Bauhaus-Universität Weimar
Fakultät Medien
Studiengang Medieninformatik

Effizientes Rendering einer isometrischen Blockwelt

Bachelorarbeit

Benedikt S. Vogler
geboren am: 04.11.1994 in Viersen

Matrikelnummer: 113294

1. Gutachter: Prof. Dr. Charles Wüthrich
2. Gutachter: Jun.-Prof. Dr. Florian Echtler

Datum der Abgabe: 8. September 2017

Abstract

Für das hochaufgelöste Rendering einer dreidimensionalen Welt mit axonometrischer Projektion können 2D-Grafiken, die Sprites, genutzt werden. Ein Problem dabei ist das Rendering in der korrekten Reihenfolge. Diese ist unter Berücksichtigung der korrekte Darstellung von Sprites mit Transparenz nicht trivial zu ermitteln. Dazu müssen Pixel oder die Geometrie sortiert werden. Die Arbeit untersucht die isometrische Projektion von block-basierten Computerspielen als Aspekt einer Game Engine, welche modifizierbare, statische Landschaftsdaten und frei bewegliche Objekte benutzt. In der Arbeit werden Implementationen zur Sortierung auf der CPU anhand eines Tiefenwertes, einer topologischen Sortierung, sowie die Pixel-basierte-Sortierung auf der Graphics Processing Unit betrachtet.

Die Sortierung Anhand des Tiefenwerts ist schnell, aber führt zu kritischen Fehlern im Bild. Eine topologische Sortierung eines Verdeckungsgraphen löst das Sortierproblem mit einer höheren Belastung der CPU als andere Methoden. Es kommt in manchen Konfigurationen zu Fehlern bei der Reihenfolge der Entities. Das Depth Peeling Verfahren eignet sich besser für Rechner mit dedizierte Grafikeinheit. Dieses Verfahren benötigt korrekte Geometriedaten und limitiert den Einsatz von übereinandergelegter Transparenz. Je nach Anwendungsfall, kann eines dieser beiden Verfahren ausgewählt werden.

Inhaltsverzeichnis

1 Einleitung 2

2 Verwandte Arbeiten 5
- 2.1 Ausgewählte Spiele . . . 5
- 2.2 Reihenfolge bei isometrischen Spielen . . . 7
  - 2.2.1 Wissenschaftliche Betrachtung . . . 7
- 2.3 GPU-Rendering kurz erklärt . . . 9
  - 2.3.1 Sortierungsproblem . . . 9
  - 2.3.2 Sortierung mit Tiefenbuffer . . . 11
  - 2.3.3 Weitere GPU-Verfahren . . . 12

3 Isometrische Projektion 14
- 3.1 Anpassung für Pixelrasterdisplays . . . 16
- 3.2 Betrachtungswinkel . . . 19
- 3.3 Projektionsmatrizen . . . 21

4 Datenstruktur 24
- 4.1 Modellierung der Spielwelt . . . 24
- 4.2 Rendering . . . 26
- 4.3 Implementation . . . 28

5 Sortierverfahren 29
- 5.1 Korrekte Reihenfolge . . . 29
- 5.2 Trivialer Algorithmus . . . 30
- 5.3 Sortierung mittels Tiefenwert . . . 32
- 5.4 Sortierung mittels topologischer Sortierung . . . 33
  - 5.4.1 Erstellen des Graphs . . . 34

5.4.2 Optimierungen der „Besucht“-Markierung 37
5.4.3 Optimierungen und Laufzeit 38
5.4.4 Ungelöste Probleme 39
5.5 Sortierung mittels Tiefenbuffer 39
5.6 Sortierung mittels Depth-Peeling 42

6 Auswertung 45
6.0.1 Ergebnisse der Messungen 46
6.0.2 Diskussion 48

7 Zukünftige Arbeiten 51

A Anhang 52
A.1 Quelltexte 57

Literaturverzeichnis 61

1 Einleitung

Game Engines sind Spiegel der aktuellen technologischen Möglichkeiten und des Fortschritts der Computergrafik. Im Gegensatz zu Offline-Rendering, bei dem das Ergebnis in eine Bild- oder Videodatei gespeichert wird, muss bei der Echtzeit-Berechnung das Bild in streng begrenzter Zeit berechnet werden, denn sonst stockt der Animationsfluss. Eine Reihe von speziellen Techniken wurde entwickelt, um möglichst realistische oder beeindruckende Bilder entstehen zu lassen. Eine dieser Techniken ist die Verwendung von axonometrischer Projektion. Die Isometrie ist eine spezielle Axonometrie (z. B. Abb. 1.1), welche in dieser Arbeit besonders betrachtet wird.

Abbildung 1.1: Presse-Screenshot des Spiels „Tyranny“ mit isometrischer Projektion, Obsidian Entertainment

Isometrie benutzt man im technischen Zeichnen. Diese Technik zur Bilderzeugung übernahm man in den 1980er Jahren für Computerspiele, was mit geringem Rechenaufwand dreidimensionale Bilder zu erzeugen ermöglichte [Sch04]. Heute ist der Gebrauch perspektivischer Projektionen bei teuren Computerspielproduktionen die Regel. Mit dem Wechsel auf die heutzutage einfach umzusetzende Perspektive nahm das kommerzielle und wissenschaftliche Interesse an axonometrischen Projektionen in der Computergrafik ab. Geschickt eingesetzt jedoch, bietet Isometrie mit aktueller Hard- und Software neues Potenzial für ästhetische Grafiken. Ein herausragendes Beispiel für den Gebrauch aktueller

Techniken zusammen mit Isometrie ist das Spiel „Tyranny“[1] (Obsidian Entertainment, 2016, siehe Abb. 1.1).

In den letzten Jahren beschäftigten wir uns mit der Entwicklung einer isometrischen Game Engine mit dem Namen „Wurfel Engine“. Der Name kommt von den Würfeln, aus denen die Spielwelt besteht. Die Game Engine erlaubt es, Spiele mit ihr zu entwickeln, ohne „das Rand neu erfinden“ zu müssen, und gibt einen Rahmen zum Game Design vor. Wurfel Engine ist frei verfügbar unter einer erweiterter Open Source BSD-3 Lizenz[2]. Die Engine ist in der Programmiersprache „Java“ geschrieben. Die Anforderungen an die Engine ergaben sich aus der Entwicklung des Spiels „Caveland“, welches die Engine nutzen. In Caveland steuert man eine oder im Split-Screen-Modus zwei Spielfiguren, die von einer resp. zwei virtuellen Kameras verfolgt werden. Die Figuren können Blöcke der Welt zerstören und auch neue platzieren. Ebenfalls kann die Spielwelt über einen internen Editor verändert werden.

Abbildung 1.2: Typische Fehler der falschen Reihenfolge

Bei der Entwicklung tauchte schnell das Problem auf, dass die Ordnung der Grafiken nicht stimmte, wie in Screenshot 1.2 zu sehen. Die vermeintliche Lösung, die wir dann über eine lange Zeit nutzten, da wir keine bessere Lösung fand, führte in einigen Fällen dazu, dass die Spielfigur vor einem anderen Block gezeigt wurde, aber eigentlich dahinter stand. Dieser kleine grafische Fehler betrifft nur wenige Pixel im Bild, ist für die Spielerin oder den Spieler aber sehr entscheidend, denn schließlich muss man jederzeit verstehen können, wo die Spielfigur steht. Da dies nicht zufriedenstellend war, untersuchen wir in dieser Arbeit dieses Rendering-Problem genauer.

[1] `https://www.tyrannygame.com`
[2] `https://github.com/BSVogler/WurfelEngineSDK/blob/master/license.txt`

1 Einleitung

Bei einem Computerspiel handelt es sich um eine Echtzeit-Anwendung. Jede Verbesserung der Performance erlaubt es komplexere Grafik zu rendern oder auf leistungsschwächerer Hardware zu laufen. Daher legen wir besonderen Fokus auf die Effizienz der Algorithmen.

In der hier vorliegenden Untersuchung werden zuerst andere Spiele und Untersuchungen zu vergleichbaren Problemen betrachtet, dann die Projektion bei der Isometrie genauer definiert, Details zur Implementation der Algorithmen erläutert, und zuletzt folgt der Vergleich der Methoden und die Diskussion.

Alle angegebenen URL wurden am Tag der Abgabe auf ihre Gültigkeit überprüft.

2 Verwandte Arbeiten

2.1 Ausgewählte Spiele

In der Geschichte der Computerspiele wurde Isometrie schon oft benutzt. Um einen Überblick zu erhalten, welche Techniken bis jetzt benutzt wurden, wollen wir hier einige ausgewählte Spiele betrachten, die typische und besondere Charakteristika zeigen.

Bei der Isometrie gibt es keine von der Position abhängige Verzerrung der Grafik, sodass vorher offline gerenderte Grafiken benutzt werden können. Diese zweidimensionalen Grafiken nennt man Sprites. Sprites wurden in den Anfängen der Computerspiele sehr häufig genutzt, denn es war die einzige Möglichkeit, wie man Pixel-Grafiken rendern konnte. Es gab sogar Sprite-Renderer in Hardware (z. B. am Amiga [Com91, p. 97ff]). Ein typisches Beispiel für den Einsatz von Sprites ist „Diablo 2“ (Blizzard Entertainment, 2000). Eine Besonderheit hier ist, dass es einen Modus gibt, bei dem eine Annäherung an die Perspektive aktiviert werden kann. Dafür werden Parallax-Verzerrungen der einzelnen 2D-Grafiken genutzt.[Trü14] Heute würde man dies mit Perspektive und 3D-Modellen ohne Sprites umsetzen, wie sich an der Fortsetzung „Diablo 3“ (Blizzard Entertainment, 2012) sehen lässt.

Eine beliebte Technik, die ebenfalls mit Sprites arbeitet, ist das Kacheln (tiles) von Sprites, um damit die Landschaft zu modellieren. Mit wenigen Kacheln lassen sich so komplexe Landschaften bauen. Kacheln können bei allen orthografisch projizierten Spielen genutzt werden.

Steam bietet mit mehr als 16.700 Titeln[1] die größte Bibliothek im Videospielevertrieb. Die Website „steamspy“[2] untersucht die öffentlich verfügbaren Daten der Steam-Datenbank

[1] `http://store.steampowered.com`
[2] `http://steamspy.com`

mittels Hochrechnungen und bereitet diese auf. Dort werden unter dem Tag „isometric“ 192 Spiele gelistet. Unter den Ergebnissen befindet sich allerdings eine Reihe von Spielen, die perspektivische Projektionen nutzen. Es scheint, als habe sich „Isometrie“ zu einem Synonym für Spiele mit einer Third-Person-Perspektive etabliert. Der verbreitetste tatsächlich isometrische Titel ist „Age of Empires II HD“ mit 4.203.992 Besitzern. Bei Age of Empires ist das Rendering der Landschaft Kachel-basiert, und die restlichen Grafiken werden über Sprites gerendert.

Microsoft kündigte eine Neuveröffentlichung von Spielen aus der Age of Empires Reihe an [aoe]. Das Spiel „Age of Empires: Definitive Edition“ wird 20 Jahre nach seiner Erstveröffentlichung erscheinen. Es bleibt weiterhin tile-basiert, und auch die Figuren werden mit Sprites gerendert. Eine Modernisierung ist die Erhöhung der Auflösung für die Sprites, da die Bildschirmauflösungen heutzutage größer sind.

Das Spiel „Q*Bert“ (Gottlieb, 1982) ist eines der ersten Spiele, das die Isometrie benutzte. Dieses Spiel ist ein besonderer Fall, denn man blickt auf eine Schräge mit bunten Würfeln (vgl. 2.1). Bei isometrischen Spielen gibt es meistens wenige Ebenen, und die Höhe wird ebenfalls wenig genutzt, wie sich an den anderen Beispielen auch sehen lässt.

Abbildung 2.1: Screenshot des Spiels „Q*Bert“ [qbe]

Die Engine der „Cossacks“-Reihe, bezeichnet mit dem Namen „dmrc“, benutzt eine 3D-Engine für das Terrain (vgl. 2.2). Einheiten und Gebäude werden mit Sprites gerendert [dmc].

Abbildung 2.2: Screenshot vom Spiel „Cossacks: Back to War“ [cos]

Abbildung 2.3: Screenshot des Spiels „Sacred“, Quelle: selbst aufgenommen

Die Engine des Spiels „Sacred“ (Ascaron, 2004) arbeitet genau umgekehrt. Da das Rollenspiel das Tragen verschiedener Rüstungsgegenstände mit einer Vielzahl von Animationen erlaubt, werden Spielfiguren zu Echtzeit in 3D berechnet (vgl. 2.3). Dies erlaubt flüssigere Animationen. Das statische Terrain wird über gekachelte 2D-Grafiken erzeugt.

2.2 Reihenfolge bei isometrischen Spielen

2.2.1 Wissenschaftliche Betrachtung

Die wissenschaftliche Betrachtung der Computerspiele geschieht aus unterschiedlichen Perspektiven. Das Feld der *Game Studies* untersucht seinem „multidisziplinären Charakter“

nach Spiele u. a. in Bezug auf die Narration, Spielmechanik, Bedienung und auf den sozialen Kontext hin [Gam12]. Die Informatik hat, sofern Spiele untersucht werden, demnach Überschneidungen mit den Game Studies, denn sie beschäftigt sich mit der Beschleunigung von Datenverarbeitung und den daraus resultierenden Möglichkeiten. Das Teilgebiet der Computergrafik kann sich wegen der Begrenztheit von Rechenleistung immer nur mit approximativer Bilderzeugung beschäftigen. Genau wie die Computergrafik demnach Grenzen im Design vorgibt, gibt es noch ästhetische und ludonarrative Gründe, die beeinflussend auf die Modellierung der Spielwelt einwirken und damit wiederum die Technik, wie die Wahl der Algorithmen, beeinflussen. Die wissenschaftliche Untersuchung von Computerspielen aus der Perspektive der Informatik kann also nur unter der Annahme von Modellierungskonzepten geschehen. Dies erschwert bzw. verhindert die Abstraktion und Generalisierung von entwickelten Techniken.

Eine Besonderheit bei der akademischen Betrachtung von Computergrafik ist, dass häufig Spezialfälle betrachtet werden, welche in der Praxis der Computerspieleentwicklung nicht so oft auftauchen. Spezialfälle kommen z. B. aus dem Fokus auf CAD[3]-Software. Viele Käufer dieser Spiele kaufen diese für Systeme, welche nicht dem aktuellen Stand der Technik entsprechen und oft viele Jahre alt sind. Deshalb dauert es oft einige Jahre, bis neue Standards weit genug verbreitet sind, bevor diese als einzige Option eingesetzt werden können. Es kann Lösungen geben, die nur auf gewissen Hardware- und Betriebsystemen funktionieren. Ein Beispiel ist die Verwendung von OpenGL[4] Version > 4.2, welche nicht von macOS unterstützt wird. Solche systemabhängigen Besonderheiten haben Auswirkungen auf die Entwicklungskosten, denn es wird Entwicklungszeit und Expertise für das jeweilige System benötigt. Vertreiber von Computerspielen oder Grafikanwendungen wünschen sich, dass eine Implementation auf möglichst vielen Plattformen läuft. Diesen Wunsch berücksichtigen auch Game Engine Produzenten, welche zunehmend den Export für eine Vielzahl von Plattformen ermöglichen. Bei der Betrachtung von Lösungen macht es also auch Sinn, diese Einschränkungen zu berücksichtigen. Es besteht daher ein Bedarf an Untersuchungen der Möglichkeiten, die am Markt orientiert sind.

[3] computer-aided design, oder rechnerunterstütztes Konstruieren
[4] Eine genauere Erklärung folgt in Kapitel 2.3

2.3 GPU-Rendering kurz erklärt

Zum weiteren Verständnis dieser Arbeit wird ein grobes Verständnis der Pipeline beim Rendering benötigt. Daher folgt nun ein kurzer Überblick.

Grafiken bzw. die Geometrie werden als eine Menge von Ecken (vertices) mit weiteren Eigenschaften auf der CPU berechnet. Um daraus ein Bild (Frame) zu generieren, wird häufig der Grafik-Standard *OpenGL* zur Kommunikation mit der Graphics Processing Unit (GPU) genutzt. Andere Standards sind *DirectX* für Windows-Plattformen, Apples *Metal*, oder *Vulkan* der Khronos Group. Nachdem die Geometriedaten auf die GPU, oft zu finden in einer dedizierten Grafikkarte, geladen wurden, wird auf jedes Vertex (Eckpunkt) parallel ein Programm ausgeführt, der sogenannte Vertex-Shader. Der Vertex-Shader kann frei programmiert werden und wird i. d. R. dazu genutzt, die Projektionen anzuwenden. Danach findet die Rasterung statt, die Pixel, oder, im Wortschatz von OpenGL, „Fragmente“ berechnet. Die Berechnung der Farbe und weiterer Werte zu den Fragmenten lässt sich, analog zu Vertex-Shadern, über den Gebrauch von Fragment-Shadern programmieren. Aus allen Fragmenten ergibt sich dann ein Einzelbild. Viele aufeinanderfolgende Einzelbilder ergeben die Animation. Deshalb wird das Verfahren kontinuierlich in einer Schleife ausgeführt, die mit einer möglichst hohen Wiederholfrequenz laufen sollte. Aktualisierungsgeschwindigkeiten der Bildschirme setzen hier üblicherweise Grenzen bei 60, 100 oder 120 Hertz. Ausgangspunkt für das Bild ist die virtuelle Kamera, welche ähnlich wie eine reale Kamera funktioniert und in der virtuellen Welt positioniert wird. Das Kapitel 3 widmet sich vertieft der Projektion der Kamera.

2.3.1 Sortierungsproblem

Isometrische Spiele arbeiten oft mit 2D-Grafiken, sogenannten Sprites. Sprites sind oft einfacher als 3D-Modelle zu erstellen und können mit 2D- oder 3D-Programmen erstellt werden. Viele kombinierte 2D-Grafiken ergeben dann die gesamte Szene. Ein wiederkehrendes Problem ist die Bewerkstelligung der korrekten Rendering-Reihenfolge der 2D-Grafiken.

Für ein korrektes Rendering ist die Reihenfolge der Objekte in der Tiefe und damit die sich überdeckenden Pixel (mit unterschiedlicher Tiefe) entscheidend. An einer Position eines Fragmentes befinden sich oft mehrere Objekte. Der Maler-Algorithmus arbeitet daher

wie ein Maler ein Bild malt: Hintere Ebenen werden zuerst gerendert, Ebenen, die davor liegen, werden danach gerendert und überdecken alle vorher gesetzten Pixel. Dies wird „back-to-front-Rendering“ genannt. Die Überdeckung hinterer Pixel ist abhängig vom Wert der Transparenz des Pixels, dem Alpha-Wert (alpha blending). Dies lässt sich in der folgenden Gleichung ausdrücken:

$$C_{final} = \alpha_{src} C_{src} + (1 - \alpha_{src}) C_{dest}$$

wobei α_{src} die Opazität und C_{src} die Farbe des Pixels im Vordergrund und C_{dest} die Farbe des Pixels im Hintergrund beschreibt [RB05, p. 200]. Für das Pixel im Hintergrund gilt rekursiv die selbe Gleichung.

Der Maleralgorithmus setzt allerdings die Kenntnis über die Renderreihenfolge voraus. Diese muss erst ermittelt werden, was wir als Sortierproblem bezeichnen. Zur Lösung des Sortierproblems gibt es drei Kategorien von Ansätzen: Eine Kategorie versucht die korrekte Sortierung der Sprites mit der CPU über Weltkoordinaten herauszufinden. Ein anderer Ansatz ist die Fragment-Sortierung auf der GPU, welche auch sich schneidende Geometrie berücksichtigen kann. Der dritte Ansatz sind Verfahren zur approximativen Lösung der o. g. Alpha-Blending-Gleichung. Bei isometrischen Spielen wird gewöhnlicherweise auf den trivialen Algorithmus zur Objektsortierung auf der CPU verwiesen, den in Kapitel 5.2 genauer vorgestellt wird.

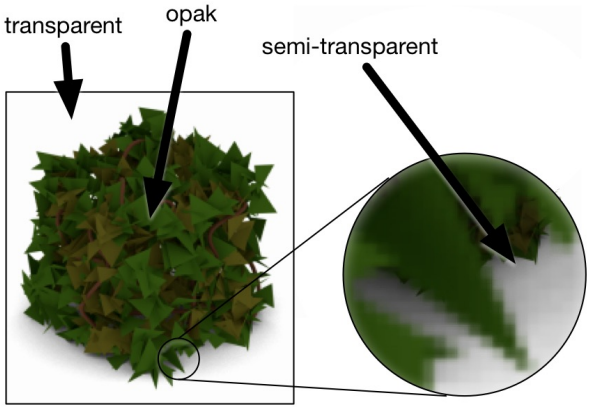

Abbildung 2.4: Unterschiedliche Arten von Pixeln in einem Sprite

Sprites bestehen meistens aus volltransparenten Pixeln ($A = 0$), vielen opaken ($A = 1$) und wenigen semi-transparenten Pixeln ($0 < A < 1$). I. d. R. werden die semi-transparenten

Pixel wegen des Anti-Aliasings oder Schatten an den Rändern (vgl. 2.4) benutzt. Spiele benötigen exakte Renderings, damit keine Verwirrung über die Positionierung von Objekten erzeugt wird. Da viele opake Pixel benutzt werden, sind approximative Lösungen der o. g. Gleichung wie z. B. der gewichtete Durchschnitt [BM08] nicht gut geeignet.[5] Diese approximativen Methoden eignen sich besser für komplexe Modelle mit viel semi-transparenten Oberflächen und werden deshalb in dieser Arbeit nicht weiter behandelt.

In dieser Arbeit werden wir verschiedene Verfahren zur Erstellung der Reihenfolge auf der CPU betrachten.

2.3.2 Sortierung mit Tiefenbuffer

Ein GPU-basiertes Verfahren ist der Tiefenbuffer (depth buffer) oder auch genannt Z-Buffer. Dieses Verfahren lässt sich vergleichsweise einfach implementieren, da es in Hardware in Grafikprozessoren implementiert ist und über den OpenGL-Standard zur Verfügung gestellt wird.

Hierbei wird die Tiefe, also die Distanz von der Kamera, der einzelnen Fragmente betrachtet und mit dem aktuellen Maximum verglichen. Fragmente, die eine größere Tiefe als den aktuellen Maximalwert haben, werden nicht weiter betrachtet (discard). Der Tiefenwert ist der z-Wert der Pixel-Koordinate nach Transformation in den clip-space[6]. Daher ist es auch wichtig, dass bei diesem Verfahren die z-Koordinate durch die Projektion nicht auf 0 gesetzt wird, obwohl die Z-Achse in der projizierten zweidimensionalen Bildebene nicht sichtbar ist. Die Z-Achse liegt orthogonal zur Bildebene, also macht die Verwendung für das resultierende Bild keinen Unterschied. Der Wert wird nur für Operationen benötigt, die diesen voraussetzen.

Probleme bei der Verwendung des Tiefenbuffers sind Präzisionsfehler bei großen Szenen, wie z. B. beim Blick zum Horizont.[7] Isometrische Spielszenen sind in der Praxis hiervon nicht betroffen, da die Kamera auf den Boden gerichtet ist.

[5] Bilder dazu mit gewichtetem Durchschnitt A.3, und gleiche Szene exakt gerendert A.2

[6] normalisierter Raum, in dem sich alle zu rendern Fragmente nach Anwendung der Kameratransformation befinden.

[7] Eine tiefergehende Betrachtung dieses Problems: `https://developer.nvidia.com/content/depth-precision-visualized`

Beim Blending werden sich überlagernde Pixel zusammengerechnet. Blending wird von hinten nach vorne durchgeführt. Bei der Benutzung des Tiefenbuffers wird nicht die Reihenfolge, sondern nur das vorderste Pixel ermittelt. Ist der Tiefenbuffer aktiviert und ein vorderes Objekt wird zuerst gerendert, so werden dahinterliegende Objekte nicht berücksichtigt. Wir stellen fest, dass das Bild beim Gebrauch des Z-Buffers weiterhin abhängig von der Reihenfolge ist (order-dependent). Eine genauere Betrachtung zur Implementation des Tiefenbuffers findet sich in 5.5.

Bei perspektivischen Darstellungen mit 3D-Modellen wird hier meistens nach einem Algorithmus vorgegangen: opake Objekte als erstes mit aktivierten Z-Buffer rendern, transparente danach [RB05, p. 201]. Nur die transparenten Objekte müssen jetzt vor dem Rendering sortiert sein. Bei dem Gebrauch von Sprites mit Semi-Transparenz ist allerdings *jeder* Sprite den transparenten Objekten zuzuordnen. Alternativ kann auf die Verwendung von Kantenglättung an den Rändern verzichtet werden, um so die Menge der Sprites mit Transparenz zu reduzieren.

2.3.3 Weitere GPU-Verfahren

Für die Lösung der Problemklasse des fehlenden Blending bzw. des order-dependent Blendings bei Benutzung des Tiefenbuffers sind zwei Arten von Verfahren üblich, welche mit Order-independent Transparency (OiT) bezeichnet werden. Bei OiT werden nicht die Objekte bzw. Dreiecke, sondern die Fragmente sortiert, was sich auf der GPU parallelisieren lässt. Die Sortierung der Fragmente kann mit dem Tiefenbuffer oder mit Listen geschehen.

Für OiT kann „Depth-Peeling“ [Eve01] oder die Verbesserung „Dual-Depth-Peeling“ [BM08, MBC12] genutzt werden. Die zugrunde liegende Funktionalität ist der Gebrauch des Tiefenbuffers, um das vorderste Fragment zu erhalten, mit einer Kombination aus multi-pass Rendering und multi-texturing, was sich historisch betrachtet aus dem shadow-mapping entwickelt hat. Mit jedem weiteren Aufruf kann durch den Vergleich mit dem Framebuffer aus dem letztem Rendering das dahinterliegende Fragment erhalten werden. Es werden zwei Tiefenbuffer benutzt, damit nicht einer zugleich gelesen und geschrieben wird. Der Buffer, welche die Informationen über die vorherige Ebene enthält, wird gelesen, der andere wird gelesen und geschrieben.

Das zweite OiT-Verfahren ist die Sortierung der Fragmente nach ihrer Tiefe. Hierbei werden nach Knowles et al. [KLZ14] „per-pixel linked lists (PPLLs)“oder „packed per-pixel arrays (PPPAs)“ genutzt. Die zu erwartende Anzahl der Fragmente liegt bei einer FullHD-Auflösung (1920 x 1080) mindestens über 2 Mio. In einem Beispiel in ihrer Arbeit mit PPLLs wird die für Echtzeitrendering nötige Performance bei 16 Mio. Fragmenten in der GLSL-Implementation auf einer Nvidia GTX Titan, also einer Grafikkarte im aktuell oberen Preissegment, erreicht. 16 Mio. Fragmente entsprechen etwa 8 Ebenen pro Pixel bei einer FullHD-Auflösung, was komplexer ist als übliche Szenen einer virtuellen Welt. Für weniger komplexe Szenen liegen wenig Daten vor. Mit einer NVIDIA Geforce 280 GTX (veröffentlicht 2008) sind bei 70.000 Dreiecken, 8 Ebenen und einer Auflösung von 512 x 512 alle vorgestellten Algorithmen schneller als 120 Bildern pro Sekunde [Liu13]. Diese Auflösung liegt natürlich weit unter aktuellen Bildschirmauflösungen.

Liu et al. [LHLW10] fanden einen OiT-Ansatz mit „FreePipe System“ mit der Verwendung der CUDA-API [NVI08b]. Den Autoren nach [Liu13] erreichte es in Vergleichstests die höchste Geschwindigkeit der getesteten Verfahren. CUDA erlaubt die Programmierung von massiv parallelen Berechnungen mit der GPU. Für eine von der Plattform unabhängige Lösung müsste dieser Algorithmus in OpenCL geschrieben werden.

Govindaraju und Henson [GH05] entwickelten eine Lösung, bei der die Objekte und nicht die Fragmente auf der GPU mit dem Depth Buffer und Occlusion Queries sortiert werden. Dieser Algorithmus nutzt die Ähnlichkeit zwischen zwei Frames aus, denn die Reihenfolge ändert sich meistens nur an wenigen Stellen. Der Input zum nächsten Frame ist dann zu einem Großteil vorsortiert. Diesen Algorithmus betrachten wir in dieser Arbeit nicht weiter.

Auch Barta und Kovács [BK11] sprechen von einem Overhead für wenig komplexe Szenen. Es ist davon auszugehen, dass aufgrund des Overheads für die Sortierung die Performance bei weniger Fragmenten nicht wesentlich besser ist. In dieser Arbeit werden diese deshalb nicht weiter betrachtet.

3 Isometrische Projektion

Virtuelle Welten werden mittels Raumkoordinaten modelliert. Beim Prozess der Bilderstellung (Rendering) werden die Raumkoordinaten in Bildkoordinaten mit einer Projektion f projiziert:

$$f : \mathbb{R}^3 \to \mathbb{R}^2$$

Die zugehörigen Räume werden *world space* oder *game space* für die Raumkoordinaten, genannt. In der Bildebene spricht man von *screen space*. Zunächst wollen wir eine spezielle Projektion, die Isometrie, einordnen. Die Isometrie gehört zu den axonometrischen Projektionen. Die Herkunft der Bezeichnung für Axonometrie (gr. áxōn = Achse und métron = Maß) erklären Fucke, Krich und Nickel [FKN93, p. 191]: „Weil, aus dem Bild durch Abmessen der Koordinaten auf den Achsen die Raumlage von Punkten ermittelt werden kann, heißt diese Darstellung die axonometrische Projektion.“ Dies ist aber nur dann möglich, wenn von rechten Winkeln zwischen den Achsen im Raus ausgegangen wird, was beim technischen Zeichnen die Regel ist. Eine Rückprojektion aus dem *screen space* zum *world space* ist wegen des Informationsverlustes wegen der Nicht-Injektivität der Projektion im Allgemeinen *nicht* eindeutig. Mögliche Raumpunkte befinden sich auf einer Geraden. Eine Besonderheit der Axonometrie ist, dass parallele Geraden allgemein auch in der Projektion parallel in der Bildebene bleiben [FKN93, p. 190].

Axonometrische Projektionen sind durch zwei Parameter beschreibbar. Wir definieren das projizierte Achsenkreuz beschreibende Paar aus zwei Vektoren $(\vec{\alpha}, \vec{e})$ mit Winkeln der Achsen zueinander (in Gradmaß) $\vec{\alpha}$ und Verzerrungen $\vec{e}$.

Isometrie (griechisch „gleiches Maß“) bezeichnet den Fall, dass die Verkürzung der Achsen durch die Projektion im gleichen Verhältnis erfolgt. Durch die Projektion bzw. Abbildung wird also die Metrik zwischen zwei Punkten nicht verändert. Formal ausgedrückt: „Sind V und W zwei normierte Vektorräume mit Norm $\|\cdot\|_V$ bzw. $\|\cdot\|_W$ und ist $f\colon V \to W$ eine lineare Abbildung, so ist diese Abbildung genau dann eine Isometrie, wenn sie die Norm

erhält, wenn also für alle $v \in V$:$\|f(v)\|_W = \|v\|_V$"[Wikb] Das Verzerrungsverhältnis ist also für alle Achsen das Gleiche mit $e_x = e_y = e_z$.

Bei der Dimetrie bleibt die Metrik für nur zwei der drei Achsen erhalten. Bei der Trimetrie besitzen alle Achsen verschiedene Verzerrungen.

Die Basis eines Koordinatenursprungskreuzes muss allgemein betrachtet nicht orthonormal,[1] also keine Orthonormalbasis sein. Da allerdings von jeder Basis immer eine Basistransformation auf eine Orthonormalbasis durchgeführt und damit jedes Koordinatenursprungskreuz als Orthonormalbasis betrachtet werden kann, gehen wir im Folgenden von dem orthonormalen Koordinatenkreuz mit den vier Eckpunkten (0,0,0), (1,0,0), (0,1,0), (0,0,1) aus.

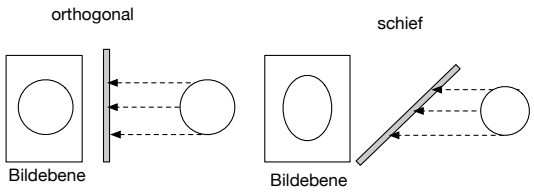

Abbildung 3.1: Orthogonale und schiefe Projektion eines Kreises

Es wird zwischen der orthogonalen und schiefen Axonometrie unterschieden. Bei der orthogonalen treffen die Projektionsstrahlen der Bildpunkte orthogonal respektiv schräg auf die Bildtafel/Bildebene [wika]. Eine schräge Projektion sorgt für eine affine[2] Transformation in die Bildebene. Kugeln erscheinen daher nicht mehr als Kreise, sondern als Ellipsen (vgl. Abb. 3.1). Das Benutzen einer schiefen Projektion hat den gleichen Effekt wie die Wahl von Längenverzerrung der Achsen.

Wir betrachten nun den Fall eines projizierten Einheitswürfels, wie er gewöhnlich anzutreffen ist (Isometrie 1, Abb. 3.2).

Herleitung von $\vec{\alpha}$ Da alle Winkel in der Bildebene im gleichen Verhältnis zueinander stehen, ist $\alpha_x = \alpha_y = \alpha_z$. Der Kreisumfang τ[3] wird gleichmäßig auf die drei Achsen verteilt.

[1] Achsen stehen orthogonal zueinander und haben die Länge 1
[2] Erhalt der Verhältnisse von Längen und Winkeln
[3] In diesem Falle vereinfacht die Nutzung von τ anstelle von 2π folgende Gleichungen. Vgl. „The Tau Manifesto" `http://www.equipeducation.org/Downloads/The%20Tau%20Manifesto.pdf`

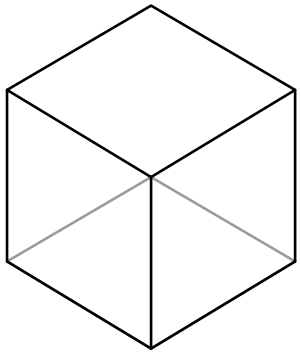

Abbildung 3.2: Ein isometrisch projizierter Einheitswürfel (Isometrie 1)

Herleitung von $\vec{e}$ Der Neigungswinkel der Achsen gegen die Bildebene berechnet sich wie folgt: $cos(\beta) = \frac{e_x}{e}$ [FKN93, p. 193], e: unverzerrte Länge. Der Neigungswinkel ist in diesem Falle $\beta = 45° = \pi/4 = \tau/8$, damit die Isometrie gewährleistet wird. Mit $e = 1$ für den Einheitswürfel folgt daraus:

$$e_x = e_y = e_z = cos(\tau/8) = 2^{-1/2} \approx 0,707$$

Für die Standard-Isometrie gilt daher:

$$\vec{\alpha} = \begin{pmatrix} \tau/3 \\ \tau/3 \\ \tau/3 \end{pmatrix} = \frac{\tau}{3} * \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \vec{e} = \begin{pmatrix} 2^{-1/2} \\ 2^{-1/2} \\ 2^{-1/2} \end{pmatrix} = 2^{-1/2} * \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

3.1 Anpassung für Pixelrasterdisplays

Pixelrasterdisplays benötigen eine Rasterung von Vektordaten. Dies führt zu Aliasing-Effekten, weshalb man mit Anti-Aliasing-Techniken arbeiten. Einige Spiele, insbesondere ältere, konnten aber kein Anti-Aliasing benutzen, denn die Pixeldichte war damals zu gering. Den Grafikstil mit hart abgegrenzten, manuell gesetzten Pixeln nennt man Pixel Art. Bei der Isometrie 1 erhalten wir dann stufige Linien (Abb. 3.3).

Für die Pixel-Art-Isometrie (Isometrie 2) soll daher für die Diagonalen das Verhältnis 2:1 (Winkel $\beta = atan(1/2) \approx 26,56°$) genutzt werden (Abb. 3.4). So sind die Stufen der Pixellinie gleichmäßig, aber der Winkel ist verändert.

Abbildung 3.3: Unregelmäßige Stufen

Abbildung 3.4: Linie mit gleichmäßigen Stufen

Mit dem verändertem Winkel erhalten wir:

$$\vec{\alpha'} = \begin{pmatrix} \tau/8 + atan(1/2) \\ \tau/8 + atan(1/2) \\ \tau/4 - 2atan(1/2) \end{pmatrix}$$

Es ergibt sich folgendes Bild (3.5):

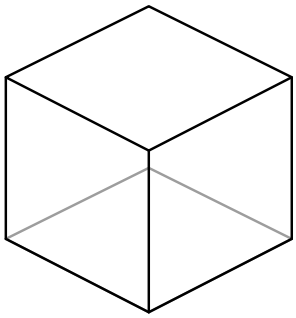

Abbildung 3.5: Ein isometrisch projizierter Einheitswürfel mit verändertem $\vec{\alpha}$ (Isometrie 2)

Besonders ist hier, dass sich hintere Ecke der Grundfläche nicht mit der vorderen Ecke der Oberseite in einem Punkt in der Bildebene schneiden. Ist dies gewünscht, so muss eine andere Verzerrung der Achsen gewählt werden (Dimetrie 1, Abb. 3.6). Wir verkürzen also die projizierte Höhe e''_z so, dass sie gleich der Länge der winkelhalbierenden projizierten Diagonalen g durch die Grundrissebene ist: $e''_z = g$. Durch die Veränderung der Verzerrungen wird diese Projektion zur Dimetrie, denn $e''_z \neq e''_x = e''_y$.

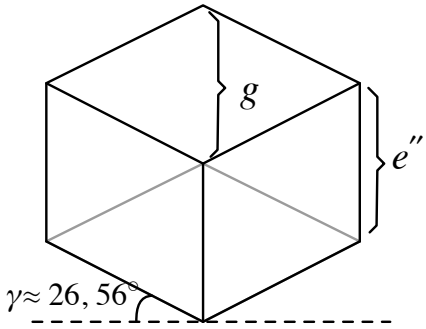

Abbildung 3.6: Ein dimetrisch projizierter Einheitswürfel mit verändertem $\vec{\alpha}$ und e_z (Dimetrie 1)

Wenn γ den Winkel der Achsen auf der Grundfläche, also die X- und Y-Achse, zur Bild-Horizontalen beschreibt, lässt sich g berechnen.

$$sin(\gamma) = \frac{g/2}{e''_x}$$

$$\Updownarrow g = 2e_x sin(\gamma)$$

Für die Dimetrie 1 setzen wir $e_x = 2^{-1/2}$ und $\gamma = atan(1/2)$ ein:

$$g = 2sin(\gamma) * 2^{-1/2} = \sqrt{2}sin(\gamma)$$

$$e''_z = g = \sqrt{2} * sin(atan(1/2)) \approx 0,632455532$$

$$\vec{e''} = \begin{pmatrix} 2^{-1/2} \\ 2^{-1/2} \\ \sqrt{2} * sin(atan(1/2)) \end{pmatrix} = 2^{-1/2} * \begin{pmatrix} 1 \\ 1 \\ 2 * sin(atan(1/2)) \end{pmatrix}$$

Das Verhältnis der Seiten ist $0,707 : 0,707 : 0,632 \approx 1 : 1 : 0,894 \approx 1,118 : 1,118 : 1$.

Bei der Isometrie 2 sind mehrere übereinanderliegende Ebenen in der Projektion zueinander leicht versetzt (Abb. 3.7). Überdecken sich Ober- und Unterseite der Blöcke in

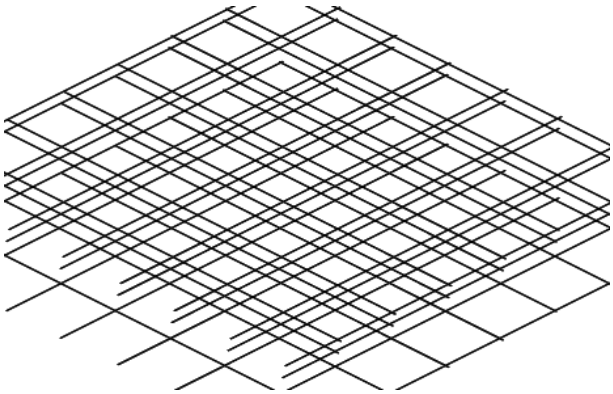

Abbildung 3.7: Versetztes Gitter mit drei Ebenen

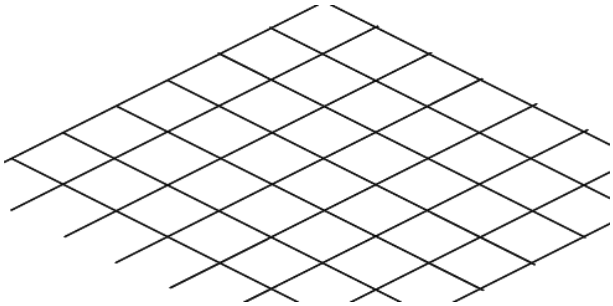

Abbildung 3.8: Sich überdeckende Ebenen

einem Punkt, wie bei der Isometrie 1 und der Dimetrie 1, so überdecken sich auch mehrere übereinandergestapelte Blöcke exakt entlang derselben Kanten und Punkte (Abb. 3.8).

3.2 Betrachtungswinkel

Der Betrachtungswinkel ist der Winkel der Projektionsstrahlen zur Grund(riss)ebene, welche durch die X- und Y-Achse definiert ist. Für eine Rücktransformation der Bild- zu Raumkoordinaten wird der Betrachtungswinkel benötigt. Bei der Rücktransformation ergibt die Multiplikation mit der Inversen der View-Projektionsmatrix den Ort, wo sich ein Pixel im world space befindet. Durch den Informationsverlust von 3D auf 2D finden sich die rücktransformierten mögliche Punkte auf einem Strahl. Um den Punkt auf dem Strahl eindeutig zu identifizieren, muss dieser mit der Welt-Geometrie geschnitten werden.

Die zur projizierten Diagonalen g äquivalente Strecke im Raum hat die Länge $\sqrt{2}$. Der Faktor ist daher $e_{xy} = \frac{g}{\sqrt{2}}$. Es folgt der Ansatz zur Berechnung des Betrachtungswinkels δ zur Grundrissebene:

$$tan(\beta) = \frac{e_{xy}}{e_z} = \frac{g}{\sqrt{2} * e_z} \tag{3.1}$$

$$\Updownarrow$$

$$\beta = atan(\frac{\sqrt{2} * e_x * sin(\gamma)}{e_z}) \tag{3.2}$$

An der Formel 3.2 lässt sich sehen, dass das Verhältnis aus γ, e_z und e_z für den Betrachtungswinkel entscheidend ist. Transformiert man eine orthografische Projektion in eine schiefe Projektion, so verändert sicht nicht der Betrachtungswinkel. Der Beweis ist trivial: Bei einer schiefen Projektion wird g sowie e_z mit einer Konstante verzerrt, sodass sich diese Konstante nach der Gleichung 3.1 wieder herauskürzt. Verändert man die Achsenwinkel oder die Verzerrung der Achsen unabhängig voneinander, führt dies demnach zu einem anderen Betrachtungswinkel.

Schiefe Parallelprojektionen lassen sich durch lineare Verzerrung in orthografische überführen, denn nach dem Satz von Pohlke [BF91] kann man zu jeder Axonometrie eine ähnliche Projektion eines parallel projizierten orthonormalen Achsenkreuzes erhalten. Anschaulich lässt sich dies sehen, wenn man das erzeugte Bild verzerrt. Der Betrachungswinkel bleibt dabei der Gleiche. Wenn die Schräge der Projektion bekannt ist, kann diese Verzerrung rückgängig gemacht und aus der nun bekannten Verzerrung der Achsen (durch Ablesen am Einheitswürfel) mittels der Gleichung 3.2 der Betrachtungswinkel α errechnet werden.

Für die Standard-Isometrie (Isometrie 1) mit $\gamma = \tau/12, e_z = 2^{-1/2}$ erhalten wir:

$$\beta = atan(\frac{sin(\tau/12)}{e_z}) = atan(2^{-1/2}) = 35,26^\circ$$

Für die Pixelgrafik-Isometrie (Isometrie 2) mit $\gamma = atan(1/2)$:

$$\beta' = atan(\frac{sin(atan(1/2))}{e'_z}) = 32,311^\circ$$

Und für die verzerrte Pixelgrafik-Isometrie mit $\gamma = \tau/12$, $e''_z = \sqrt{2}sin(atan(1/2))$ (Dimetrie 1) erhalten wir:

$$\beta'' = atan(\frac{sin(atan(1/2))}{e''_z}) = atan(\frac{sin(atan(1/2))}{\sqrt{2} * sin(atan(1/2))}) = atan(2^{-1/2}) = 35,26^\circ$$

Die Pixelraster-Isometrie mit Überdeckung (Dimetrie 1) kann also auch als schiefe Projektion der Standard-Isometrie aufgefasst werden.

3.3 Projektionsmatrizen

In der Computergrafik benutzt man eine Projektionsmatrix M, um Punkte p mit einer Matrixmultiplikation in einen anderen Raum S zu projizieren: $S = M * p$. Eine wichtige Anwendung davon ist die Überführung der Raumkoordinaten in Bildkoordinaten.

In der Wurfel Engine werden die Raumkoordinaten in einem anderen Koordinatensystem beschrieben. Das benutzte Koordinatensystem hat den Vorteil, dass eine Umwandlung in die Projektion einfacher bzw. intuitiver ist, da die X-Achse in Weltkoordinaten in die Bildebene im Verhältnis 1:1 übertragen wird.

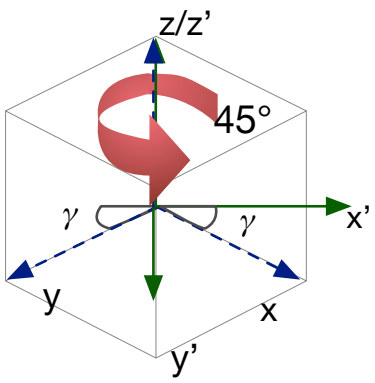

Abbildung 3.9: Rotiertes Koordinatensystem des Wurfel Engine

Im Folgenden wird die Projektion aus diesem Koordinatensystem genauer vorgestellt und untersucht. Modelliert werden die Koordinaten im world space in einem um die Z-Achse um 45° rotierten Koordinatensystem (vgl. Abb. 3.9). Die Axonometrie wird also auf das

neue Koordinatensystem übertragen, wobei die Darstellung die gleiche bleibt. Zu einer Umwandlung von Koordinatensystemen kann eine Basistransformation durchgeführt werden.

Die Projektionsmatrix kann auch direkt gebildet werden, ohne Basistransformation. Der Faktor entlang der projizierten Vertikalen ist wie bereits erwähnt $e_{xy} = \frac{g}{\sqrt{2}}$. Die Y-Achse der Spielwelt wird nach unten entlang dieser abgetragen: $e_y^{(IV)} = -e_{xy}$. Die Verzerrung auf der Z-Achse e_z bleibt von der Rotation unberührt. Die Y- und Z-Achse teilen sich in der Projektion eine Linie bzw. Achse, können also addiert werden. Die Tiefe wird bei der Projektion einfach übernommen. Für die Projektion der X-Achse gilt das Gleiche wie bei der Y-Achse, nur dass der Winkel γ um $45° = \tau/4$ versetzt ist. Die $sin()$-Funktion kann einfach durch die $cos()$-Funktion ersetzt werden, denn $sin(x + \tau/4) = cos(x)$. Es ergibt sich die Projektionsmatrix:

$$M = \begin{pmatrix} \frac{2e_x cos(\gamma)}{\sqrt{2}} & 0 & 0 \\ 0 & \frac{-2e_x sin(\gamma)}{\sqrt{2}} & e_z \\ 0 & 0 & 1 \end{pmatrix}$$

Wir wollen die X-Achse unverzerrt, also mit Faktor 1, projizieren lassen, skalieren also dahingehend die Projektion:

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -tan(\gamma) & \frac{e_z\sqrt{2}}{2e_x cos(\gamma)} \\ 0 & 0 & \frac{\sqrt{2}}{2e_x cos(\gamma)} \end{pmatrix}$$

Für die vorgestellten Projektionen ergibt sich daraus:

Name	Isometrie 1	Isometrie 2	Dimetrie 1
γ	$\tau/12$	$atan(1/2)$	$atan(1/2)$
M	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -0,57 & 0,816 \\ 0 & 0 & 1,15 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -0,5 & 0,79 \\ 0 & 0 & 1,12 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -0,5 & 2^{-1/2} \\ 0 & 0 & 1,12 \end{pmatrix}$
β	$35,26°$	$32,311°$	$35,26°$

Natürlich ist man nicht auf die Wahl dieser drei Projektionen beschränkt. Aus den beschrieben Gründen empfiehlt sich aber die Verwendung dieser. Die Wahl der Projektion ist eine Designentscheiden, deren Rolle wir nun genauer betrachten.

4 Datenstruktur

4.1 Modellierung der Spielwelt

Die Wahl der Modellierung der Spielwelt ist fundamental bedeutend für die Grafik und die Möglichkeiten in einem Spiel. Viele Game Engines versuchen die Natur möglichst exakt abzubilden, wobei andere Spiele aus technischen Gründen und aus Designentscheidungen bewusst Einschränkungen als Gestaltungselement nutzen. Genau dies wurde bei der Wurfel Engine getan. Die drei-dimensionale Landschaft der Spielwelt ist in ein Gitter unterteilt, und besteht nur aus Blöcken, welche die Landschaft formen.

Technisch kann diese Modellierung der verschiedenen Blöcke durch die Verwendung einer 8-bit-Zahl pro Blocktyp geschehen. Damit ist man auf $2^8 = 256$ Blockarten begrenzt. Jeder Block wird über eine Koordinate identifiziert. Der Speicherort eines Blocks mit einer Koordinate nennen wir eine Zelle.

Virtuelle Welten enthalten für gewöhnlich nicht nur Landschaftsdaten, sondern Objekte, die sich frei innerhalb der Zellen bewegen können, sogenannte Entities. Entities können Spielfiguren, Deko-Objekte oder Grafikeffekte wie Partikeleffekte etc. sein. In der Wurfel Engine werden Entities in einer Liste separat gespeichert. Ein Random-Access-Zugriff auf n Entities ist daher im Schnitt nur mit einer Laufzeit $n/2$ möglich. In der Praxis wird Random-Access selten durchgeführt, und der Hauptteil der Zugriffe iteriert über alle Objekte der Liste, z. B. für Aktualisierungen in jedem Frame.

Die virtuelle Welt, oft genannt „Karte“ oder engl. „Map“, soll beliebig groß sein können. Da der Computerspeicher aber limitiert ist, gibt es zwei Möglichkeiten: eine Einteilung in Abschnitte (engl. Levels) oder das Streaming der relevanten Inhalte. Die Wurfel Engine benutzt ein Streaming-Konzept, bei dem die Karte in gleich große Abschnitte eingeteilt

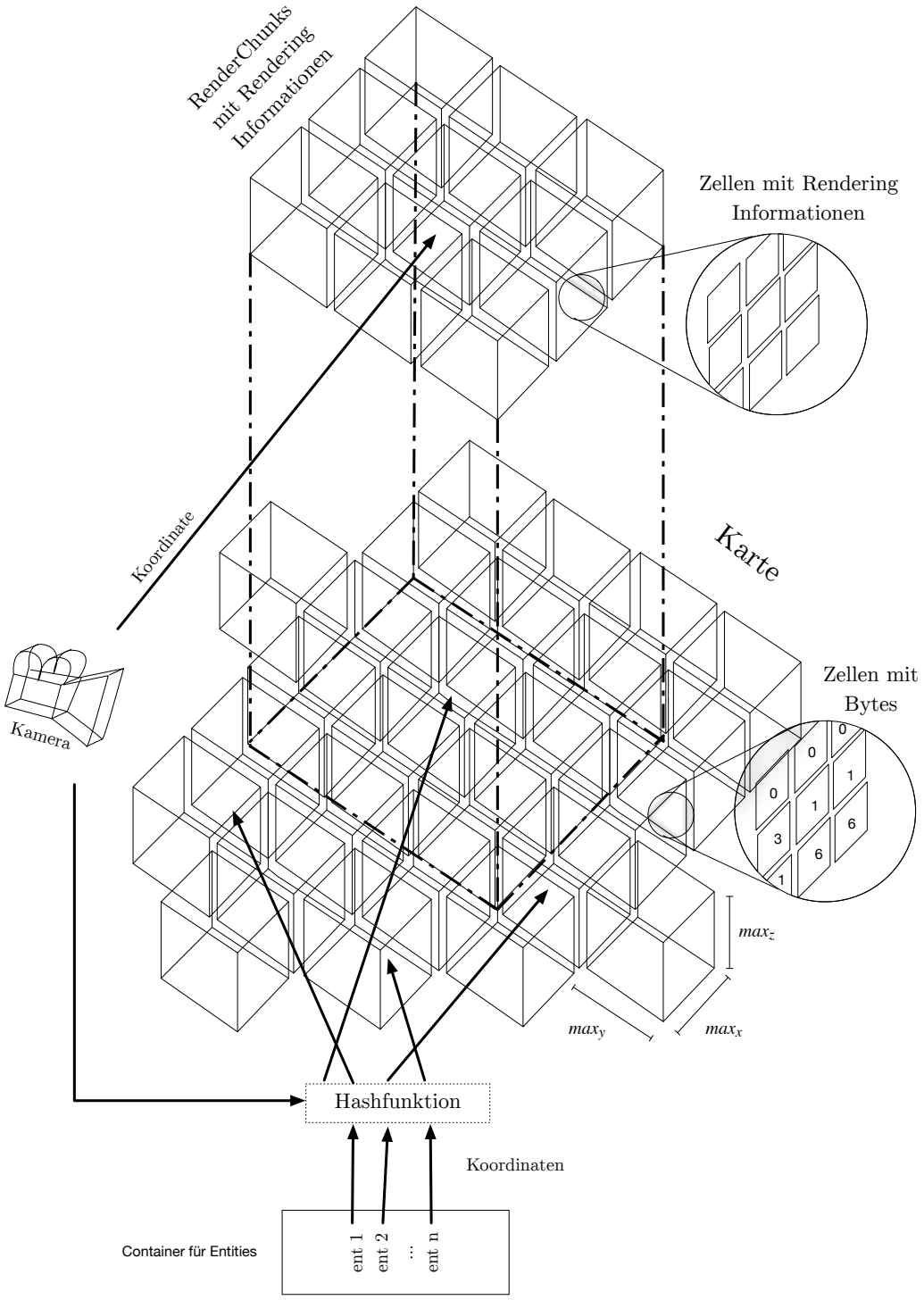

Abbildung 4.1: Ein Schaubild, welches die Datenstruktur zeigt.

wird, sogenannte „Chunks“ (engl. Stück, Brocken). Die Blöcke der Chunks werden intern in dreidimensionalen Arrays gespeichert. Die Koordinaten lassen sich so in die Adressierung umwandeln, sodass eine Zugriffszeit mit $O(1)$ möglich ist. Der Zugriff auf die Chunks via einer Hashmap ist ebenfalls mit $O(1)$ möglich. Es können sowohl der Indexbereich der Hashmap als auch der maximale Speicherverbrauch durch Chunks konfiguriert werden.

Das Schaubild 4.1 zeigt die Datenstruktur.

Damit alle Rendering-Algorithmen auf weit weniger Chunks arbeiten, werden dafür nur die Chunks in der Nähe der Kamera genutzt. Das sind drei Chunks in jeder Richtung. Diese 3 mal 3, also 9 Chunks, werden separat mit allen Rendering-Informationen gespeichert. Dies Begrenzung auf die umliegenden Chunks begrenzen die Möglichkeiten herauszuzoomen.

Die Dimension der Chunks kann je nach Anforderung pro Karte geändert werden. Die Standard-Werte sind $max_x = 10$, $max_y = 40$, $max_z = 10$. Größere Werte vergrößern die benötige Zeit für viele Algorithmen, denn die Menge der Zellen wächst kubisch mit der Verlängerung der Seitenkanten.

4.2 Rendering

Bei Echtzeitanwendungen werden in einer Schleife aufeinanderfolgend die Inhalte aktualisiert, und danach wird der Frame gerendert. Bei Spielen wird dies *game loop* genannt. Die Aktualisierungsrate vieler Monitore und bei Computerspielen von 60 Herz ist quasi der Standard. Damit eine Anwendung Bilder mit einer Frequenz von 60 Bildern pro Sekunde erzeugen kann, darf die Berechnung eines Frames maximal $\frac{1}{60}s = 16,6ms$ dauern.

Die Wurfel Engine benutzt keine 3D-Modelle für die Entities. Da die Objekte immer unverzerrt frontal abgebildet werden, können diese in einem 3D-Programm, z. B. Blender, erstellt werden und dann als sog. Billboards [AMH02, p. 320] gerendert werden. Diese Technik hat den Vorteil, dass keine Aliasing-Effekte auftreten und die Qualität des Bildes via Raytracing einem in Echtzeit erzeugtem Bild deutlich überlegen ist. Ein Nachteil

ist, dass so einige (Geometrie-)Informationen fehlen. Diese können aber mit dem Gebrauch von weiteren überlagerten Billboards ergänzt werden (z. B. depth map, normal map).

Da jedes Objekt nicht teure API-Befehle zum Rendering (einen *Drawcall*) benötigen soll, werden mehrere Objekte zusammengefasst. Ein Batch sammelt zu zeichnende Objekte und macht daraus ein großes Geflecht von Vertices, das Mesh, was mit einer Textur gerendert wird. Alle Sprites müssen also in einer Textur zu finden sein, genannt *Spritesheet*. Da Texturen in der Größe begrenzt sind, beschränkt dies die Anzahl der möglichen Billboards. Die Größe ist abhängig von der Hardware. Der Wert kann mit dem OpenGL-Befehl mit *GL_MAX_TEXTURE_SIZE* ausgelesen werden.

Weitere Texturen benötigen weitere zeitaufwändige Änderungen von Render States und Drawcalls. Die Nutzung von mehreren Texturen z. B. über Texture Arrays erlaubt die parallele Benutzung von mehreren Spritesheets. Damit sollte es möglich sein, mehr Sprites mit einem texture bind, also einer Zuweisung einer Textur an eine Textureinheit, zu benutzen.

Um die Geschwindigkeit zu erhöhen, werden Objekte vor der Rasterung überprüft: Liegen sie außerhalb des sichtbaren Bereiches der Kamera, werden sie nicht gerendert (visibility culling, frustum culling [AMH02, p. 357]).

Für weitere Geschwindigkeits-Optimierungen filtert ein Algorithmus Seiten von Blöcken heraus, die nicht sichtbar sind (occlusion culling) [AMH02, p. 368], sodass unnötiges Übermalen (overdraw) vermieden werden kann. Damit dies möglich ist, werden Blöcke auch nicht als Ganzes gerendert, sondern die drei sichtbaren Seiten sind im Spritesheet als einzelne Sprites gespeichert und werden zur Laufzeit zusammengesetzt. Es gibt aber auch Zellen, die als Ganzes gerendert werden müssen, da sie keine Blöcke aus drei Seiten enthalten. Zellen können beliebige Sprites rendern. Diese Sprites werden auch unzerteilt im Spritesheet gespeichert.

Überdecken sich die Ebenen genau, so ergibt sich ein Raster aus vielen Rauten (vgl. 3.8). Man kann erkennen, dass sich die vielen Rauten in doppelt so viele Dreiecke halbieren lassen. Hier kann man den von mir vorgestellten Raytracing-Algorithmus zum Culling von Blockseiten verwenden [Vog13]. Die Wurfel Engine wechselte im Laufe ihrer Entstehung von der Dimetrie 1 zur Isometrie 2, sodass sich die Zellen nun nicht mehr genau

überdecken. Damit wurde auch der Code für diesen Algorithmus entfernt. Eine alte Implementation befindet sich im Anhang der Arbeit (A.1).

4.3 Implementation

Die Wurfel Engine soll plattformunabhängig (d.h. für Windows, Linux, macOS tauglich) sein und nutzt deshalb die Programmiersprache Java. Java wird von einer virtuellen Machine (JVM) interpretiert, sodass Java-Code auf allen Geräten läuft, die eine JVM laufen lassen können. Für die Hardware-Beschleunigung durch Grafikhardware wird OpenGL 2 benutzt. Für die Verwaltung der low-level-Pipeline wird die Open Source Bibliothek libGDX[1] verwendet. LibGDX baut wiederum auf der Bibliothek LWJGL auf und bietet damit über eine Java-API auch Zugriff auf OpenGL. LibGDX beschränkt sich auf die OpenGL ES Untermenge der OpenGL-Befehle, sodass die Software auch auf Android laufen kann.

LibGDX benutzt eine Klasse, die einen Sprite-Batch implementiert.[2] Ein SpriteBatch ist eine Klasse, die das Rendering von vielen Sprites sammelt und gebündelt an die Grafikhardware übergibt. Diese Klasse lädt die Sprites aber nur im projection space, also mit 2D-Informationen, auf die Grafikkarte. Man modifiziert daher den Code, sodass die Koordinaten im world space hochgeladen werden und die Projektion mit der Projektionsmatrix im vertex shader angewandt wird. Dies hat den Vorteil, dass nun im Fragment Shader die Koordinaten als Raumkoordinaten vorhanden sind. Dadurch können viele Effekte wie z. B. Nebel oder lokale Beleuchtung einfach per Fragment berechnet werden.

[1] `http://libgdx.badlogicgames.com`
[2] Source Code der Klasse com.badlogic.gdx.graphics.g2d.SpriteBatch: `https://github.com/libgdx/libgdx/blob/master/gdx/src/com/badlogic/gdx/graphics/g2d/SpriteBatch.java`

5 Sortierverfahren

5.1 Korrekte Reihenfolge

Um die Reihenfolge im Maleralgorithmus zu bestimmen, überführen wir ein Datenformat, in dem die Objekte gespeichert sind, in eine geordnete Liste. Die Reihenfolge in der Objekte gerendert werden, ergibt sich aus einer Vielzahl von Abhängigkeiten. Einige Objekte müssen vor anderen gerendert werden. Diese Abhängigkeiten lassen sich daher gut im Modell eines gerichteten Graphs, den Verdeckungsgraph, modellieren. Ein Knoten beschreibt dabei ein zu renderndes Element und die Kanten die Elemente, die zuvor kommen.

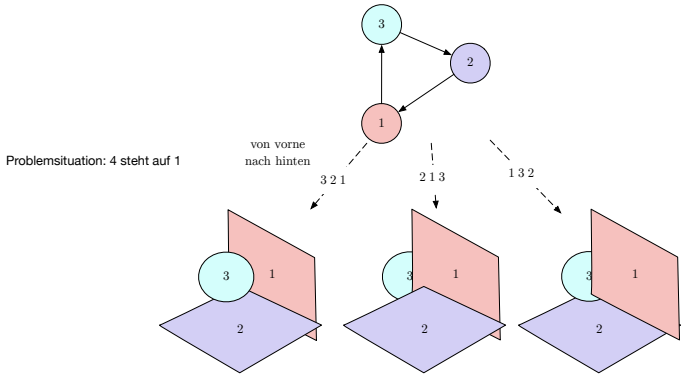

Abbildung 5.1: Unterschiede in der relativen Reihenfolge und in deren Resultat.

Betrachten wir nun ein Szenario, was so in Blockwelten häufig vorkommt (Abb. 5.1). Nehme man an, wir hätten zwei Blöcke (1 und 2). Hinter Block 1 befindet sich ein Entity (3). Die Grafik dieses Entitys ragt zudem über Block 2 hinüber, welcher sich davor eine Ebene tiefer befindet. Wenn nun die Abhängigkeiten in diesem Szenario in einen Graph überführt wird,

erhalten wir einen Graph mit Zyklus. Kann ein Graph mit Zyklus eine eindeutige Reihenfolge bzw. Sortierung haben? Nein, denn die Relation ist nicht transitiv: Das Ergebnis der Reihenfolge hängt vom Objekt ab, womit begonnen wird.

Hinsichtlich der Wahrnehmung durch die verschiedenen möglichen Reihenfolgen gibt es Unterschiede (vgl. Abb. 5.1). Die Reihenfolge von vorne nach hinten (3, 2, 1) verwirrt am stärksten, denn hier erscheint ein Objekt vor einer Kante, obwohl es sich eigentlich hinter dieser befindet. Die Reihenfolge (2, 1, 3) hat den Fehler, dass das Entity unter dem Boden gerendert wird und damit tiefer erscheint, obwohl es sich über diesem befindet. Die Reihenfolge (1, 3, 2) erzeugt ein scheinbar akzeptables Bild. Hierbei ist der vordere Block vom hinteren verdeckt, was in dieser abstrakten Darstellung nicht negativ auffällt. Was als akzeptabel angesehen wird, ist also auch von den verwendeten Grafiken abhängig. Wenn sich die Blöcke streng auf ihr Raster beschränken und nicht überdecken, so ist die Reihenfolge (1, 3, 2) korrekt.

5.2 Trivialer Algorithmus

Ein Großteil der isometrischen virtuellen Welten benötigen keinen komplizierten Algorithmus. Die Sortierungen geschieht einfach von hinten nach vorne anhand der Objektkoordinaten. Wenn wir unser Koordinatensystem von zuvor benutzen, geschähe die Sortierung erst entlang der X-Achse, dann nach der Y- und zuletzt nach der Z-Achse.

```
for z
  for y
    for x
      draw(x,y,z)
```

Da die Welt auch noch Entities enthält, werden diese ebenfalls in die Reihenfolge eingefügt. Iteriert man über die Blöcke, so müssen hiervor die Entities in die Reihe der Blöcke einsortiert werden. Zur Identifikation der Position der entsprechenden Zelle kann die Oberkante des Entities benutzt werden.

Wenn man sich auf zwei Ebenen beschränkt, kann die Zuweisung der Entities zu einer jeweiligen Z-Ebene geschehen, und dadurch werden Entities innerhalb der Z-Ebene nur nach der Y-Koordinate sortiert.

In beiden Fällen ist die Laufzeitklasse $O(N)$ für N Elemente.

Dieser Ansatz funktioniert für komplexe 3D-Szenen nicht. Eine zu hohe Komplexität gibt es beim Gebrauch von mehr als drei Ebenen auf der Z-Achse oder vom Raster losgelösten Z-Positionen. Beim trivialen Ansatz werden die Achsen voneinander isoliert betrachtet. Beispielsweise liefet der Algorithmus falsche Ergebnisse, wenn ein Objekt B tiefer auf der Z-Achse als ein Objekt A positioniert ist, aber B sich auf der Y-Achse weiter in Richtung der Kamera als A befindet (vgl. Abb. 5.2). Im Rendering wird B dann fälschlicherweise von A verdeckt (vgl. Abb. 5.3).

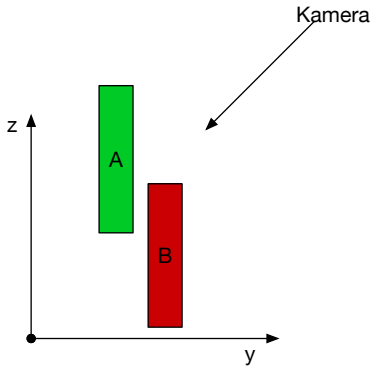

Abbildung 5.2: Seitenansicht zweier Objekte.

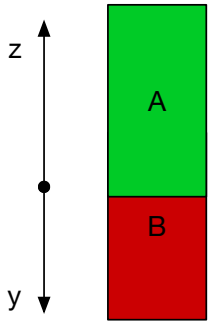

Abbildung 5.3: Ergebnis des Maleralgorithmus nach YZ.

Die Zellen, über die iteriert wird, liegen im Speicher nicht hintereinander in einer Liste. Mit einer Zwischenspeicherung in einer Liste kann nach Messungen ein Geschwindigkeitsvorteil pro Frame von etwa 600 ms erreicht werden. Diesen Geschwindigkeitsvorteil können alle Verfahren nutzen, welche über alle Zellen iterieren.

5.3 Sortierung mittels Tiefenwert

Wichtiges Entscheidungskriterium für die Sortierung von Objekten oder Fragmenten ist die Distanz zur Kamera. Der Abstand zur Kamera, der Tiefenwert, lässt sich aus der Position der Objekte berechnen. Anhand dieses Tiefenwertes kann dann sortiert werden. Da die Kamera durch die orthografische Projektion im Unendlichen liegt, wird einfach reziprok die Entfernung zum Boden berechnet und verwendet.

$$depth = e_{xy} * y + e_z * z$$

Die Laufzeit mit einem generischen Sortieralgorithmus für N Elemente ist in $O(N * log(N))$. Durch die eher regelmäßige Verteilung der Blöcke kann mit speziellen Sortieralgorithmen, wie z. B. Bucketsort[1] oder Counting-Sort[2], die Sortierung beschleunigt werden. Ebenfalls könnte die Frame-zu-Frame-Kohärenz ausgenutzt und ein Algorithmus genutzt werden, der von einer Vorsortierung stark profitiert.

Dieser Algorithmus hat den Vorteil, dass die Relation transitiv und damit eindeutig ist. Allerdings erzeugt er auch Fehler, wenn ein Entity nahe einer Blockkante steht. Da das Entity höher als der Block endet, ist auch der Tiefenwert größer, sodass es eine Grenze gibt, an der die Reihenfolge falsch ist (vgl. 5.4).

Abbildung 5.4: Die Figur müsste vom vorderen, dunklen Block verdeckt werden.

[1] Ein Sortieralgorithmus bei dem Elemente in geordnete „Buckets“ einsortiert und diese Behälter dann sortiert werden, vgl. `https://en.wikipedia.org/wiki/Bucket_sort`

[2] Ein Sortieralgorithmus bei dem der Index der Sortierung durch ein Histogram berechnet wird, vgl. `https://en.wikipedia.org/wiki/Counting_sort`

5.4 Sortierung mittels topologischer Sortierung

Ein ganz anderer Ansatz funktioniert mittels des Gebrauchs des in Kapitel 5.1 vorgestellten gerichteten Verdeckungsgraphs. Alle zu rendernden Objekte werden als Knoten eines Abhängigkeits-Graphs behandelt, und mit einer *topologischen Sortierung* wird aus der Menge der Objekte eine Ordnung erzeugt. Jedes Objekt speichert über gerichtete Kanten zu Nachbarknoten die Objekte, welche zuvor gerendert werden sollen.

Die Grundidee der topologischen Sortierung geht von zwei Phasen aus: In der ersten Phase wird der Graph geladen oder erstellt. Wie ist der Graph aufgebaut und welche Knoten enthält er? Mit Laufzeit $O(N^2)$ lässt sich feststellen, welche N Objekte von welchen verdeckt werden. Hierfür wird jedes Objekt mit jedem verglichen.[3] Da sich alle in einem Raster befindet, kann dies genutzt und nur direkte Nachbarzellen betrachtet werden. Das hat den negativen Effekt, dass leere Zellen, die nicht gerendert werden, als ein Knoten dem Graph hinzugefügt werden müssen und damit der Speicherverbrauch gegenüber anderer Methoden größer ist.

Zu einer Zelle (x,y,z) sind die vorherig zu rendernden hinteren Nachbarzellen, die sich schräg links, schräg rechts und direkt dahinter befinden, in der Ebene darunter, in der gleichen und in der Ebene darüber. Das sind zusammen neun Zellen / Kanten pro Zelle.

Neben den Blöcken müssen natürlich auch die Entities in den Graphen eingeordnet werden. Entities haben andere bzw. mehr Abhängigkeiten als Blöcke. Befinden sich mehrere Entities innerhalb einer Zelle, so teilen sie die gleichen Abhängigkeiten. Daher können diese zusammen in den Graphen mit einem Knoten einfließen. Wenn wir alle Entities innerhalb einer Zelle zusammen betrachten, können wir diese untereinander mittels der Tiefenzahl-Methode (5.3) sortieren, denn die Methode arbeitet hier fehlerfrei.

Wir wissen nun, wie der Graph konstruiert ist. Das Verfahren zur Erzeugung des Graphs mit Entities wird in Kap. 5.4.1 genauer betrachtet, weshalb wir an dieser Stelle davon ausgehen, dass der Graph vollständig im Speicher vorhanden ist. Nachdem der Graph

[3] Eine Untersuchung dieses Problems: `https://mazebert.com/2013/04/18/isometric-depth-sorting/`

in der ersten Phase gebaut wurde, muss in der zweiten Phase eine topologische Sortierung durchgeführt werden. Das Ergebnis der Sortierung ist dann die zu rendernde Reihenfolge.

Eine topologische Sortierung entspricht einer Tiefensuche. Eine Tiefensuche funktioniert nur bei kreisfreien Graphen. Wird ein Kreis bei der Traversierung entdeckt, so wird diese Kante ignoriert und damit nicht weiter in die Tiefe gegangen, sodass die Transitivität der Sortierung sichergestellt ist. Für die Detektion der Kreise muss der erfolgte Besuch eines Knotens gespeichert werden. Dafür wird eine Markierung in den Knoten gesetzt. Der Algorithmus im Pseudo-Code:

```
createGraph();

method visit(x,y,z){
  if (not marked):{
    mark as visited;
    for every neighbour cell:{
      visit()
    }
    append(x,y,z);
  }
}
```

Da der Graph nicht zusammenhängend sein muss, muss jede Zelle in einer äußeren Schleife einmal „besucht“ werden. So kann garantiert werden, dass der Algorithmus für alle Knoten aufgerufen wird. Der Pseudo-Code zur äußeren Schleife:

```
for x, y, z {
  vist(x,y,z);
}
```

5.4.1 Erstellen des Graphs

Den Graphen kann man nach dem Prinzip der *lazy evaluation* erzeugen: Ein Knoten wird erst dann angelegt, wenn er angefordert wird. Dadurch wird der Performance-Einbruch im Falle einer Neuerstellung etwas abgemildert, schließlich werden nicht alle Knoten besucht und damit nicht der gesamte Graph neu angelegt.

Auch bei dieser Sortiermethode ergibt sich wieder die Frage, wie Entities in den Graphen einzutragen sind. Die Sichtbarkeit von Blöcken verändert sich in der Regel nicht von Frame zu Frame. Daher können Subgraphen mit Nachbarzellen dauerhaft (bis zu einer Änderung) gespeichert werden. Im Gegensatz dazu ändert sich die Position von Entities sehr häufig, i. d. R. von Frame zu Frame. Daher werden Entities jeden Frame neu in den Graphen eingefügt. Über die Position der Entities werden die zugehörigen Knoten bestimmt. Dann nutzt man in jedem der Knoten neben einer Liste mit den Nachbarzellen eine weitere Liste für Entities, in welche die zu diesem Knoten gehörigen Entities in jedem Frame neu gefüllt werden. Nach dem Rendering des Frames werden alle diese Listen wieder geleert. Nun enthält die Datenstruktur alle benötigten Informationen, aber noch sind die Entities nicht über Kanten mit dem Rest des Graphs verknüpft. Wie muss der Graph für die Entities erweitert werden? Entities haben eine Zelle zugewiesen bekommen, über den Knoten, in dem sie sich befinden. Der Block der Zelle muss zuerst und danach die Entities in selbiger gerendert werden. Im Abhängigkeitsgraph kommen sie also vor der Zelle, in der sie positioniert sind.

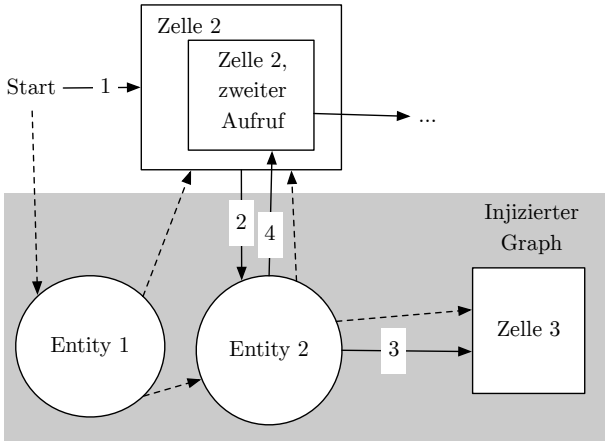

Abbildung 5.5: Visualisierung der Entity Injection in den Graphen. Rekursionsaufrufe durchgezogene Linie, Verdeckungsgraph gestrichelt.

Da der Graph über eine lazy evaluation entsteht, muss eine Injektion der Entity-Graphen in den Gesamtgraph zur Laufzeit der Rekursion geschehen (Abb. 5.5).

Dies ergibt den vollständigen Algorithmus:

```
method visit(x,y,z){
  if (not marked):{
    if (hasEntities and first Entity not marked){
      c = firstEntity.occludedNeighborCells
       for every c:
         visit(c.coordinates)
       append(entities in this cell);
       return;//normalen Aufruf beenden, da Graph injiziert wurde
     }
     markiere als besucht;
     for every occluded neighbor cell c:
      visit(c.coordinates)
    append(x,y,z);
  }
}

method depthsort(){
   invertMark();

   listeZellenMitEntities = neue Liste
   for each zu renderndes Entity e: {
      e in Graph eintragen
      zelle = e.getCoordinate();
      listeZellenMitEntities.add(zelle)
   }

   for x, y, z: {
      vist(x,y,z);
   }

   for each listeZellenMitEntities z:{
      in z: leere Liste mit Entities;
   }
}
```

Veränderungen von Blöcken Grundsätzlich gilt, dass in jedem Frame so wenig wie möglich neu berechnet werden sollte, bis auf die Dinge, die sich von Frame zu Frame ändern. Die Blöcke in den Zellen der Chunks verändern sich in der Regel selten, allerdings

können sie sich auch in jedem Frame verändern. Mit diesen Veränderungen kann man auf mindestens vier Arten umgehen:

1. Den Graphen bei Veränderung neu erstellen (selbe Laufzeitklasse wie beim Ablaufen, aber teurere Operationen).
2. Sind die Knoten und Zellen zusammen in der gleichen Struktur gespeichert, müssen bei Veränderungen an den Zellen über eine Suche entsprechende Knoten ($O(N)$) erst lokalisiert werden, bevor sie ausgetauscht werden können.
3. Den Graphen als bidirektionalen Graph konstruieren, sodass die Suche entfällt und verweisende Nachbarknoten so direkt gefunden ($O(1)$) und aktualisiert werden können.
4. Die Suche kann vermieden werden, wenn der Graph separat in einer parallel zu den Zellen existierende Datenstruktur gespeichert wird. Der Graph speichert dann in den Knoten Verweise auf die Zellen und umgekehrt die Zellen auf die Knoten. Bei Veränderungen kann so der entsprechende Knoten direkt gefunden werden und seine Referenz auf die neue Zelle gesetzt werden.

Die letzte Option ist jene, die in der Wurfel Engine implementiert wurde.

5.4.2 Optimierungen der „Besucht“-Markierung

Bei gängiger Konfiguration (40*10*10 Blöcke je Chunk, mit 9 aktiven Chunks, also 36,000 Blöcke) beträgt die Zeit, die nur für das Zurücksetzen der Markierungen benötigt wird, etwa 820 ms, also fast 5 % der pro Frame maximal zur Verfügung stehenden Zeit (bei 60 Bildern pro Sekunde). Diese Zeit kann mit einer Optimierung fast komplett eingespart werden. Damit die Markierung nicht für jeden Frame in jedem Knoten neu initialisiert werden muss, kann hierzu einfach ein globales Bit benutzt werden, welches in jedem Frame invertiert wird. Dieses Bit wird zur Feststellung des Besucht-Zustandes herangezogen. Durch den zusätzlichen Vergleich gibt es natürlich einen minimalen Mehraufwand im Algorithmus, welcher aber vernachlässigbar ist.

Damit neue Objekte nicht einen Frame zu spät angezeigt werden, muss das Inverter-Bit in neuen Objekten die richtige Invertierung haben, also dem Vergleichsbit nach passend initialisiert werden.

Die Markierung muss für jede Kamera separat erfasst und darf nicht zwischen ihnen geteilt werden, da sonst nicht besuchte Knoten in den folgenden Kameras veraltet sind. Wenn ein Byte als Träger für diese Boolean-Werte herangezogen wird, können bis zu acht Kamera-Markierungen in nur einem Feld gespeichert werden:

```
public final boolean isMarkedDS(final int cameraId) {
  return ((marked >> cameraId) & 1) == ((currentMarkedFlag >> cameraId)
    & 1);
}
```

5.4.3 Optimierungen und Laufzeit

Das Einfügen der E Entities jeden Frame geschieht in der Laufzeitklasse $O(E)$. Die Erzeugung der Teilgraphen geschieht auf Anforderung und fließt daher in die Laufzeitberechnung als Konstante ein und verändert damit nicht die Komplexitätsklasse.

Die Laufzeit der Tiefensortierung (im Pseudocode die Methode visit()) und damit der topologischen Sortierung beträgt $O(N + M)$[TW15, s. 98] mit N Knoten und M Kanten. Trotz der äußeren Schleife bleibt die Komplexitätsklasse linear, denn jeder Knoten wird maximal einmal besucht.

Der Algorithmus muss jeden Knoten einmal besuchen. Es kann so über die Knoten iteriert werden, sodass durch die folgende Rekursion alle Knoten einmal besucht werden. Da die Struktur der Teilgraphen bekannt ist, reicht es, über die obere und vordere Kante des mehrdimensionalen Arrays des sichtbaren Bereichs der Karte zu iterieren. Durch die Rekursion wird dann der gesamte sichtbare Bereich abgedeckt. Die äußere Schleife ruft n_s-mal die Sortierung auf. Ohne diese Optimierung ist $N_s = N$, mit ihr ist $N_s \ll N$. Es sollte möglich sein, diese Optimierung auch für andere axonometrischen Projektionen zu benutzen.

Es gibt Graphen, die sich durch die Anzahl der Kanten unterscheiden, aber dennoch nach der Sortierung das gleiche Ergebnis liefern. Formal ausgedrückt $\exists H : H \subseteq G \wedge s(G) = s(H)$, wobei s den Sortieralgorithmus und H und G die Graphen bezeichnen. Ein Beispiel findet sich in Abb. 5.6. Beide Graphen liefern durch die topologische Sortierung das äquivalente Ergebnis (3, 2, 1). Daher kann durch den Gebrauch von H eine Kante gespart werden. Im Beispiel ist dies die Kante zwischen 1 und 3. Diese Blöcke werden durch die

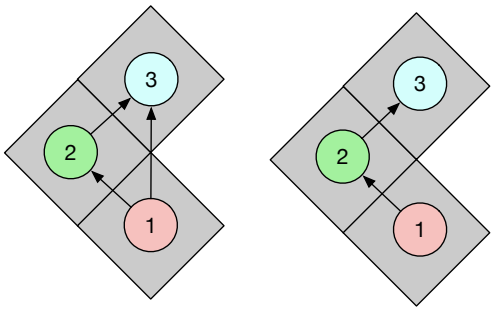

Abbildung 5.6: Zwei Graphen, welche das gleiche Ergebnis liefern.

Graphen der seitlichen Nachbarzellen erfasst (hier 2). Die seitlichen Blöcke werden auf jeden Fall zuvor gerendert. Durch diese Optimierung können die Kanten von $M = 9N$ auf $M = 6N$ optimiert werden.

Die Komplexitätsklasse ist dann mit der äußeren Schleife $O(N_s + 6N + N) = O(N)$. Mit dem Einfügen der Entities beträgt die Komplexitätsklasse des gesamten Algorithmus $O(E + N)$.

5.4.4 Ungelöste Probleme

Befinden sich zwei Entities in zwei benachbarten Zellen, so wird die Reihenfolge unter Umständen falsch bestimmt. Die Sortierung mit der Methode der Tiefenzahl erfolgt nur innerhalb einer Zelle. Befindet sich ein Entity vor einem anderen, ist aber einer hinteren Zelle zugeordnet, so ist die Sortierung falsch. Diese Fehler machen sich am stärksten bei Bewegungen bemerkbar. Den Wechsel der Reihenfolge macht sich als ein Flackern bei Partikeleffekten bemerkbar. Eine Mögliche Lösung gibt es durch eine veränderte Grapherstellung.

5.5 Sortierung mittels Tiefenbuffer

Der Tiefen-Test wird mit

```
glEnable(GL_DEPTH_TEST)
```

aktiviert. Neben dem Farbbuffer (color buffer) muss dann auch der Tiefenbuffer (depth buffer) jeden Frame mit

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

zurückgesetzt werden.

Der Schreibzugriff auf den Buffer darf nicht deaktiviert sein (glDepthMask(true)). Wie bereits beschrieben (Kap. 4.3) benutzt die Wurfel Engine eine modifizierte Klasse aus der libGDX 2D-Bibliothek. Die ursprüngliche Klasse deaktivierte bei jedem Frame den Schreibzugriff (vermutlich zur Verbesserung der Performance). Ohne Schreibzugriff schlägt jedoch das ganze Verfahren fehl. Deshalb wurden entsprechende Zeilen aus der modifizierten Klasse entfernt.

Prinzipiell können alle Objekte unsortiert gerendert werden. Fragmente, die den depth test bestehen, werden mittels einer Funktion zusammengeführt. Damit Entities keine schwarzen Flächen und Kanten im Bild an semi-transparenten Stellen erzeugen, wird erst die Landschaft gerendert und danach alle Entities. Die umgekehrte Reihenfolge ließe Entities erst rendern und damit an diesen Pixeln den Z-Buffer beschreiben. An diesen Stellen werden keine dahinterliegende Blöcke mehr gerendert. Da hierbei transparente Blöcke vor den Entities gerendert werden, führt dies zu Fehlern wie in Screenshot 5.7. Eine

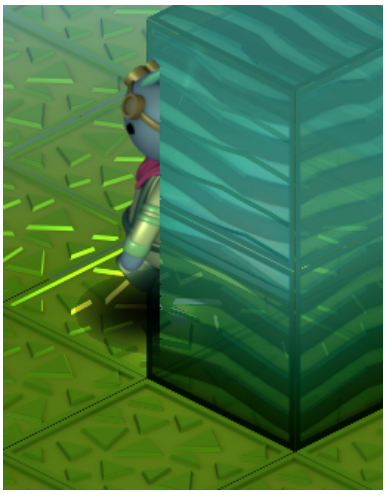

Abbildung 5.7: Figur wird hinter transparentem Block nicht gerendert

andere Reihenfolge kann dieses Problem lösen. Um aber diese Reihenfolge zu finden, muss man eine Objekt-Sortierung auf der CPU vornehmen, die die GPU-Sortierung überflüssig macht.

Abbildung 5.8: Entities schneiden darüber liegende Blöcke

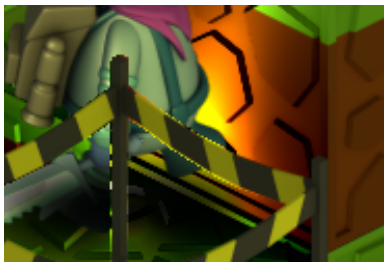

Abbildung 5.9: Vormals semi-transparente Kanten ergeben harte Kanten mit Aliasing-Effekt

Es ergeben sich zwei neue Klassen von fehlerhaften Darstellungen. Die erste Fehlerklasse ist sich schneidende Geometrie. In gewisser Weise wird das Bild korrekter, denn Schnitte in der Geometrie werden korrekt dargestellt. Sprites, die durch andere hindurchragen, sind an den entsprechenden Stellen dann sichtbar. Gerade Schnitte (Abb. 5.8) entstehen, da die Geometrie planar ist. Die geraden Schnittlinien können sich der Form des Objektes anpassen, wenn die Tiefenwerte durch Bumbpmapping[4] mit einer zusätzlichen Depth-Map präzisiert werden. Die Depth-Map fügt fehlende Tiefeninformationen in der Geometrie hinzu, und die Schnitte befinden sich dann an den Stellen, wo ein Objekt tatsächlich ein anderes durchragt. Mit der gleichen Technik ist Normal-Mapping[5] in der Wurfel Engine bereits implementiert. In den Fällen, wo sich ein Block über den Entities befindet, können die Vertices in der Größe begrenzt werden. Die zweite Fehlerklasse ergibt sich aus dem fehlenden Blending durch die stattfindenden Filterung von Fragmenten: Verläuft die Reihenfolge der Fragmente entgegen des Maler-Algorithmus, sodass ein vorderes Fragment

[4] Die Verwendung von Texturen, um hochaufgelöste Geometrie zu simulieren
[5] Das Hinzufügen der Normalen über eine Textur. Normal-Mapping kann für Licht-Berechnung benutzt werden.

vor einem hinteren gerendert wird, dann finden alle dahinterliegenden Fragmente keine Berücksichtigung mehr.

5.6 Sortierung mittels Depth-Peeling

Beim Depth-Peeling werden zwei Tiefenbuffers benötigt. Es wird mit einem regulären Tiefentest das vorderste Fragment bestimmt, also wird dieser gelesen und geschrieben. Der zweite Buffer besitzt die Tiefenwerte des vorherigen Layers und wird daher nur gelesen. Die gesamte Szene wird jetzt so oft gerendert, wie man transparente Ebenen hat. Nach jedem Durchgang werden die Buffer getauscht, so dass der zweite Buffer immer die Daten vom vorherigen Pass enthält. Der Buffer, der zuvor geschrieben wird, wird nun gelesen und zum Entfernen der Fragemente benutzt, welche sich davor, also näher zur Kamera, befinden. Das Bild wird also Ebene für Ebene von vorne nach hinten berechnet. Daher stammt die Bezeichnung „peeling“ (schälen).

Der Depth-Peeling Algorithmus in Pseudo-Code:

```
aktiviere depth test
deaktiviere blending
for i < n; i++:{
    benutze zum lesen frame buffer i % 2
    benutze zum schreiben frame buffer i % 2 +1
    render scene (color + depth) //ergibt i-te Ebene von vorne
    lade depth buffer i % 2 +1 als Textur auf GPU
}
lade blending shader
aktiviere blending
rendere gespeicherte Fragment-Ebenen von hinten nach vorne
```

Im benutzten Shader gibt es eine Erweiterung zum regulären Rendern. Als erstes werden die Tiefenbuffer-Werte aus der Textur mit der Tiefe des aktuellen Fragmente verglichen.[6] Mit dem Befehl *discard* werden dann die Fragmente entfernt, die Näher zur Kamera als die vorherige Ebene liegen.

[6] Beim Vergleich muss ein Epsilon-Vergleich durchgeführt werden (hier 0.00008), da Rundungsfehler zu schraffierten Flächen führen, welche falsch gefiltert werden. (Screenshot A.1)

Dual-Depth-Peeling kombiniert den Algorithmus in zwei verschieden Richtungen: Von vorne nach hinten und von hinten nach vorne. Depth-Peeling oder Dual-Depth-Peeling benötigt n geometry-passes, respektiv $n/2+1$, also Geometrie-Transformationen in den screen space. Ab vier Ebenen ist Dual-Depth-Peeling also schneller. In der Regel sind Blöcke opak. Der Regelfall in einem Spiel nicht mehr als maximal vier Ebenen. Aus diesem Grund reicht das einfache Depth-Peeling aus, was in der Wurfel Engine implementiert ist. Das Ergebnis sieht man in Abb. 5.10.

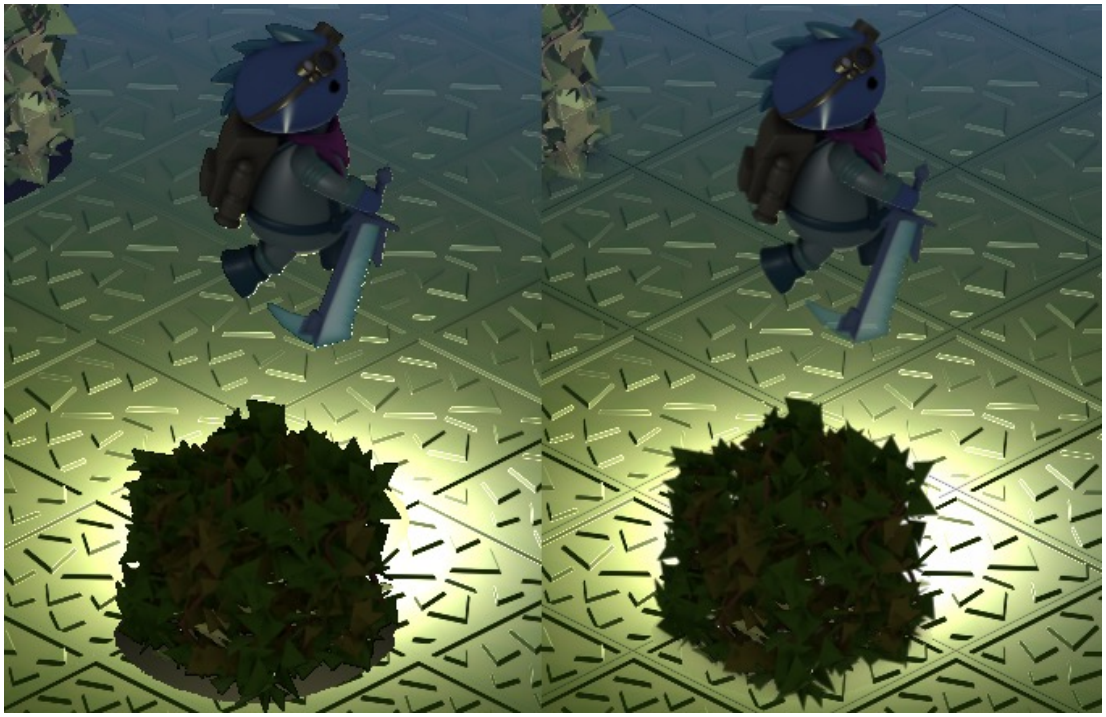

Abbildung 5.10: links nur der vorderste Layer, rechts zwei Layer

Beim Depth-Peeling wird die gleiche Geometrie mehrmals gerendert (Multi-Pass-Rendering). Deshalb wird nur ein mal die Liste der zu rendernden Objekte erstellt, und so muss für das gleiche Bild nicht die gesamte Rendering-Pipeline auf der CPU durchlaufen werden.

Das early-z Discarding beschleunigt das Rendering, indem Fragmente, die den Depth-Test nicht bestehen, vor Ausführung des Fragment-Shaders aus der Pipeline entfernt werden. Werden Fragmente im Fragment-Shader über den Befehl *discard* entfernt, so kann diese Optimierung nicht funktionieren, denn die Phase kann erst nach dem Fragment-Shader ge-

schehen [nvi08a]. Dies wird vermutlich auch für andere Grafikkartenhersteller gelten. Daher könne beim Depth-Peeling keine early-z discarding genutzt werden.

6 Auswertung

Eine Testszene wird genutzt, um die resultierenden Bilder aller Verfahren vergleichbar zu machen. Die Bilder befinden sich im Anhang (A.4, A.5, A.6, A.7, A.8).

Bei der Messung der Performance betrachten wir die Zeit in Millisekunden. Bei Echtzeitanwendungen wird oft die Anzahl der Frames pro Sekunde (FPS) als Metrik gewählt. Dieser Wert ist aber nur relevant, um festzustellen, ob Grenzwerte wie 30, 60 oder 120 erreicht werden. Ansonsten verhält sich per Definition die FPS nicht linear, da $FPS = 1/t_{Frame}$.

Ein Benchmark prüft verschiedene Stufen. Alle Stufen werden hintereinander mit der Kamera abgefahren. Nach einer gewissen zurückgelegten Strecke wird der Render-Cache neu gebaut, da sich die 3x3-Chunks verschieben müssen. Diese Verschiebung sollte zu einem kurzen Einbruch der Performance führen. Als Java virtual machine (JVM) wird die HotSpot-VM von Oracle genutzt. Ein internes Tool speichert die Zeit zwischen den Frames erst im Arbeitsspeicher und schreibt diese zwischen den Messungen auf den Festspeicher. Bevor die Messungen in einer Stufen gestartet werden, gibt es eine Verzögerung von 2 Sekunden, sodass die JVM Optimierungen vornehmen kann.

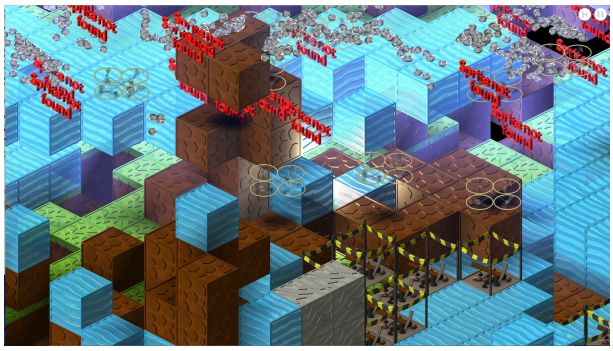

Abbildung 6.1: Stufe 4 mit vielen semi-transparenten Blöcken und sich bewegenden Sprites

Die Stufen beschreiben sich wie folgt:

Stufe 1 testet die Performance ohne Rendering.
Stufe 2 testet die Performance mit wenig Blockseiten.
Stufe 3 testet die Performance mit viel overdraw.
Stufe 4 testet die Performance mit overdraw und vielen sich bewegende Sprites (Screenshot 6.1).

Die erste Stufe ist nötig, um zu verifizieren, dass mit der Messung alles in Ordnung ist und keine Hintergrundprozesse die Leistung milden. Da hier gar kein Rendering stattfindet, dürften sich die Werte nicht zwischen den Verfahren stark voneinander unterscheiden. Stufe 2 hat keine Besonderheiten beim Rendering und zeigt eine flache Ebene mit Blöcken. Besonderheiten werden in den folgenden Stufen getestet. Es gibt viel mehr sichtbare Blockseiten und viele semi-transparente Pixel. Für die Verfahren, die den Tiefenbuffer nutzen, ist die Anzahl der Fragmente entscheidend. Die Laufzeit der anderen Verfahren ist abhängig von der Menge an Blöcken und Entities.

6.0.1 Ergebnisse der Messungen

Getestet wurde auf zwei Systemen im Vollbildmodus bei der Auflösung 1920 x 1080. Auf einem Apple MacBook Pro Early 2015 Intel Core i5 2,7 GHz mit macOS 10.12 in Tabelle 6.1

Tabelle 6.1: Messergebnisse Intel Core i5 (in ms)

Verfahren	Stufe 1	Stufe 2	Stufe 3	Stufe 4
keine Sortierung	0.93	3.07	4.24	8.66
Tiefenwert	0.96	3.16	4.37	8.86
topologisch	0.99	4.27	5.69	10.36
Tiefenbuffer	0.98	3.84	5.30	9.31
Depth Peeling	1.01	7.24	10.79	15.46

Mit einem AMD FX-4100 Quad-Core-Prozessor und Grafikkarte AMD Radeon R9 200 auf Window 10 in Tabelle 6.2.

Tabelle 6.2: Messergebnisse AMD FX + AMD Radeon R9 200 (in ms)

Verfahren	Stufe 1	Stufe 2	Stufe 3	Stufe 4
Tiefenwert	0.38	1.84	2.00	14.14
topologisch	0.39	1.98	2.24	14.55
topologisch	0.39	2.55	2.78	15.61
Tiefenbuffer	0.42	1.87	2.02	14.06
Depth Peeling	0.43	2.20	2.53	14.59

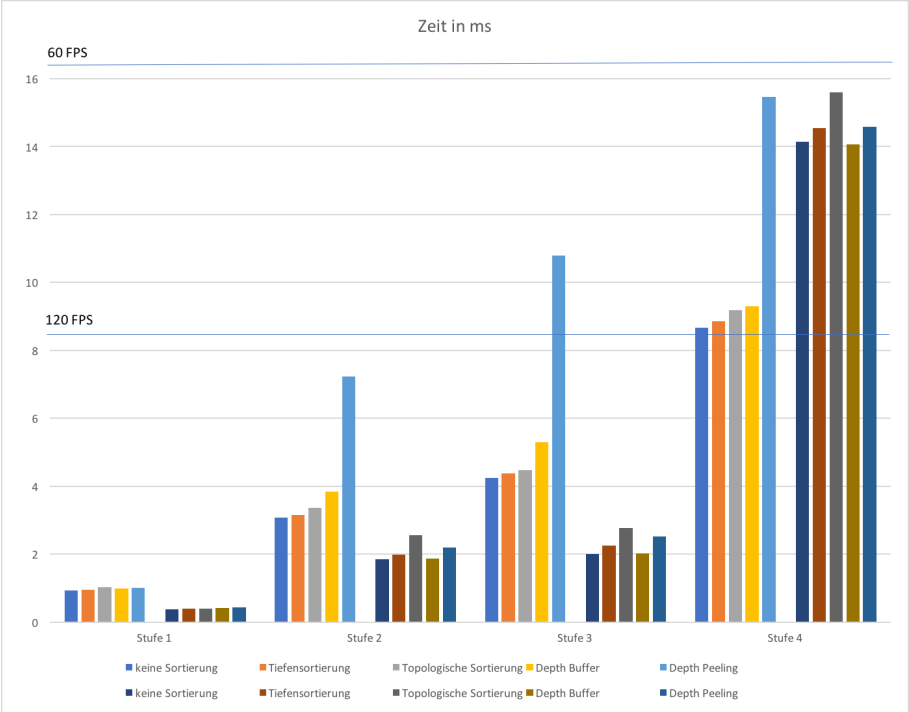

Abbildung 6.2: Ergebnisse im Balkendiagramm, helle Farben MacBook Pro, dunkle AMD FX + AMD Radeon R9 200

Die Daten sind in Abb. 6.2 visualisiert. Wenn gar keine Sortierung statt findet, gibt es natürlich die schnellstmögliche Laufzeit. Dies kann man als Referenzwert für einen relativen Vergleich heranziehen (vgl. Abb. 6.3).

Auffällig ist, dass die Geschwindigkeit aller Verfahren in der letzten Stufen auf dem AMD-Rechner deutlich unterliegt. Eine mögliche Ursache ist die schlechtere CPU-Leistung bei Single-Thread Anwendungen. Laut der Website cpuboss.com ist die single-core Bewertung beim Intel Prozessor 40% besser [cpu]. Die Anforderungen an die CPU sind in dieser

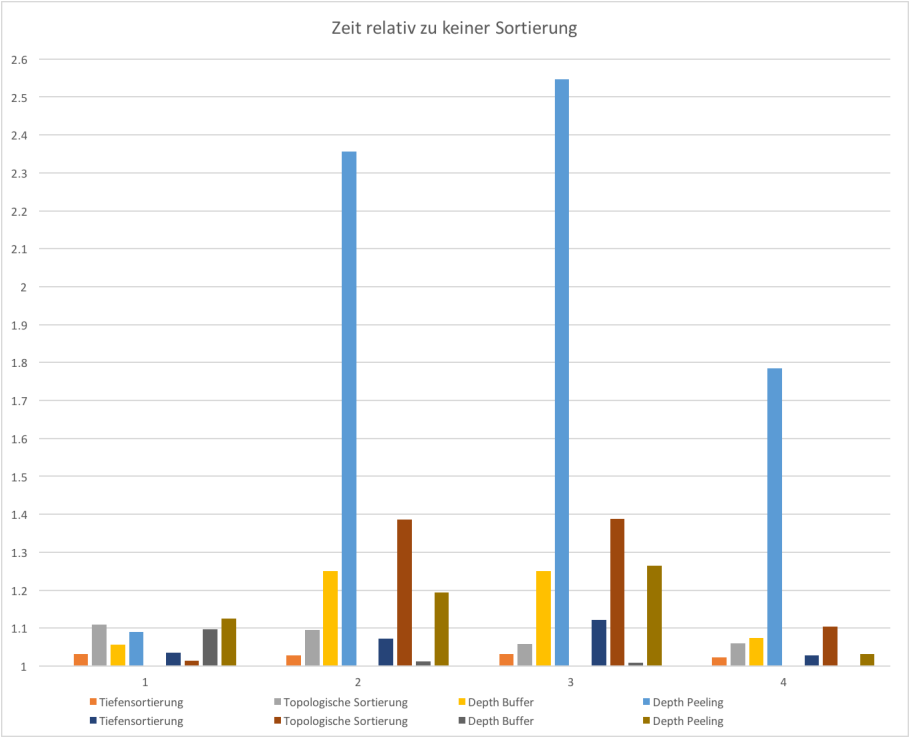

Abbildung 6.3: Relativ zu „keiner Sortierung“

Stufe am höchsten. Im Vergleich zwischen den Systemen zeigt sich, dass die dedizierte Grafikkarte die Verfahren mit Depth Buffer und Depth Peeling relativ besser laufen als die topologische Sortierung, welche die CPU mehr belastet.

6.0.2 Diskussion

Um den besten Algorithmus zu finden, müssen die Genauigkeit des resultierenden Bildes und die Performance bei der Erstellung dieser miteinander verglichen werden.

Schaut man nur auf die Performance, so ist die Sortierung mit dem Tiefenwert das beste Verfahren.

Bei der topologischen Sortierung (t. S.) werden für die Entities Container-Knoten benötigt, in die sie in den Graph eingefügt werden können. Deshalb werden verschiedene Knoteninstanzen für jede leere Zelle benötigt. Wenn darauf verzichtet wird, kann eine einzige geteilte Objektinstanz dafür benutzt werden. Es könnte möglich sein, dass auch eine Reduktion des Graphs effizient durchzuführen ist. Wenn ein Knoten selber nicht gerendert

wird, können alle seine Abhängigkeiten in den vorherigen Knoten gespeichert werden. So wird bei der Rekursion eine Rekursionsebene übersprungen. Abseits davon sehen wir kein großes Potential, die Performance deutlich zu verbessern.

Die Fehler der Reihenfolge bei der t. S. sind bei Bewegungen von Entities, abhängig von der Situation, sehr schwerwiegend.

Die Tiefenwertsortierung könnte mit einem speziellen Sortieralgorithmus noch beschleunigt werden. Da die Geschwindigkeit sich aber nur marginal vom Verfahren mit keiner Sortierung unterscheidet, ist die Beschleunigung hier praktisch unnötig.

Bei der Implementation des Depth Peeling (D. P.) wurden eventuell Fehler gemacht, welche die Performance unnötig beschränken. Die Performance des D. P. muss auf jeden Fall schlechter als mit einem einzigen Pass mit dem Tiefenbuffer sein.

Diskussion des Renderings Das Auslassen einer Sortierung und die Tiefenwertsortierung unterliegen aufgrund der visuellen Fehler den anderen Verfahren. Wenn keine Transparenz genutzt wird, kann der Tiefenbuffer genutzt werden. Andernfalls ergibt diese Methode harte Kanten, wenn Entities in der falschen Reihenfolge gerendert werden. D. P. liefert ein besseres Ergebnis als der Tiefenbuffer. Wie vermutet, reichen zwei Ebenen für fast alle Inhalte aus. D. P. führt zu Problemen bei vielen übereinandergelegten, transparenten Sprites. So nutze wir z. B. zwei semi-transparente Sprites zur Modellierung von Schlagschatten. Beim D. P. funktioniert dies nicht mehr. Für den praktischen Einsatz in einer Anwendung bleiben in der Regel also das D. P. Verfahren und die t. S.

Im Vergleich zwischen der t. S. und dem D. P. würden wir das D. P. bevorzugen. Korrekte Geometrie-Schnitte sind hier einfach möglich und die Stärke des Verfahrens. Eine Möglichkeit wäre es, beide Verfahren zu implementieren und je nach Leistungsfähigkeit der Hardware zwischen den Verfahren umzuschalten. Idealerweise würde eine weitere Stufe drei oder mehr DP-Ebenen erlauben.

In der aktuellen Implementation erfolgt das Rendering Single-Threaded. Der Programmcode wird in der render-Methode der libGDX-Bibliothek ausgeführt, welche ein mal pro Frame aufgerufen wird: „The ApplicationListener.render() method on your main game object will be invoked by this render thread once for every screen refresh (so it should

be about 60hz), so just put the body of your render loop in your implementation of this method.“ [Use]

Wenn ein Frame auf der CPU erstellt wird, wird dieser gerendert und erst nach dem Tausch der Back-Buffer[1] wird der nächste Frame berechnet. Mit Multi-Thread-Rendering könnte der nächste Frame auf der CPU berechnet werden, während parallel zum aktuellem auf der GPU gerendert wird [Gre14, p. 371]. Dann stellt sich die Frage, ob die Anwendung CPU- oder GPU-begrenzt ist. Ist die Anwendung CPU limitiert, kann ohne Auswirkung auf die Framerate länger auf der GPU gerechnet werden.

Bei isometrischen Projektionen mit Sprites ist von einer CPU Limitierung auszugehen. Nach einer Messung auf dem Intel Rechner beträgt das Verhältnis CPU- zu GPU-Laufzeit 9:1 Eine höhere Belastung der GPU mit gleichzeitiger Entlastung der CPU führt also zu einer Beschleunigung. Dies sollte beim Depth-Peeling-Verfahren der Fall sein. Meine Versuche mit Multi-Threaded-Rendering führten bisher nicht zum Erfolg.

[1] Es wird immer ein Frame gerendert, der gerade nicht sichtbar ist. Ist dieser fertig berechnet, so wird er mit dem aktuell sichtbaren Framebuffer getauscht. So wird sichergestellt, dass nur fertig berechnete Frames zu sehen sind.

7 Zukünftige Arbeiten

Da sich aus dem Multi-Threading eine hohe Beschleunigung ergeben kann, sollte dies genauer untersucht werden. Ebenfalls besteht mit den neuen Grafik-APIs „Metal“ und „Vulkan“ die Möglichkeit, dass auf bestimmten Plattformen deutlich höhere Geschwindigkeiten mit den GPU-basierten Ansätzen erreicht werden [Wel16].

Die Wurfel Engine hat den Fokus auf Desktop-Betriebssysteme gelegt. Mobile Systeme wie Smartphones und Tablets haben i. d. R. eine niedrigere Prozessor-Taktung und weniger Grafikspeicher. Es gibt einen lauffähigen Prototypen für Android mit einer alten Engine-Version. Da die Ressourcen-Anforderungen bei Sprite-basierten Spielen relativ niedrig sind, sollte es möglich sein, die Engine bei mindestens 30 FPS laufen zu lassen. Die maximalen Textur-Dimensionen sind ebenfalls i. d. R. geringer, sodass eine einzelne Textur für viele Spiele nicht ausreichen sollte und Texture-Arrays benutzt werden müssten. Im Vergleich zu vielen anderen Spielen kann durch den Gebrauch von offline gerenderten Texturen ein höhere Grafikqualität erreicht werden. Die starke Verbreitung der mobilen Geräte führt hier auch zu einem großen Bedarf von Seiten der Industrie. Weitere Arbeiten könnten also ausgewählte Verfahren auf Mobilgeräten untersuchen.

Die Beschränkung auf zwei Dimensionen in 2D-Rendering-Bibliotheken führt ebenfalls zu einer Beschränkung auf 2D-Techniken. Mit der Wurfel Engine zeigten wir, dass es möglich ist, trotz des klassischen Sprite-basierten Renderings, neue 3D-Techniken zu benutzen. Erweiterungen von 2D-Bibliotheken auf 3D sind ohne eine Steigerung der Software-Engineering-Komplexität möglich. Obwohl viele Spiele nur zwei Dimensionen für die Spielwelt nutzen, könnten sie von einem Rendering mit 3D-Informationen profitieren.

A Anhang

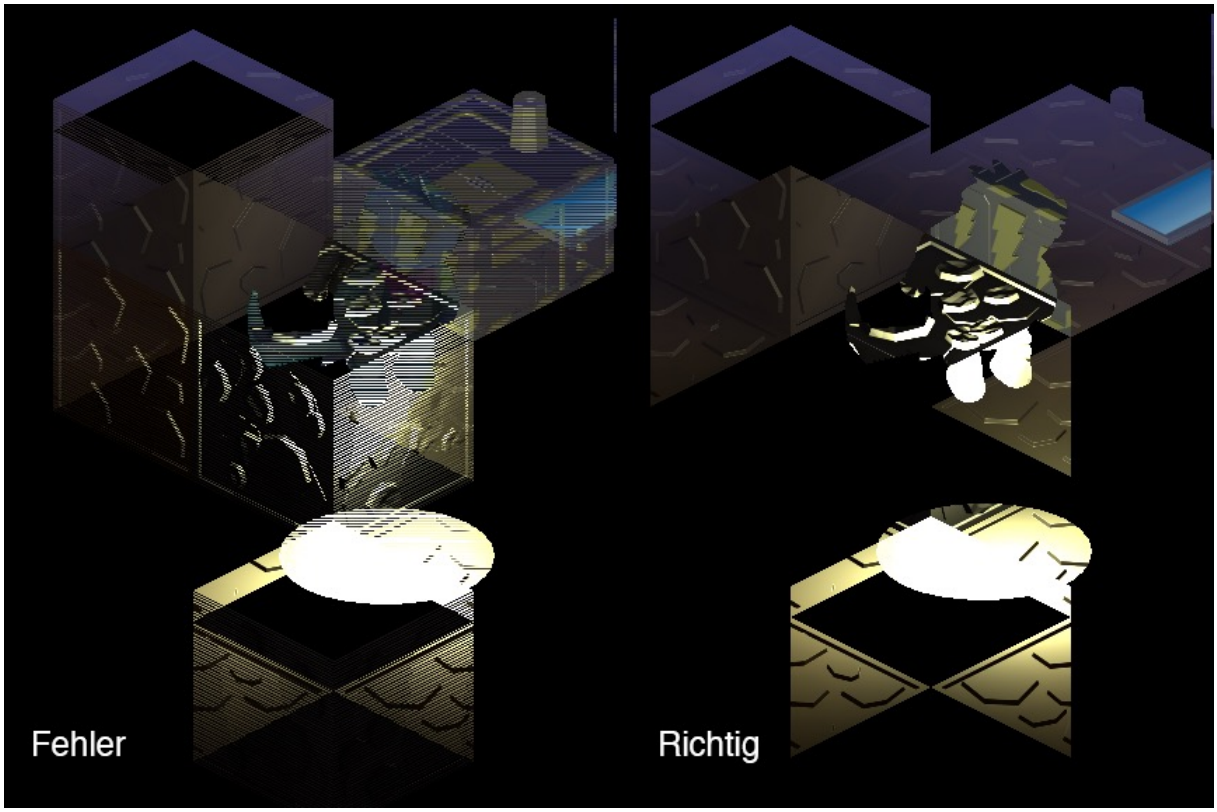

Abbildung A.1: Fehlerhafte Darstellung (links), korrekte (rechts).

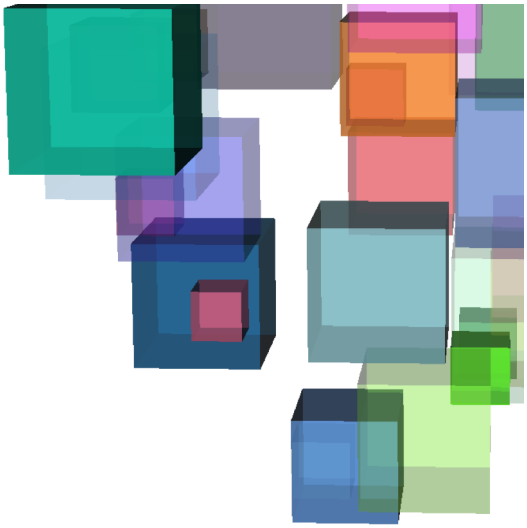

Abbildung A.2: Dual Depth Peeling, [Rig08]

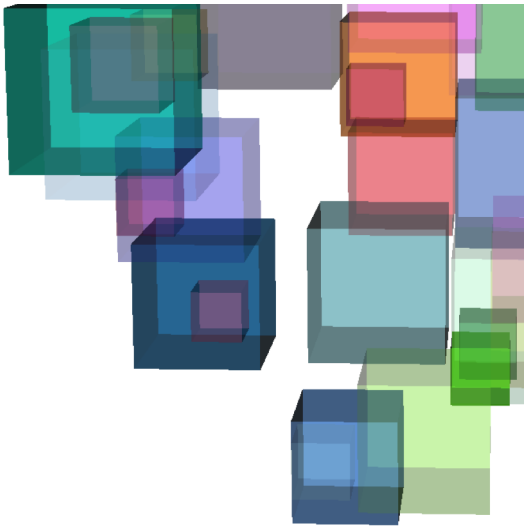

Abbildung A.3: Weighted Average, [Rig08]

Abbildung A.4: keine Sortierung

Abbildung A.5: Sortierung mit Tiefenwert

Abbildung A.6: topologisch

Abbildung A.7: Sortierung mit depthbuffer

Abbildung A.8: Sortierung mit depth peeling

A.1 Quelltexte

Die Verfügbarkeit von Implementationen vom Depth-Peeling im Internet und Literatur fällt eher gering aus. Gründe liegen vermutlich bei den benötigten hohen Komplexität an OpenGL-Kenntnissen sowie einem geringen Bedarf in der Praxis. Aus diesem Grund gibt es hier noch eine Liste von gefundenen Implementationen.

https://github.com/naohisas/DepthPeeling
https://github.com/elect86/f2b_zf
https://github.com/dartarrow/jzy3d-depthpeeling
https://github.com/Kopakc/Visu
http://www.glowscript.org/docs/GlowScriptDocs/technical.html

Clipping mit Raytracing

```
//c109373  on 28 Oct 2013

/**
 * Filters every block (and side) wich is not visible. ↲
   Boosts rendering
 * speed.
 */
public void rayCastingClipping() {
  if (zRenderingLimit > 0) {
    //prepare clipping
    for (int x = 0, maxX =Chunk.getBlocksX(); x < maxX; x↲
      ++) {
      for (int y = 0, maxY =Chunk.getBlocksY(); y < maxY; ↲
        y++) {
        //ground layer
        clipping[x][y][0][0] = true;//clip left side
        clipping[x][y][0][1] = false;//render only top
        clipping[x][y][0][2] = true;//clip right side

        //clip by default
        for (int z = 0; z < zRenderingLimit; z++) {
          setClipped(x,y,z,true);
        }
      }
    }

    //send the rays through top of the map
    for (
      int x = getIndexedLeftBorder(), maxX =↲
        getIndexedLeftBorder()+Chunk.getBlocksX();
      x < maxX;
      x++
    ) {
      for (
        int y = getIndexedTopBorder(), maxY =↲
          getIndexedTopBorder()+Chunk.getBlocksY() + ↲
          zRenderingLimit * 2;
        y < maxY;
        y++
      ) {
        castRay(x, y, Side.LEFT);
        castRay(x, y, Side.TOP);
        castRay(x, y, Side.RIGHT);
      }
    }
  }
}

/**
 * Traces a single ray.
 *
 * @param x The starting x-coordinate on top of the map.
 * @param y The starting y-coordinate on top of the map.
 * @param side The side the ray should check
 */
private void castRay(int x, int y, Side side) {
  int z = zRenderingLimit - 1;//start always from top

  //get coordaintes of border index position
```

```
int bottomIndex = getIndexedBottomBorder();
int topIndex    = getIndexedTopBorder();
int leftIndex   = getIndexedLeftBorder();
int rightIndex  = getIndexedRightBorder();

boolean left  = true;
boolean right = true;
//true if entered a liquid
boolean liquidfilter = false;
boolean leftliquid   = false;
boolean rightliquid  = false;

//bring ray to start position
if (y >= bottomIndex) {//if y coordiante is outside of
  clipping range
  z -= (y - bottomIndex) / 2;//move down for halfe the
    distance which is over the limit
  if (y % 2 == 0) {
    y = bottomIndex - 1;
  } else {
    y = bottomIndex - 2;
  }
}

y += 2;
z++;
Coordinate currentCor;
do {
  y -= 2;
  z--;

  currentCor = new Coordinate(x, y, z);
  if (side == Side.LEFT) {
    //direct neighbour groundBlock on left hiding the
      complete left side
    if (Controller.getMap().getBlock(x, y, z).hasSides()
      //block on top
      && x > leftIndex && y < topIndex - 1
      && currentCor.hidingPastBlock((y % 2 == 0 ? -1 :
        0), 1, 0)) {
      break; //stop ray
    }
    //liquid
    if (Controller.getMap().getBlock(x, y, z).isLiquid()
      ) {
      if (x > leftIndex && y < topIndex - 1
        && Controller.getMap().getBlock(x - (y % 2 == 0
          ? 1 : 0), y + 1, z).isLiquid()
      ) {
        liquidfilter = true;
      }

      if (x > leftIndex && y < topIndex - 1 && z <
        zRenderingLimit - 1
        && Controller.getMap().getBlock(x - (y % 2 == 0
          ? 1 : 0), y + 1, z + 1).isLiquid()
      ) {
        leftliquid = true;
      }

      if (y < topIndex - 2
        && Controller.getMap().getBlock(x, y + 2, z).
          isLiquid()
      ) {
        rightliquid = true;
      }

      if (leftliquid && rightliquid) {
        liquidfilter = true;
      }
    }

    //two blocks hiding the left side
    if (x > leftIndex && y < topIndex && z <
      zRenderingLimit - 1
      && currentCor.hidingPastBlock((y % 2 == 0 ? -1 :
        0), 1, 1)
```

```
  ) {
    left = false;
  }
  if (y < topIndex - 2
    && currentCor.hidingPastBlock(0, 2, 0)
  ) {
    right = false;
  }

} else if (side == Side.TOP) {//check top side
  if (Controller.getMap().getBlock(x, y, z).hasSides()
    //block on top
    && z  < zRenderingLimit - 1
    && currentCor.hidingPastBlock(0, 0, 1)) {
    break;
  }

  //liquid
  if (Controller.getMap().getBlock(x, y, z).isLiquid()
    ) {
    if (z < zRenderingLimit - 1 && Controller.getMap()
      .getBlock(x, y, z + 1).isLiquid()) {
      liquidfilter = true;
    }

    if (x > leftIndex && y < topIndex - 1 && z <
      zRenderingLimit - 1
      && Controller.getMap().getBlock(x - (y % 2 == 0
        ? 1 : 0), y + 1, z + 1).isLiquid()
    ) {
      leftliquid = true;
    }

    if (x < rightIndex - 1 && y < topIndex - 1 && z <
      zRenderingLimit - 1
      && Controller.getMap().getBlock(x + (y % 2 == 0
        ? 0 : 1), y + 1, z + 1).isLiquid()
    ) {
      rightliquid = true;
    }

    if (leftliquid && rightliquid) {
      liquidfilter = true;
    }
  }

  //two 0- and 2-sides hiding the side 1
  if (x > leftIndex && y < topIndex - 1 && z <
    zRenderingLimit - 1
    && currentCor.hidingPastBlock((y % 2 == 0 ? -1 :
      0), 1, 1)
  ) {
    left = false;
  }

  if (x < rightIndex - 1 && y < topIndex - 1 && z <
    zRenderingLimit - 1
    && currentCor.hidingPastBlock((y % 2 == 0 ? 0 : 1)
      , 1, 1)
  ) {
    right = false;
  }

} else if (side == Side.RIGHT) {
  //block on right hiding the whole right side
  if (Controller.getMap().getBlock(x, y, z).hasSides()
    //block on top
    && x < rightIndex - 1 && y < topIndex - 1
    && currentCor.hidingPastBlock((y % 2 == 0 ? 0 : 1)
      , 1, 0)
  ) {
    break;
  }

  //liquid
  if (Controller.getMap().getBlock(x, y, z).isLiquid()
    ) {
    if (x < rightIndex - 1 && y < topIndex - 1
```

```
    && Controller.getMap().getBlock(x + (y % 2 == 0
      ? 0 : 1), y + 1, z).isLiquid()
  ) {
    liquidfilter = true;
  }

  if (y < topIndex - 2
    && Controller.getMap().getBlock(x, y + 2, z).
      isLiquid()
  ) {
    leftliquid = true;
  }

  if (x < rightIndex -1 && y < topIndex - 1 && z <
    zRenderingLimit - 1
    && Controller.getMap().getBlock(x + (y % 2 == 0
      ? 0 : 1), y + 1, z + 1).isLiquid()
  ) {
    rightliquid = true;
  }

  if (leftliquid && rightliquid) {
    liquidfilter = true;
  }
}

//two blocks hiding the right side
if (y < topIndex - 2
  && currentCor.hidingPastBlock(0, 2, 0)
) {
  left = false;
}
```

```
    if (x < rightIndex - 1 && y < topIndex - 1 && z <
      zRenderingLimit -1
      && currentCor.hidingPastBlock((y % 2 == 0 ? 0 : 1)
        , 1, 1)
    ) {
      right = false;
    }
  }

  if ((left || right) && !(liquidfilter && Controller.
    getMap().getBlock(x, y, z).isLiquid())) { //unless
    both sides are clipped don't clip the whole block
    liquidfilter = false;
    setClipping(x, y, z, side, false);
  }
} while (y > topIndex && z > -1 //not on back or bottom
  of map
  && (left || right) //left or right still visible
  && (!currentCor.hidingPastBlock(0, 0, 0))
);

if (side==Side.TOP){
  //bottom layer is visible if it hit ground level and
    left or right still visible
  setClipping(
    x,
    y,
    -1,
    Side.TOP,
    !((z <= -1) && (left || right))
  );
}
}
```

Literaturverzeichnis

[AMH02] Akenine-Möller, Tomas; Haines, Eric: *Real-Time Rendering*. A K Peters, 2002

[aoe] *Microsoft Store: Age of Empires: Definitive Edition*. https://www.microsoft.com/en-us/store/p/age-of-empires-definitive-edition/9n2kmdvlk85d?ocid=SCCEducation_soc_omc_win_tw_Text_lrn_ba787&utm_source=t.co&utm_medium=referral&wa=wsignin1.0

[BF91] Barner, Martin; Flohr, Friedrich: *Darstellende Geometrie*. Quelle & Meyer, 1991

[BK11] Barta, Pál; Kovács, Balsázs: Order Independent Transparency with Per-Pixel Linked Lists / Budapest University of Technology and Economics. 2011. – Forschungsbericht

[BM08] Bavoil, Louis; Myer, Keving: Order Independent Transparency with Dual Depth Peeling. In: *Nvidia* (2008), February

[Com91] Commodore-Amiga, Inc.: *Amiga Hardware Reference Manual*. dritte Ausgabe, 1991

[cos] *Screenshot vom Spiel „Cossacks: Back to War“*. https://images.g2a.com/images/1024x768/1x1x0/7c6db06a42a7/59110f0bae653ab40b6172c5

[cpu] cpuboss: *Vergleich Intel Core i5 5257U vs AMD FX 4100*. http://cpuboss.com/cpus/Intel-Core-i5-5257U-vs-AMD-FX-4100#performance

[dmc] *About the dmcr game engine*. http://swordandarrow.ucoz.com/index/about_the_dmcr_game_engine/0-23

[Eve01] Everitt, Cass: Interactive Order-Independent Transparency / NVIDIA OpenGL Applications Engineering. 2001. – Forschungsbericht

[FKN93] Fucke; Kirch; Nickel: *Darstellende Geometrie für Ingenieure*. 15. Auflage. Fachbuchverlag Leipzig-Köln, 1993

[Gam12] *Kapitel* 0. In: GamesCoop: *Theorien des Computerspiels*. Junius, 2012

[GH05] Govindaraju, NK; Henson, M: Interactive visibility ordering transparency computations among geometric primitives in complex environments. In: *Proceedings of the 2005 symposium on interactive 3D graphics games* (2005), S. 49–56

[Gre14] Gregory, Jason: *Game Engine Architecture*. CRC Press, 2014

[KLZ14] Knowles, Pyarelal; Leach, Geoff; Zambetta, Fabio: Fast Sorting for Exact OIT of Complex Scenes. In: *CGI2014* (2014), Nr. 86

[LHLW10] Liu, Fang; Huang, Meng-Cheng; Liu, Xuehui; Wu, Enhua: FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects. In: *SI3D*, 2010

[Liu13] *Kapitel* Efficient Rendering of Order Independent Transparency on the GPUs. In: Liu, Fang: *GPU Solutions to Multi-scale Problems in Science and Engineering*. Springer Berlin-Heidelberg, 2013, 437-455

[MBC12] Myers, K.; Bavoil, L.; Cebenoyan, M.C.: *Method for depth peeling and blending*. `https://www.google.de/patents/US8094152`. Version: Januar 10 2012. – US Patent 8,094,152

[nvi08a] *GPU Programming Guide GeForce 8 and 9 Series*. NVIDIA Corporation, 2008

[NVI08b] NVIDIA Corporation: *NVIDIA CUDA: Compute unified device architecture*. 2008

[qbe] *Screenshot vom Spiel „QBert“*. `http://images.eurogamer.net/2013/06/18/185320-atari-qbert.gif`

[RB05] Reynolds, Tom M.; Blythe, David: *Advanced Graphics Programming Using OpenGl*. Morgan Kaufmann, 2005

[Rig08] Rigazzi, Alessandro: *Order Independent Transparency*. `https://www.slideshare.net/acbess/order-independent-transparency-presentation`. Version: 2008

[Sch04] Schuster, Thomas: *Entwicklung einer isometrischen Graphik-Engine in Java*, Universität Koblenz-Landau, Diplomarbeit, August 2004

[Trü14] Trümpler, Simon: Don't starve, Diablo – Parallax 7. (2014). https://simonschreibt.de/gat/dont-starve-diablo-parallax-7/

[TW15] Turau, Volker; Weyer, Christoph: *Algorithmische Graphentheorie*. 4. Auflage. De Gruyter Studium, 2015

[Use] User „P.T.“: *Stackoverflow Antwort zu „java Threads in libGDX“*. stackoverflow.com/questions/12671089

[Vog13] Vogler, Benedikt: *Erstellung einer auf Blöcken aufbauenden Spiel-Engine in Java: Schriflicher Arbeitsbericht der besonderen Lernleistung*. http://www.wurfelengine.net/downloads/Lernleistung.pdf. Version: 2013

[Wel16] Wellings, Matthew: Depth Peeling Order Independent Transparency in Vulkan. (2016), July. https://matthewwellings.com/blog/depth-peeling-order-independent-transparency-in-vulkan/

[wika] *Darstellende Geometrie Unterkapitel Abbildungsverfahren*. https://de.wikipedia.org/wiki/Darstellende_Geometrie#Abbildungsverfahren

[Wikb] Wikipedia: *Isometrie*. https://de.wikipedia.org/wiki/Isometrie

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum Unterschrift