auch das Vorhandensein eines Backends für OpenGL kann über Makros abgefragt werden. Fällt diese Abfrage negativ aus, wird automatisch auf den Software-Renderer zurückgegriffen und eine Warnung ausgegeben. Die Ergebnisse und die Laufzeit sollen sich jedoch so wenig wie möglich unterscheiden.

2.5. Beleuchtung

Bis jetzt war der Software-Renderer nur in der Lage eine weiße, gleichmäßige und maximale Ausleuchtung aller Flächen beziehungsweise Dreiecke zu ermöglichen. Dies ist eine Variante des *Ambient Lighting* aus [Vri14], bei welchem eine omnidirektionale Lichtquelle mit konstanter Intensität und fester Farbe die Szene ausleuchtet. Diese einfachste Art der Beleuchtung kommt zum Einsatz, wenn ein Überblick über eine Szene und die darin enthaltenen Objekte gewonnen werden soll. Zur Realisierung ist dabei lediglich das Multiplizieren jeder Komponente der Farbe eines Pixels mit gegebenen Konstanten notwendig, die dann die Farbintensität und das Mischverhältnis bestimmen.

(a) *Ambient Lighting* (b) *Diffuse Lighting*

Abbildung 2.8.: Zwei Beleuchtungsmöglichkeiten derselben Szene

In einer dreidimensionalen Szene kann jedoch die Position und Farbe einer punktuellen Lichtquelle und dadurch die Richtung des Lichteinfalls definiert werden, die dann im *Diffuse Lighting* [Vri14] resultiert. Abhängig davon sind Flächen, die nahezu senkrecht zum Lichteinfall liegen, sehr intensiv beleuchtet und solche, deren Rückseite beleuchtet wird, dunkel. Durch diesen Effekt wirkt die Szene wesentlich realistischer, weil eine bessere Wahrnehmung der Tiefe entstehen kann und Schattierungen einen dreidimensionalen Eindruck erwecken. Die hardwarebeschleunigte Variante des GR3 unterstützt diese Art der Beleuchtung [Rhi12]. Im direkten Vergleich unterscheiden sich die Beleuchtungsvarianten für ein konkretes Beispiel wie in Abbildung 2.8 dargestellt.

Aus mathematischer Sicht ergibt sich die Beleuchtungsstärke an einer konkreten Stelle abhängig vom Winkel zwischen der Richtung des Lichteinfalls und dem orthogonal auf der Stelle stehenden Vektors [Vri14]. Je größer der Winkel, desto weniger wird diese Stelle beleuchtet. Wird die Rückseite einer Stelle beleuchtet, beträgt der Winkel zwischen der Normale und dem Lichteinfall mehr als 90°, sodass diese dann gänzlich unbeleuchtet bleibt. In der Regel kommen mehrere Beleuchtungstechniken in Kombination zum Einsatz, jedoch unterstützt GR3 ausschließlich Kombinationen aus dem *Diffuse Lighting* und dem *Ambient Lighting*.

2.5.1. Transformation der Normalen

Zu jedem Eckpunkt ist ein Normalenvektor in *Local-Space*-Koordinaten (vgl. [Rit19]) gegeben, welcher senkrecht auf diesem steht. Bevor diese jedoch zur Berechnung der Beleuchtungseffekte verwendet werden können, müssen Transformationen vorgenommen werden, bis schließlich *View-Space*-Koordinaten vorliegen, in denen die Berechnung der Farbe der Pixel, in die die Beleuchtung einfließt, stattfindet. Um die Koordinaten in den *View-Space* zu transformieren kann nicht einfach mit der *Model*- und anschließend mit der *View*-Matrix multipliziert werden, wie es bei Eckpunkten möglich ist.

Im Software-Renderer des GR3 werden analog zu OpenGL homogene Koordinaten für die Hinterlegung von Orts- und Richtungsvektoren verwendet, sodass die eigentlich dreidimensionalen Koordinaten um eine vierte Dimension erweitert werden. Bei den Ortsvektoren, wie zum Beispiel Eckpunkten, hat die vierte Komponente den Wert Eins, damit durch lineare Operationen, wie eine Matrizenmultiplikation, eine Translation formuliert werden kann. Da die Normalen Richtungsvektoren und keine Ortsvektoren sind, ist deren letzter Wert Null, sodass Translationen, die in der vierten Spalte der *Model*-Matrix ausgedrückt werden, keinen Einfluss nehmen, weil Richtungen unabhängig von ihrer Ausgangsposition sind. Daher ist nur der Inhalt der oberen linken 3×3 Matrix relevant, in der Rotationen und Skalierungen vorgenommen werden.

Es können durch eine Matrix achsenweise verschiedene Skalierungen, also nicht orthonormale Transformationsmatrizen, zu einer Änderung des Normalenvektors führen. Veranschaulicht wird dieser Zusammenhang in der folgenden Abbildung

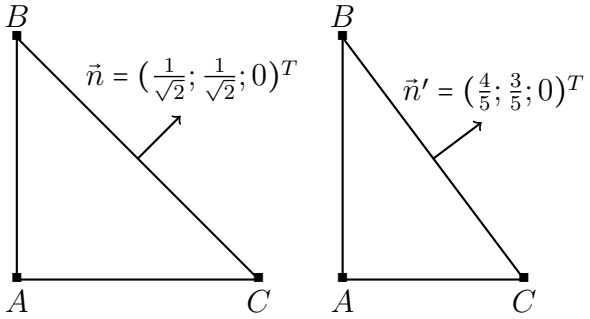


Abbildung 2.9.: Achsenweise unterschiedliches Skalieren und der Einfluss auf den Normalenvektor

Skaliert wurden die Eckpunkte hier beispielsweise mit der Matrix (obere linke 3×3-Matrix):

$$M = \begin{pmatrix} \frac{3}{4} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Der Normalenvektor hat sich durch die Skalierung verändert, eine Verwendung vom ursprünglichen Normalenvektor bei der Berechnung der Beleuchtung würde zu einem falschen Ergebnis führen. Dies liegt darin begründet, dass im Unterschied zu den Eckpunkten der Normalenvektor nicht die Differenz zwischen zwei Punkten beschreibt, wie es zum Beispiel bei der Kante in einem Dreieck der Fall ist, sondern als ein zur Oberfläche orthogonaler Richtungsvektor definiert wird.

Zur korrekten Berechnung wird in diesem Fall statt der *ModelView*-Matrix M (Produkt aus *Model*- und *View*-Matrix) die sogenannte *Normal*-Matrix N [Len02] verwendet.

Sei $\vec{v}$ die Richtung, auf der der Normalenvektor $\vec{n}$ senkrecht steht. Dann muss gelten:

$$\begin{aligned} 0 = \vec{n} \cdot \vec{v} &= (N\vec{n}) \cdot (M\vec{v}) \\ &= (N\vec{n})^T (M\vec{v}) \\ &= \vec{n}^T N^T M \vec{v} \end{aligned}$$

Nach der Voraussetzung gilt, dass $\vec{n} \cdot \vec{v} = 0$, also wenn gilt, dass $N^T M = I$, ist genau diese erfüllt. Aus der Bedingung ergibt sich aus der oberen linken 3×3 Matrix der *ModelView*-Matrix M die Formel für die gesuchte *Normal*-Matrix N:

$$\begin{aligned} I &= N^T M \\ \Leftrightarrow N &= (M^{-1})^T \end{aligned}$$

Mit dieser werden alle Normalenvektoren vor deren Interpolation und der darauf folgenden Berechnung der Farbe jedes Pixels multipliziert, sodass die Normalen nach diesem Schritt in *View*-Space Koordinaten vorliegen. Für das genannte Beispiel wäre

dies

$$N = \begin{pmatrix} \frac{4\sqrt{2}}{5} & 0 & 0 \\ 0 & \frac{3\sqrt{2}}{5} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Damit gilt in Abbildung 2.9 die Gleichung $N\vec{n} = \alpha \cdot \vec{n}'$, also erzeugt diese Transformation den korrekten Normalenvektor, der normiert dargestellt ist.

2.5.2. Interpolation der Normalen

Genau wie die Farben sind die Normalen nur an den Eckpunkten aller Dreiecke definiert. Zum korrekten Einfärben wird jedoch für jeden Pixel nicht nur die interpolierte Farbe, sondern auch der zum Pixelmittelpunkt orthogonale Vektor benötigt, der die Farbe durch das *Diffuse Lighting* beeinflusst. Die baryzentrischen Koordinaten λ_i müssen ohnehin für spätere Berechnungen zur Interpolation der Tiefe und Farbe berechnet werden. Außerdem wurde bei der perspektivisch korrigierten Interpolation der Farbe die Interpolation der invertierten Tiefenwerte

$$z'_p = \lambda_0 \frac{1}{z_{v_0}} + \lambda_1 \frac{1}{z_{v_1}} + \lambda_2 \frac{1}{z_{v_2}}$$

durchgeführt. Der Wert z'_p wird im Anschluss verwendet, um auch die Normalen perspektivisch korrigiert interpolieren zu können, da im *Screen Space* nicht der Normalenvektor n_p selbst, sondern $\frac{n_p}{-z}$ linear auf der Oberfläche des Dreiecks abhängig von den Normalen der Eckpunkte variiert [Rit19]. Analog können mit den selben Werten auch die in den *View*-Space transformierten Normalen perspektivisch korrigiert interpoliert und pixelweise bereit gestellt werden.

$$\vec{n_p}' = \lambda_0 \frac{n_{v_0}}{z_{v_0}} + \lambda_1 \frac{n_{v_1}}{z_{v_1}} + \lambda_2 \frac{n_{v_2}}{z_{v_2}}$$
$$\Rightarrow n_p = \frac{n'_p}{z'_p}$$

2.5.3. Berechnung der Farbe unter Berücksichtigung des Lichteinfalls

Bei der Rasterisierung werden zur Einfärbung der Pixel die Farben, die ausschließlich an den Eckpunkten der Dreiecke definiert sind, perspektivisch korrigiert interpoliert. Hinzu kommt nun die Berücksichtigung des Lichteinfalls. Dabei wird ein Pixel umso intensiver beleuchtet, je mehr dessen Normalenvektor in die Richtung des Lichteinfalls zeigt beziehungsweise je kleiner der Winkel zwischen diesen beiden Vektoren ist.

Der Normalenvektor ist durch Interpolation der Normalen der Eckpunkte gegeben. Die Richtung des Lichts l kann in *World-Space*-Koordinaten vorgegeben werden und muss ebenfalls wie die Normalenvektoren in den *View-Space* transformiert werden. Ist sie nicht definiert, wird von einer Lichtquelle an der Position des Betrachters aus-

gegangen, welches in *View-Space*-Koordinaten dem Vektor $l = (0, 0, 1)^T$ entspricht. Aufgrund der Einfachheit dieses Standardvektors im *View-Space* findet die Berechnung in diesen Koordinaten statt, da sonst in einem anderen Koordinatensystem der entsprechende Vektor ermittelt werden müsste.

Die folgende Erläuterung ist aus [Vri14] entnommen. Sind also nun sowohl die Richtung des Lichteinfalls $\vec{l}$ und der Normalenvektor $\vec{n_p}$ eines Pixels bekannt, kann der Cosinus des Winkel fac zwischen diesen beiden Vektoren mit dem Skalarprodukt aus den normierten Vektoren berechnet werden.

$$\text{fac} = \frac{l}{||l||} \cdot \frac{n_p}{||n_p||}$$

Ist der Wert $\text{fac} < 0$, so wird die Fläche vom Licht nicht beeinflusst und trifft auf die Rückseite, sodass $\text{fac} = 0$ gesetzt wird, was einer schwarzen Fläche entspricht. Andernfalls wird der Wert fac übernommen und dient als Multiplikator aller Farbkomponenten. Somit ergibt sich die Farbe $\vec{c_p}'$ eines einzufärbenden Pixels nach Multiplikation mit der durch perspektivisch korrigierte Interpolation ermittelten Farbe $\vec{c_p}$

$$\begin{aligned} \text{fac}_{dif} &= \max(\text{fac}, 0) \\ \vec{c_p}' &= (\text{fac}_{dif} + \text{fac}_{amb}) \cdot c_p \end{aligned}$$

Der Wert fac_{amb} bezieht sich auf die Stärke des *Ambient Lighting*, hat aber standardmäßig den Wert Null. Um die Farbe des Lichtes zu berücksichtigen, die vom Nutzer definiert werden kann, wird zum Schluss nochmal der Vektor $\vec{c_p}'$ komponentenweise mit einem Vektor multipliziert, welcher in jeder Komponente eine Intensität aus $[0, 1]$ für jede Farbe des Lichtes enthält.

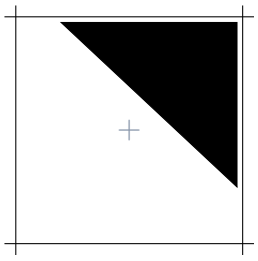
2.6. Kantenglättung

2.6.1. Ziel

In der hardwarebeschleunigten Variante des GR3 wird Kantenglättung unterstützt. Die darin implementierte Variante nennt sich *Supersample Anti-Aliasing* [Hug+14], kurz `SSAA`. Ziel ist es, unerwünschte optische Effekte, die auf einem Pixelraster entstehen, zu minimieren.

Letztendlich besteht der Rasterisierungsvorgang aus der Zuweisung einer Farbe für jeden Pixel. Diese Farbe wird dabei berechnet, indem die Bildbeschreibung ausschließlich an den Mittelpunkten der Pixel abgetastet wird und für diese Koordinaten die resultierende Farbe berechnet dem Pixel zugewiesen wird. Alle Punkte des Bildes außer die Mittelpunkte der Pixel fließen gar nicht in die Berechnung beziehungsweise Darstellung ein (vgl. Abbildung 2.10 (a)), sodass Dreiecksgitter, die sich innerhalb eines Pixels befinden, aber dessen Mittelpunkt nicht überschneiden überhaupt keine Berücksichtigung in der Visualisierung finden. Bei regelmäßiger Anordnung kleiner Dreiecksgitter gehen all diese Daten verloren, was einer der unerwünschten Effekte

ist, die `SSAA` zu beheben versucht. Besonders ins Gewicht fallen kann dieser Effekt bei der Darstellung eines schachbrettähnlichen Musters, wenn zwischen zwei Abtastpunkten ein ganzes Feld verloren geht (vgl. Abbildung 2.10 (b)) und deswegen zwei gleiche Farben in zwei Reihen aufeinander folgen. In derartigen Fällen wird die Grafik als unterabgetastet bezeichnet. Im Allgemeinen kann nie die exakte Darstellung einer Bildbeschreibung garantiert werden, da sonst unendlich viele Abtastungspunkte und Pixel notwendig wären, um theoretisch beliebig kleine Dreiecksgitter genau darstellen zu können.

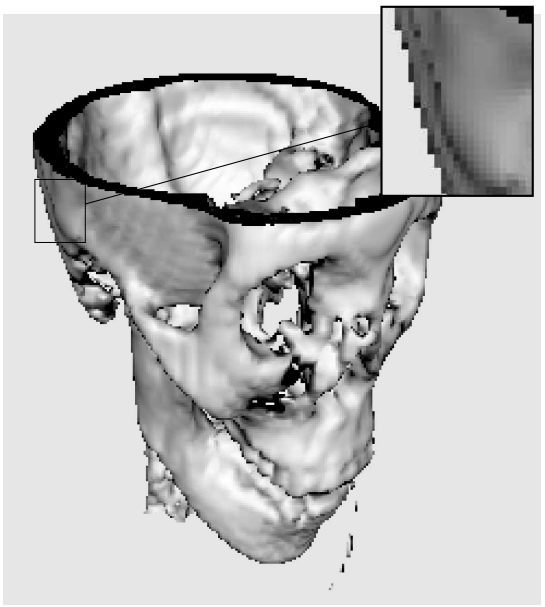


(a) Dreieck innerhalb eines Pixels, das durch die Abtastung nicht registriert wird

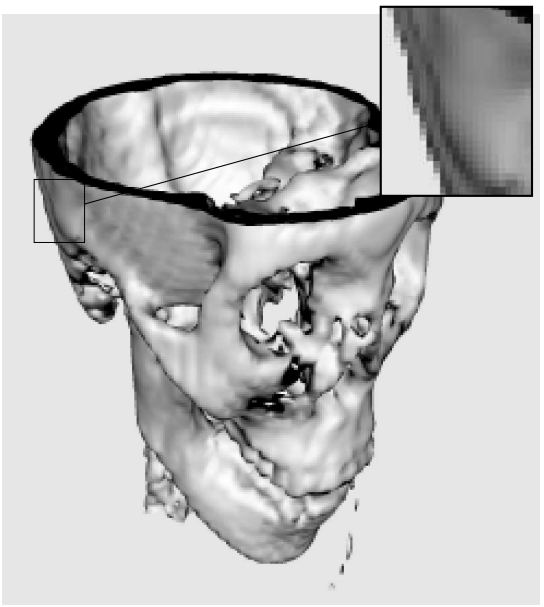
(b) Abtastung eines schachbrettähnlichen Musters unter Verlust eines Feldes

Abbildung 2.10.: Unerwünschte Effekte ohne Kantenglättung

Ein weiterer negativer Effekt von Rastergrafiken ist der Treppeneffekt, welcher das kantige Erscheinungsbild gerasterter Objekte beschreibt. Besonders präsent wird dieser Effekt in Animationen, weil sich dort Teile des Bildes ruckartig bewegen oder gar zu flimmern scheinen.



(a) Ohne Kantenglättung



(b) Mit Kantenglättung

Abbildung 2.11.: Die selbe Szene ohne und mit Kantenglättung (Downsampling der vierfachen Auflösung) zur Verdeutlichung des Treppeneffekts

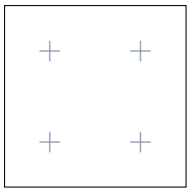
2.6.2. Funktionsweise

Ohne `SSAA` wird die Bildbeschreibung an allen Pixelmittelpunkten des schlussendlich berechneten Bildes abgetastet, um eine möglichst gleichmäßige und genaue Darstellung zu ermöglichen. Der Pixel wird in der Farbe eingefärbt, die die Bildbeschreibung in dessen Mittelpunkt annimmt.

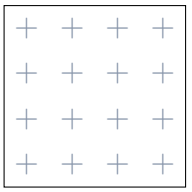
Mit `SSAA` wird die Bildbeschreibung innerhalb eines Pixels an n gleichmäßig verteilten Positionen ausgewertet. Durch einen Rekonstruktionsfilter wird anschließend aus diesen n Farbinformationen der abgetasteten Stellen eine Farbe berechnet und dem Pixel schlussendlich zugewiesen. Dabei unterscheiden sich verschiedene Arten des `Supersampling` sowohl durch die unterschiedlichen Anordnungen und Anzahlen der Abtastpositionen pro Pixel, als auch durch die Wahl des Rekonstruktionsfilters. Für gewöhnlich besteht dieser aber aus der Mittelung aller berechneten Farben der zu einem Pixel gehörigen Abtastpunkte. Dabei können sich Pixel auch Abtastpunkte teilen, wenn sie auf den Grenzen liegen. Dies wird `Sample Sharing` genannt [Gra16].

2.6.3. Realisierung

In GR3 existiert die Funktion `gr3_setquality(int quality)`, mit der die Anzahl der Abtastpunkte pro Pixel und dadurch die Qualität der erzeugten Grafik festgelegt wird. Beim Aufruf ersetzt `GR3_QUALITY_OPENGL_<`q`>X_SSAA` den Parameter `quality`, wobei q Werte aus der Menge {2,4,8,16} annehmen kann. Letzten Endes resultiert dies in q^2 Abtastungspunkten pro Pixel, welche gleichmäßig verteilt werden. [Hug+14]



(a) Abtastpunkte innerhalb eines Pixels für $q = 2$

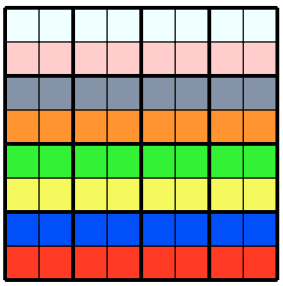


(b) Abtastpunkte innerhalb eines Pixels für $q = 4$

Abbildung 2.12.: Abtastpunkte für verschiedene q, an denen der Farbwert errechnet und zusammen mit den anderen gemittelt dem Pixel zugewiesen wird

Wird diese Methode vom Nutzer aufgerufen und mit der Qualität q seiner Wahl spezifiziert, wird die Grafik intern zunächst in q^2-facher Auflösung gerendert, d.h. mit q-facher Breite und Höhe. So wird pro Abtastpunkt ein Pixel erzeugt, in dessen Mitte sich der Abtastpunkt befindet. Aus dieser höher aufgelösten Grafik werden nun die Farbinformationen von den Quadraten mit jeweils q^2 Pixeln mittels Rekonstruktionsfilters zusammengefasst, indem der Wert jedes Farbkanals von jedem dieser Pixel addiert und dann mit $\frac{1}{q^2}$ multipliziert wird. Die daraus resultierende Farbe wird dem entsprechenden Pixel der fertigen Grafik, die wiederum nicht q^2-fach sondern

einfach aufgelöst ist, zugewiesen. Dieser Vorgang simuliert das mehrfache Abtasten der Bildinformation pro Pixel in der endgültigen Grafik.



(a) Höher aufgelöste Grafik mit $q = 2$

(b) Resultierendes Ergebnis durch Zusammenfassung der höher aufgelösten Grafik

Abbildung 2.13.: Übersetzung der $q^2 = 4$-fach aufgelösten Grafik zur Endgrafik (einfach aufgelöst)

3. Optimierung

Wie in Kapitel 1 erwähnt soll die Laufzeit des Software-Renderers möglichst gering sein, um aus Datenmengen resultierende Darstellungen möglichst in Echtzeit darstellen zu können. Dazu wird im folgenden die benötigte Zeit für den bislang nicht parallelen Renderingprozess reduziert. Im Fokus sind dabei die Vermeidung aufwändiger Speicheroperationen, die effizientere Berechnung baryzentrischer Koordinaten und vor allem die Beschleunigung der Rasterisierung.

3.1. Reduktion von Speicheroperationen

Die Koordinaten der Eckpunkte sowie deren Farben und Normalenvektoren werden vom Software-Renderer als Array aus `float` Werten empfangen. Dabei wird zudem ein Farb- und Tiefenpuffer, welche sich auf dem Heap befinden, zur Verfügung gestellt, in denen pro Pixel vier Farbkanäle beziehungsweise ein Tiefenwert hinterlegt werden können. Bevor die Eckpunkte unterschiedliche Transformationen durchlaufen, werden sie in ein `struct vertex_fp` umgewandelt. Dieses besitzt als Attribute die vier Koordinaten und ein `struct color` (Farbe) sowie ein `struct vector` (Normalenvektor), sodass ein Eckpunkt mitsamt aller seiner Informationen auf diese Art hinterlegt werden kann.

Der Kopiervorgang ist nötig, da auf den Eckpunkten verschiedene Transformationen vorgenommen werden, welche mit diesen Datentypen arbeiten. Dazu gehören sowohl die auf den Eckpunkten operierenden *Model-*, *View-*, *Perspective-* und *Viewport-*Transformationen, als auch die Transformationen für die Beleuchtung.

Die Verwendung von Speicherplatz auf dem Heap muss minimiert werden, da die damit zusammenhängenden Allokieroperationen langsam sind [AS96]. Übergeben werden an alle Methoden ausschließlich Zeiger, da das in der Programmiersprache `C` verwendete Call-by-Value ansonsten bei jedem Aufruf eine Kopie erzeugt, was zu Lasten der Effizienz fällt. Vor dieser Optimierung war die Ausführungszeit deutlich erhöht, konnte aber durch diese simple Änderung verringert werden.

Eine Ausnahme bildet dabei die Übergabe von baryzentrischen Koordinaten an eine Methode. Hierbei werden die drei Werte einzeln übergeben, da sich dies als schneller herausstellte, als die Übergabe eines Zeigers auf ein Feld mit drei Werten. Ursache dafür ist, dass die mehrfache Dereferenzierung die Ausführungszeit erhöht.

3.2. Berechnung baryzentrischer Koordinaten

Baryzentrische Koordinaten sind im Falle eines Dreiecks drei Werte λ_0, λ_1 und λ_2. Die Koordinaten werden genutzt, um mithilfe der Eckpunkte V_0, V_1 und V_2 des Dreiecks einen beliebigen Punkt P in der Form $P = \lambda_0 V_0 + \lambda_1 V_1 + \lambda_2 V_2$ auf der Dreiecksfläche darstellen zu können. Im Wesentlichen besteht der Aufwand der Überprüfung, ob ein Pixel zum Zeichnen eines Dreiecks eingefärbt werden muss darin, die baryzentrischen Koordinaten für dessen Mittelpunkt zu berechnen. Ist mindestens eine der Koordinaten negativ, liegt der Pixelmittelpunkt außerhalb des Dreiecks.

Selbst wenn die Überprüfung nicht nötig wäre, soll der Aufwand zur Berechnung baryzentrischer Koordinaten möglichst gering sein, da diese zur Tiefen-, Farb- und Normaleninterpolation dienen. Zur Berechnung der Koordinaten für einen Punkt P werden die Teilflächen betrachtet, die durch Verbinden dieses Punktes mit den drei Eckpunkten entstehen. Der Anteil einer Teilfläche an der Gesamtfläche des Dreiecks entspricht der baryzentrischen Koordinate des Punktes P für den gegenüber der Teilfläche liegenden Eckpunkt. Daraus folgt die Darstellung des Punktes in der Form

$$P = \frac{A_{V_0}}{A_{ges}} V_0 + \frac{A_{V_1}}{A_{ges}} V_1 + \frac{A_{V_2}}{A_{ges}} V_2 \tag{3.1}$$

die in der folgenden Grafik veranschaulicht wird.

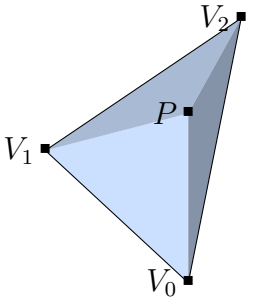


Abbildung 3.1.: Teilflächen zur Berechnung baryzentrischer Koordinaten

Das doppelte des Flächeninhalts A_{V_i} jedes Teildreiecks ergibt sich aus dem Betrag des Kreuzproduktes von zwei seiner Kanten. Der Wert ist negativ, falls P außerhalb des Gesamtdreiecks liegt. Zum Beispiel gilt für das Teildreieck gegenüber des Eckpunktes V_0

$$\begin{aligned} A_{V_0} &= \|\overrightarrow{V_1 P} \times \overrightarrow{V_1 V_2}\| \\ &= (V_{2_y} - V_{1_y})(P_x - V_{1_x}) - (V_{2_x} - V_{1_x})(P_y - V_{1_y}) \\ &= (V_{1_y} - V_{2_y})P_x + (V_{2_x} - V_{1_x})P_y + (V_{1_x} V_{2_x} - V_{1_y} V_{2_x}) \\ &= A_0 \cdot P_x + B_0 \cdot P_y + C_0 = A_{V_0}(P_x, P_y) \end{aligned}$$

Die Idee dieser Optimierung stammt aus [Gie13]. Die Eckpunkte V_i sind für jedes Dreieck konstant, sodass eine lineare Funktion in P entsteht. Der Algorithmus ar-

beitet jeden potentiell zum Dreieck gehörenden Pixel ab und vollzieht daher innerhalb jeder Zeile Schritte in x-Richtung nach rechts beziehungsweise beim Wechsel der Zeile in y-Richtung nach oben. Zum Beispiel wird die Teilfläche A_{V_0} des rechten Nachbarn eines Pixels P dabei durch $A_{V_0}(P_x + 1, P_y)$ berechnet, analog dazu die des oberen Nachbarn mittels $A_{V_0}(P_x, P_y + 1)$. Auf Grund der Linearität der Funktion gilt $A_{V_i}(P_x + 1, P_y) - A_{V_i}(P_x, P_y) = A_i$ beziehungsweise $A_{V_i}(P_x, P_y + 1) - A_{V_i}(P_x, P_y) = B_i$, was darin resultiert, dass die baryzentrischen Koordinaten inkrementell berechnet werden können.

Dazu werden zunächst alle drei Teilflächen des ersten zu prüfenden Pixels mittels $A_{V_i}(P_x, P_y)$ berechnet. Sollen als nächstes die Koordinaten des rechten Nachbarn ermittelt werden, so werden alle Teilflächen A_{V_i} um den Wert A_i erhöht, während analog dazu die des oberen Nachbarn sich durch Erhöhung jeder Teilfläche um B_i ergeben.

Bislang wird die Berechnung der Dreiecksfläche A_{ges} und der Teilflächen A_{V_i} bei der Berechnung der Koordinaten jedes Pixels erneut gemäß Formel 3.1 durchgeführt. Die Fläche A_{ges} (das doppelte der Gesamtfläche des Dreiecks), muss nur ein Mal berechnet und hinterlegt werden und resultiert ebenfalls aus dem Kreuzprodukt zweier Kanten des Dreiecks. Durch diese alternative Art baryzentrische Koordinaten zu berechnen sind nun für die Berechnung aller Teilflächen zu einem Pixel nur drei Additionen notwendig, anstatt wie vorher fünf Additionen und zwei Multiplikationen.

3.3. Rasterisierung

3.3.1. Optimierung des vorhandenen Algorithmus

3.3.1.1. Frühzeitiges Clipping

Ein frühes Clipping (also das Verwerfen nicht in den darzustellenden Bereich fallender Teile) minimiert die Anzahl der Operationen mit Koordinaten, die am Ende verworfen werden. Das umgebende Rechteck aus Abschnitt 2.2 wird daher nicht nur von den maximalen bzw. minimalen x- und y-Werten des Dreicks limitiert, sondern auch von der maximalen Breite und Höhe des Darstellungsbereiches. So muss nicht bei jedem Pixel geprüft werden, ob seine Koordinaten außerhalb dieses Bereiches liegen, sondern das Kriterium wird schon bei der Festlegung der `Bounding-Box` erfüllt, welche nur pro Dreieck berechnet werden muss.

Zusätzlich wird vor der Interpolation der Farbe inklusive Beleuchtungsberechnung im Tiefenpuffer geprüft, ob an diesem Pixelmittelpunkt ein Wert hinterlegt ist, der zu einem Objekt gehört, das sich dort näher am Betrachter befindet. Da der Pixel ohnehin im Falle einer positiven Prüfung seinen Farbwert behält, müssen keine Farb- und Beleuchtungsberechnungen stattfinden. Ein Pixel, welcher nicht eingefärbt wird, weil dort ein näheres Objekt Tiefenpuffer hinterlegt ist, hat also lediglich die Interpolation der Tiefe zu Folge.

Allgemein ist es vorteilhaft, so viele Operationen wie möglich vor dem Durchlaufen der Schleife zu vollziehen. Dazu gehört die Berechnung des Flächeninhalts des Dreiecks, durch die die baryzentrischen Koordinaten zur Skalierung auf den Bereich

[0,1] geteilt werden müssen. Der Divisor muss nicht für jeden Pixel erneut berechnet werden.

3.3.1.2. Abbruchbedingung innerhalb jeder Zeile

Der Algorithmus mit begrenzendem Viereck aus Abschnitt 2.2 arbeitet jede Zeile von links nach rechts ab und färbt sie entsprechend ein. Innerhalb jeder Zeile ist bekannt, ob in dieser schon ein Pixel eingefärbt wurde, oder nicht. Falls ja, ist der aktuelle Ort des Rasterisierungsvorgangs entweder im Dreieck oder rechts davon, falls nicht, ist er links davon. Ist ersteres der Fall und der Algorithmus befindet sich aktuell nicht auf einem einzufärbenden Pixel, kann mit der nächsten Zeile fortgefahren werden, da ohnehin kein einzufärbender Pixel mehr in dieser Zeile existieren kann. Dadurch wird das Rechteck rechtsseitig ausgefranst. In der folgenden Abbildung sind die eingefärbten Pixel mit einem grauen Kreuz versehen. Die mit einem roten Kreuz markierten Pixel sind jene, in denen mit der nächsten Zeile fortgefahren werden kann.

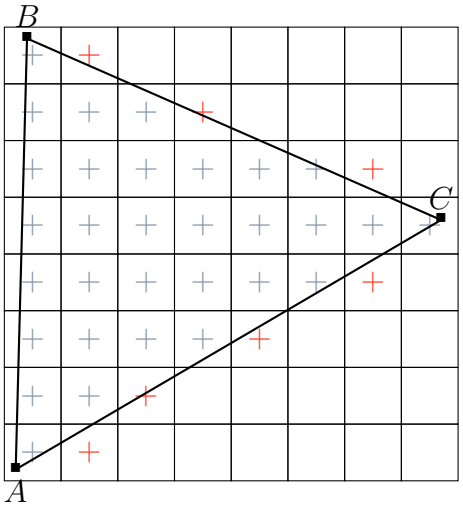


Abbildung 3.2.: Der zeilenweise frühere Abbruch bei der Rasterisierung

Mit Hilfe dieser Optimierung wird im Durchschnitt bei jedem Dreieck ungefähr die Hälfte der fälschlich geprüften Pixel gespart, in diesem Fall sogar mehr. Insgesamt werden für halb so viele Pixel, wie sich im Dreieck befinden, die baryzentrischen Koordinaten berechnet nur um festzustellen, dass sie sich nicht im Dreieck befinden.

3.3.1.3. Rasterisierung von links und rechts

Der Algorithmus arbeitet in vertikaler Richtung von unten nach oben und in horizontaler von links nach rechts. In jeder Zeile wird die Abbruchbedingung geprüft, die im vorherigen Abschnitt erläutert wurde, und kann damit für eine Ersparnis sorgen. Im Worst-Case müssen aber dennoch die baryzentrischen Koordinaten für alle Pixelmittelpunkte des umgebenden Rechtecks berechnet werden, wodurch weitere Rechenzeit

verbraucht wird. Einen solchen Worst-Case bildet jedes Dreieck, das eine Kante hat, die den rechten Rand des Rechtecks wie beispielsweise in Abbildung 3.3 zeigt, gänzlich abdeckt.

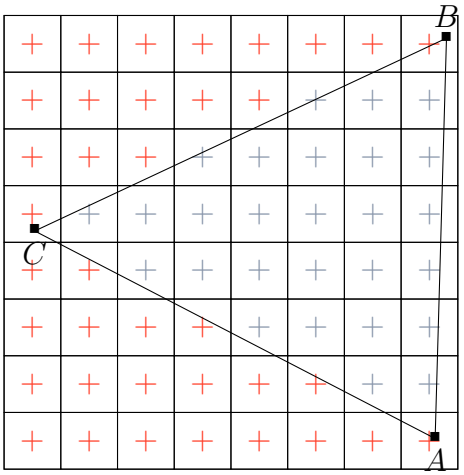


Abbildung 3.3.: Beispielhafter Worst-Case für die Rasterisierung bezüglich der Bedingung aus 3.3.1.2

In diesem Fall können keine Pixel durch die Optimierung aus Abschnitt 3.3.1.2 eingespart werden, weil die zeilenweise Rasterisierung nach Betreten des Dreiecks selbiges nicht mehr verlässt, und dementsprechend keine Pixel übersprungen werden können.

Würde die Rasterisierung von rechts nach links verlaufen, könnten alle Pixel nach dem ersten fälschlich geprüften pro Zeile eingespart werden, also eine hohe Ersparnis erzielt werden und der Worst-Case würde zum Best-Case umgewandelt werden, da eine höhere Ersparnis nicht möglich ist. Die Funktionsweise ist dabei analog zu der, wenn von links nach rechts rasterisiert wird. Das Rechteck kann linksseitig ausgefranst werden, weil sich nach dem Verlassen des Dreiecks keine Pixel mit ihren Mittelpunkten innerhalb des Dreiecks befinden können.

Nun fehlt ein Kriterium zur Entscheidung, ob das Dreieck von links nach rechts oder umgekehrt rasterisiert werden soll. Es bietet sich an, die x-Koordinate des Schwerpunktes des Dreiecks zu bestimmen. Je nach dem an welcher vertikalen Kante des Rechtecks bzw. wo im Intervall $[x_{min}, x_{max}]$ sich der Wert befindet, wird von dieser Seite beginnend rasterisiert, da erwartet wird, dass sich dort der Großteil aller zum Dreieck gehörigen Pixel befindet. In Abbildung 3.3 hätte die x-Koordinate des Schwerpunktes S bei einem Rechteck der Breite Eins den Wert $\frac{2}{3}$, welcher näher an der rechten Kante $x_{max} = 1$ liegt. Somit wird das Dreieck beginnend von rechts nach links rasterisiert.

Durch diese Methode kann erwartet werden, dass nur noch 25% aller Pixel, die innerhalb des Rechtecks negativ auf Zugehörigkeit des Dreiecks getestet würden überhaupt noch geprüft werden.

3.3.2. Window Search Rasterisierung

3.3.2.1. Kritik an der Rasterisierung mit umgebenden Rechteck

Der Ansatz der Rasterisierung aus Abschnitt 2.2 ist zwar simpel und garantiert die vollständige Einbeziehung aller Pixel innerhalb des Dreiecks, jedoch involviert er auch die Abtastung von Pixelmittelpunkten, welche nicht zum Dreieck gehören und dementsprechend nicht gezeichnet werden. Für solche müssen die baryzentrischen Koordinaten berechnet werden, um in Folge dessen zu prüfen, ob sie das gleiche Vorzeichen haben. Nur in diesem Fall liegen sie wirklich innerhalb des Dreiecks und es kommt zur Einfärbung des Pixels. In der folgenden Abbildung 3.4 sind die Pixelmittelpunkte, die abgetastet werden, ohne sie einzufärben, mit einem roten Kreuz gekennzeichnet. Besonders unter diesen leidet die Performance unnötig, da häufig viele Dreiecke gezeichnet werden müssen und die Vielzahl an negativen Prüfungen Zeit in Anspruch nimmt.

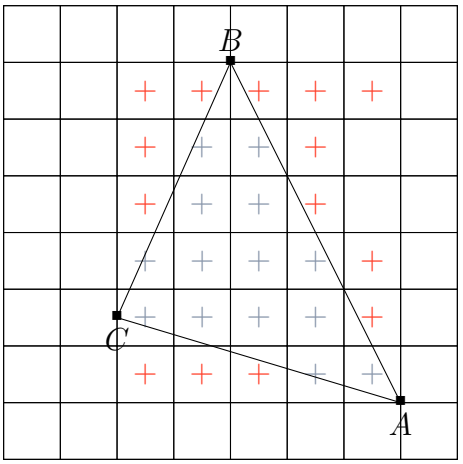


Abbildung 3.4.: Abtastpunkte der Rasterisierung mit umgebenden Viereck

Insgesamt werden 28 Pixel geprüft, sodass von allen die baryzentrischen Koordinaten berechnet werden müssen. Besonders auffällig ist die obere Reihe, die über die ganze Breite geprüft wird, da der Algorithmus innerhalb dieser nie einen Pixemittelpunkt im Dreiecks findet und dadurch nicht frühzeitig abbrechen kann, weil ihm nie bekannt ist, dass er die Fläche des Dreiecks schon überschritten hat. Ungefähr 50% aller Überprüfungen fallen negativ aus und resultieren nicht in einem Einfärbevorgang. Wünschenswert wäre ein Algorithmus, der keine negativen Prüfungen vorweisen kann, sodass nicht nur die inkrementelle Berechnung baryzentrischer Koordinaten, sondern allgemein im Quelltext die gesamte Überprüfung auf das gleiche Vorzeichen jener Koordinaten, die in der bisherigen Variante für jeden Pixel anfällt, eingespart werden kann. Durch die Kanten des Dreiecks kann jedoch mathematisch bestimmt werden, welche Pixel innerhalb und außerhalb liegen.

Weiterhin spart die Optimierung aus Abschnitt 3.3.1.2 insgesamt nur die Überprüfung von zwei Pixeln im Vergleich zum üblichen umgebenden Viereck ein. Für kleine

Dreiecke, wie sie bei fein detaillierten, dreidimensionalen Strukturen üblich sind, ist die Einsparung durch die Optimierung allgemein nur gering. Grund dafür ist, dass pro Zeile auch immer hinter dem Dreieck ein Pixel geprüft wird, der nicht im Dreieck liegt, um zu registrieren, dass die Rasterisierung dieser Zeile das Dreieck wieder verlassen hat. Die Optimierung merkt nicht, dass sie sich am letzten Pixel innerhalb des Dreiecks befindet, sondern erst, wenn sie sich am ersten Pixel außerhalb befindet. Dadurch endet der Algorithmus beim Durchlaufen einer Zeile häufig bei dem Pixel, der ohnehin auch in der Bounding Box der letzte Pixel gewesen wäre, wie es in Zeile drei und vier der Fall ist. Pro Zeile können also durch frühzeitiges Verlassen häufig gar keine oder wenige Pixel eingespart werden. Wüsste der Algorithmus, dass er sich am letzten Pixel innerhalb des Dreiecks befindet und nun die Zeile wechseln kann nicht erst dann, wenn er bereits am ersten außerhalb ist, wäre die dadurch erzielte Einsparung drei Mal so groß für das Dreieck in der obigen Abbildung. Abgesehen davon werden alle Pixel außerhalb des Dreiecks auf der Seite des Startes des Zeilendurchlaufes negativ geprüft und dies nimmt Rechenzeit in Anspruch. Worst-Case bezüglich dieser Optimierung ist ein symmetrisches Dreieck, da dann die Einsparung von links und rechts gleich ist und minimal wird.

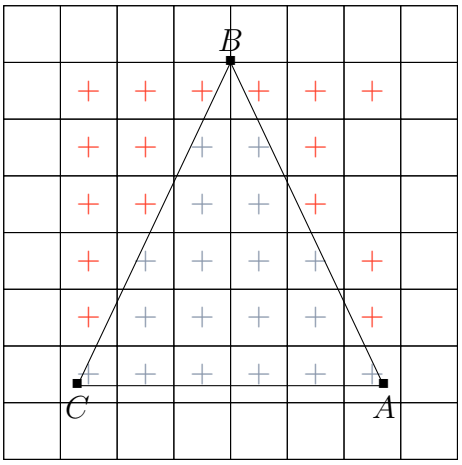


Abbildung 3.5.: Worst-Case für die Optimierung des Zeilendurchlaufes für beide Richtungen

Die Entscheidung, ob von links oder rechts rasterisiert werden soll, brachte für diverse Testbeispiele keinen Performancegewinn, sondern tendierte eher zum Verlust. Pro Dreieck ist die Berechnung der x-Koordinate des Schwerpunktes, bestehend aus der Mittelung der x-Koordinaten aller Eckpunkte, und die Abfrage der Differenz zu den beiden äußeren x-Werten nötig. Die damit verbundene Einsparung ist aber aus den oben genannten Gründen nur gering, vor allem, weil die Dreiecke sehr klein sind. Häufig ist sogar die Einsparung bei einer Rasterisierung der Zeilen von links oder rechts identisch oder weicht nur minimal ab, sodass die Berechnung unnötigen Zusatzaufwand darstellt, wodurch der Zusatzaufwand zur Ermittlung der Richtung des Durchlaufs mehr Zeit in Anspruch nimmt, als eingespart wird.

3.3.2.2. Ziel und Funktionsweise der „Window Search Rasterisierung“

Die Rasterisierung soll nun durch eine performantere ersetzt werden, deren grundlegende Idee aus [Hen+11] stammt. In diesem wird sie als „Window Search Bounding Box“ aufgeführt. Wie im vorausgegangenen Abschnitt beschrieben, sollen ausschließlich die Pixel innerhalb eines Dreiecks abgefragt werden. In Abbildung 3.5 entfallen also beispielsweise alle roten Kreuze.

Das Dreieck wird in dieser Implementierung, ähnlich wie bei der Implementierung mit dem Bresenham-Algorithmus aus [Rit19], von unten nach oben zeilenweise rasterisiert. Die Aufteilung des Dreiecks in zwei Hälften inklusive der Definition eines vierten Eckpunktes auf Höhe des mittleren Eckpunktes auf dessen gegenüberliegender Kante entfällt in dieser Implementierung, da es unnötigen Aufwand darstellt. Dies bildet sowohl einen Speicher als auch einen Geschwindigkeitsvorteil. Es soll eine Schleife jede Reihe von Pixeln angefangen von der Höhe des tiefsten Eckpunktes endend bei der Höhe des höchsten durchlaufen. In jeder Iteration werden für die aktuelle, ganzzahlige Pixelreihe die x-Koordinaten für die linke und die rechte Kante ermittelt, zwischen denen sich alle einzufärbenden Pixel befinden müssen. Anschließend iteriert eine Schleife angefangen vom kleineren endend beim größeren x-Wert auf dieser Höhe, um alle in dieser Zeile liegenden und zum Dreieck gehörigen Pixel einzufärben. Dabei werden die baryzentrischen Koordinaten für den Pixel ganz links über die Teilflächeninhalte ermittelt [Rit19], die restlichen Koordinaten in dieser Zeile ergeben sich inkrementell. Jede Zeile wird in diesem Algorithmus auf ihre Schnittpunkte mit dem zu rasterisierenden Dreieck untersucht.

Zur Realisierung müssen die drei Eckpunkte v_1, v_2 und v_3 zunächst der y-Koordinate nach aufsteigend sortiert werden, aber bleiben zusätzlich in der ursprünglichen Reihenfolge hinterlegt. Beim Backface Culling werden nämlich die dem Beobachter nicht zugewandten Seiten nicht gezeichnet; alle Koordinaten haben in dem Fall ein negatives Vorzeichen, was nur der Fall ist, wenn die Eckpunkte der Berechnung in der gegebenen Reihenfolge übergeben werden. Außerdem sollen gegen den Uhrzeigesinn definierte Dreiecke nicht gezeichnet werden, daher ist die ursprüngliche Reihenfolge wichtig.

Sind nun die Eckpunkte bezüglich ihrer y-Koordinate aufsteigend sortiert, ergibt sich die untere Grenze der Schleife über die Zeilen durch $y_{min} = \lceil v_{1_y} \rceil$ und die obere durch $y_{max} = \lfloor v_{3_y} \rfloor$. Auf den Höhen des unteren bzw. oberen Eckpunktes kann nur in der Reihe der Pixel darüber bzw. darunter ein Pixel existieren, der zu dem Dreieck gehört. Nun muss im Wesentlichen für jede ganzzahlige Höhe $y_i \in \{y_{min}, ..., y_{max}\}$ die Limitierung links und rechts auf dieser Höhe als Intervall $x_{min}(y_i)$ und $x_{max}(y_i)$ ermittelt werden, um für jede Zeile die einzufärbenden Pixel innerhalb dieses Intervalls zu ermitteln. Um den Algorithmus zu veranschaulichen, wird er anhand des Beispiels aus Abbildung 3.5 teilweise vorgeführt.

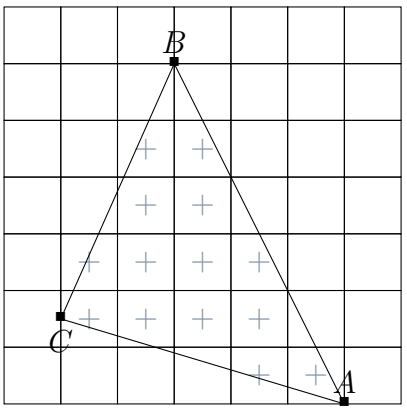


Abbildung 3.6.: Abtastpunkte der Window Search Rasterisierung

Auf den ersten Blick ist ersichtlich, dass nur die tatsächlich zum Dreieck gehörenden Pixel abgetastet werden, was der gewünschten Anforderung entspricht. Dadurch entsteht häufig eine enorme Einsparung, die größer ist, als der zusätzliche Overhead der im Zuge der Berechnung der Schnittpunkte der Kanten mit den Höhen fällig wird.

A hat in Abbildung 3.6 die Koordinaten (6,5|0,5), B(3,5|6,5) und C(1,5|2,1). Zur Ermittlung der begrenzenden x-Werte werden die beiden Kanten, die aus A herausgehen, als lineare Funktionen interpretiert. Eine verläuft bis zum obersten Eckpunkt B, die andere bis zum mittleren Eckpunkt C. Das Dreieck zeigt nach links, da C links von der Kante zwischen A und B liegt. Demnach bildet die Kante $\overrightarrow{AC}$ die linke Begrenzung für die x-Werte und die Kante $\overrightarrow{AB}$ die rechte. Bei einem nach rechts zeigenden Dreieck gilt umgekehrtes und in der folgenden Erklärung müssen x_{max} und x_{min} getauscht werden.

Die unterste Linie, die durchlaufen wird, besitzt die Höhe Eins. In der Schleife über die verschiedenen Höhen kann nun anhand der Information, ob sich die aktuelle Höhe unter oder über der y-Koordinate des mittleren Pixels befindet, festgelegt werden, ob für die linke Begrenzung die als Gerade interpretierte Kante $\overrightarrow{AC}$ oder $\overrightarrow{CB}$ verwendet werden muss. Die Steigung von $\overrightarrow{AC}$ bezüglich y ergibt sich durch $m_{ac} = \frac{A_x - C_x}{A_y - C_y}$, die von $\overrightarrow{AB}$ durch $m_{ab} = \frac{A_x - B_x}{A_y - B_y}$.

Für die linke begrenzende Kante unterhalb der Höhe des mittleren Eckpunktes C gilt nun allgemein: $x_{min}(y_i) = \lceil A_x + m_{ac} \cdot (y_i - A_y) \rceil$, da sie als lineare Funktion mit A_x als y-Achsenabschnitt interpretiert wird und von dort an ein Schritt der Länge $y_i - A_y$ gemacht wird, um die aktuelle Höhe zu erreichen. Die obere Gaußklammer kann hier verwendet werden, da dies der erste in Frage kommende x-Wert ist, der zu dem Dreieck gehört. Für den Fall $y_i = 1$ aus der Abbildung hat x_{min} den Wert 5 und entspricht dem ersten Pixel von links auf der Höhe 1, der eingefärbt werden muss. Die Formel für $x_{max}(y_i)$ ergibt sich analog durch ersetzen von m_{ac} durch m_{ab} und dem Tausch der oberen mit der unteren Gaußklammer. So gelangt der Algorithmus zu dem Ergebnis $x_{max} = 6$, welcher den letzten zum Dreieck gehörenden Pixel auf der Höhe 1 darstellt. Nach dem Einfärben aller Pixel zwischen x_{min} und x_{max} (inklusive inkre-

menteller Berechnung baryzentrischer Koordinaten und der daraus folgenden Farb- und Beleuchtungsberechnung) auf der Höhe wird mit der nächsten Zeile fortgefahren. In dieser arbeitet der Algorithmus analog, da die Höhe $y_i = 2 < C_y$ ist, weswegen die Kante $\overrightarrow{AC}$ die linke Limitierung bildet.

In der nächsten Zeile ändert sich lediglich die Vorschrift zur Bestimmung von x_{min}, weil die aktuelle Zeile der Rasterisierung dort oberhalb des mittleren Eckpunktes liegt. Deswegen wechselt m_{ac} zu m_{cb} (Steigung der Kante $\overrightarrow{CB}$) und A nun zu C, da der Ursprung der Gerade im Punkt C statt wie vorher A definiert wird. Die Ermittlung der linken Grenzen läuft nun analog mit der Formel für $x_{min}(y_i) = \lceil C_x + m_{cb} \cdot (y_i - C_y) \rceil$.

4. Parallelisierung

Die Laufzeit des Software-Renderers soll möglichst gering sein. Insbesondere für Animationen ist sie relevant, weil jene mit einer möglichst hohen Bildrate dargestellt werden müssen, um eine auf den Benutzer flüssig wirkende Bewegung zu erschaffen. Eine Laufzeit wie in der Variante mit OpenGL und Hardwarebeschleunigung ist nahezu unmöglich, da die Hardware einer Grafikkarte auf Visualisierungsprobleme spezifiziert ist, während die CPU allgemeiner für diverse Anwendungsgebiete der Datenverarbeitung gedacht ist. Dabei liegt der Vorteil der GPU-Hardware insbesondere in der Vielzahl von Kernen, die Transformationen und Visualisierungsoperationen, wie beispielsweise die Rasterisierung von Dreiecken, in hohem Maße parallel abfertigen können.

Nichtsdestotrotz hat die Mehrheit moderner CPUs mehrere Kerne, sodass die Rechnungen blockweise parallel vollzogen werden können. Offenbar muss die Parallelisierung schematisch anders implementiert werden, als bei der hardwarebeschleunigten Variante, da sie dort darauf ausgelegt ist, auf einem System mit sehr vielen Kernen zu arbeiten. Die Skalierung dieser Art der Parallelisierung auf CPUs ist im Vergleich zu GPUs suboptimal. Eine Reduzierung der Laufzeit soll daher durch andere Mechanismen und Konzepte paralleler Programmierung realisiert werden, die im folgenden erläutert und verglichen werden.

Wie in Abschnitt 2.3 beschrieben, werden pro Bild 0 bis n Dreiecksgitter gezeichnet. Da die Anzahl an Dreiecksgittern theoretisch beliebig variieren kann, bietet sich eine Parallelisierung dieser nicht an, weil die Anzahl verfügbarer Kerne konstant ist, die Anzahl an Dreiecksgittern jedoch häufig geringer ist als die Anzahl der Kerne. So werden im folgenden zwei mögliche Varianten für Aufteilungen vorgestellt und diskutiert.

4.1. Paralleles Zeichnen von Dreiecksgitterteilen

Die Grundidee der ersten Variante ist Anzahl der Dreiecke jedes Dreiecksgitters in gleich große Teile zu unterteilen und den Threads jeweils einen solchen Teil zuzuweisen, der von ihm abgearbeitet werden soll. Somit bildet die Variante eine Aufteilung hinsichtlich des darzustellenden Inhaltes.

Die Koordinaten der Eckpunkte eines Dreiecksgitters sind in einem Feld `Eckpunkte` hinterlegt, jeder Eintrag enthält einen Eckpunkt mit dessen vier Koordinaten, sowie dessen Farbe und Normale [Rit19]. Für ein Dreiecksgitter kann ein sogenannter `Indexpuffer` der Länge l_i hinterlegt sein, welcher aus einem Feld mit Indizes besteht, deren Elemente in Blöcke der Größe drei unterteilt werden können und als solche

zusammen ein Dreieck durch Referenzierung des Feldes der Eckpunkte bilden.

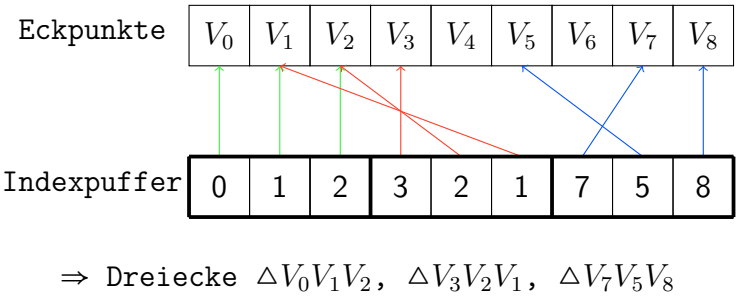


Abbildung 4.1.: Visualisierung eines Indexpuffers

Ist kein Indexpuffer definiert, werden die Eckpunkte in ihrer Reihenfolge durchlaufen. Dabei sind jeweils drei Einträge des Feldes `Eckpunkte`, das die Länge l_e besitzt, zusammen als ein Dreieck zu interpretieren. Abhängig davon, ob ein Indexpuffer existiert, sind demnach wenn er definiert ist $A_d = \frac{l_i}{3}$ und sonst $A_d = \frac{l_e}{3}$ Dreiecke zur Darstellung eines Dreiecksgitters von Nöten. Die Anzahl der Dreiecke eines Dreiecksgitters A_d kann gleichmäßig auf die Threads aufgeteilt werden, indem sie durch die Anzahl der Threads n_t geteilt werden. Jeder Thread übernimmt dann seinen Teil des Dreiecksgitters. Sie werden also nacheinander, aber jeweils in sich parallel gezeichnet.

4.1.1. Mutex-Lock des kritischen Bereiches

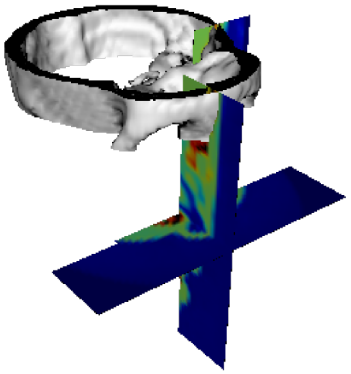
Während der parallelen Abarbeitung der Arbeitspakete, die aus einer Teilmenge aller Dreiecke eines Dreiecksgitters bestehen, greifen die Threads alle auf dieselben Farb- und Tiefenpuffer zu, falls ohne wesentliche Änderung die Parallelisierung in das vorhandene Programm eingefügt wird. Dies ist sowohl beim Tiefen- als auch beim Farbpuffer der Fall. Über die Positionierung der Dreiecke im Zusammenhang mit ihrer Threadzugehörigkeit ist nichts bekannt, sie können sich an beliebigen Stellen über die gesamten Farbpuffer verteilt befinden. Besonders relevant ist dabei der Fakt, dass mehrere Threads unter Umständen auf den exakt selben Speicherbereich zugreifen, falls zwei der Threads jeweils ein Dreieck rasterisieren, welches zur Einfärbung desselben Pixels führt. Die Speicherbereiche der Farbpuffer und des Tiefenpuffers müssen also beim Eintritt eines Threads für alle anderen gesperrt werden, sonst könnte es zu einer Race Conditon kommen.

Beispielsweise fragt ein Thread T_1 den Tiefenpuffer an einer Stelle (x, y) ab und realisiert, dass das Dreieck, welches gerade durch ihn rasterisiert wird, näher am Beobachter ist ($z_1 = 0{,}5$), als das bisher an dieser Stelle hinterlegte ($z_{x,y} = 0{,}6$). Somit muss ein Einfärbevorgang stattfinden. Bevor der Pixel eingefärbt wird, setzt ein anderer Thread T_2 ein, der ebenfalls an der Stelle (x, y) eine Einfärbung vornehmen will. Dieses Dreieck besitzt dort den Tiefenwert $z_2 = 0{,}4$ und ist somit das zum Betrachter nächstgelegene, deswegen wird es auch eingefärbt. Nun setzt die Abarbeitung des Threads T_1 erneut ein; dieser hat allerdings bereits die Abfrage des Tiefenpuffers

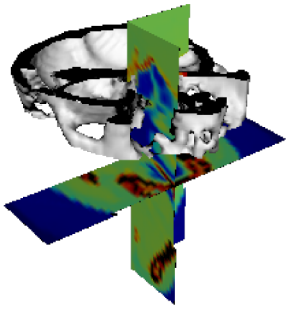
überwunden und überschreibt aus diesem Grund den von Thread T_2 hinterlegten Farbwert, welcher eigentlich der für diesen Pixel korrekte gewesen wäre. In diesem und ähnlichen Fällen kann die Korrektheit der Darstellung nicht mehr gewährleistet werden.

Die erste intuitive Idee ist es, den Bereich der Abfrage des Tiefenpuffers und den Einfärbevorgang durch ein Mutex-Lock zu schützen, sodass nur ein Thread gleichzeitig dort eintreten darf. Jedoch ist in diesem Fall nicht nur der Overhead durch das allzuhäufige Sperren und Entsperren des Mutex-Locks sehr hoch, sondern die Parallelisierung an sich durch das ständige Warten gestört, sodass es in der tatsächlichen Ausführung der sequentiellen Abarbeitung nahe kommt. In diversen Beispielen war die Laufzeit dieser Implementierung deutlich höher als die der Ursprünglichen ohne jegliche Parallelität.

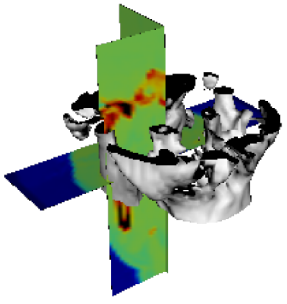
4.1.2. Mehrere Farb- und Tiefenpuffer mit anschließendem Mischen



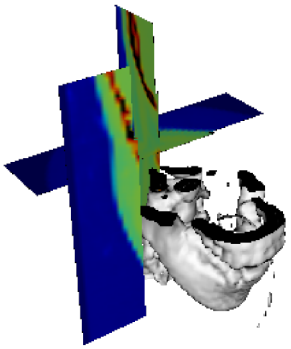
(a) Dreiecksgitterteile im Farbpuffer des ersten Threads



(b) Dreiecksgitterteile im Farbpuffer des zweiten Threads



(c) Dreiecksgitterteile im Farbpuffer des dritten Threads



(d) Dreiecksgitterteile im Farbpuffer des vierten Threads

Abbildung 4.2.: Darstellung der Dreiecksgitterteile in den Farbpuffern der Threads

Besonders das Warten auf das Mutex-Lock bremst die Ausführungsgeschwindigkeit enorm. Praktisch wäre es also, wenn jeder Thread ohne Mutex seinen Teilblock an Dreiecken (d.h. $D_i = \frac{A_d}{n_t}$) zeichnen kann und dabei keine Gefahr von Race Conditions besteht. Aus diesem Grund werden so viele Farbpuffer und Tiefenpuffer angelegt, wie Threads vorhanden sind. Somit hat jeder Thread seinen eigenen Speicherbereich, in den sein Teil eines Gitters gezeichnet werden kann. Sind mehrere Dreiecksgitter pro Bild vorhanden, zeichnet jeder Thread seinen Anteil an jedem dieser in den zu ihm gehörigen Farbpuffer und hinterlegt dementsprechend die Tiefenwerte in seinem Tiefenpuffer. Ergebnis dieses Vorgangs sind mehrere Farb- und Tiefenpuffer, deren Anzahl jeweils der Anzahl an verwendeten Threads entspricht. In diesen Farbpuffern ist pro Dreiecksgitter ein Anteil hinterlegt.

Diese Farbpuffer müssen nun mit Hilfe der vorhandenen Tiefenpuffer zusammengefügt werden, sodass sie im Verbund das ursprünglich gewünschte Bild ergeben. Dazu wird über jeden Pixel des Farbpuffers iteriert und das Minimum über alle an dieser Stelle in den verschiedenen Tiefenpuffern hinterlegten Tiefenwerte gesucht. Der zu dem Tiefenwert im entsprechenden Farbpuffer hinterlegte Wert bildet dann den Farbwert des Ergebnisses an dieser Stelle. Bildlich gesehen werden die Farbpuffer gemischt, indem an jeder Stelle die Tiefenwerte aufsteigend sortiert werden und die Farbpuffer dort analog angeordnet werden. Das Endergebnis besteht dann aus dem Zusammenschluss aller Pixel, die sich am Ende ganz vorne befinden.

Diese Operation kann auf simple Art und Weise parallelisiert werden. Das Bild wird in so viele Streifen unterteilt wie Threads verfügbar sind. Jeder Thread kümmert sich dann um einen Teilbereich, sodass niemals von zwei Threads auf dieselbe Adresse zugegriffen werden kann und demzufolge auch keine Mutex-Locks oder ähnliche Mechanismen von Nöten sind.

Diese Variante stellte sich als effizienter als die sequentielle Abarbeitung heraus, obwohl zusätzlich die Operation des Zusammenfügens anfällt. Der Speicherbedarf ist enorm erhöht, da der von den Farb- und Tiefenpuffern benötigte Speicherplatz sich um den Faktor n_t vervielfacht.

4.1.3. Teile von Dreiecksgittern in Queues

4.1.3.1. Motivation

Die Implementierung auf die beschriebene Art und Weise impliziert diverse Nachteile. Zum Beispiel müssen für jedes Dreiecksgitter die Threads neu erstellt und ihnen die Aufgaben zugewiesen werden. Die Erstellung eines Threads ist zeitintensiv. Besonders, wenn nur kleine Dreiecksgitter vorhanden sind lohnt sich die mehrfache Initalisierung der Threads und die Aufteilung der Aufgaben kaum, da der Overhead zur Erstellung zu groß ist. Allgemein führen viele Dreiecksgitter zur häufigen wiederholten Erstellung von Threads, welche aber prinzipiell immer nur die gleiche Aufgabe verrichten, die im Zeichnen eines Teils des Dreiecksgitters besteht. Der Aufruf der Methode `pthread_join()` zum Warten auf Fertigstellung der Aufgaben der Threads wäre sinnvoller vor Beendigung des gesamten Programmes und nicht nach jedem Drei-

ecksgitter, da erst ganz am Ende des Programmdurchlaufs den Threads mit Sicherheit keine neue Aufgaben mehr zugeteilt werden müssen. Praktischer und zeitsparender wäre also die einmalige Erstellung von Threads beim Start des Programmes, die dann immer wieder neue Arbeitspakete verschiedener Dreiecksgitter und Bilder zum Zeichnen übergeben bekommen und sich um die Darstellung jener in dem dazugehörigen Farbpuffer kümmern. Sie sollen immer zur Verfügung stehen und nach der Beendigung der Abarbeitung eines zu einem Dreiecksgitter gehörigen Teilstückes wieder bereit sein, den nächsten Job entgegen nehmen zu können.

Weiterhin muss die Abarbeitung nach dem Zeichnen jedes Dreiecksgitters beziehungsweise dem Zeichnen von dessen Teilen auf die Fertigstellung von dessen langsamsten Teilstückes warten, bevor das Zeichnen des nächsten oder, im Falle des letzten Dreiecksgitters, der Mischvorgang beginnen kann. Dieser Nachteil ist jedoch eher weniger relevant, da sich die Größe der Arbeitspakete auf Grund der gleichmäßigen Aufteilung pro Dreiecksgitter maximal um ein Dreieck unterscheiden.

Außerdem werden in der Implementierung nach dem beschriebenen Schema neue Threads erstellt und gestartet, um das Mischen der Farbpuffer zu realisieren. Hier treten die gleichen Nachteile auf, wie eingangs in diesem Abschnitt beschrieben. Ziel ist es, das Zeichnen der Dreiecksgitterteile und deren Mischvorgang für alle Bilder in denselben Threads zu erledigen, die lediglich ein Mal pro Darstellungsvorgang erzeugt werden und vor Terminierung des GR3 wieder gelöscht werden.

4.1.3.2. Implementierung

Damit die Threads nur ein Mal pro Szene erstellt werden müssen, wird eine Queue implementiert, die alle Jobs beinhaltet die die einmalig initialisierten Threads dann bei Verfügbarkeit entgegennehmen. Dieses Prinzip ähnelt dem Producer-Consumer-Problem. Die Grundidee ist dabei die folgende: Pro Thread wird eine Queue erstellt, in der jedes Element die Informationen über den zu diesem Thread gehörigen Teil eines Dreiecksgitters enthält.

In einer Szene werden jeder der Queues also so viele Elemente hinzugefügt, wie Dreiecksgitter vorhanden sind, nur, dass jeder Eintrag nicht das gesamte Dreiecksgitter beinhaltet, sondern lediglich einen Teil. Die Aufteilung jedes Dreiecksgitters in seine Teile übernimmt der Main-Thread während der Iteration, der sogleich die Informationen in die Queue einfügt und damit die Verarbeitung anstößt. Jeder Thread entnimmt dann seiner Queue die Elemente und zeichnet die daraus resultierenden Dreiecksgitterteile in seinen Farbpuffer. Hat jeder Thread die Abarbeitung aller seiner Jobs erledigt, beginnt der Mischvorgang, der ebenfalls parallel stattfindet. Jeder Thread bekommt bei seiner Erstellung übergeben, welchen Teil des Farbpuffers er durchlaufen beziehungsweise mit den anderen vermischen soll. Ist auch der Mischvorgang beendet wird das Bild dargestellt. Die Threads sind dann wieder in der Lage, die Teile der Dreiecksgitter eines möglichen neuen Bildes entgegenzunehmen. Eine Übersicht des gesamten Ablaufs ist in der Abbildung 4.3 visualisiert.

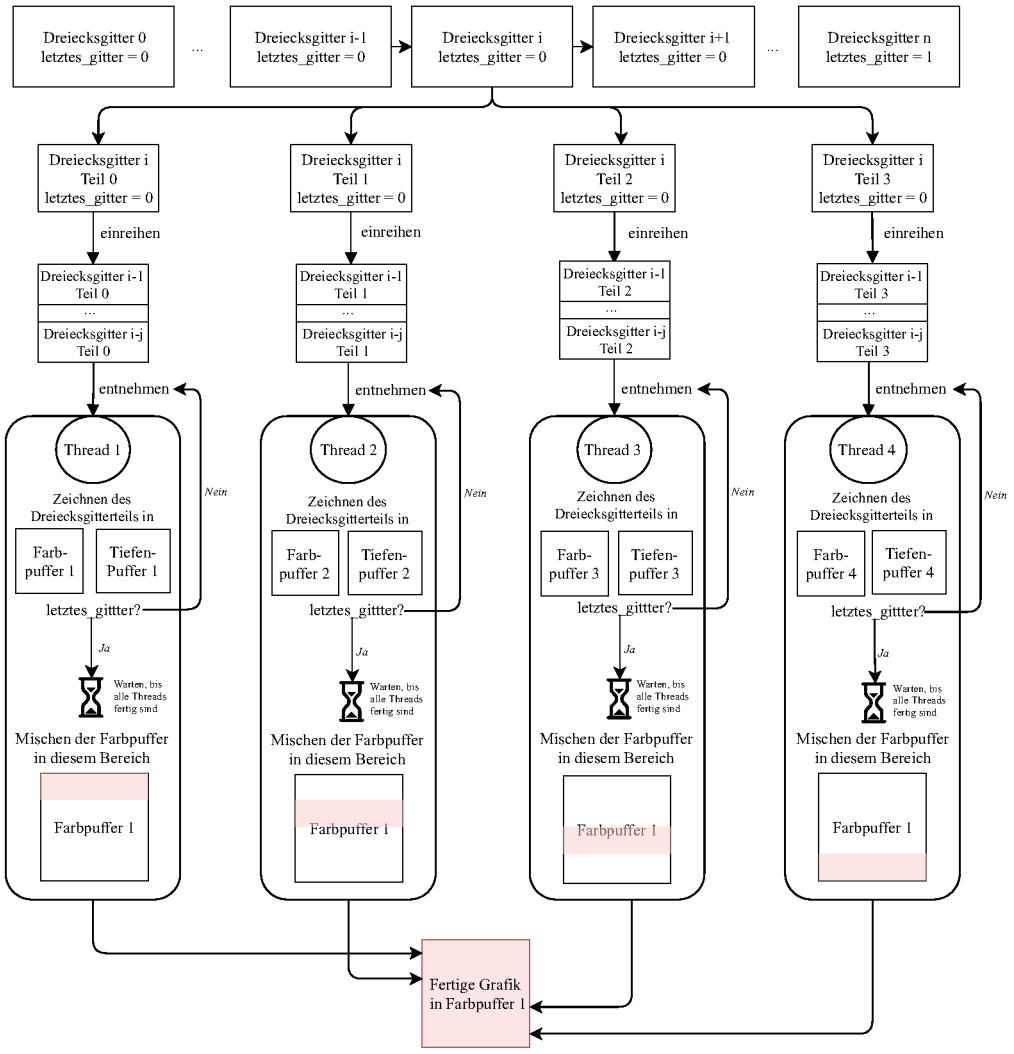


Abbildung 4.3.: Grober Ablauf der ersten Variante des parallelen Renderns für $n_t = 4$ Threads

Beim Start des Programmes werden die Threads mit `pthread_create()` initialisiert. Dabei wird ihnen ein horizontaler Streifen, der einen Anteil von $\frac{1}{n_t}$ an der Gesamthöhe hat, in Form von Starthöhe und Endhöhe zugeordnet. Dieser dient später als Basis des parallelen Mischvorganges. Ebenso wird für jeden Thread eine Queue erstellt, in die die Dreiecksgitterteile eingereiht werden. Der Methode `pthread_create()` wird als Parameter im Wesentlichen die folgende Methode übergeben, die die Jobs entgegennimmt.

```
1  while (teildreiecksgitter_info = queue_dequeue(queue)){
2    zeichne_teildreiecksgitter(teildreiecksgitter_info);
3    if (teildreiecksgitter_info.letztes_gitter){
4    /* Letzte Teildreiecksgitter dieses Threads ? */
5      threads_fertig += 1;
6      if (threads_fertig == ANZAHL_THREADS){
7      /* Alle Threads fertig mit Zeichnen? */
8        pthread_cond_broadcast(&warte_auf_mischen);
9      }else{
10       pthread_cond_wait(&warte_auf_mischen, &lock);
11     }
12     mische_puffer(thread.start_y, thread.end_y);
13     /* Mischen des Bildauschnittes */
14     threads_fertig += 1;
15     if (threads_fertig == 2 * ANZAHL_THREADS){
16     /* Alle Threads fertig mit Mischen */
17       pthread_cond_signal(&warte_nach_mischen);
18     }
19   }
20 }
```

Abbildung 4.4.: Vereinfachte Abarbeitung für jeden Thread (zur Übersichtlichkeit sind Mutex-Locks nicht aufgeführt)

Jeder Thread blockiert dabei in der ersten Zeile, falls die Queue keine Elemente enthält. Es wird in der Methode `queue_dequeue()` auf ein Condition-Signal gewartet, wenn die Queue leer ist. Während also n_t Threads auf die Entgegennahme von Arbeitspaketen warten, füllt der Main-Thread den Farbpuffer mit Hintergrundfarbe. Danach iteriert er über alle Dreiecksgitter und teilt pro Iterationsschritt das aktuelle in gleich große Teile ein, damit sie den verschiedenen Queues hinzugefügt werden können. Für jedes Dreiecksgitter werden dann einmal im Main-Thread die Transformationsmatrizen sowohl für die Koordinaten selbst, als auch für die Normalen initialisiert und miteinander multipliziert, um diesen Aufwand nicht in jedem Thread durchführen zu müssen. Dann wird einer von n_t gleich großen Teilen des Indexpuffers oder der gegebenen Eckpunkte in die jeweils zu einem Thread gehörige Queue eingereiht.

```
int letztes_gitter;
while(dreiecksgitter){
  int anteil;
  letztes_gitter = dreiecksgitter->next == NULL;
  if (anzahl_indizes != 0){
  /* Indexbuffer definiert */
    anteil = anzahl_indizes/ANZAHL_THREADS;
  }else{
  /* Kein Indexbuffer definiert */
    anteil = anzahl_eckpunkte/ANZAHL_THREADS;
  }
  for (thread_idx = 0; thread_idx < NUM_THREADS; thread_idx++){
    queue_enqueue(  queues[thread_idx], malloc_arg(dreiecksgitter,
        matrizen, letztes_gitter, thread_idx*anteil,
        (thread_idx+1)*anteil));
  }
  dreiecksgitter = dreiecksgitter->next;
}
pthread_cond_wait(&warte_nach_mischen, &lock);
```

Abbildung 4.5.: Vereinfachte Einreihung der Teilaufgaben in die einzelnen zu den Thread gehörigen Queues

Übergeben wird das zu zeichnende Dreiecksgitter, die für die Transformation notwendigen Matrizen und der Start- und der Endindex im Indexpuffer bzw. Eckpunktfeld des zu zeichnenden Anteils. Der Wert *letztes_gitter* gibt Auskunft darüber, ob es sich um das letzte Dreiecksgitter eines Bildes handelt. In diesem Fall kann nämlich nach der Beendigung aller Jobs mit *letztes_gitter* = 1 der Prozess des Mischens der Farbpuffer beginnen. Nach der Iteration über die Dreiecksgitter wartet der Main-Thread mit der tatsächlichen Darstellung des Bildes auf ein Signal der anderen Threads, die ihm mitteilen, wenn sie sowohl das Zeichnen als auch das Mischen erledigt haben.

Beim Kommando `queue_enqueue()` wird ein Condition-Signal an den zu der Queue gehörigen Thread gesendet, der danach aufwacht und die `queue_dequeue()` Funktion fortsetzt. Damit beginnt die Abarbeitung aus Abbildung 4.4, die das Zeichnen dieses Dreiecksgitterteils beinhaltet. Ist ein Thread mit dem Zeichnen fertig, bevor der nächste Job eingereiht wird, wartet er auf ein Condition Signal, das heißt auf das nächste Einreihen eines Jobs in der Bedingung der `while`-Schleife.

Ist ein Job abgearbeitet, der zum letzten Dreiecksgitter eines Bildes gehört, wird eine Zählvariable *threads_fertig* erhöht. Ein Thread, der einen Job aus der Queue entnimmt und realisiert, dass dies der letzte zu dem Bild gehörige ist, kann warten, falls die anderen Threads noch nicht fertig sind, also falls $threads\_fertig < n_t$. Gleicht der Wert dieser Variable allerdings der Anzahl an vorhandenen Threads, ha-

ben alle Threads ihre Dreiecksgitterteile gezeichnet und der Vorgang des Mischens kann beginnen. Ein Flaschenhals ist hier unwahrscheinlich, da jeder Teil eines Dreiecksgitters gleich groß ist und damit jeder Thread einen gleich großen Anteil der Arbeit erledigt. Der Thread, der als letztes mit seiner Abarbeitung fertig wird, signalisiert dies allen anderen bereits fertigen und auf ihn wartenden Threads mit dem Befehl `pthread_cond_broadcast()`. Dieser Befehl weckt alle mit eine spezifizierte Condition-Variable wartenden Threads auf, was in dem Fall allen Threads bis auf den Main-Thread entspricht. Jeder der Threads mischt dann seinen festgelegten horizontalen Streifen des Bildes und sucht für jeden darin enthaltenen Pixel den zum Betrachter am nächsten befindlichen, um dessen Farbwert in dem am Ende dargestellten Farbpuffer zu hinterlegen. Dabei kann es auf Grund der Disjunktheit der Streifen nicht zu Race-Conditions kommen. Jeder Thread, der seinen Teil fertig gemischt hat, erhöht die Variable $threads\_fertig$ erneut. Sind alle fertig mit dem Mischvorgang, hat die Variable den Wert $2n_t$. In dem Fall erhält der Main-Thread, der nach der Einreihung aller Jobs auf die Fertigstellung der anderen Threads gewartet hat, ein Signal. Dieser weiß nun, dass das gesamte Bild gezeichnet ist und der parallele Teil der Abarbeitung ist beendet. Somit kann das Bild entweder einer weiteren Verarbeitung unterzogen werden (z.B. Kantenglättung) oder dargestellt beziehungsweise exportiert werden.

4.2. Zeilenweise Verteilung der Arbeit auf die Threads

Der zweite Ansatz verfolgt die Idee, nicht den darzustellenden Inhalt, sondern den darzustellenden Bereich gleichmäßig auf die Threads aufzuteilen. Dadurch müssen nicht mehrere Farb- und Tiefenpuffer angelegt werden, da jeder Thread innerhalb der Puffer seinen eigenen, disjunkten Bereich hat, auf den niemals ein anderer Thread lesend oder schreibend zugreifen wird. Im Vergleich zur Implementierung aus Abschnitt 4.1.3.2 müssen daher nicht mehrfach große Speicherbereiche wie der Farb- und Tiefenpuffer allokiert werden. Dementsprechend entfällt auch der Schritt des Mischens, da alle Threads auf denselben Puffern arbeiten und innerhalb dieser ausschließlich auf die ihnen zugeteilten Bereiche zugreifen. Insgesamt setzt sich daraus dann das vollständige Bild im Farbpuffer zusammen.

Unter der Prämisse, dass alle Threads einen möglichst gleichen Teil der Rasterisierung übernehmen sollen, bietet es sich nicht an, die Puffer in n_t gleich große horizontale oder vertikale Streifen zu unterteilen, weil die Bildinformation häufig zusammenhängend in der Mitte der Szene konzentriert ist. Deswegen besteht beim Einteilen in gleich große Streifen zusammenhängende die Gefahr, dass die Bildinformation ungleich auf die Threads verteilt wird und zum Beispiel die Threads, die sich um den obersten und untersten Streifen kümmern deutlich weniger Rasterisierungsarbeit erledigen, als die für das Zentrum verantwortlichen.

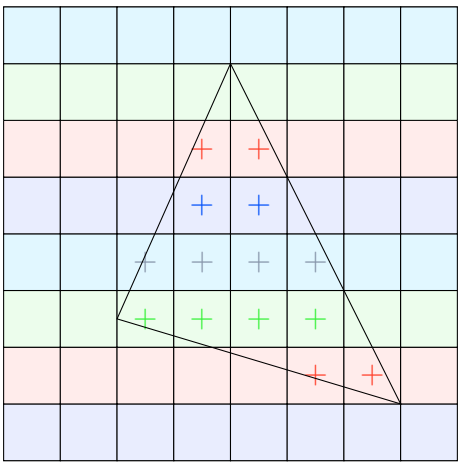


Abbildung 4.6.: Einteilung der Puffer auf die verschiedenen Threads

Sinnvoller hingegen ist es, jedem Thread t_i den $t_i + n_t$-ten Streifen des Farb- und Tiefenpuffers zuzuweisen. Der Thread t_i kümmert sich also um alle Zeilen z aus dem Farb und Tiefenpuffer, bei denen $z \equiv t_i \pmod{n_t}$ gilt. Der erste Thread übernimmt beispielsweise die Zeile $0, n_t, 2n_t, ...$, der zweite $1, 1 + n_t, 1 + 2n_t, ...$ Diese Aufteilung erhöht die Wahrscheinlichkeit einer gleichmäßigen Aufteilung der zu rasterisierenden Bildinformation, da ein Dreieck eines Gitters, dessen Höhe n_t überschreitet, unabhängig von seiner Positionierung gleichmäßig auf die Threads aufgeteilt wird. In der obigen Abbildung 4.6 hat jeder der $n_t = 4$ Threads eine eigene Farbe und es ist eingezeichnet, für die Rasterisierung welcher Zeilen er verantwortlich ist. Eine schematische Visualisierung des Gesamtablaufs ist in Abbildung 4.7 dargestellt. Jene Pixel, die eingefärbt werden, sind mit einem Kreuz markiert.

Bei der Rasterisierung wird zunächst innerhalb des Main-Threads über die Gitter und schließlich innerhalb jedes Threads über die Dreiecke iteriert, indem sie der Queue entnommen werden. Liegt ein Dreieck vollständig in einem Bereich, das keine Zeile beinhaltet, für die der aktuelle Thread verantwortlich ist, wird es übersprungen und mit dem nächsten fortgefahren. Um zu ermitteln ob dies der Fall ist, sind jedoch Berechnungen von Nöten, die bei der Rasterisierung ohnehin anfallen würden. Die dadurch erzielte Einsparung ist dementsprechend überschaubar. Die Motivation für die Queue ist ähnlich wie in Abschnitt 4.1.3.1.

Obwohl dieser Ansatz den Vorteil hat, weniger Speicherplatz beanspruchen zu müssen, ist er in der Praxis deutlich langsamer, als die in Abschnitt 4.1 beschriebene Alternative. Ursache dafür ist unter anderem, dass jeder Thread alle Dreiecke eines Bildes durchlaufen muss, unabhängig davon, ob ein Dreieck eine für den Thread relevante Zeile enthält. Innerhalb jedes Threads t_i wird deswegen geprüft, ob eine Zeile z innerhalb des Dreiecks liegt, bei der $z \equiv t_i \pmod{n_t}$. Ist diese Prüfung negativ, wie es bei vielen kleinen Dreiecken häufig der Fall ist, war die gesamte Rechnung für den Thread irrelevant und es wird mit dem nächsten Dreieck fortgefahren. Die beschriebene Prüfung ist jedoch zeitintensiv und muss für ausnahmslos jedes Dreieck in jedem Thread geschehen.

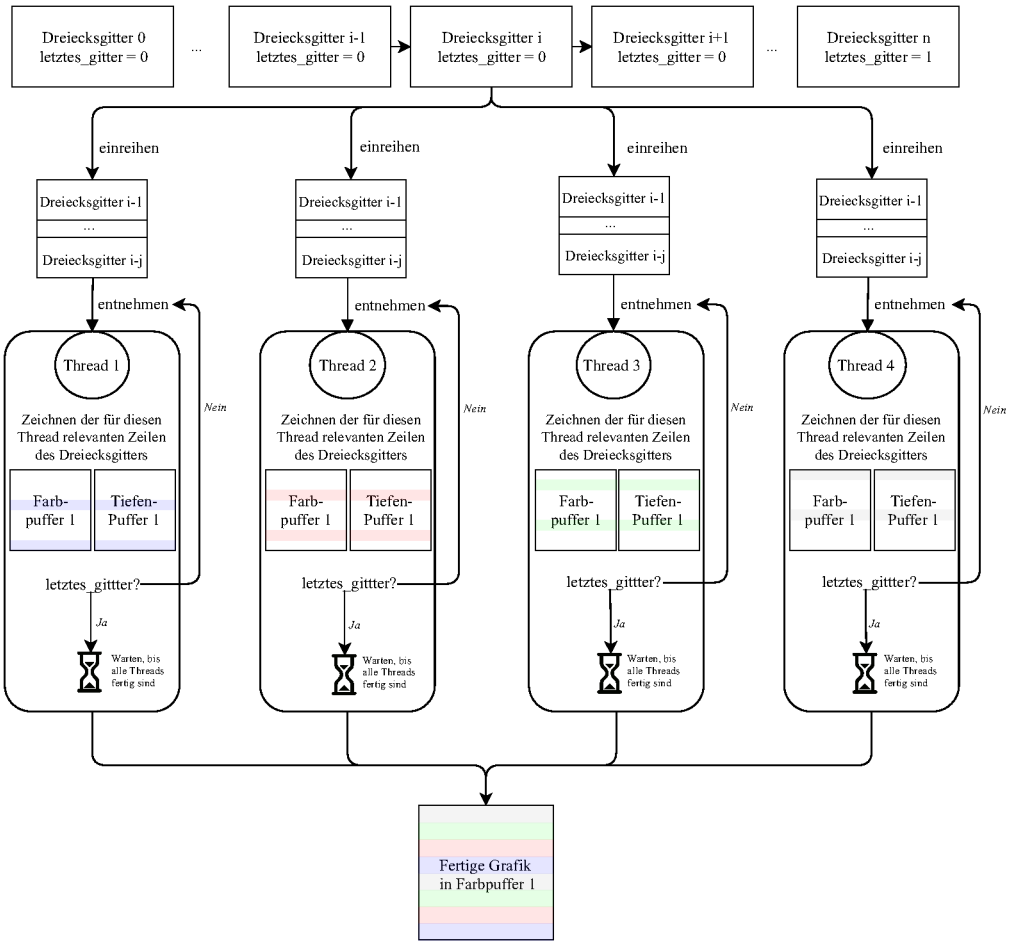


Abbildung 4.7.: Grober Ablauf der zweiten Variante des parallelen Renderns für $n_t = 4$ Threads

Andernfalls ist bei positiver Prüfung der Overhead zur Ermittlung der Schnittmenge der durch den Thread zu rasterisierenden Zeilen und der zum Dreieck gehörigen Zeilen höher als das einfache Durchlaufen aller Dreieckszeilen, besonders bei kleinen Dreiecken, die häufig in präzisen Darstellungen auftreten.

Des Weiteren ist der Rasterisierungsalgorithmus darauf ausgelegt, durch die Initialisierung der zur Rasterisierung benötigten Werte anschließend durch wenige Operationen ein zeilenweises Durchlaufen des Dreiecks von unten nach oben zu ermöglichen. Durch das Überspringen von für den Thread nicht relevanten Zeilen wird neben anderem der Vorteil, der durch die Implementierung der inkrementellen Berechnung baryzentrischer Koordinaten entsteht, nicht optimal ausgenutzt, da in jedem Thread eine initiale Berechnung stattfinden muss. Ähnliches gilt für die Steigung der Dreieckskanten, die nur einmal pro Dreieck berechnet wird. Zusammengefasst bedeutet dies, dass das Rasterisieren jeder n_t-ten Zeile eines Dreiecks bei weitem nicht nur $\frac{1}{n_t}$ der Zeit benötigt.

In seiner Struktur ist diese Variante simpler und zudem benötigt sie weniger Speicherplatz, aber auf Grund der beschriebenen Nachteile und ihres Einflusses auf die Ausführungszeit wurde die Implementierung verworfen. Stattdessen wird die in Abschnitt 4.1 beschrieben wurde übernommen, auf die sich im folgenden Abschnitt bezogen wird.

4.3. Weitere Optimierungen

4.3.1. Nutzen allokierter Teile des letzten Bildes

Allokationen sind relativ aufwändig. Jedes Bild setzt sich nach dem Zeichnen der Dreiecksgitterteile aus n_t Farbpuffern und den dazugehörigen n_t Tiefenpuffern zusammen. Um nicht in jedem Bild die genannten Speicherbereiche allokieren zu müssen, werden nach dem Zeichnen eines Bildes die Speicherbereiche nicht freigegeben, sondern sind für alle weiteren Bilder verfügbar. Hat sich dann im nächsten Bild die Höhe und Breite des Bildes nicht verändert, wie es bei Animationen zum Beispiel oft der Fall ist, können die Farb- und Tiefenpuffer aus dem Bild davor erneut verwendet werden. Andernfalls müssen sie reallokiert werden, was in der Praxis jedoch häufig nicht der Fall ist. In der Implementierung werden dazu Variablen hinterlegt, die die Höhe und die Breite des letzten Bildes speichern, um beim nächsten Bild die Gleichheit überprüfen zu können. Das Prinzip der Wiederverwendung über die Bilder hinweg findet sich auch bei den Threads wieder, wie in Abschnitt 4.1.3.1 beschrieben.

4.3.2. Füllen der Farb- und Tiefenpuffer

Die Methode `gr3_getpixmap_(pixmap)` bekommt einen Farbpuffer (*engl.* Pixmap) übergeben, in den letztendlich das fertige Bild eingefügt soll. Vom Software-Renderer selbst allokiert werden müssen also nur die restlichen $n_t - 1$ Farb- und n_t Tiefenpuffer. Der übergebene Farbpuffer wird initial mit der Hintergrundfarbe des Bildes gefüllt,

die vom Nutzer spezifiziert werden kann. Der dazugehörige Tiefenpuffer wird mit dem Wert Eins zurückgesetzt, da tiefere Elemente nicht gezeichnet, sondern abgeschnitten werden. Das in diesen beiden Fällen zum Füllen verwendete Durchlaufen der Farbpuffer und des Tiefenpuffers ist zeitintensiv. Aus diesem Grund werden alle restlichen Farbpuffer durch `malloc()` allokiert, anstatt auch die Hintergrundfarbe zugewiesen zu bekommen. Alle Tiefenpuffer, bis auf den zum letztendlich dargestellen Farbpuffer gehörigen, werden wegen der guten Performance mit einem `memset()` initialisiert. Dieses setzt eine spezifizierte Anzahl von Bytes beginnend bei einer Startadresse auf einen Wert, sodass später, bei der Interpretation von vier aufeinanderfolgenden Bytes eine Fließkommazahl resultiert, die größer als Eins ist. Alle Pixel, die aufgrund eines zu zeichnenden Teildreiecksgitters in den Farbpuffern eingefärbt werden, bekommen dann ohnehin den korrekten Tiefenwert in Zuge der Rasterisierung zugewiesen.

Nach einem Bild muss nur der Farbpuffer, der am Ende dargestellt wird, noch einmal mit der Hintergrundfarbe gefüllt und jeder Wert des dazugehörigen Tiefenpuffers auf den Wert Eins gesetzt werden. Die anderen Tiefenpuffer werden dann wie oben beschrieben durch die performante Operation `memset()` auf einen Tiefenwert größer als Eins gesetzt. Der Inhalt aller Farbpuffer bis auf den ersten ist zu diesem Zeitpunkt irrelevant, da die darin enthaltene Information in Kombination mit den Tiefenpuffern ohnehin in einem Mischvorgang nie verwendet wird. Die Tiefenpuffer geben nämlich Aufschluss darüber, welche der Daten aus den dazugehörigen Farbpuffern sich hinter dem Hintergrund aus dem ersten Farbpuffer befinden. Deswegen müssen die Farbpuffer nicht zurückgesetzt werden, wodurch Zeit eingespart wird. Beim Rasterisierungsvorgang werden Elemente, die tatsächlich vor dem Hintergrund sind, auch mit einem passenden Tiefenwert in den anderen Farbpuffern hinterlegt, sodass dies beim Mischen keine Komplikationen verursacht. Lediglich die initale Füllung ist nicht von Bedeutung, da wegen der Tiefe ohnehin die korrekte, im Zielfarbpuffer befindliche, Hintergrundfarbe präferiert wird. Wichtig sind also dann nur die von der Rasterisierung veränderten Pixel, welche auch einen passenden Tiefenwert besitzen.

4.3.3. Arbeit mit und ohne Indexpuffer

Wie in Abschnitt 4.1 erwähnt, können die Dreiecke entweder über einen Indexpuffer definiert sein, oder in die Eckpunkte in der Reihenfolge angegeben sein, in der sie die Dreiecke bilden.

Im ersten Fall lohnt es sich, vor Beginn der Abarbeitung zunächst alle Eckpunkte inklusive ihrer Normalenvektoren vom Main-Thread transformieren zu lassen, ohne den Indexpuffer dabei zu berücksichtigen. Der Indexpuffer greift dann im weiteren Verlauf des Programmes auf die bereits transformierten Eckpunkte zu. Zwar muss dafür ein Feld mit Eckpunkten allokiert werden, jedoch erwies sich diese Variante in Tests als effizienter. Alternativ könnten zwar die Eckpunkte eines Dreiecks immer unmittelbar vor dessen Zeichenvorgang transformiert werden, allerdings würde in diesem Fall derselbe Eckpunkt mehrfach transformiert werden, wenn der Indexpuffer ihn in mehreren Dreiecken referenziert.

Im Falle ohne Verwendung eines Indexpuffers können die Transformationen unmit-

telbar vor dem Zeichenvorgang des Dreiecks geschehen. Damit entfällt die zeitaufwändige Allokation des Speichers und die Eckpunkte können in transformierter Form in einem Feld auf dem Stack hinterlegt werden. Des Weiteren sind dadurch die Transformationen ebenfalls parallelisiert. Hier ist das Programm ohnehin nicht in Kenntnis darüber, ob der gleiche Eckpunkt mehrfach transformiert wird, oder nicht, da die Informationen im ursprünglichen Feld der Eckpunkte redundant hinterlegt wären.

5. Zusammenfassung und Ergebnisse

5.1. Zusammenfassung

Der Software-Renderer aus [Rit19] wurde in seiner Funktionalität erweitert, sodass nun alle von GR3 erzeugten Dreiecksgitter gezeichnet werden können, was vorher nur mit Hilfe einer OpenGL-Implementierung möglich war. Dadurch können alle Szenen des GR3 automatisch auch gänzlich ohne Hardwarebeschleunigung dargestellt werden, falls diese nicht verfügbar ist. Um den gesamten Funktionsumfang abzudecken mussten `SSAA` zur Kantenglättung und zusätzliche Beleuchtungsberechnungen implementiert werden.

Zur Optimierung des Programmes wurde zunächst die Anzahl der benötigten Berechnungen minimiert. Dazu gehört hauptsächlich die Implementierung der „Window Search Rasterisierung“, ein alternativer, schnellerer Algorithmus zur Rasterisierung von Dreiecken, der die ursprüngliche Implementierung der „Rasterisierung mit umgebenden Rechteck“ vollständig ersetzt. In dem neuen Algorithmus werden zudem baryzentrische Koordinaten inkrementell berechnet und die Anzahl der benötigten Allokier- und Kopieroperationen möglichst minimiert.

Weiterhin wurde die Implementierung parallelisiert. Dazu wurden zwei Methodiken verglichen. In der ersten werden Teile von Dreiecksgittern jeweils von einem Thread in seinen Farbpuffer gezeichnet und am Ende die Puffer aller Threads bereichsweise parallel zusammengemischt. Die zweite Methodik teilt den Farb- und Tiefenpuffer in Streifen mit einem Pixel Höhe auf, sodass jeder Thread die gleiche Anzahl solcher gleichmäßig verteilten Streifen zugewiesen bekommt und nur diesen Teil der Pixmap visualisieren müssen. Auf den Bereich greifen die anderen Threads niemals zu. Wegen des Geschwindigkeitsvorteils setzte sich die erste Variante durch und wurde folglich weiter untersucht und optimiert.

5.2. Ergebnisse

GR3 beinhaltet nun einen Software-Renderer, auf den automatisch zurückgegriffen wird, wenn keine hardwarebeschleunigte OpenGL Implementierung vorliegt. Explizit ausgewählt werden kann der Software-Renderer durch das Setzen der Umgebungsvariablen `GR3_USE_SR`. Dies funktioniert auch in Umgebungen, die Hardwarebeschleunigung unterstützen. Die erzeugten Grafiken unterscheiden sich von jenen, die mit OpenGL erzeugt wurden, nur minimal auf Grund von Ungenauigkeiten der Fließkommaarithmetik. Die Testbeispiele wurden alle, sofern nicht explizit anders erwähnt, auf einem Ubuntu System mit 16 Kernen erzeugt. Die Zeiten in Zahlen befinden sich im Anhang A.

5.2.1. Test mit hoher geometrischer Komplexität

Der erste Test besteht aus der Visualisierung mehrerer Schädel. Jeder Schädel ist ein Dreiecksgitter bestehend aus 76382 zu rasterisierenden Dreiecken. Die Schädel sind in einem Rechteck angeordnet, in einer Reihe sind 15 Stück und es gibt acht Reihen, somit ergeben sich insgesamt 120 Schädel. Um das Wachstum der Laufzeit mit wachsender Anzahl an Schädeln zu ermitteln und zu vergleichen, wurden Tests mit zunehmender Anzahl an Reihen r_i mit $r_i \in \{1,2,3,4,5,6,7,8\}$, also angefangen bei 15 bis hin zu 120 Schädeln in Schritten der Größe 15, durchgeführt. Sind alle Schädel dargestellt, sieht die Grafik wie folgt aus.

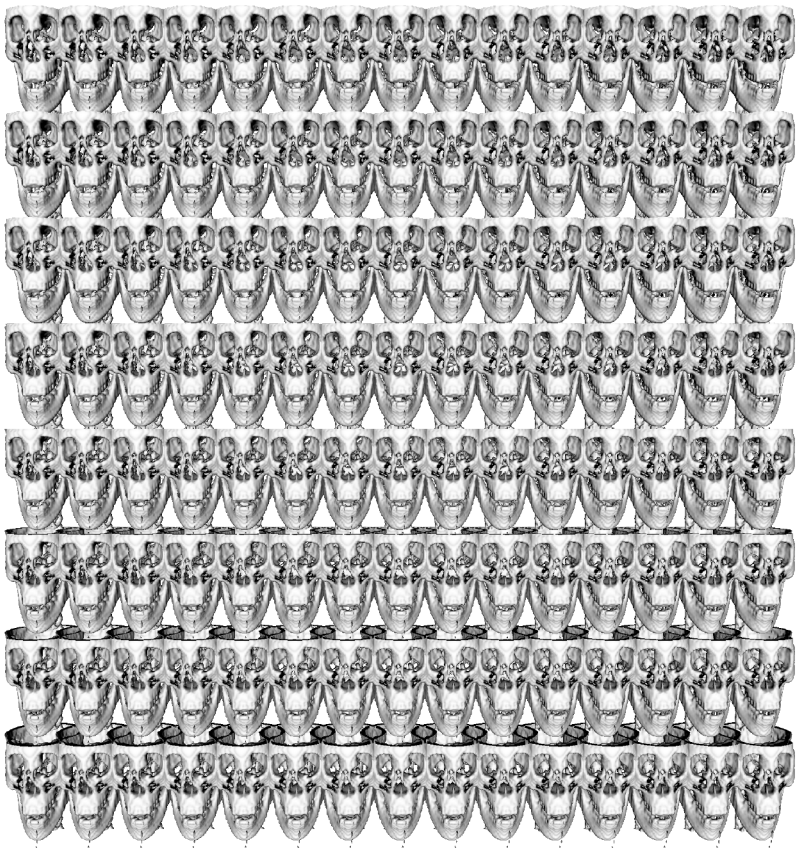


Abbildung 5.1.: Testgrafik mit 120 Schädeln

Dazu wurde die Laufzeit der Erzeugung von 3000 Bildern gemittelt, also die durchschnittliche Zeit zur Erzeugung eines Bildes errechnet. Der Laufzeitvergleich liefert dieses Ergebnis.

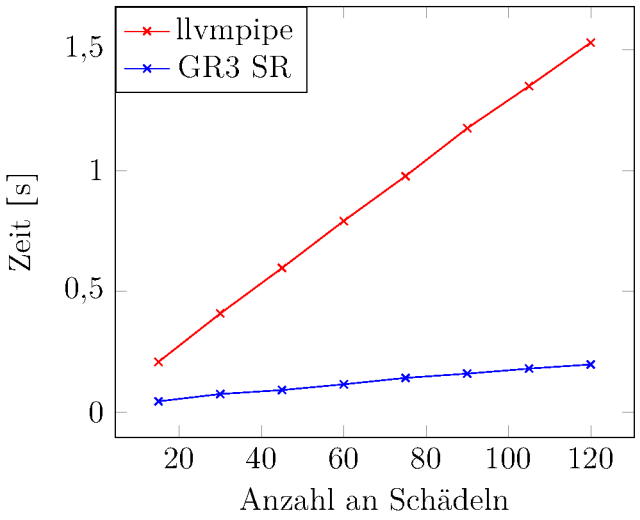


Abbildung 5.2.: Testergebnisse bei variabler Anzahl an Schädeln

Der Software-Renderer des GR3 ist im Falle von 120 Schädeln ungefähr acht Mal schneller als die Implementierung von llvmpipe. Für 15 Schädel liefert der Software-Renderer des GR3 ungefähr 25 Bilder pro Sekunde, was in einer Animation annähernd flüssig wäre. Der konkurrierende Software-Renderer llvmpipe schafft unter gleichen Bedingungen lediglich fünf Bilder pro Sekunde.

Eine Besonderheit an diesem Beispiel ist die enorm hohe Anzahl von sehr kleinen Dreiecken, die größtenteils nur zur Einfärbung weniger oder gar keiner Pixel führen. Für eine hohe geometrische Komplexität mit geringer Anzahl zu füllender Pixel schneidet der Software-Renderer des GR3 diesem Beispiel nach zu urteilen besser ab als llvmpipe. Dies legt die Frage nahe, wie ein Laufzeitvergleich mit wenigen großen Dreiecken im Gegensatz dazu aussieht.

5.2.2. Test mit geringer geometrischer Komplexität

Wie im vorherigen Abschnitt erläutert soll nun die Ausführungsgeschwindigkeit beim Füllen mehrerer großer Dreiecke getestet werden. Dazu werden zwei Dreiecke gezeichnet, die das gesamte Bild füllen. In GR3 bilden sie zusammen ein Dreiecksgitter, das wie folgt aussieht.

Abbildung 5.3.: Testgrafik mit zwei das gesamte Bild abdeckenden Dreiecken

Um ein repräsentatives Ergebnis zu erhalten, werden die dargestellten Dreiecke mehrmals gezeichnet. Dabei befinden sich die Dreiecke in der zu zeichnenden Reihenfolge übereinander, sodass der Tiefentest keinen Einfluss auf das Ergebnis hat. So wird das Wachstum der Laufzeit im Vergleich zur wachsenden Anzahl an großen und übereinanderliegenden Dreiecken getestet, was der wachsenden Anzahl an Schädeln aus dem vorherigen Test entspricht, wobei das Verhältnis zwischen geometrischer Komplexität und Anzahl zu füllender Pixel deutlich geringer ist. Getestet wurde mit einer Auflösung von 1000×1000 Pixel. Die Anzahl der Dreiecke ist dabei ein Parameter, der von 200 bis 3000 pro Bild variiert und in Schritten der Größe 200 wächst. Der Laufzeitvergleich liefert visualisiert das folgende Ergebnis.

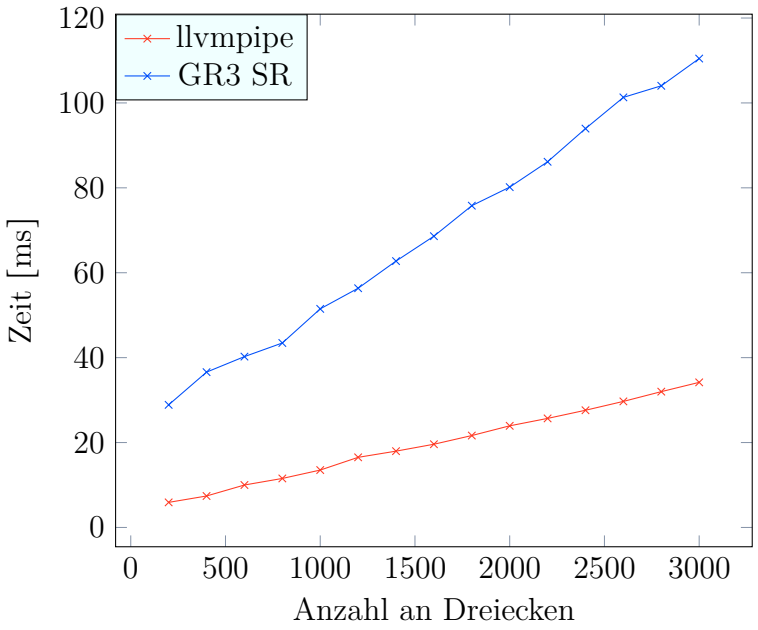


Abbildung 5.4.: Testergebnisse bei variabler Anzahl an Dreiecken

Der Software-Renderer ist für jede getestete Anzahl an Dreiecken langsamer. Die Implementierung mit llvmpipe braucht für zusätzliche 200 Dreiecke circa 2,5 ms länger. Der im Rahmen dieser Bachelorarbeit implementierte Software-Renderer benötigt für denselben Schritt circa 5ms länger. Beide Laufzeiten wachsen linear mit der Anzahl der Dreiecke, jedoch die des Software-Renderers des GR3 deutlich steiler, sodass dieser tendetiell eher füllratenlimitiert ist.

Ein Unterschied in der Laufzeit lässt sich schon bei den ersten 200 Dreiecken erkennen, da sie dort für den Software-Renderer des GR3 bei der Messung mit der geringsten Anzahl an Dreiecken ungefähr 5 mal so hoch ist. Dies hängt mit dem Aufwand für die Initialisierung der Threads, Tiefen- und Farbpuffer zusammen, die auch für wenige Dreiecke schon viel Zeit in Anspruch nehmen und daher anteilig bei einer simpleren Aufgabe mehr ins Gewicht fallen. Die Füllrate, also die Anzahl der Pixel, die pro Sekunde eingefärbt werden können, scheint bei llvmpipe höher zu sein.

5.2.3. Test mit variabler geometrischer Komplexität und konstanter Anzahl zu füllender Pixel

Die bisherigen Tests lassen vermuten, dass viele kleine Dreiecke, also eine hohe geometrische Komplexität, schneller mit dem im GR3 implementierten Software-Renderer erzeugt werden können und wenige große schneller mit llvmpipe dargestellt werden können. Dies legt einen Test nahe, bei dem die Geometrie veränderlich ist, aber die Anzahl insgesamt zu füllender Pixel gleich bleibt. Es wird also die gleiche Fläche gefüllt, während die Anzahl der Dreiecke, aus der sich die Fläche zusammensetzt, variiert. So lässt sich nachvollziehen, wie die Größe und Anzahl der Dreiecke Einfluss auf die Geschwindigkeit nimmt. Demnach lässt sich durch diesen Test zeigen, ob es ein Verhältnis zwischen Komplexität und Anzahl zu füllender Pixel gibt, bei welchem der Software-Renderer des GR3 schneller ist.

In diesem Testbeispiel wurde ein Dreieck erzeugt, welches rekursiv in gleich große Teildreiecke zerlegt wird. Die einfache und doppelte Zerlegung eines Dreiecks sieht dabei wie folgt aus.

(a) Nicht zerlegt (b) Einfach zerlegt (c) Doppelt zerlegt

Abbildung 5.5.: Testgrafik zur Zerlegung eines Dreiecks in Teildreiecke

Dadurch entsteht bei gleichbleibender Anzahl an gefüllten Pixeln eine höhere geometrische Komplexität. Interessant ist hier der Vergleich der Laufzeit bei wachsender Anzahl Unterteilungen. Aufgetragen ist im folgenden die Anzahl der Dreiecke, die durch die Unterteilung entstehen und die dafür benötigte Zeit. Erstellt wurden jeweils 40000 Bilder mit einer Auflösung von 500×500 Pixeln.

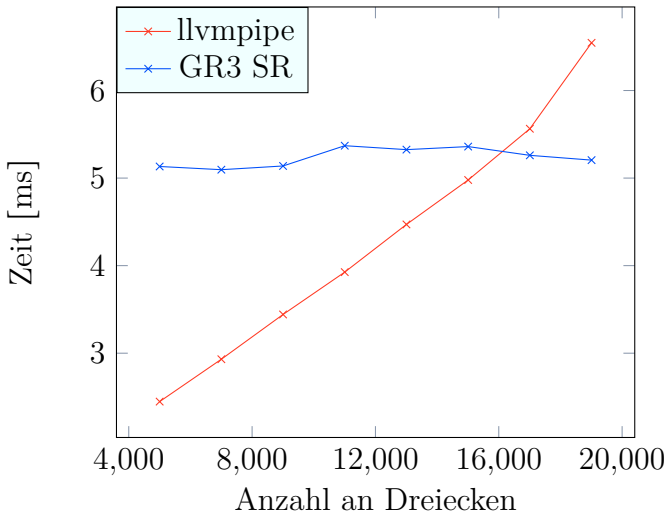


Abbildung 5.6.: Testergebnisse bei variabler Anzahl an durch Zerlegung entstehenden Dreiecken

Die Laufzeit von llvmpipe wächst linear mit der Anzahl der Dreiecke an, obwohl die Anzahl einzufärbender Pixel gleich bleibt. Hingegen ist die benötigte Zeit vom im GR3 implementierten Software-Renderer nahezu konstant, genauso wie die Anzahl eingefärbter Pixel. Im Falle von über circa 16000 Dreiecken, also ungefähr einer siebenfachen rekursiven Unterteilung des Dreiecks, führen viele der Teildreiecke überhaupt nicht zur Einfärbung eines Pixels. Trotzdem nehmen besagte Dreiecke bei llvmpipe nahezu die gleiche Zeit in Anspruch, wie welche, die zur Einfärbung führen, was daran erkennbar ist, dass sie gleichermaßen für einen Anstieg der Laufzeit sorgen.

5.2.4. Test mit konstanter geometrischer Komplexität und variabler Anzahl zu füllender Pixel

Im Beispiel aus Abschnitt 5.2.3 variierte die geometrische Komplexität, die Anzahl zu füllender Pixel blieb jedoch konstant. In diesem Beispiel soll dies umgekehrt werden, sodass geprüft wird, wie sich bei gleichbleibender geometrischer Komplexität die Laufzeit bei steigender Anzahl zu füllender Pixel bei den beiden Varianten verhält. Dazu wird die gleiche Grafik wie aus dem Testbeispiel mit geringer geometrischer Komplexität verwendet, jedoch werden pro Bild keine überlappenden, sondern nur zwei zusammen das gesamte Bild abdeckende Dreiecke gezeichnet. Die variable Größe ist die Auflösung des gesamten Bildes, mit ihr variiert entsprechend die Anzahl insgesamt einzufärbender Pixel. Getestet wird eine Auflösung von 10×10 bis zu 200×200 ansteigend mit Schrittgröße 10. Die Laufzeitmessung liefert das folgende Ergebnis für die Zeit pro Bild.

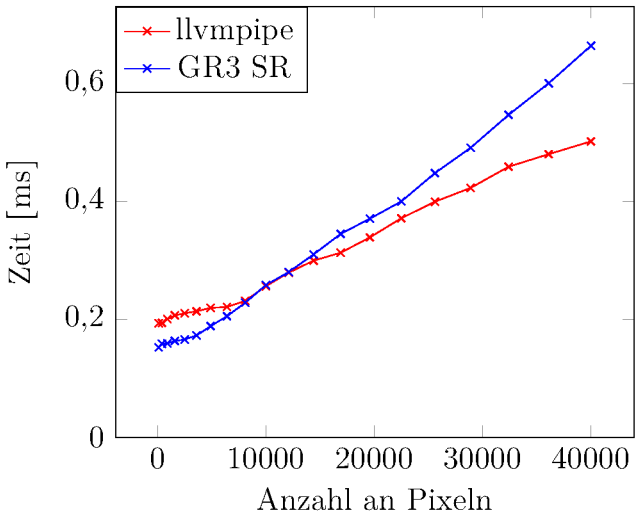


Abbildung 5.7.: Testergebnisse bei variabler Auflösung

Die aus den vorherigen Testbeispielen hergeleitete Vermutung, für kleine Dreiecke wäre die Laufzeit beim in GR3 implementierten Software-Renderer geringer als bei llvmpipe, bestätigt sich für dieses Testbeispiel. Mit steigender Auflösung wächst die Laufzeit beider Testbeispiele, jedoch die von llvmpipe flacher. Dies ist demnach das entsprechende Gegenbeispiel zum vorherigen Testbeispiel, bei dem der Software-Renderer des GR3 die Laufzeit von llvmpipe bei steigender Komplexität unterschreitet, da hier bei steigender Anzahl an auszufüllenden Pixeln und gleicher Komplexität llvmpipe verhältnismäßig schneller wird. Für kleine Dreiecke ist der Software-Renderer des GR3 schneller.

5.2.5. Test der Skalierung bei variabler Anzahl an Threads

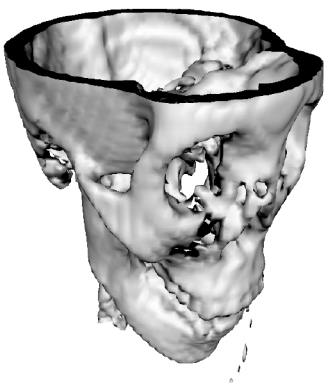


Abbildung 5.8.: Testgrafik mit einem Schädel

Das letzte Testbeispiel soll darstellen, wie die Performance des Software-Renderers des GR3 abhängig von der Anzahl verfügbarer CPU Kerne skaliert. Getestet wird eine Grafik mit hoher geometrischer Komplexität, aber auch einer insgesamt großen Anzahl zu füllender Pixel. Die Testgrafik zeigt den selben Schädel wie im Test mit hoher geometrischer Komplexität, jedoch wird er dieses Mal größer skaliert und lediglich einmal dargestellt, wie der folgenden Abbildung 5.8 entnommen werden kann.

Auf dem verwendeten System sind nun anders als in den vorherigen Beispielen 12 Kerne beziehungsweise 24 Threads verfügbar [Int]. Für diesen Test wird die Anzahl verwendeter Threads variiert und dabei die Laufzeit gemessen. Visualisiert ergibt sich die folgende Grafik.

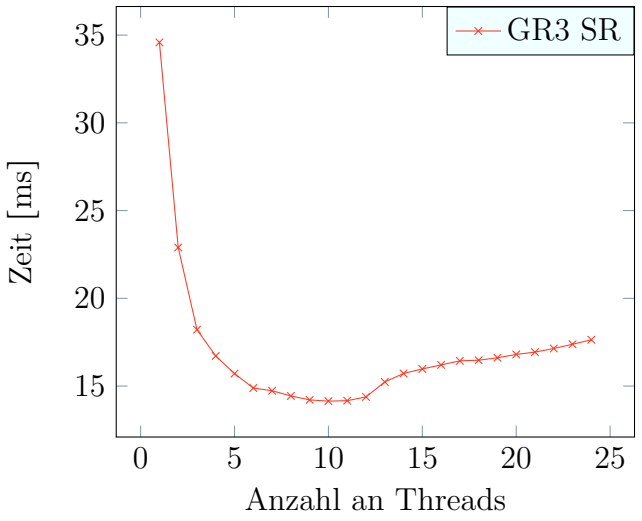


Abbildung 5.9.: Testergebnisse bei variabler Anzahl an Threads

Die Laufzeit nimmt besonders durch Hinzufügen eines zweiten, dritten und vierten Threads ab. Anschließend fällt sie nur noch leicht, bis sie dann circa ab dem zwölften Thread wieder wächst. Dies liegt daran, dass dann insgesamt zwölf Threads und zusätzlich ein Main Thread arbeiten, sodass kein echter CPU-Kern mehr verfügbar ist, sondern nur noch solche, die durch Hyperthreading entstehen. Des Weiteren bewirkt die Parallelisierung für hohe Anzahlen an Threads keinen Geschwindigkeitsvorteil, weil dann der Zusatzaufwand pro Thread (inklusive Allokationen) die aufgeteilte Arbeit des Rendering Prozesses dominiert.

5.2.6. Schlussfolgerung aus den Tests

Die Tests legen nahe, dass der implementierte Software-Renderer primär durch die Anzahl zu füllender Pixel limitiert ist. Die Laufzeit wächst hauptsächlich mit steigender Anzahl von insgesamt zu füllenden Pixeln, wie dem Beispiel mit geringer geometrischer Komplexität entnommen werden kann. Hingegen kann eine feste Anzahl an Pixeln unabhängig von seiner Geometrie in nahezu konstanter Zeit dargestellt werden, wie dem Testbeispiel 5.2.3 entnommen werden kann.

Die vom Software-Renderer llvmpipe benötigte Laufzeit wächst primär mit der geometrischen Komplexität, wie Beispiel 5.2.3 zeigt. Die Füllrate ist dort häufig nicht der ausschlaggebende limitierende Faktor der Rasterisierung, also wird ein Objekt der gleichen geometrischen Komplexität in linear mit seiner Größe ansteigender Zeit rasterisiert, wie Beispiel 5.2.4 zeigt. Das Wachstum der Laufzeit im Verhältnis zur dargestellten Größe ist dabei deutlich geringer, als die beim Software-Renderer des GR3, also ist die Rate an befüllten Pixeln pro Sekunde bei llvmpipe höher. Das Testbeispiel 5.2.1 untermauert das starke Laufzeitwachstum bei wachsender geometrischer Komplexität, während die Anzahl einzufärbender Pixel pro Schädel langsamer wachsen als die geometrische Komplexität.

Für übliche Anwendungsfälle ist der in GR3 implementierte Software-Renderer geeigneter, da häufig sehr detaillierte Strukturen mit vielen kleinen Dreiecken gezeichnet werden müssen. Die Füllrate ist häufig nicht so hoch, dass sie an ihre Grenzen stößt. Aus diesem Grund wäre ein Wachstum der Zeit, das primär von der geometrischen Komplexität abhängt, ungeeigneter als eins, das von der Anzahl der zu füllenden Pixel abhängt.

5.3. Ausblick

Aus den Testfällen ergibt sich unmittelbar die Verbesserungsmöglichkeit, die Füllrate zu erhöhen. Dadurch würden mehr Pixel pro Zeiteinheit eingefärbt werden können und die Laufzeit insbesondere für große Dreiecke enorm reduziert werden. Erreicht werden könnte dies zum Beispiel durch Vektorisierung der Rasterisierung, auch wenn der erste Versuch diesbezüglich die Laufzeit ansteigen ließ. Außerdem könnte sich das Parallelisieren des Zeichenvorgangs eines großen Dreiecks ebenfalls als lohnenswert herausstellen.

Der Software-Renderer könnte insofern erweitert werden, als dass andere Kantenglättungsalgorithmen implementiert werden. GR3 unterstützt bislang ausschließlich das rechenintensive Verfahren `SSAA`, welches auf der CPU erledigt wird und gute Ergebnisse liefert. Zum einen könnten die dafür benötigten Operationen parallelisiert werden. Zum anderen können alternative und effizientere Kantenglättungsalgorithmen implementiert werden, wie zum Beispiel `FXAA` und `SMAA` aus [Gra16]. Dies liefert zwar andere Ergebnisse, jedoch ist die Ausführungsgeschwindigkeit niedriger und die Ergebnisse sind qualitativ trotzdem deutlich besser als ohne Verwendung von Kantenglättung.

Sind die Daten eines Dreiecksgitter mit einem Indexpuffer gegeben, finden die Transformationen von Eckpunkten aus den in Abschnitt 4.3.3 erläuterten Gründen nicht parallel statt. Der Main-Thread nimmt die Transformationen aller Eckpunke vor und der restlichen Threads arbeiten dann auf den bereits transformierten Eckpunkten. Stattdessen kann die Arbeit vorher auf die Threads verteilt werden, sodass jeder Thread zunächst einen Anteil an Eckpunkten transformiert und anschließend seinen Anteil an Dreiecken rasterisiert.

Teile von Dreiecksgittern, die am Ende nicht innerhalb des darzustellenden Be-

reiches liegen, werden bei dem im Rahmen dieser Bachelorarbeit implementierten Software-Renderer erst in Pixelkoordinaten abgeschnitten (Clipping). Die Dreiecke, die sich gänzlich außerhalb des Bildes befinden, könnten schon vorher (z.B. im *View-Space*) verworfen werden, sodass sie nicht mehr den Berechnungen von der Rasterisierung unterzogen werden. Dies bringt vor allem dann einen Geschwindigkeitsvorteil, wenn viele Dreiecksgitterteile außerhalb des darzustellenden Bereiches liegen.

A. Anhang

A.1. Ergebnisse der Laufzeitmessungen

Im folgenden werden die Zeiten der Laufzeitmessungen aus Kapitel 5 aufgeführt.

A.1.1. Test mit hoher geometrischer Komplexität

	Laufzeit in ms	
Anz. Schädel	llvmpipe	GR3 SR
15	208	44
30	408	75
45	596	91
60	791	115
75	977	141
90	1175	159
105	1349	180
120	1531	197

A.1.2. Test mit geringer geometrischer Komplexität

	Laufzeit in ms	
Anz. Dreiecke	llvmpipe	GR3 SR
200	5,91	28,87
400	7,41	36,58
600	9,99	40,24
800	11,54	43 45
1000	13,52	51,48
1200	16,51	56,34
1400	17,97	62,74
1600	19,60	68,60
1800	21,64	75,79
2000	23,94	80,16
2200	25,68	86,13
2400	27,60	93,96
2600	29,69	101,30
2800	31,99	104,03
3000	34,17	110,45

A.1.3. Test mit variabler geometrischer Komplexität und konstanter Anzahl zu füllender Pixel

	Laufzeit in ms	
Anz. Dreiecke	llvmpipe	GR3 SR
5002	2,44	5,13
7003	2,93	5,09
9004	3,44	5,13
11005	3,92	5,36
13006	4,47	5,32
15007	4,97	5,35
17008	5,56	5,25
19009	6,54	5,20

A.1.4. Test mit konstanter geometrischer Komplexität und variabler Anzahl zu füllender Pixel

	Laufzeit in ms	
Anz. Pixel	GR3 SR	llvmpipe
100	0,152	0,192
400	0,158	0,194
900	0,159	0,200
1600	0,163	0,206
2500	0,165	0,210
3600	0,172	0,213
4900	0,189	0,219
6400	0,205	0,221
8100	0,228	0,235
10000	0,258	0,256
12100	0,280	0,279
14400	0,310	0,299
16900	0,345	0,313
19600	0,371	0,339
22500	0,400	0,371
25600	0,448	0,399
28900	0,491	0,423
32400	0,547	0,459
36100	0,600	0,470
40000	0,665	0,492

A.1.5. Test der Skalierung bei variabler Anzahl an Threads

Anz. Threads	Laufzeit in ms GR3 SR
1	34,58
2	22,89
3	18,21
4	16,71
5	15,70
6	14,88
7	14,73
8	14,43
9	14,21
10	14,14
11	14,16
12	14,38
13	15,23
14	15,71
15	15,97
16	16,20
17	16,43
18	16,47
19	16,61
20	16,80
21	16,93
22	17,14
23	17,38
24	17,63

Literatur

[AS96] Andrew W. Appel und Zhong Shao. „Empirical and analytic study of stack versus heap cost for languages with closures“. In: *Journal of Functional Programming* 6.1 (1996), S. 47–74. DOI: 10.1017/S095679680000157X.

[Bre62] J. E. Bresenham. „Seminal graphics“. In: *IBM Systems Journal* (1962), S. 25–30.

[Gie13] Fabian Giesen. „Optimizing the basic rasterizer“. In: (2013). URL: https://fgiesen.wordpress.com/2013/02/10/optimizing-the-basic-rasterizer/ (besucht am 10. 07. 2019).

[Gra16] Alexander Grahn. „An Image and Processing Comparison Study of Antialiasing Methods“. Bachelor's Thesis. Blekinge Institute of Technology, Department of Creative Technologies, 2016, S. 50. URL: http://www.diva-portal.org/smash/get/diva2:972774/FULLTEXT02.pdf.

[Hen+11] Hengyong Jiang u. a. „A novel triangle rasterization algorithm based on edge function“. In: *Proceedings of 2011 Cross Strait Quad-Regional Radio Science and Wireless Technology Conference*. Bd. 2. Juli 2011, S. 1235–1238.

[Hug+14] J.F. Hughes u. a. *Computer Graphics: Principles and Practice*. The systems programming series. Addison-Wesley, 2014. ISBN: 9780321399526.

[Int] Intel. *INTEL® CORE™ i9-7920X PROZESSOR Spezifikation*. URL: https://www.intel.de/content/www/de/de/products/processors/core/x-series/i9-7920x.html (besucht am 19. 08. 2019).

[Len02] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, 2002. ISBN: 1584500379.

[Rhi12] Florian Rhiem. „Integration von 3D-Visualisierungstechniken in 2D-Grafiksystemen“. Bachelor's Thesis. FH Aachen – University of Applied Sciences, 2012. URL: https://pgi-jcns.fz-juelich.de/pub/doc/Bachelor/Bachelorarbeit_FlorianRhiem.pdf (besucht am 16. 08. 2019).

[Rit19] Jonas Ritz. „Entwicklung eines Software-Renderers zur Visualisierung bivariater Funktionen“. Seminar Paper. FH Aachen – University of Applied Sciences, 2019. URL: https://pgi-jcns.fz-juelich.de/pub/doc/Seminararbeiten/Seminararbeit_JonasRitz.pdf (besucht am 17. 08. 2019).

[VMw] VMware. *Gallium LLVMpipe Driver*. URL: https://www.mesa3d.org/llvmpipe.html (besucht am 01. 08. 2019).

[Vri14] Joey de Vries. *Basic Lighting in OpenGL*. 2014. URL: `https://learnopengl.com/Lighting/Basic-Lighting` (besucht am 01. 07. 2019).