

# Efficient Tiled Front-to-Back Rendering of Grass Blades

Bachelorarbeit von

**Yannick Tanner**

An der Fakultät für Informatik  
Institut für Visualisierung und Datenanalyse,  
Lehrstuhl für Computergrafik

Erstgutachter: Prof. Dr.-Ing. Carsten Dachsbacher  
Zweitgutachter: Prof. Dr. Hartmut Prautzsch  
Betreuernder Mitarbeiter: MSc. Tobias Zirr

28. Juli 2016 – 3. November 2016



## Abstract

### Efficient Tiled Front-to-Back Rendering of Grass Blades

This thesis describes the implementation of a new method of rendering grass in real time. The approach uses a compute shader to draw every single grass blade in parallel by separating the screen into tiles, each of them worked on by a work group of 32 threads. The grass blades in each tile are rendered front to back to avoid using depth tests for obstruction and be able to only draw visible parts of grass blades.

The developed program produces a good approximation for medium to far distance grass at a reasonable speed. The implementation can't compete with image-based methods but with further optimizations may prove valid as an alternative to geometry-based grass for medium to far distances.

## Zusammenfassung

### **Effizientes unterteiltes Front-to-Back Rendering von Grashalmen**

Diese Arbeit beschreibt die Implementierung einer neuen Methode Gras in Echtzeit darzustellen. Der Ansatz nutzt einen Compute Shader, um parallel jeden einzelnen Grashalm zu zeichnen, indem der Bildschirm in Bereiche aufgeteilt wird. Jeder der Bereiche wird von einer Arbeitsgruppe mit 32 Threads bearbeitet. Die Grashalme in jedem Bereich werden von vorne nach hinten gezeichnet, dies erlaubt Verdeckung ohne Tiefentests und ermöglicht es, für jeden Halm nur den sichtbaren Teil zu zeichnen.

Das entwickelte Programm produziert angemessen schnell eine gute Approximation für mittel bis weit entferntes Gras. Die Implementierung kann sich allerdings nicht gegen bildbasierten Methoden behaupten, könnte aber, mit weiteren Optimierungen, eine Alternative zu geometriebasiertem Gras auf mittlere bis weite Entfernung darstellen.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Stand der Forschung</b>	<b>3</b>
2.1 Darstellung von Grashalmen . . . . .	3
2.1.1 Bildbasierte Methoden . . . . .	3
2.1.2 Volumetrische Texturen . . . . .	5
2.1.3 Zeichnen jedes einzelnen Halmes . . . . .	5
2.1.3.1 Generelle Vorgehensweise . . . . .	5
2.1.3.2 Generierung der Halme . . . . .	6
2.1.4 Gemischte Methoden . . . . .	7
2.2 Beeinflussung des Grases . . . . .	8
2.2.1 Variation . . . . .	8
2.2.2 Externe Kräfte . . . . .	8
2.2.2.1 Windsimulation . . . . .	8
2.2.2.2 Kollision . . . . .	10
<b>3 Referenzarbeit: Darstellung von Gras durch Erzeugung von Geometrie</b>	<b>13</b>
3.1 Erzeugung der Grashalme . . . . .	13
3.1.1 Zufällige Wahl der Eigenschaften . . . . .	13
3.1.2 Tesselierung und LOD . . . . .	14
3.1.3 Geometrieerzeugung mit Hilfe von Bézierkurven . . . . .	15
3.2 Darstellung einer unendlichen Grasfläche . . . . .	16
3.2.1 Unterteilung der sichtbaren Fläche in Quadrate . . . . .	16
3.2.2 Weitere Unterteilung der Quadrate . . . . .	19
<b>4 Zeichnen von Gras unter Nutzung eines Compute Shaders</b>	<b>23</b>
4.1 Konzept . . . . .	23
4.2 Finden von Positionen zum Zeichnen der Grashalme . . . . .	24
4.2.1 Unterteilung des Bildschirms . . . . .	24
4.2.2 Vorarbeit für die Front-to-Back Abarbeitung . . . . .	25
4.2.3 Aufteilung der Gitterzellen auf Threads . . . . .	26
4.2.3.1 Grundlegende Vorgehensweise . . . . .	26
4.2.3.2 Reduktion der Gittergröße mit der Entfernung . . . . .	28
4.2.3.3 Weicher Übergang zwischen Gitterstufen . . . . .	29
4.2.3.4 Implementierung . . . . .	30
4.3 Erzeugen und Zeichnen der Halme . . . . .	35
4.3.1 Bestimmung der Eigenschaften . . . . .	35
4.3.2 Scanline-Rasterisierung . . . . .	36
4.3.3 Shading . . . . .	37
4.3.4 Anti-Aliasing . . . . .	39
4.3.5 Nutzung der Front-to-Back Eigenschaft für Überdeckung . . . . .	39
4.4 Kopieren in den Framebuffer . . . . .	40

4.5 Windsimulation . . . . .	40
<b>5 Ergebnisse</b>	<b>43</b>
5.1 Visuelles . . . . .	43
5.2 Performance . . . . .	44
5.2.1 Aufgliederung der Zeit . . . . .	44
5.2.2 Auswirkungen verschiedener Entfernung . . . . .	45
5.2.3 Weitere Testfälle . . . . .	46
5.2.4 Einsparungen durch Rasterisierungsabbruch und Shared Memory . .	47
5.2.5 Vergleich von Compute Shader Gras mit Geometriegras . . . . .	47
<b>6 Fazit</b>	<b>49</b>
<b>7 Ausblick</b>	<b>51</b>
7.1 Möglichkeiten zur Weiterentwicklung . . . . .	51
7.1.1 Performanceverbesserung . . . . .	51
7.1.2 Flexibilität . . . . .	52
7.1.3 Weiches Aus- und Einblenden einzelner Halme . . . . .	52
7.1.4 Einbindung in eine Szene . . . . .	53
7.1.5 Terrain . . . . .	53
7.1.6 Texturen . . . . .	53
7.1.7 Shading . . . . .	54
7.1.8 Externe Beeinflussung der Grashalme . . . . .	54
7.1.8.1 Windanimation . . . . .	54
7.1.8.2 Kollision . . . . .	54
7.1.9 Weitere Pflanzenarten . . . . .	55
7.1.10 Kombination mit anderen Methoden . . . . .	55
<b>8 Technische Details</b>	<b>57</b>
8.1 Umgebung . . . . .	57
8.2 Applikation . . . . .	57
8.2.1 User Interface . . . . .	58
8.2.1.1 Hauptanwendung . . . . .	58
8.2.1.2 Referenzanwendung . . . . .	60
8.2.2 Struktur . . . . .	60
8.2.3 Bekannte Fehler . . . . .	60
<b>Literatur</b>	<b>61</b>

# 1. Einleitung

Sei es in der Stadt oder auf dem Land: Gras ist ein fester Bestandteil vieler Szenerien. Es ist also wünschenswert bei der Nachbildung der Realität, sei es in einem animierten Film oder in einem Videospiel, Methoden zu haben, die Gras gut nachbilden können. Allerdings stellte dies bei Echtzeitanwendungen, bei denen ein Bild so schnell wie möglich generiert werden soll, schon immer eine Herausforderung dar. Das Hauptproblem besteht darin, dass eine Grasfläche zwar, im Gegensatz zu anderen dargestellten Szenen, konzeptionell relativ primitiv ist, jedoch die schiere Anzahl von einzelnen Grashalmen eine Echtzeitanwendung in hohem Maße beanspruchen kann. Trotzdem ist Gras in vielen dieser Anwendungen nicht vernachlässigbar. Deshalb wurden diverse Techniken entwickelt um trotzdem zumindest die Illusion einer Grasfläche zu erzeugen. Diese reichen von einer Fläche mit dem Bild einer Grasfläche zu versehen bis hin dazu, Grashalme zu modellieren, zu platzieren und zu rendern.

Mit dieser Bachelorarbeit soll eine neue Methode zur Echtzeitdarstellung von Grashalmen entwickelt und untersucht werden. Die Methode soll jeden Grashalm einzeln darstellen, allerdings sollen die Grashalme nicht die normale Renderpipeline durchlaufen, sondern die Erzeugung und das Zeichnen soll in einem eigens dafür programmierten Compute Shader passieren. Ein Compute Shader ist eine Möglichkeit alle möglichen Berechnungen auf der GPU durchzuführen. Der Shader wird massiv parallel auf der Grafikkarte ausgeführt. Durch die Nutzung des Compute Shaders kann jeder Aspekt des Prozesses kontrolliert werden, wodurch es einige Möglichkeiten zur Optimierung gibt. So soll die Verdeckung der Grashalme ausgenutzt werden. Dafür sollen diese von vorne nach hinten gezeichnet werden, um dann bei den weiter hinten liegenden Halmen nur noch zu zeichnen was nicht von anderem Gras überdeckt wird. Der Viewport wird dabei in mehrere Teile unterteilt, die jeweils von Arbeitsgruppen aus mehreren Threads bearbeitet werden. Dieses Verfahren umgeht das Erzeugen und Rendern von Geometrie für das Gras und zeichnet trotzdem jeden einzelnen Grashalm. Dadurch können feingranulare Änderungen ermöglicht werden, die bei bisherigen, nicht geometriebasierten, Methoden nicht so einfach oder gar nicht möglich waren. So können zum Beispiel Physiksimulationen, die Bewegung oder Verformung der Halme steuern, verwendet werden. Das Verfahren soll hauptsächlich für weiter entfernte Grashalme eine gute und effiziente Approximation liefern, während immer noch einzelne Halme gezeichnet werden.



## 2. Stand der Forschung

Im Folgenden soll ein Überblick über den Stand der Forschung zur Verarbeitung und Darstellung von Gras in Echtzeit gegeben werden.

### 2.1 Darstellung von Grashalmen

Wie im vorherigen Kapitel kurz angerissen bestehen schon einige Methoden Gras in Echtzeit darzustellen. In diesem Abschnitt soll beschrieben werden welche Methoden es gibt.

#### 2.1.1 Bildbasierte Methoden

Die einfachste Methode ist, sich einer Fläche auf die ein Bild einer Grasfläche abgebildet ist, zu bedienen. Diese Methode vereinfacht Gras zu einer grünen Fläche, was für manche Anwendungen hinreichend sein mag. Sie wurde vor allem eingesetzt, als andere Möglichkeiten die Leistungsfähigkeit von damaliger Hardware überforderten.

Eine Methode, die immer noch viel Einsatz findet ist Flächen senkrecht zum Boden zu stellen und diese mit einer teilweise transparenten Textur, die Grashalme zeigt, zu versehen. Diese Methode ist einfach skalierbar, da die Flächen automatisch zufällig verteilt werden können. Außerdem gibt es wenig Geometrie, relativ zu der Anzahl der Grashalme, da diese als Textur dargestellt werden (Vgl. [Pel04]). Ein Nachteil dieser Lösung ist, dass es oft bei genauerer Betrachtung offensichtlich ist wie die Methode funktioniert (siehe Abbildung 2.1). Um diesen Eindruck zu verbessern, müssen mehr Flächen hinzugefügt werden, was wiederum schlechtere Performance zur Folge hat. Allerdings gibt es diverse Optimierungen und Verbesserungen dieses Verfahrens. Dazu zählen zum Beispiel das Verwenden verschiedener Texturen, die für diesen Zweck optimiert wurden oder das Verwenden von Büscheln Gras, um es weniger auffällig zu machen, dass es sich um zweidimensionale Flächen handelt. Hempe beschreibt in „Real-Time Rendering Approaches for Dynamic Outdoor Environments“ [Hem16] eine Optimierung, bei der ein Geometry Shader benutzt wird, um die Flächen auf verschiedene Entfernung zu beeinflussen. So generiert der Shader auf kurze Distanzen zur Kamera mehrere in einem Büschel angeordnete Flächen, die alle mit einer entsprechenden Vegetationstextur versehen sind. Auf sehr große Entfernung wird nur noch eine einzelne Fläche verwendet, die immer zur Kamera zeigt. Auf diese Distanz ist es nicht auffällig, dass die Flächen sich mit der Kamera drehen. Gleichzeitig kann das Gras von oben betrachtet werden, ohne dass es zu offensichtlich ist, dass es sich nur um einzelne Flächen handelt.

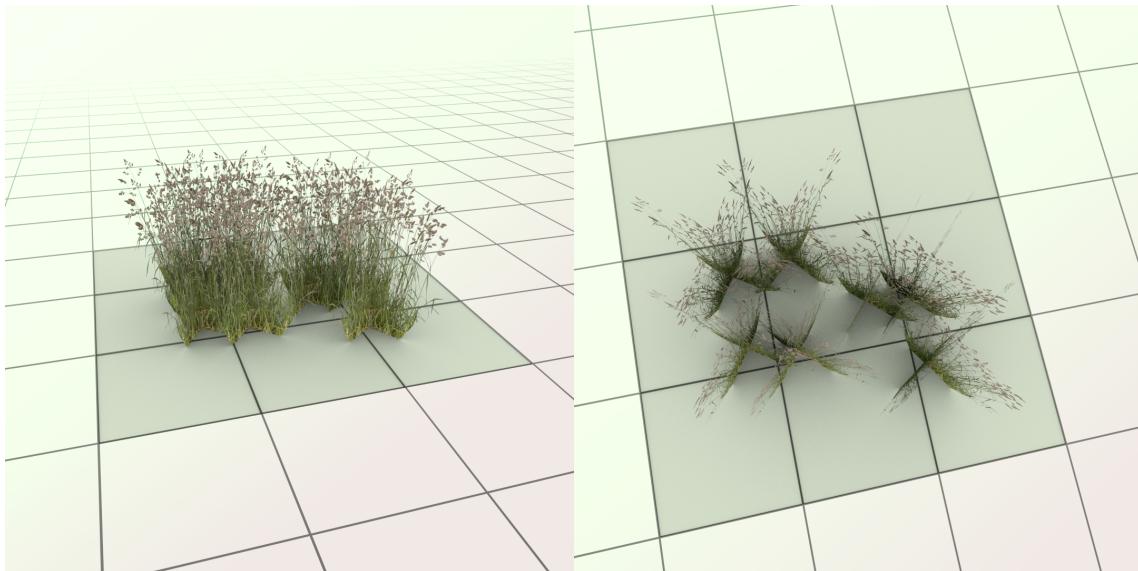


Abbildung 2.1: Gras als Flächen mit einer semitransparenten Textur dargestellt. Links: von der Seite (der gewünschte Betrachtungswinkel) Rechts: von oben (Einzelne Flächen deutlich zu erkennen)

Semitransparente texturierte Flächen haben den Nachteil, dass durch die Transparenz auch alles hinter der vordersten Fläche betrachtet werden muss, um ein korrektes Ergebnis zu erhalten. Normalerweise wird hier Alpha-Blending benutzt, was dazu führt, dass alle Flächen von hinten nach vorne betrachtet werden müssen um die korrekten Farben für einen Pixel zu erhalten. Im Falle von Gras sind dies potentiell sehr viele. Habel, Wimmer und Jeschke implementieren deshalb in „Instant Animated Grass“ [HWJ07] im Fragment Shader eines, mit Gras zu bedeckenden, Objekts einen Ray Tracer, welcher nur die texturierten Flächen beachtet und entsprechend frühzeitig abbricht, wenn ein Pixel seine endgültige Farbe hat oder eine Iterationsgrenze erreicht ist. Die Methode verhält sich also ähnlich zu Bump oder Parallax Mapping und hat ähnliche Probleme. So kann die Kamera nicht in das Gras eintauchen. Allerdings implementieren Habel, Wimmer und Jeschke eine Methode die es dem Gras erlaubt andere Objekte zu überdecken, was bei Bump oder Parallax Mapping ein Problem darstellt.

Ein anderer Ansatz der ebenfalls texturierte Flächen einsetzt wird von Bakay, Lalonde und Heidrich in „Real-time animated grass“ [BLH02] beschrieben hierbei werden mehrere horizontale semitransparente Flächen verwendet, um den Eindruck von Gras zu erwecken. Die übereinanderliegenden Flächen werden an den Stellen, an denen ein imaginärer Grashalm sie schneiden würde, mit einer grünen Textur versehen. Ein Halm wird also durch eine Reihe übereinander schwebender grüner Punkte dargestellt. Ähnlich wie bei der vorher genannten Methode ergibt sich ein akzeptabler Eindruck solange nicht zu genau hingeschaut wird. Ein Vorteil ist, dass ein Betrachter weniger wahrscheinlich einen kritischen Winkel einnimmt, von dem aus offensichtlich ist, dass es sich nur um übereinander liegende Flächen handelt. Weiterhin kann mit insgesamt weniger Geometrie auskommen werden. Auch bietet sich die Möglichkeit einfache Bewegungen im Wind darzustellen, indem die Flächen entsprechend verschoben werden. Da es relativ viele Flächen gibt, ist die Verbiegung der Halme detaillierter im Vergleich zu dem was mit horizontalen Flächen möglich ist.

Bildbasierte Methoden, hauptsächlich semitransparente Flächen rechtwinklig zum Boden, sind nach wie vor vorherrschend und das aus verschiedenen Gründen. Zum Einen kann

durch eine Textur viel einfacher ein realistisches Ergebnis erzeugt werden, indem Fotos zur Erzeugung der Textur verwendet werden. Da es sich um eine direkte Abbildung der Wirklichkeit mit all ihren Details handelt ist diese Vorgehensweise nahezu unersetzbar für die Darstellung von Vegetation. Zum Anderen wird wesentlich weniger Geometrie verwendet, um komplexe Formen darzustellen. Was dies, zumindest für lange Zeit, zur einzigen vertretbaren Methode zur Darstellung für Gras in Echtzeit machte. Es existiert außerdem viel Potential für Verfeinerungen und Level of Detail (LOD) Optimierungen wie zum Beispiel in „Real-Time Rendering Approaches for Dynamic Outdoor Environments“ [Hem16] beschrieben.

### 2.1.2 Volumetrische Texturen

Um das Aufkommen von Geometrie beim Darstellen von Gras weiter zu minimieren, können zum Beispiel volumetrische Texturen verwendet werden, die für jeden Punkt im Raum eine Farbe speichern. Es gibt inzwischen einige Methoden um diese auch in Echtzeit zu zeichnen (Vgl. [SKP05; Iki+04]), allerdings ist der Speicherverbrauch sehr hoch (Vgl. [SKP05]). Es gibt aber verschiedene Ideen, die auf einem ähnlichen Ansatz beruhen. So kann man die zuvor genannte Vorgehensweise, die übereinander liegende texturierte Flächen verwendet ([BLH02]), als vereinfachtes Erstellen und Rendern einer volumetrischen Textur betrachten.

In „Real-time rendering of realistic-looking grass“ [SKP05] verfolgen Shah, Kontinnen und Pattanaik ein ähnliches Konzept, indem für eine Grasfläche, von einem darüber liegenden Betrachter, Tiefeninformationen und eine BTF (Bidirectional Texture Function) abgespeichert werden. Es werden dabei Positions- und Beleuchtungsinformationen zu verschiedenen Blickwinkeln gespeichert. Mit diesen vorberechneten Informationen kann das Gras auf einer Fläche gezeichnet werden und mit den Tiefeninformationen können dem Ganzen auch Tiefe und Verdeckungseigenschaften verliehen werden.

### 2.1.3 Zeichnen jedes einzelnen Halmes

Die intuitivste Methode ist natürlich eigene Geometrie für jeden Grashalm zu nutzen und die Halme in einer realistischen Art und Weise auf einer Fläche zu verteilen. Dies ist eine sehr anspruchsvolle Methode, doch mit immer stärker werdender Hardware rückt sie immer mehr ins Mögliche. So gibt es schon jetzt in einigen Videospielen Optionen für AMDs TressFX und NVIDIAAs Hairworks die Haare und Fell simulieren und zeichnen, dabei werden auch zumindest Strähnen einzeln berücksichtigt.

#### 2.1.3.1 Generelle Vorgehensweise

In „Grass Rendering and Simulation with LOD“ [HL15] beschreiben Han und Li eine Vorgehensweise zum Zeichnen einzelner Grashalme, bei der Modelle für zweidimensionale Halme generiert und dann zufällig skaliert, rotiert und platziert werden. Für die Echtzeitarstellung wird optimiert, indem ein sehr einfacher Shader für die Beleuchtung des Grases benutzt wird. Außerdem werden verschiedene LODs benutzt. Umso weiter die Grashalme von der Kamera entfernt sind umso weniger Details werden gezeichnet.

Ein sehr ähnlicher Ansatz zu dem später in der Referenzarbeit (Kapitel 3) umgesetzten, ist in „Interactive Grass Rendering Using Real-Time Tessellation“ [JW13] von Jahrmann und Wimmer beschrieben worden. In der Arbeit werden Quads zu Grashalmen tesseliert. Die Quads werden also automatisch verfeinert und gleichzeitig in die Form eines Grashalmes gebracht. Auf diese Weise kann durch die Tesselierung<sup>1</sup>, die auf der GPU geschieht, sehr effizient Geometrie für jeden einzelnen Grashalm dargestellt werden. Da die Generierung der

---

<sup>1</sup>engl. tessellation

meisten Geometrie auf der Grafikkarte passiert, kann auch einfach LOD umgesetzt werden, indem die Geometrie mit wachsender Entfernung weniger unterteilt wird. Eine weitere in der Arbeit umgesetzte LOD Optimierung ist das Berechnen von Patches mit verschiedenen Grasdichten. Je nach Kameraentfernung wird ein anders dichtes Patch angezeigt. Um die Übergänge glatt zu gestalten, werden einzelne Halme dazwischen weggelassen.

Mit dem gleichen Thema beschäftigt sich Papavasiliou in „Real-Time Grass (and Other Procedural Objects) on Terrain“ [Pap15], hier werden als Erweiterung noch andere Pflanzenarten gerendert. Genauso wird festgestellt, dass jede Geometrie, die sich auf der GPU generieren lässt, genutzt werden kann. Als Beispiel werden Steine und Kiesel genannt. Der Autor beabsichtigt mit seiner Methode eine Alternative zu den komplexeren gemischten Methoden (siehe Unterabschnitt 2.1.4) darzustellen, indem kontinuierliches Level of Detail benutzt wird. So wird sich in der Arbeit auf große Entfernungen einfach auf die Textur, der unter dem Gras liegenden Landschaft, verlassen.

### 2.1.3.2 Generierung der Halme

Die einzelnen Grashalme werden bei der Mehrheit aller Methoden in irgend einer Form automatisch generiert. Wobei Han und Li erwähnen, dass von Hand modellierte Halme auch möglich sind. In der, in „Grass Rendering and Simulation with LOD“ [HL15] beschriebenen, Methode werden nur fünf verschiedene Halme generiert, diese werden später zufällig rotiert und skaliert. Die Vorgehensweise wird damit begründet, dass eine Hand voll Halme, die eine realistische Form haben, sehr vielen verschiedenen unrealistischen Halmen vorzuziehen sind. Insbesondere wird die Methode Grashalme mit Partikeln zu generieren referenziert. Diese Grashalme haben immer die Form einer quadratischen Kurve und seien somit nie realistische Halme.

Allgemein wird in den meisten Arbeiten jedoch ein prozeduraler Ansatz genutzt. Algorithmmisch kann so relativ einfach vielfältiges Gras ohne erkennbare Muster erzeugt werden. Die Generierung von Halmen funktioniert, in den meisten Arbeiten, unter Nutzung von Kurven. Dabei gibt es zwei Varianten. Zum Ersten die bereits erwähnten Partikel. Dabei werden Partikel simuliert, die mit einer bestimmten Geschwindigkeit in eine zufällige Richtung fliegen und durch Gravitation beeinflusst werden. Die Position der Partikel wird an verschiedenen Zeitpunkten aufgezeichnet. Es ergeben sich also eine Menge an Punkten auf einer quadratischen Kurve. Aus diesen kann dann ein Grashalm generiert werden. Deusen et al. schlagen in „Interactive visualization of complex plant ecosystems“ [Deu+02] zum Beispiel dazu vor, dünne und lange Pflanzenstrukturen als einfache Linien zu zeichnen. Hier also Linien zwischen, den durch die Partikel generierten, Punkten. Die Farbe und Dicke der Linien kann angepasst werden, um einen dreidimensionalen Eindruck zu erzeugen. Boulanger benutzt in „Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting“ [Bou08] ebenfalls Partikel um die Form der Grashalme zu erzeugen, jedoch werden hier die Punkte mit Quads verbunden die dann auf beiden Seiten texturiert werden.

Eine andere Methode für die Generierung der Form der Halme, ist Kurven beziehungsweise Splines direkt zu erzeugen und darauf die Form aufzubauen. Han und Li beschreiben in „Grass Rendering and Simulation with LOD“ [HL15] wie zwei Splines zur Tesselierung einer Folge von Quads benutzt werden. Die Splines ergeben sich aus den Vertexpositionen der Ausgangsgeometrie und Kontrollpunkten, diese bestimmen die Krümmung des Halmes. Oft wird jedoch nur eine Kurve genutzt, um die Form eines Halmes zu bestimmen. Andere Kurven, wie zum Beispiel kubische hermitesche Splines, die durch Anfangs- und Endpunkt sowie Tangenten an diesen beschrieben sind, sind analog umsetzbar.

### 2.1.4 Gemischte Methoden

Es bietet sich an, verschiedene Methoden für verschiedene Entfernungen zu benutzen. So kann zum Beispiel auf eine weite Entfernung Gras einfach durch eine grün texturierte Fläche dargestellt werden, während nahe Grashalme tatsächlich einzeln gezeichnet werden. Mit einem solchen Konzept beschäftigt sich unter anderem Boulanger in „Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting“ [Bou08]. Hier wird ein Ansatz beschrieben bei dem in nahen Bereichen Geometrie für jeden Grashalm gezeichnet wird. Auf höhere Distanz werden vertikale und horizontale texturierte Flächen eingesetzt. Auf sehr weite Entfernungen ist das Gras nur noch als texturierte horizontale Fläche dargestellt. Boulanger nutzt fertige Bereiche mit denen eine Grasfläche gefüllt wird. Diese Bereiche enthalten das Gras, das je nach LOD verschieden dargestellt wird. Um nicht Informationen für jeden einzelnen Bereich zu speichern wird ein Bereich immer wiederverwendet. Sich zu sehr wiederholende Muster werden vermieden, indem die Bereiche je nach Position zufällig gespiegelt und rotiert werden.

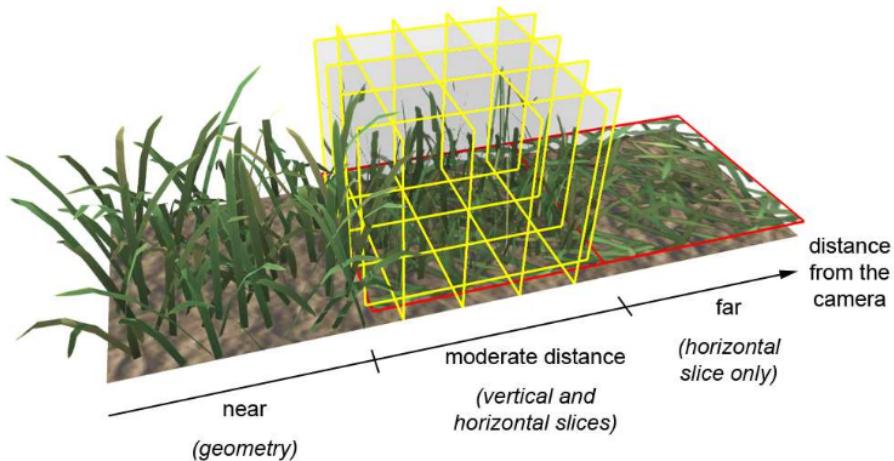


Abbildung 2.2: Gemischer Ansatz mit drei Levels of Detail. Bild aus „Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting“ [Bou08]

In „Real-Time Rendering and Animating of Grass“ [LDY12] von Li, Ding und Yan wurde eine ganz ähnliche Methode zur Darstellung des Grases verwendet. Es werden ebenfalls drei Detailstufen berechnet. Zwischen den Stufen wird eine Übergangszone festgelegt, in der beide Detailstufen gezeichnet werden. Dabei nimmt die Sichtbarkeit der hochauflösenden Stufe entsprechend kontinuierlich mit der Entfernung ab und die der weniger hochauflösenderen umgekehrt proportional zu.

Bei beiden Methoden wird für die mittlere Stufe ein pseudovolumetrischer Ansatz genutzt, bei dem die Geometrie der Halme der ersten Stufe mit den horizontalen Flächen der zweiten geschnitten werden um die Textur für diese zu erhalten. Dieser Ansatz ermöglicht die zwei Detailstufen so ähnlich wie möglich aussehen zu lassen

In „Screen-Space Grass“ [Pan14] stellt Pangerl eine Approximation von weit entferntem Gras vor. Der Autor implementiert eine Methode, bei der zunächst das Terrain mit einer Grastextur gerendert wird und danach, in der Nachverarbeitung des Ergebnisses, einzelne Graspixel vertikal verschmiert werden. Es entstehen vertikale Linien, die mit Hilfe von Tiefen- und Stencil-Buffer auch dort gezeichnet werden wo die Grasfläche sich befindet. Nachträglich wird eine Rauschfunktion auf das Ergebnis angewandt um den Eindruck von natürlichem Chaos zu erzeugen. Diese Vorgehensweise erweist sich als extrem schnell und ergibt ein gutes Ergebnis für entsprechende Entfernungen (Vgl. [Pan14]).

Eine etwas weniger konventionelle Methode, die Qualität von gerendertem Gras zu verbessern, wurde von Brkic et al. in „Cross-modal Affects of Smell on the Real-time Rendering of Grass“ [Brk+09] untersucht. Die Arbeit beschäftigt sich mit dem Konzept Geruch von Gras zu nutzen, um einem Betrachter eine Abbildung einer Graslandschaft realistischer erscheinen zu lassen. Die Autoren hoffen mit diesem Konzept bei weniger Rechenaufwand die gleiche wahrgenommene Qualität zu erreichen. Tatsächlich erkennen Testpersonen weniger wahrscheinlich, dass es sich um eine synthetische Szene handelt wenn ein entsprechender Geruch involviert ist (Vgl. [Brk+09]).

## 2.2 Beeinflussung des Grases

Vegetation, wie zum Beispiel Gras, wird durch verschiedene Faktoren über die Zeit oder auch über den Raum beeinflusst. Deshalb soll, in diesem Abschnitt, eine Übersicht über die gängigsten Ansätze gegeben werden welche Faktoren, bei der Erzeugung von Gras, zu berücksichtigen.

### 2.2.1 Variation

Gras ist keine absolut homogene Fläche. Es gibt Variationen von Länge, Dichte, Form, Farbe und Dicke. In einem großen Teil der sich mit dem Thema befassenden Arbeiten, kommen zumindest zufällige Variationen zum Einsatz um eine Grasfläche natürlicher wirken zu lassen (zum Beispiel [Fan+15; HL15]). Variationen sind jedoch oft nicht komplett zufällig und hängen von anderen Faktoren in der Umgebung ab. So sieht man zum Beispiel langes Gras in der Nähe von Wasser oder wenig bis gar kein Gras auf einem Wildpfad. Ein populärer Ansatz solche Faktoren zu beeinflussen, ist Texturen für verschiedene Eigenschaften zu verwenden. Diese geben für ein Terrain an was für Eigenschaften das darauf wachsende Gras haben soll (Vgl. [JW13; HRS13; Bou08; JSK09]). Die Texturen haben den Vorteil, dass ein Anwender Kontrolle über das Aussehen einer Grasfläche hat, während er sich nicht um Eigenschaften jedes einzelnen Halms kümmern muss. Für sehr große Landschaften scheint diese Vorgehensweise jedoch sehr aufwändig. Es ist durchaus vorstellbar diverse Variationen algorithmisch zu bestimmen oder eine Textur durch ein externes Tool generieren zu lassen (Vgl. [Hen+13]).

### 2.2.2 Externe Kräfte

Gras ist, im Gegensatz zu vielen anderen Objekten, die in einer in Echtzeit zu rendernden Szene vorzufinden sind, sehr flexibel. Es lässt sich einfach von diversen Kräften beeinflussen. Somit erscheint es umso unrealistischer, falls in einer virtuellen Szene das Gras völlig statisch ist. Externe Beeinflussung der Grashalme ist also ein wichtiges Thema bei jedem Ansatz Gras in Echtzeit darzustellen.

#### 2.2.2.1 Windsimulation

Vegetation wird auch von den leichtesten Windböen bewegt. Es hilft also ungemein eine Simulation von Wind mit Einfluss auf die Vegetation zu implementieren, um eine bewegte Szene glaubwürdiger erscheinen zu lassen.

In einer sehr frühen Arbeit zur Darstellung und Simulation von Gras betrachteten Reeves und Blau [RB85] die Windsimulation von Gras. Die Konzepte wurden zwar nicht für Echtzeitanwendungen entwickelt, sind aber trotzdem übertragbar. Für die Simulation von Wind werden hier zufällig generierte Wellen aus Partikeln verwendet, die sich alle ungefähr in die Richtung des Windes bewegen. Jeder Partikel repräsentiert eine lokale Windbörse. Aus den Partikeln wird eine Textur berechnet, welche für jeden Punkt angibt, wie sehr ein Grashalm an dieser Stelle auf dem Terrain vom Wind beeinflusst wird. Der Wert ergibt

sich aus der Nähe des Punktes zu den Partikeln. Diese Textur muss für jedes Bild einer Animation einzeln berechnet werden. Ein Grashalm wird nun gebogen indem die Punkte, aus denen der Halm besteht, um eine Achse, die im Rechten Winkel zur Windrichtung steht und durch die Basis des Halmes verläuft, gedreht wird. Wie stark gebogen wird hängt von der Stärke der Böe und der Höhe des betrachteten Punktes ab. So kann simuliert werden, dass der Halm unten starrer ist als oben. Falls die Biegung durch den Wind aufhört, kehrt der Halm zu seiner ursprünglichen Position zurück. Er folgt dabei einer gedämpften Sinusfunktion. Um das Ergebnis natürlicher wirken zu lassen, werden noch diverse Eigenschaften zufällig bestimmt. So ist zum Beispiel die Biegeachse nicht immer ganz rechtwinklig zur Windrichtung und nahe aneinander liegende Grashalme benutzen leicht verschiedene Böenstärken.

Habel, Wimmer und Jeschke nutzen in „Instant Animated Grass“ [HWJ07] ebenfalls eine Textur um die Windanimation von Grashalmen zu steuern, dabei wird argumentiert, dass so prozedurale sowie von Hand erstellte Animationen unterstützt werden. Animierte wird hier nicht durch Verschiebung von Vertexkoordinaten sondern Texturkoordinaten der texturierten Flächen, die das Gras darstellen. Es wurde auch ein prozeduraler Ansatz für die Erzeugung der Animationstextur vorgestellt. Dabei kommt eine Textur aus zwei (zeitabhängigen) Perlin-Noise Funktionen zum Einsatz.

Perbet und Cani nutzen in „Animating prairies in real-time“ [PC01] für die Beeinflussung des Grases durch Wind zweidimensionale Masken, die eine beliebige Position auf der Grasfläche haben können. Die Masken enthalten Informationen zur Richtung des Windes und der Stärke, die auf die Halme unter der Maske angewandt werden. Grashalme haben in einer Simulation vorausberechnete Biegepositionen mit einem Index von minus eins bis eins, diese Positionen werden als Stärke in der Maske direkt referenziert. Die vorberechneten Biegungen helfen verschiedenen flexible Gräser umzusetzen, indem diese andere vorberechnete Biegungen haben. Die Masken können frei erzeugt bewegt und überlagert werden. Sobald keine Maske mehr Einfluss auf die Halme hat, oszillieren diese zurück zu ihrer ursprünglichen Position ähnlich wie in „Approximate and probabilistic algorithms for shading and rendering structured particle systems“ [RB85]. Diese Animation ist relativ zu den vorausberechneten Biegung ebenfalls vorausberechnet. Es gibt hier drei verschiedene Masken: Wirbelwind, leichte Brise und ein Luftstoß in alle Richtungen von einem Zentrum aus.

Häufig wird zur Nachahmung von Grasbewegung im Wind lokal eine trigonometrische Funktion benutzt. So werden in „Rendering countless blades of waving grass“ [Pel04] Flächen mit einer Grastextur animiert, indem die obersten Vertices der Vierecke gemäß einer trigonometrischen Funktion bewegt werden. Dazu übergibt die vorgestellte Implementierung einen Zeitstempel, eine Windrichtung und die Windstärke in den Vertex Shader des Grases. Dieser berechnet die Verschiebung der Vertices entsprechend und wendet diese mit pseudozufälligen Faktoren für jeden Vertex an, um ein wenig natürliches Chaos zu erzeugen. Ein ähnliches Vorgehen beschreiben Fan et al. in „Simulation and rendering for millions of grass blades“ [Fan+15]<sup>2</sup>, hier ist jeder Grashalm einzeln als Geometrie dargestellt. Die Halme werden aber auch hier im Vertex Shader animiert, indem die Vertices entlang der Windrichtung bewegt werden. Wie stark die Vertices verschoben werden, hängt von ihrer Entfernung zur Graswurzel und der Windstärke ab, die Verschiebung ergibt sich aus der Summe von Sinusfunktionen. Leichte zufällige Verschiebungen finden Anwendung um unerwünschte Muster zu vermeiden. Diese Verschiebung im Vertex Shader mit trigonometrischen Funktionen bietet den Vorteil, dass die Berechnung relativ schnell und lokal auf der Grafikkarte passiert und die Berechnung nur von der Zeit abhängt. Es wird also

<sup>2</sup>Die Arbeiten „Simulation and rendering for millions of grass blades“ [Fan+15] und „Grass Rendering and Simulation with LOD“ [HL15] haben zwei Autoren gemeinsam und stimmen in großen Teilen miteinander überein.

kein zusätzlicher Speicher benötigt. Aufgrund der Einfachheit ist diese oder eine ähnliche Methode in vielen Arbeiten zu dem Thema zu finden (so wie auch in [JW13; HRS13]).

Etwas weiter geht die Arbeit zur prozeduralen Animation von Vegetation „Vegetation procedural animation and shading in crysis“ [Sou07] von Sousa auf die Jens, Salama und Kolb in „GPU-based responsive grass“ [JSK09] eine Animationsmethode für Gras aufbauen. Die Autoren implementieren die Bewegung von Gras im Wind als die Summe geglätteter Dreiecksfunktionen, die die oberen Vertices von Quads entlang der Windrichtung verschieben. Sousa argumentiert, dass diese Funktionen günstiger auszuwerten seien als trigonometrische Funktionen. Allerdings bietet heutige Hardware auch Beschleunigung für trigonometrische Funktionen.

Han und Li ergänzen in „Grass Rendering and Simulation with LOD“ [HL15] die Animation jedes einzelnen Halmes im Wind außerdem noch um eine globale Komponente, bei der lange horizontale, ansonsten unsichtbare, Zylinder in Windrichtung über ein Grasfeld bewegt werden und Kollisionserkennung benutzt wird, um die Halme entsprechend zu biegen.

Eine mehr auf physikalischen Grundlagen beruhende Methode wird in „Real-Time Rendering Approaches for Dynamic Outdoor Environments“ [Hem16] von Hempe angewandt. Dabei wird die Bewegung von Gras im Wind als erzwungene Schwingung in einem System mit einem einzigen Freiheitsgrad modelliert. Die Kraft die auf einen Halm ausgeübt wird ergibt sich aus der Grashalmfläche und dem Winddruck welcher sich aus der Windgeschwindigkeit berechnet.

### 2.2.2.2 Kollision

Neben Wind soll Gras natürlich auch von Objekten in der virtuellen Welt beeinflusst werden, dazu wird eine Art Kollisionserkennung benötigt.

Da eine echte Physiksimulation für so viele Objekte teuer ist wird in „A procedural approach to animate interactive natural sceneries“ [Gue+03] von Guerraz et al. ein Animationsprimitiv an ein Objekt gehängt welches Gras beeinflussen soll (zum Beispiel ein Schuh). Dieses Primitiv agiert ähnlich wie die Masken in „Animating prairies in real-time“ [PC01] und biegt oder knickt die sich in der Umgebung befindenden Halme. Beim Verschwinden des Einflussfaktors kehren diese wieder langsam zu ihrer ursprünglichen Form zurück. Die Bewegung des Primitives wird aufgezeichnet und mit diesem Bewegungsvektor entschieden in welche Richtung ein Halm sich biegen oder knicken soll.

In „Simulation and rendering for millions of grass blades“ [Fan+15] ist eine Grasfläche in Bereiche unterteilt. Diese werden als aktiv markiert wenn sie ein, für Kollision in Frage kommendes, Objekt beinhaltet. Auf der GPU wird dann für jeden Vertex eines Halmes in einem aktiven Bereich überprüft, ob er sich mit einem der Objekte schneidet. Dann wird ein Bedingungserfüllungsproblem (CSP) für die Position des Vertex gelöst. Die Bedingungen sind: Keine Überlappung und Beibehalten der Länge des Grashalms. Außerdem sind schwache Bedingungen (soft constraints) zur Erhaltung der Form des Halmes gegeben. Das CSP wird iterativ innerhalb von drei Iterationen gelöst, dieser Ansatz wurde aus einer Arbeit über Haarsimulation übernommen ([HH12]). Nachdem sich kein Kollisionssubjekt mehr in einem Bereich befindet, bleiben die Bereiche ungefähr noch so lange aktiv bis sich nichts mehr verändert. So wird die tatsächliche Kollisionsberechnung nur auf einem vergleichbar kleinen Teil der Halme angewandt.

Jens, Salama und Kolb berechnen in „GPU-based responsive grass“ [JSK09] den Schnitt zwischen unterteilten texturierten Quads, die Grashalme repräsentieren, und Kollisionsobjekten, die die Form eines echten Objektes approximieren. Falls ein Vertex im Schnitt ist, wird dieser entlang der Normalen des Kollisionsobjektes verschoben bis er nicht mehr

im Schnitt liegt. Alle anderen Vertices werden entsprechend eines Federsystems verschoben, das alle Vertices eines unterteilten Quads verbindet. Genauso wie in „Simulation and rendering for millions of grass blades“ [Fan+15] kommt auch hier eine Vorauswahl von Bereichen, die für die Kollisionserkennung in Frage kommen, zur Anwendung. Allerdings gibt es eine separate Markierung für sich erholende Halme, die nicht für Kollisionen in Frage kommen, um unnötige Schnitttests zu vermeiden. Die Erholung eines Grashalms entspricht der Animation aller Vertices zu ihrem ursprünglichen Zustand. Diese passiert hier nicht linear sondern kubisch. Die Erholung erfolgt also zuerst langsam und wird dann immer schneller. Erwähnenswert ist weiterhin, dass Jens, Salama und Kolb nur eine Unterteilung eines texturierten Quads vornehmen, falls dieses kollidiert oder sich erholt.



### **3. Referenzarbeit: Darstellung von Gras durch Erzeugung von Geometrie**

Zum Vergleich der neu entwickelten Methode wurde zunächst eine eher konservative Methode zum Darstellen von Grashäuten implementiert. Hierbei war das Ziel den „Brute-Force“ Ansatz zu nutzen, bei dem Geometrie für jeden einzelnen Grashalm erzeugt und gerendert wird. Dies geschieht unter Nutzung der Standard Renderpipeline. Das Verfahren ist auf diverse Weisen optimiert, die wahrscheinlich auch in einer vergleichbaren und sich im professionellen Gebrauch befindenden Anwendung zum Einsatz kommen.

Dieser Ansatz wurde als Referenz ausgewählt, da er vergleichbare Möglichkeiten zu der geplanten Methode dieser Arbeit bietet. Jeder Grashalm kann einzeln berücksichtigt und beeinflusst werden und wird auch einzeln gezeichnet. Des Weiteren ist der Ansatz dieser Referenz mit höchster Wahrscheinlichkeit derjenige, der semitransparente Flächen mit Grastexturen zumindest zum Teil ablösen wird. Dies ist anzunehmen, da weitere Verfeinerung dieser Flächen, was mit steigender Grafikleistung zu einer realistischen Möglichkeit wird, irgendwann dazu führt Geometrie für jeden einzelnen Grashalm zu zeichnen.

#### **3.1 Erzeugung der Grashalme**

Die Grashalme werden in der Anwendung komplett in Shadern erzeugt und verarbeitet. Das Programm über gibt nur Grenzen eines Bereichs und bestimmt, wie oft der Vertex Shader aufgerufen wird indem ein Vertex Array der entsprechenden Größe an OpenGL übergeben wird. Dieses enthält ansonsten keine Informationen.

##### **3.1.1 Zufällige Wahl der Eigenschaften**

Wie oft der Vertex Shader aufgerufen wird, bestimmt hier semantisch wie viele Grashalme gezeichnet werden. Deshalb übernimmt der Vertex Shader hier auch Berechnungen, die für jeden Grashalm nur ein mal ausgeführt werden müssen und nichts weiteres erfordern. So werden hier Eigenschaften wie Position (begrenzt durch den übergebenen Bereich), Rotation, Höhe, Breite und Krümmung festgelegt. All diese Eigenschaften werden mit einem Pseudozufallszahlengenerator erzeugt, der als Seed VertexID und x-Koordinate des Bereichs, in dem Grashalme generiert werden, benutzt. So sind die Ergebnisse des Shaders stabil bezüglich des Bereichs.

Der Shader gibt vier Vertices und einen Einflusspunkt aus. Die Vertices beschreiben ein Viereck das ungefähr der Form des gewünschten Grashalms entspricht. Der Einflusspunkt

bestimmt in welche Richtung sich der Grashalm biegen soll. Diese Daten werden aus zufälliger Position sowie aus drei Kontrollpunkten berechnet, die die Form des Halmes beschreiben und sich aus den restlichen zufällig gewählten Eigenschaften ergeben.

In Abbildung 3.1 sind die Daten, die im Vertex Shader berechnet werden, veranschaulicht. Im Bild sind in Grün und Orange die Ausgabepunkte des Shaders zu sehen. Hierbei handelt es sich um die Vertices (Grün) und einen Einflusspunkt (Orange). Diese Punkte ergeben sich aus den Kontrollpunkten (Blau und Orange).

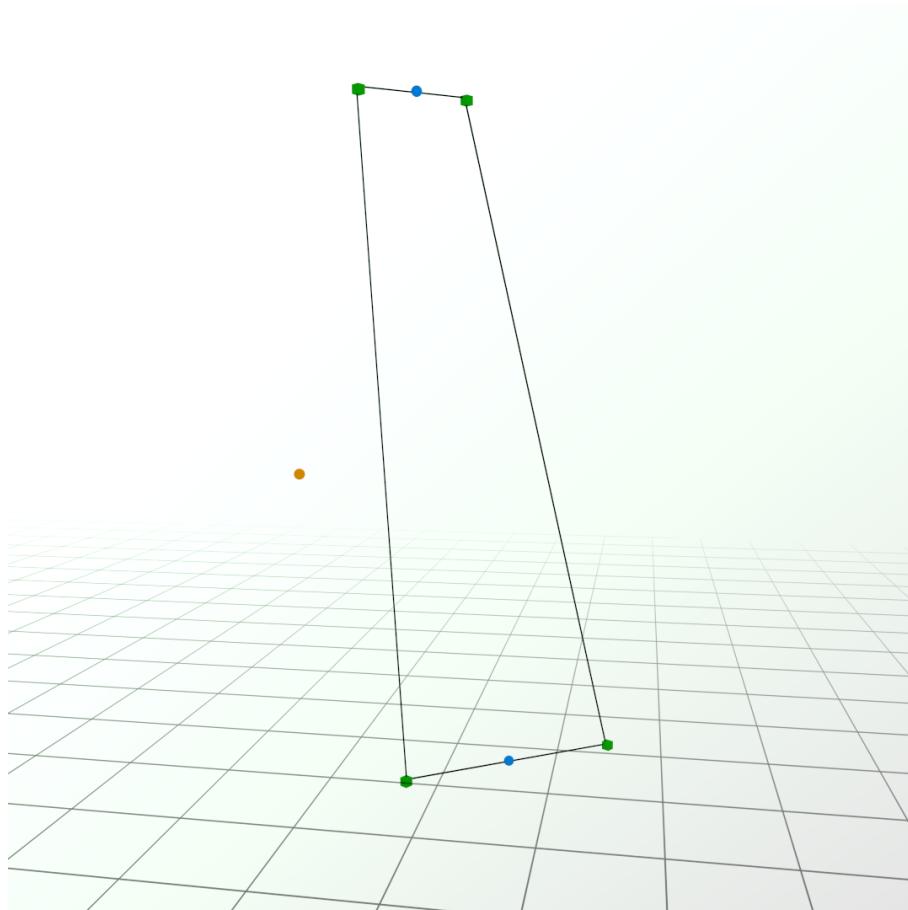


Abbildung 3.1: Grashalm nach Verarbeitung im Vertex Shader.

In Grün: Eckpunkte der Ausgangsgeometrie für den Halm.

In Blau und Orange: Kontrollpunkte des Halses

In Orange: Einflusspunkt

### 3.1.2 Tesselierung und LOD

Die tatsächliche Erzeugung der Geometrie des Grashalms erfolgt über Tesselierung. Diese wird allgemein benutzt um Objekte dynamisch zu unterteilen und somit ein qualitativ höheres Ergebnis zu erzeugen. In diesem Fall wurde sie benutzt, um aus dem Ergebnis des Vertex Shaders einen Grashalm zu generieren. Dies bietet den Vorteil, dass die Generierung der Halme parallel auf der GPU geschehen kann. Die Methode ist somit schneller als Gras auf der CPU zu generieren und dynamischer als Geometrie zu importieren.

In OpenGL erfolgt Tesselierung in drei Schritten. Zuerst wird für jeden Vertex eines zu tesselierenden Teils eines Objektes der Tessellation Control Shader aufgerufen. Dieser gibt eine Anzahl Vertices sowie Unterteilungsgrade für verschiedene Kanten aus. Als zweites

werden zusätzliche Vertices generiert, die Anzahl bestimmt durch den Unterteilungsgrad, den der Control Shader ausgegeben hat. Diese Stufe nennt man Tessellation Primitive Generation und erfolgt im Tesselierungsprozess automatisch und kann nicht programmiert werden. Zuletzt wird der Tessellation Evaluation Shader ausgeführt dessen Aufgabe es ist, den neu generierten Vertices passende Positionen zuzuweisen (Vgl. [Ope16]).

Der Tessellation Control Shader gibt in dieser Anwendung die vier Vertices aus dem Vertex Shader weiter als diejenigen, die unterteilt werden sollen. Außerdem legt er fest, wie der Grashalm unterteilt wird und zwar hauptsächlich entlang der linken und rechten Kante. Der Unterteilungsgrad wird durch die Distanz zur Kamera festgelegt, so wurde einfaches und effizientes LOD umgesetzt.

### 3.1.3 Geometrieerzeugung mit Hilfe von Bézierkurven

Für jeden durch die Tessellation Primitive Generation automatisch erzeugten Vertex wird der Tessellation Evaluation Shader aufgerufen. Dieser erhält Koordinaten relativ zu dem Primitiv, das unterteilt wurde. In diesem Fall also das Quad aus den Vertices die im Vertex Shader generiert wurden. Abbildung 3.2 veranschaulicht dies. Die Koordinaten sind hier mit u und v gekennzeichnet und erstrecken sich von 0 bis 1. Aus den vorhandenen Punkten

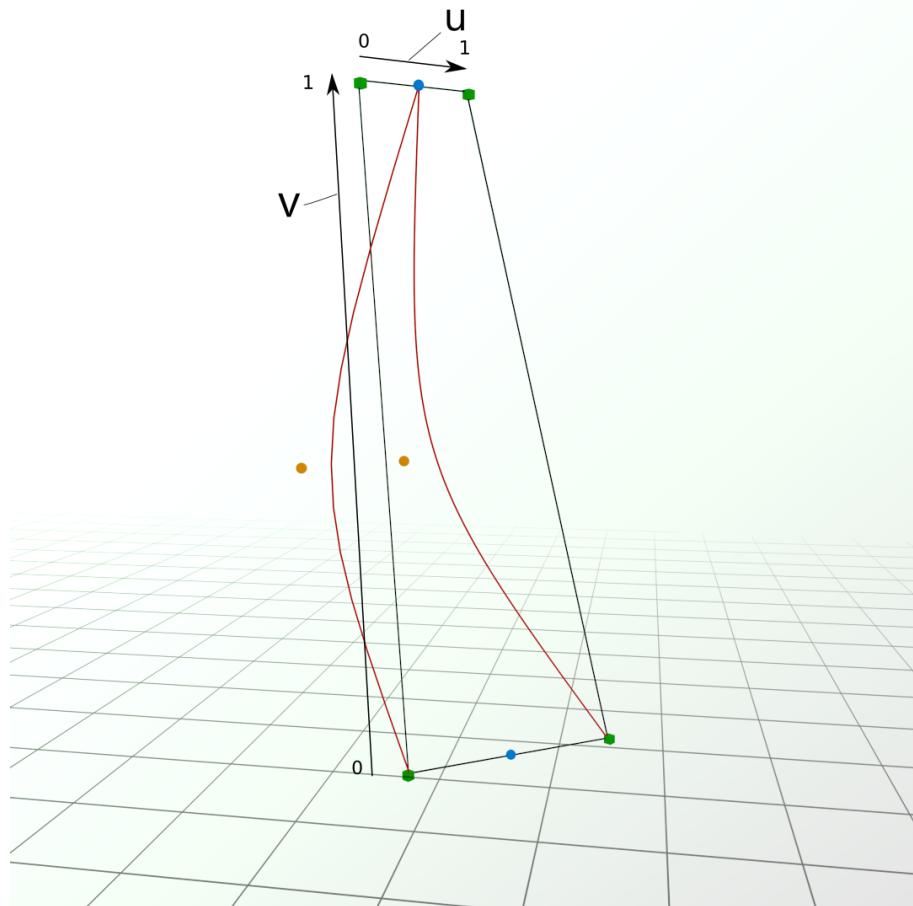


Abbildung 3.2: Veranschaulichung der Verarbeitung des Grashalms durch Tesselierung

können zwei quadratische Bézierkurven aufgestellt werden, die jeweils die rechte und linke Kante des Halmes repräsentieren. Aus den gegebenen relativen Koordinaten kann dann einfach eine Position in der gewünschten Form des Halms für den betrachteten Vertex berechnet werden:

```

left = (1 - v)2 * quadVertices[0] + 2 * (1 - v) * v * influencePoint + v2 * topCenter
right = (1-v)2*quadVertices[3]+2*(1-v)*v*influencePointRight+v2*topCenter

position = mix(left, right, u)

```

Hierbei sind `quadVertices[0..3]` die im Uhrzeigersinn geordneten Vertices des Quads und `topCenter` ist der obere Mittelpunkt. Der zweite Einflusspunkt `influencePointRight` wird durch eine Verschiebung des Einflusspunktes um den Verbindungsvektor der zwei oberen Punkte des Quads berechnet.

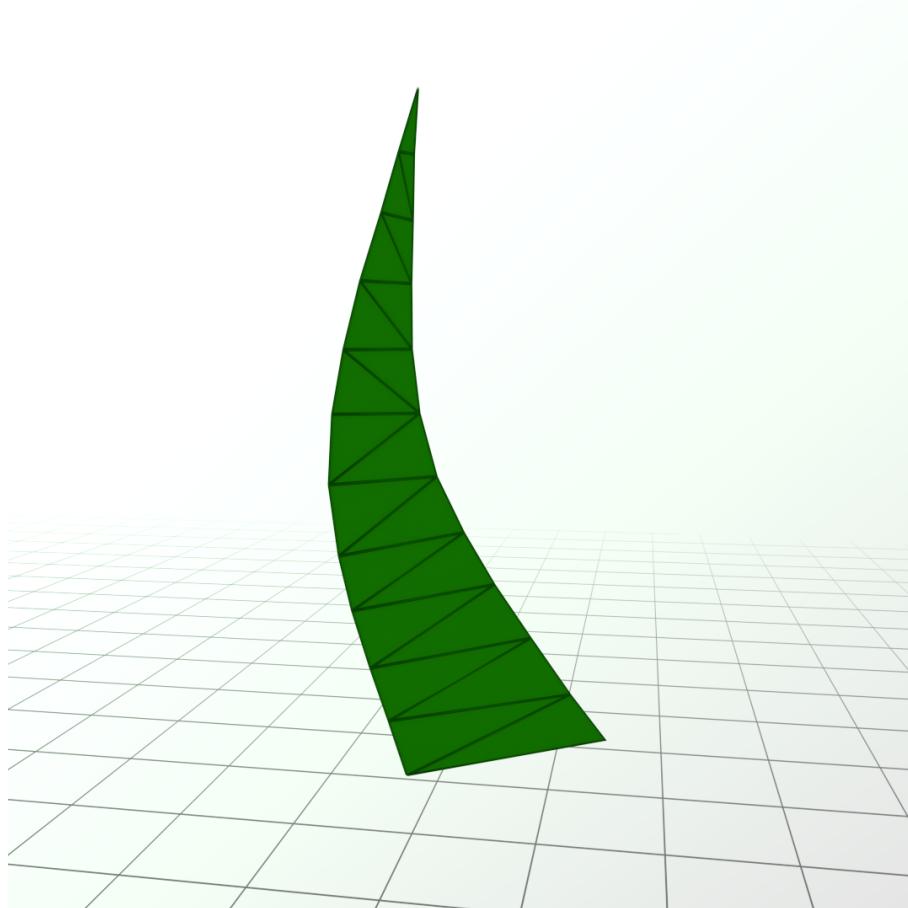


Abbildung 3.3: Ergebnis des Tessellation Evaluation Shaders

## 3.2 Darstellung einer unendlichen Grasfläche

Um die Methoden unter verschiedenen Kriterien zu vergleichen, wie zum Beispiel wie gut auf große Entfernung optimiert wird beziehungsweise werden kann, wurde eine unendliche Grasfläche auf der xz-Ebene implementiert.

### 3.2.1 Unterteilung der sichtbaren Fläche in Quadrate

Es ist nicht möglich den Bereich, in dem Grashalme generiert werden, unendlich groß zu gestalten, also wurde die xz-Ebene in ein Gitter unterteilt. Aus diesem Gitter und dem Viewing Frustum (dem Sichtbereich der Kamera) lassen sich dann alle Quadrate (im Folgenden auch Patches genannt) berechnen, die sich im sichtbaren Bereich befinden. Für diese Patches werden dann über die bereits beschriebenen Shader Grashalme generiert.

Die Quadrate, die sich im Frustum befinden sind konservativ approximiert. (siehe Algorithmus 1)

Hierbei werden alle Patches durchgezählt, angefangen beim Minimum des Frustums. Aufgehört wird nachdem das Maximum überschritten wurde. Alle durchlaufenen Quadrate werden darauf getestet, ob sie sich im Frustum befinden. Dabei wird überprüft, ob sich alle Eckpunkte des Quads außerhalb irgendeiner Ebene des Frustums befinden. Nur dann wird ein Quadrat als außerhalb des Frustums eingestuft.

Dieser Vorgang erfolgt jedes mal wenn sich das Frustum ändert, also die Kamera rotiert oder verschoben wird. Die so berechneten Bereiche werden in einem Array gespeichert und, sobald es zum Rendern des Grases kommt, jeweils mit Grashalmen gefüllt.

Das Ergebnis des Algorithmus ist in Abbildung 3.4 zu sehen. Eine sich scheinbar bis zum Horizont erstreckende Fläche. In Abbildung 3.5 ist zu sehen welche Patches tatsächlich gezeichnet werden.

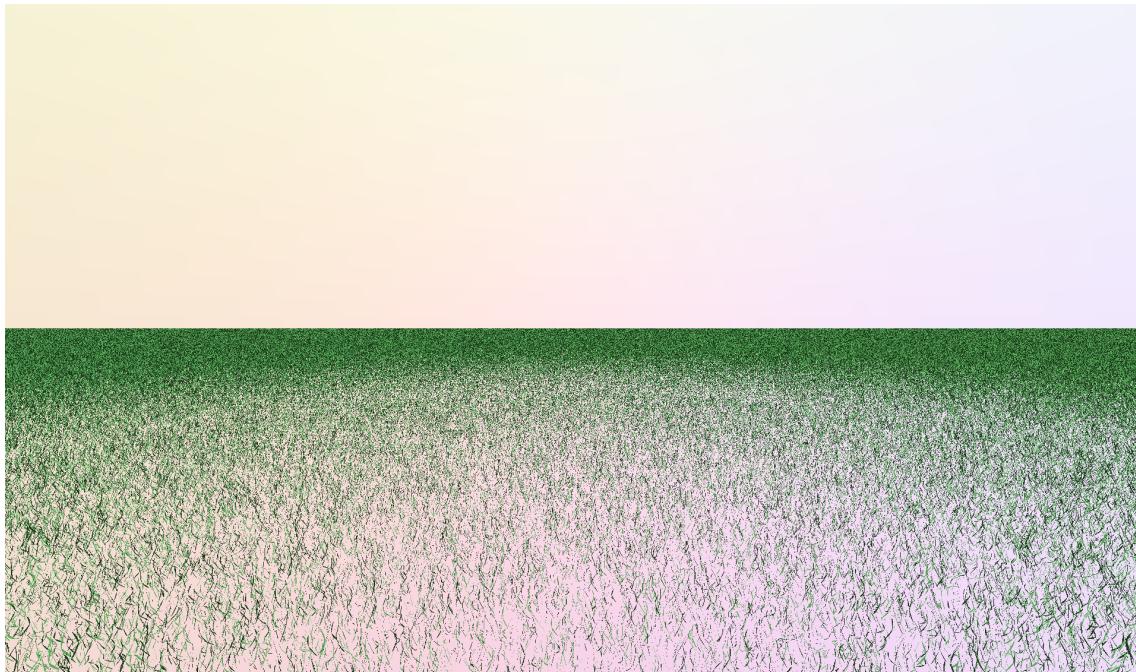


Abbildung 3.4: Eine unendliche Grasfläche

---

**Algorithmus 1** Algorithmus zur Auswahl der zu zeichnenden Quadrate (Patches) für eine unendliche Grasfläche

---

```

1: function CALCULATEPATCHES
2:   start  $\leftarrow$  ROUNDDOWNToNEXTPATCH(minFrustum)
3:   end  $\leftarrow$  ROUNDUPToNEXTPATCH(maxFrustum)
4:   x  $\leftarrow$  start.x
5:   z  $\leftarrow$  start.z
6:
7:   ▷ Füge alle Patches im Frustum zu einem Ar-
     ray aller zu zeichnenden Patches hinzu
8:   for x  $\leq$  end.x do
9:     for z  $\leq$  end.z do
10:      if ISINFRUSTUM(x, z) then
11:        ADDTOARRAY(x, z)
12:      end if
13:      x  $\leftarrow$  x + patchSize
14:    end for
15:    z  $\leftarrow$  z + patchSize
16:  end for
17: end function
18:
19: function ISINFRUSTUM(squareCoordinate)
20:   points  $\leftarrow$  GETPOINTSOFSCUARE(squareCoordinate, patchSize)
21:
22:   for all plane  $\leftarrow$  frustumPlanes do
23:     if POINTSOUTSIDEOPFLANE(points, plane) then
24:       return False
25:     end if
26:   end for
27:   return True
28: end function
29:
30: function POINTSOUTSIDEOPFLANE(points, plane)
31:   for all point  $\leftarrow$  points do
32:     if plane.normal  $\cdot$  (point - plane.point) < 0 then
33:       return False
34:     end if
35:   end for
36:
37:   return True
38: end function

```

---

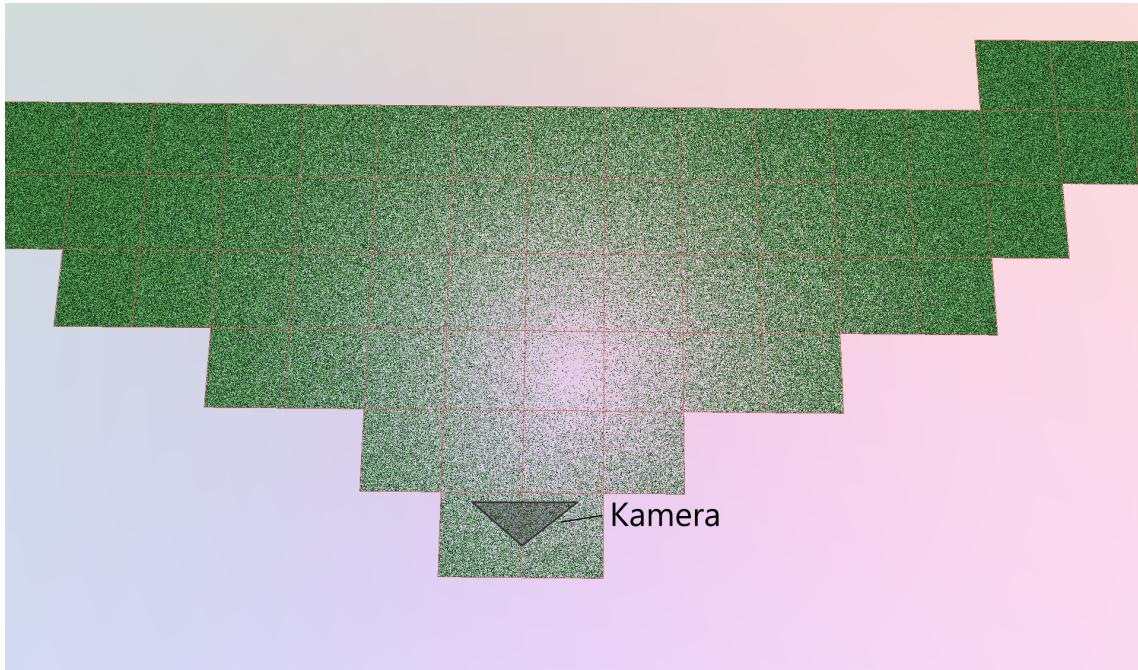


Abbildung 3.5: Top Down Ansicht von Abbildung 3.4 (Tatsächlich gezeichnete Patches)

### 3.2.2 Weitere Unterteilung der Quadrate

In einem weit entfernten Bereich ist es nicht mehr nötig so viele Grashalme zu zeichnen wie in einem nahen Bereich. Tatsächlich ist eine solche Optimierung sogar notwendig um mit einer annehmbaren Geschwindigkeit durchgehend dichtes Gras darzustellen. Dies kann erreicht werden, indem in einem näheren Bereich mehr Grashalme gezeichnet werden. Allerdings ist es auch wünschenswert, dass ein weiter entfernter Bereich größer ist. Somit wird die Anzahl der Quadrate stark beschränkt, was das Ganze wesentlich effizienter gestaltet. Es hat zusätzlich den Vorteil, dass der Übergang zwischen Bereichen weniger offensichtlich ist, da näher am Beobachter mehr Bereiche dargestellt werden. Es würde sich anbieten die Größe der Bereiche mit der Entfernung immer so zu verdoppeln, dass der Footprint eines Quadrates auf dem Bildschirm ungefähr gleich bleibt. Jedoch ergibt sich das Problem, dass das Ergebnis instabil wird. Denn Grashalme sind nur bezüglich Position und Größe des Bereiches in dem sie sich befinden stabil. Also würden sich bei dem beschriebenen Verfahren, da sich die Größe von Bereichen mit der Entfernung verändert, auch die zufälligen Eigenschaften der Grashalme verändern. In diesem Fall bleibt also höchstwahrscheinlich kein Grashalm gleich wenn sich der Beobachter bewegt und die Größe der Bereiche sich verändert.

Somit ist also gewünscht zum Einen Grashalme auszublenden, um nicht unnötig viele auf größere Entfernung zu rendern und zum Anderen nicht offensichtlich ganz andere Halme darzustellen, wenn sich die Kameraposition verändert. Die Bereichsgrößen können aus diesem Grund nicht komplett frei bestimmt werden. Es wurde deshalb als Ausgangspunkt ein größtes Gitter definiert. Die Quadrate dieses Gitters werden immer mit Grashalmen gefüllt, so ergibt sich eine bestimmte Grunddichte. Idealerweise ist diese so gewählt, dass bei der größtmöglichen Entfernung von der Kamera diese Dichte eine Grasfläche ohne auffällige Lücken ergibt.

Jedes mal, wenn das Gitter neu berechnet wird, werden nun, abhängig von der Distanz zur Kamera, die Quadrate weiter rekursiv unterteilt. Die neuen Patches werden nun mit ihren übergeordneten Patches gezeichnet. Somit bleiben von weit weg schon sichtbare Halme

immer erhalten, aber umso näher der Beobachter ist desto mehr Halme werden gezeichnet. Außerdem werden Patches kleiner und Übergänge zwischen verschiedenen dichten Patches somit feiner. Das Ergebnis des Algorithmus ist in Abbildung 3.6 und Abbildung 3.7 zu

---

**Algorithmus 2** Algorithmus zur rekursiven Unterteilung der Patches
 

---

```

1: function ADDANDSUBDIVIDE(position, size, density, recursionDepth)
2:   if ISINFRUSTUM(position) then
3:     patch : GrassPatch
4:     patch.position  $\leftarrow$  position
5:     patch.size  $\leftarrow$  size
6:     patch.density  $\leftarrow$  density
7:     ADDTOARRAY(patch)
8:
9:      $\triangleright$  Definiere eine Pseudodistanz so, dass sie
       größer als die maximale Entfernung zum
       Unterteilen ist, falls die maximale Rekur-
       sionsstiefe erreicht ist
10:
11:    pseudoDistance  $\leftarrow$  DISTANCEPATCHTOCAMERA(position, size)
12:    pseudoDistance  $\leftarrow$  pseudoDistance + (recursionDepth/maxRecurstionDepth) *
       maxDistToSubdivide
13:
14:    if pseudoDistance < maxDistToSubdivide then
15:       $\triangleright$  Erstelle vier Patches in dem das aktuelle
         geviertelt wird. Rufe diese Funktion rekur-
         siv auf diesen auf
16:
17:      newSize  $\leftarrow \frac{\text{size}}{2}$ 
18:      i  $\leftarrow 0$ 
19:      for i < 4 do
20:        newPosition.x  $\leftarrow (\text{postion.x} + (i \bmod 2) * \text{newSize})$ 
21:        newPosition.z  $\leftarrow (\text{postion.z} + (\text{int}(\frac{i}{2})) * \text{newSize})$ 
22:        newDensity  $\leftarrow \text{density} * \text{densityFactor}$ 
23:        newDepth  $\leftarrow \text{recursionDepth} + 1$ 
24:        ADDANDSUBDIVIDE(newPosition, newSize, newDensity, newDepth)
25:
26:        i  $\leftarrow i + 1$ 
27:      end for
28:    end if
29:  end if
30: end function
  
```

---

sehen. Patches nahe der Kamera werden unterteilt und erhalten eine höhere Dichte. So wird ein Eindruck einer konsistent dichten Grasfläche erzeugt. Jedoch werden auf größere Entfernungen wesentlich weniger Halme gezeichnet.

In Algorithmus 2 ist der Algorithmus zur rekursiven Unterteilung von Patches zu sehen. Dieser wird in Algorithmus 1 in Zeile 11 anstatt der if Abfrage und ADDTOARRAY aufgerufen.

Für die Unterteilung wird hier die Entfernung zur Kamera genutzt. Je nach Blickwinkel kann es allerdings dazu kommen, dass Patches nicht unterteilt werden, die viel Platz auf dem Bildschirm einnehmen und somit eine weniger dichte Grasfläche haben als sie vielleicht sollten. Eine bessere aber auch performanceaufwändigere Metrik wäre, zu berechnen wie

groß der Footprint eines Patches ist und die Unterteilung danach vorzunehmen. Die Metriken sollten je nach Anwendung gewählt werden, wenn nie ein Blickwinkel möglich ist, der Probleme verursachen kann (zum Beispiel von schräg oben) lohnt es sich wahrscheinlich nur auf die Distanz zur Kamera zu achten.



Abbildung 3.6: Unendliche Grasfläche mit Unterteilung der nahen Bereiche und entsprechender Erhöhung der Grasdichte

Bei der Kamerabewegung neue Grashalme einzufügen kann, ohne inakzeptablen Performanceverlust, nicht vermieden werden, allerdings kann es versteckt werden. Dazu gibt es diverse Techniken, die bei solchen und ähnlichen LOD Probleme Anwendung finden. Zum Beispiel kommt oft Alpha Blending zum Einsatz um das Erscheinen neuer Objekte beziehungsweise Details zu verbergen.

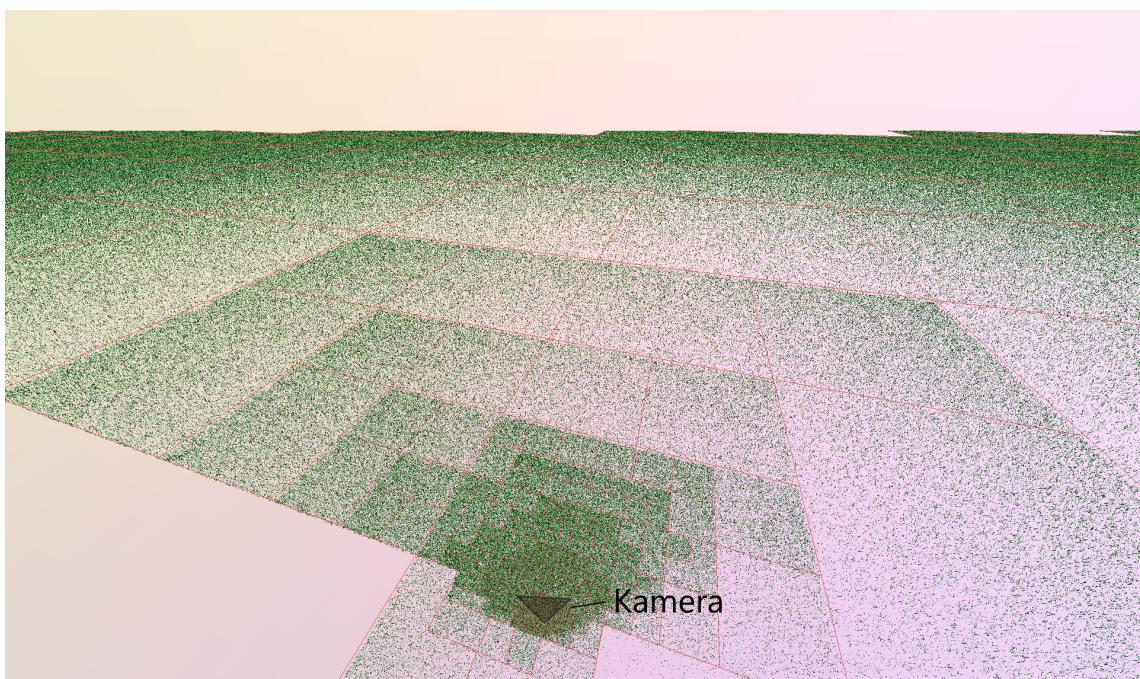


Abbildung 3.7: Top Down Ansicht von Abbildung 3.7 Unterteilung der Patches und Erhöhung der Grasdichte nahe der Kamera

## **4. Zeichnen von Gras unter Nutzung eines Compute Shaders**

Im Folgenden wird das Hauptthema dieser Arbeit behandelt: Die Entwicklung einer neuen Methode Gras zu zeichnen.

### **4.1 Konzept**

Die im Zuge dieser Arbeit entwickelte Lösung, bedient sich des in OpenGL relativ neu eingeführten Konzeptes des Compute Shaders. Der Compute Shader kann, wie CUDA oder OpenCL, beliebige Berechnungen auf der GPU durchführen. Im Gegensatz zu CUDA oder OpenCL ist er jedoch Teil von OpenGL und lässt sich damit leicht auch in einem OpenGL Kontext verwenden. Es ist somit möglich Berechnungen massiv parallel auf der GPU durchzuführen und die Ergebnisse für eine OpenGL Anwendung zu benutzen.

Es soll in dieser Arbeit ein Compute Shader für das Zeichnen von Gras verwendet werden, der Shader soll dabei einzelne Halme auf den Bildschirm zeichnen. Idealerweise unter maximaler Ausnutzung der parallelen Kapazitäten der GPU. Um dies zu bewerkstelligen, wird der Bildschirm in Bereiche unterteilt. Jeder Bereich wird durch eine Arbeitsgruppe bearbeitet, die je 32 Threads enthält. Alle Threads führen das gleiche Programm aus und sollen am Ende jeweils einen oder mehrere Grashalme zeichnen.

Es wird erwartet, dass nicht alle Grashalme gleichzeitig gezeichnet werden können, da in einer Arbeitsgruppe immer nur 32 Halme gleichzeitig verarbeitet werden. Sofern sie nicht parallel verarbeitet werden, sollen die Halme von vorne nach hinten durchlaufen werden. Das Programm soll später so konfiguriert werden, dass ein Bildschirmteil mindestens 32 Halme breit ist damit die 32 parallel verarbeiteten Halme nie hintereinander sind. Der Überdeckungstest und Alpha-Blending für einzelne Halme vereinfacht sich ungemein, da alle Pixel, die schon gezeichnet sind vor dem gerade bearbeiteten Halm liegen müssen. Weiter hinten liegende Halme können außerdem abbrechen zu zeichnen sobald auf Halm gestoßen wird, die schon gezeichnet sind. Dies bedeutet also weniger Arbeit für größtenteils überdeckte Halme.

## 4.2 Finden von Positionen zum Zeichnen der Grashalme

Die erste Aufgabe, die sich stellt, ist das Finden von Koordinaten, an denen Threads Grashalme zeichnen sollen. Gearbeitet wird hier in Weltkoordinaten die dann auf den Viewport abgebildet werden können.

### 4.2.1 Unterteilung des Bildschirms

Zuerst müssen Grenzen für die Arbeitsgruppen gesteckt werden. Jede Arbeitsgruppe soll einen Teil des Bildschirms bearbeiten. Dazu wurde der Bildschirm in  $32 \times 32$  Pixel große Bereiche unterteilt. Im Compute Shader wird mit der Größe des Viewports in Pixeln ein Teilfrustum berechnet, in dem die Arbeitsgruppe operiert. Nun muss festgestellt werden, wo die Grashalme dieser Arbeitsgruppe gezeichnet werden sollen. Dazu wird zuerst jede Kante des Teilfrustums mit dem Objekt geschnitten auf dem das Gras wachsen soll (in diesem Fall wieder die xz-Ebene). In Abbildung 4.1 ist dies veranschaulicht.

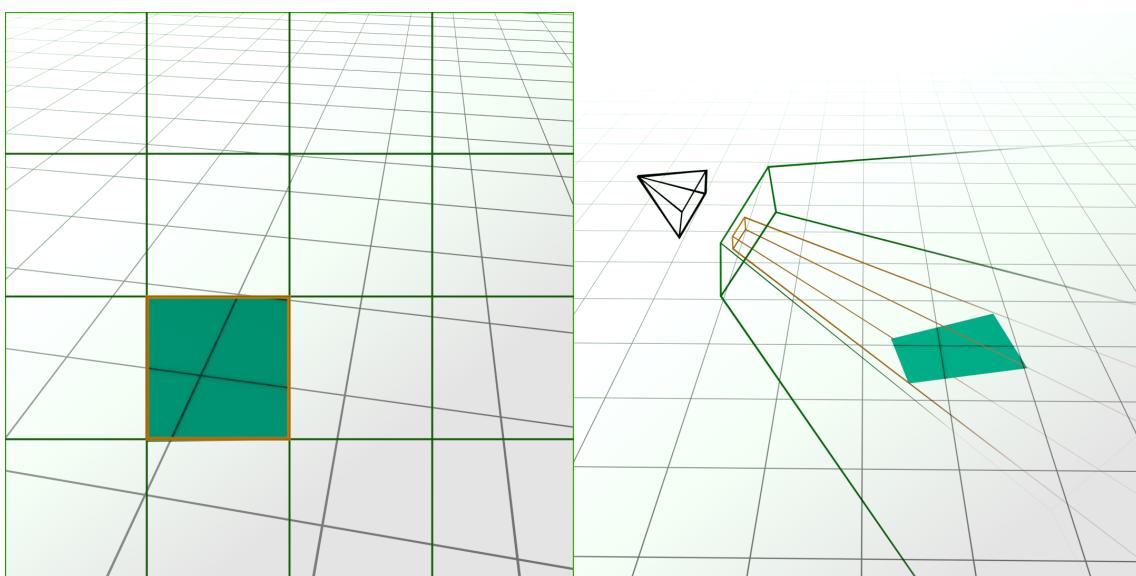


Abbildung 4.1: Unterteilung des Viewports in Quadrate für die Arbeitsgruppen und Schnitt eines Teilfrustums mit der xz-Ebene (links aus Sicht der Kamera und rechts von außerhalb)

Der Schnitt enthält nun alle Grashalme, die in dem entsprechenden Bildschirmteil ihre Wurzel haben. Es müssen jedoch auch noch alle Grashalme berücksichtigt werden die in den Bildschirmausschnitt hineinwachsen. So kann zum Beispiel die Spitze eines Grashalms, dessen unteres Ende in einem anderen Teil des Bildschirms liegt, auch in dem betrachteten Teil liegen. Alle Grashalme für die dies möglich ist, sollten also auch in diesem Bildschirmteil gezeichnet werden. Um dies zu bewerkstelligen wird konservativ approximiert. Dazu wird das Teilfrustum noch einmal mit der um die maximale Grashöhe nach oben verschobene xz-Ebene geschnitten. Der Schnitt wird auf die xz-Ebene projiziert und in NDC-Koordinaten umgewandelt, mit diesen kann das Frustum entsprechend erweitert werden (Abbildung 4.2).

Der, in dem betrachteten Bildschirmbereich, möglicherweise enthaltene Teil des Raumes wird also geschnitten mit der Ebene der möglichen Grashalmspitzen und der möglichen Grashalmwurzeln. Alle Grashalme in diesen Schnitten werden durch die Erweiterung des Frustums später betrachtet.

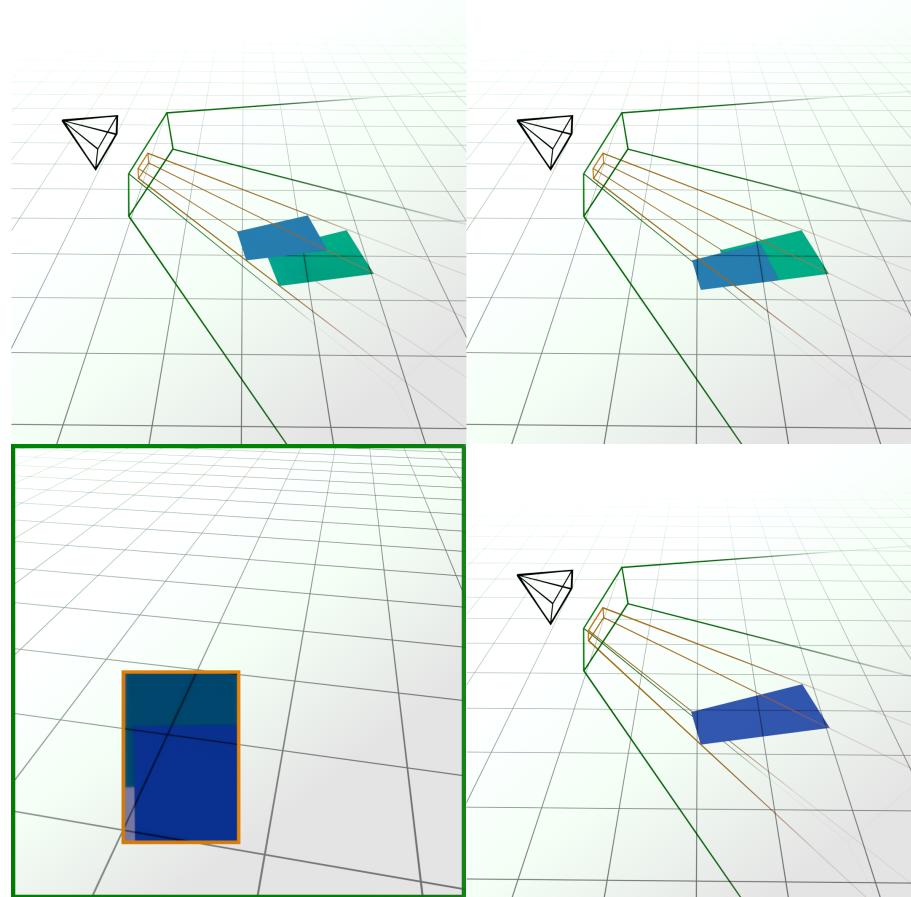


Abbildung 4.2: Links oben: Schnitt mit der Ebene der Graspitzen (blau). Rechts oben: Verschieben des Schnittes auf die xz-Ebene. Unten: Erweiterung des Teilfrustums

#### 4.2.2 Vorarbeit für die Front-to-Back Abarbeitung

Um die Grashalme im Folgenden gut abarbeiten zu können wird die Fläche auf der das Gras wachsen soll in ein Gitter aufgeteilt. In jeder der Gitterzellen wird später ein Grashalm (mit zufälliger Position in der Zelle) gezeichnet. Um die Front-to-Back Eigenschaft zu gewährleisten, sollen die Zellen später Zeile um Zeile von vorne nach hinten abgearbeitet werden. Dazu berechnet das Programm zuerst zwei Vektoren: Die Gitterrichtung in die die Kamera zeigt und ein Vektor rechtwinklig dazu. Die Richtungen sind jeweils eine von acht auf dem Gitter. Die zeilenweise Abarbeitung kann also auch diagonal geschehen. In diesem Sonderfall erhält der Vektor, der in Richtung der Kamera zeigt (Front-to-Back Vektor) die Länge  $\frac{1}{2} * \sqrt{2}$  und der rechtwinklig dazu  $\sqrt{2}$ . So ist gewährleistet, dass auch bei diagonaler Iteration alle Zellen betrachtet werden. In diesem Fall muss allerdings wenn ein Schritt vorwärts gegangen wird, entlang der neuen Zeile auf eine Gitterposition gerundet werden, da ansonsten die nächste Position in der Mitte einer Zelle läge.

Als nächstes wird der Startpunkt für die Abarbeitung als der Schnittpunkt des Teilfrustums, der bezüglich der Kamerarichtung am kleinsten ist, festgelegt. Der Vektor rechtwinklig zur Kamerarichtung wird falls nötig gespiegelt, so dass er ungefähr von der Kamera weg zeigt. Das Konzept wird in Abbildung 4.3 skizziert. Es sind die beiden Vektoren sowie der ausgewählte Startpunkt zu sehen. Weiterhin ist der abgerundete Startpunkt eingezeichnet, welcher bei der Abarbeitung als eigentlicher Anfang genutzt wird, um zu gewährleisten, dass durch alle Gitterzellen im Teilfrustum iteriert wird.

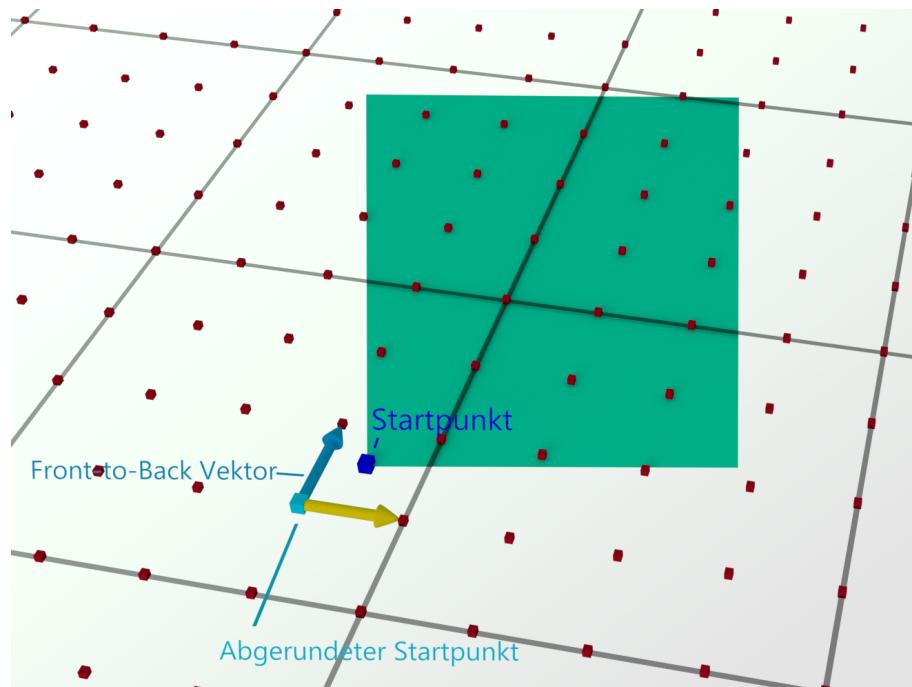


Abbildung 4.3: Reguläre Gitterrichtung der Kamera (Front-to-Back Vektor), sowie rechtwinkliger Vektor dazu (Gelb). Außerdem Startpunkt für die Iteration durch die Gitterzellen im Teilfrustumsschnitt. Eckpunkte aller Gitterzellen sind hier in rot gekennzeichnet

Später bei der Iteration muss geprüft werden, ob eine Gitterzelle außerhalb des Frustums ist, um festzustellen ob in die nächste Zeile gewechselt werden sollte. Um diesen Test zu beschleunigen, wird berechnet, welche Ebenen des Frustums geprüft werden müssen. Dies geschieht indem herausgefunden wird, in welche Richtung der zum Front-to-Back Vektor rechtwinklige zeigt und die Ebenen des Frustums in dieser Richtung bestimmt werden. Diese werden in der Iteration verwendet um zu Testen ob aus dem Frustum herausgelaufen wurde.

#### 4.2.3 Aufteilung der Gitterzellen auf Threads

Alle bisherigen Berechnungen waren nur arbeitsgruppenspezifisch. Im Folgenden werden Gitterzellen auf die 32 Threads jeder Arbeitsgruppe aufgeteilt, damit jeder Thread Grashalme an der richtigen Stelle zeichnen kann. Dabei wurde auch implementiert, dass die Gitterabstände sich mit der Entfernung vergrößern. Genauer gesagt sollen sie sich so verdoppeln, dass das Gitter auf dem Bildschirm ungefähr gleich groß bleibt. Der komplette Algorithmus ist in Algorithmus 3 beschrieben und wird von jedem Thread simultan ausgeführt.

##### 4.2.3.1 Grundlegende Vorgehensweise

Das Konzept hinter dem Algorithmus ist einfach: teile die im Teilfrustum vorhandenen Gitterzellen von vorne nach hinten auf die 32 Threads auf. Um dies zu ermöglichen gibt es zwei Schleifen: die äußere läuft so lange bis alle Zellen abgearbeitet sind, die innere so lange bis ein Thread eine Zelle gefunden hat. Ein Thread berechnet seine Position aus dem Startpunkt der aktuellen Zeile, seiner Id und der aktuellen Gittergröße.

Nun wird festgestellt, ob die berechnete Gitterzelle im Frustum liegt, falls ja hat der Thread seine Position gefunden und kann dort zeichnen, falls nicht läuft die innere Schleife weiter

bis eine Zelle gefunden ist oder bis in allen Zellen gezeichnet wurde. Das Konzept ist in Abbildung 4.4 skizziert.

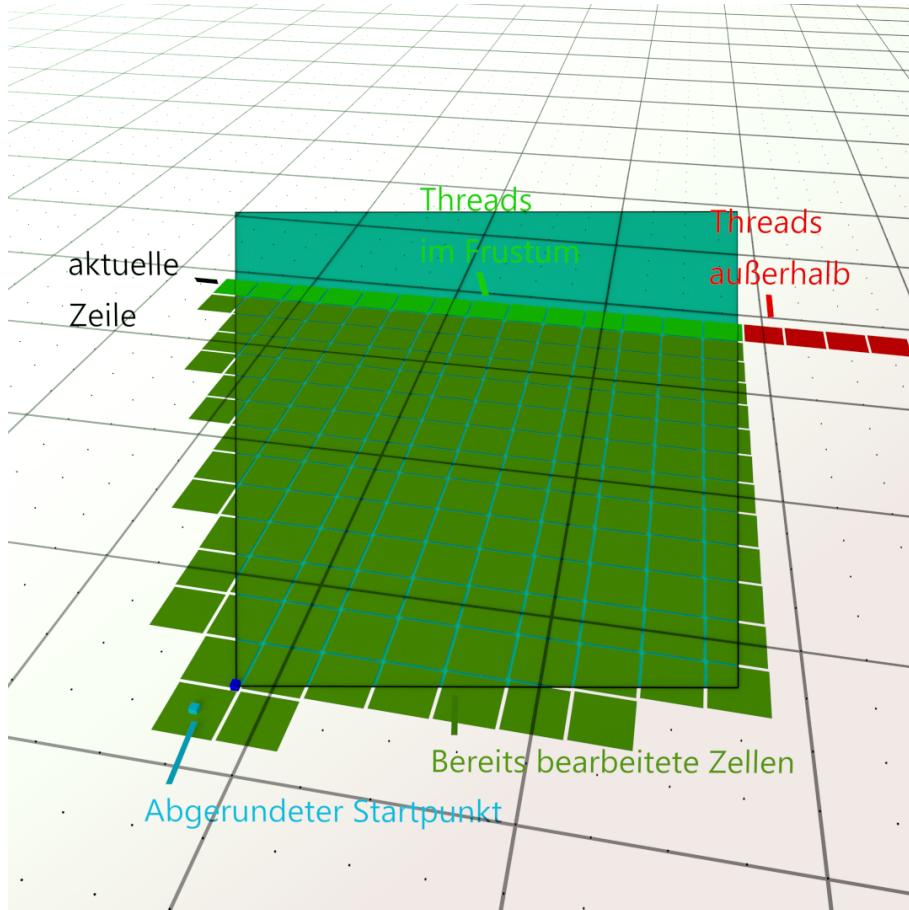


Abbildung 4.4: Iteration der Threads durch die Gitterzellen im Frustum

In der aktuellen Zeile wird für jeden Thread die in dieser Zeile zugehörige Gitterzelle berechnet und getestet, ob diese sich noch im Frustum der Arbeitsgruppe befindet. Alle Threads, die ihre Zelle gefunden haben gehen über zum Zeichnen, alle anderen suchen in der nächsten Zeile weiter. Da nun weniger Threads aktiv auf der Suche sind, müssen diese bei der nächsten Berechnung der Zelle für jeden Thread berücksichtigt werden. Deshalb wird gezählt wie viele Threads noch aktiv sind. Daraus, der Thread Id, der Gittergröße und dem Zeilenanfang wird dann wieder die Position der Zelle berechnet (Algorithmus 3 Zeile 37).

Um den Zeilenanfang zu berechnen, wird die Zeile mit dem Frustum geschnitten (Algorithmus 3 Zeile 16). Tatsächlich wird immer der Anfang der nächsten, zu der in der Iteration betrachteten Zeile, berechnet. Somit kann der Zeilenanfang für die aktuelle oder die nächste Iteration angeglichen werden, falls das Frustum noch eine Zelle jenseits des Zeilenanfangs beinhalten kann (Algorithmus 3 Zeile 21).

Abgebrochen wird sobald in einer neuen Zeile kein Thread mehr im Frustum ist (Algorithmus 3 Zeile 84).

### 4.2.3.2 Reduktion der Gittergröße mit der Entfernung

Genau wie auch bei der Erzeugung von Geometriegras (Kapitel 3), sollen hier mit größerer Entfernung weniger Grashalme angezeigt werden. Um dies zu erreichen wird die Gittergröße relativ zu der Entfernung der Kamera variiert. So kann mit größerer Entfernung ein entsprechend größeres Gitter verwirklicht werden. Die Gittergröße wird immer so bestimmt, dass eine Strecke in der Mitte des Bildschirms mit der gegebenen Entfernung, eine bestimmte Länge in Pixeln hat. So sollten Gitterzellen immer ungefähr die gleiche Größe in Pixeln haben.

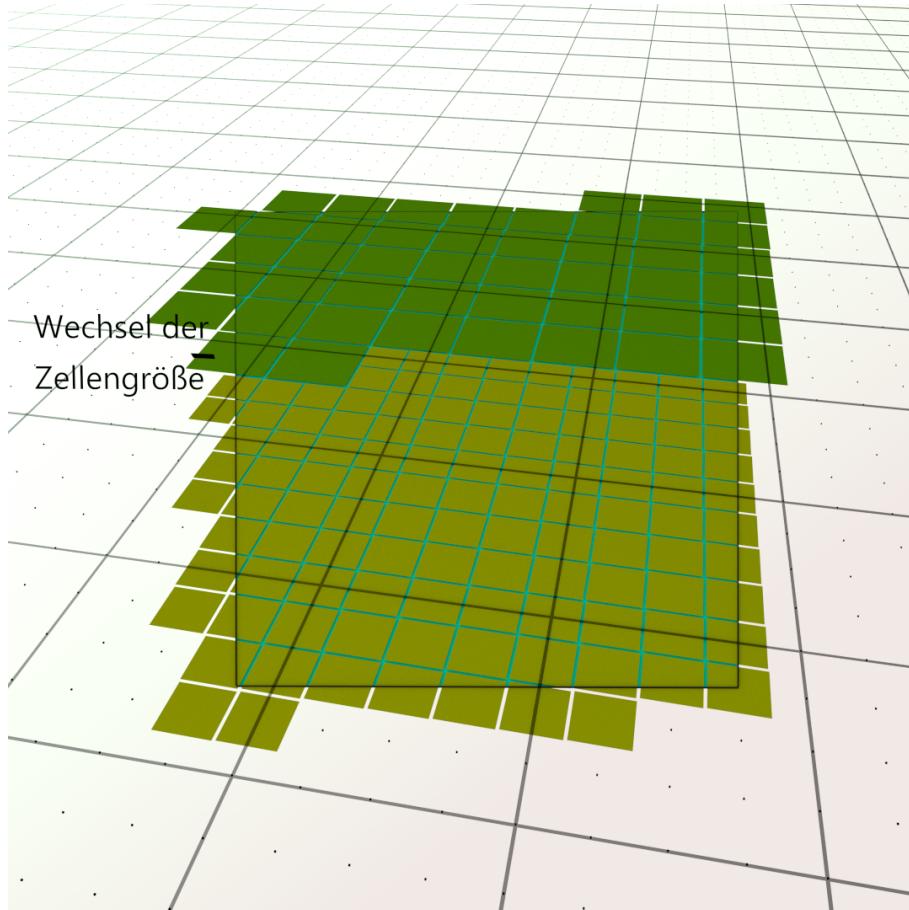


Abbildung 4.5: Variation der Gittergröße

Die so berechnete Zellengröße kann nicht ohne weiteres verwendet werden, ansonsten hätte jeder Gitterpunkt eine eigene Zellengröße. Deshalb wird auf ein Vielfaches der initialen Gittergröße gerundet, dieses Vielfache ist in diesem Fall eine ganzzahlige Zweierpotenz, so wird die Gittergröße immer nur verdoppelt. Bei dem Übergang von einer Größe auf die Nächste werden somit immer vier Zellen zu einer zusammengefasst. Die initiale Gittergröße ist bestimmt durch die Gittergröße in Pixeln auf dem Bildschirm an einer bestimmten Entfernung.

Die Gittergröße wird für die aktuelle Zeile konservativ approximiert. Das heißt es kann passieren, dass die Zeile Teile enthält, in denen die Gittergröße größer sein müsste (siehe auch Abbildung 4.7). Falls das passiert, werden entsprechend Zellen übersprungen (ausmaskiert), die in diesem Fall zu viel sind. Es wird also überprüft, um wie viel die lokale Gittergröße größer ist als die in der Iteration genutzte und dementsprechend zwischen Zellen mehrere Zellen weggelassen, um die richtige Zellengröße zu gewährleisten (Algorithmus 3 Zeile 54). Dies funktioniert, da sich die Gittergröße immer verdoppelt. Falls also

zum Beispiel einige Zellen existieren, für die eigentlich schon die nächst größere Gittergröße gelten müsste, diese aber nicht verwendet wird, wird jede zweite Zelle ausmaskiert. Die restlichen Zellen erhalten dann die entsprechend größere Gitterzellengröße. Erkennt ein Thread die gefundene Gitterzelle als ausmaskiert, sucht dieser weiter nach einer neuen Zelle.

#### 4.2.3.3 Weicher Übergang zwischen Gitterstufen

Weiterhin wird in dem vorgestellten Algorithmus zwischen den Gitterstufen ein relativ weicher Übergang implementiert, indem Zellen, je nach dem wie nahe diese an der optimalen Entfernung für eine Gittergröße sind, pseudozufällig ausmaskiert werden(Algorithmus 3 Zeile 59).

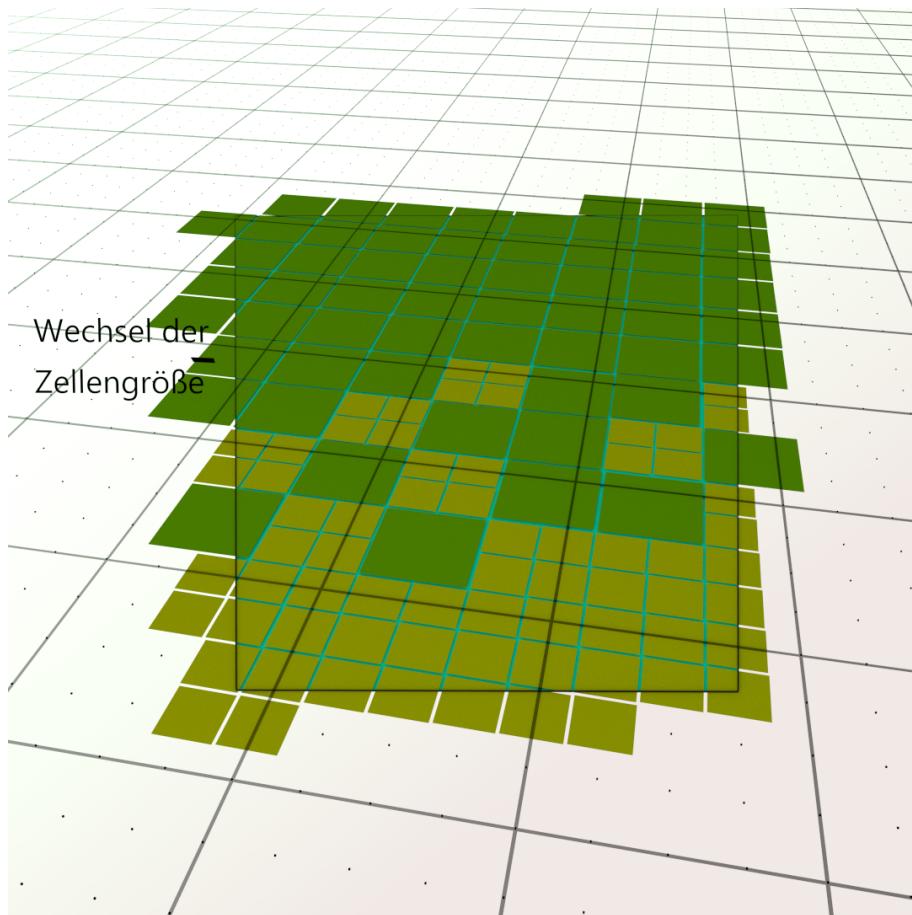


Abbildung 4.6: Weicher Übergang zwischen Gittergrößen

Um einen solchen Übergang umzusetzen, werden immer vier Zellen, die in der nächsten Gitterstufe genau eine ausmachen würden, als eine zusammengefasst indem drei davon ausmaskiert werden und die vierte entsprechend vergrößert wird. Die vier (beieinander) liegende Zellen sind zufällig ausgewählt. Die Wahrscheinlichkeit dafür ist dementsprechend größer umso näher die Zellen am Übergang zur nächsten Gittergröße sind. Dieser Übergang ist in Abbildung 4.6 veranschaulicht.

#### 4.2.3.4 Implementierung

Im Folgenden sind wichtige Aspekte der Implementierung des Algorithmus erklärt. Algorithmus 3 implementiert die Aufteilung mit Reduktion der Gittergröße und weichen Übergängen wie beschrieben. Um die Aufteilung der Zellen im Teilfrustum der Arbeits-

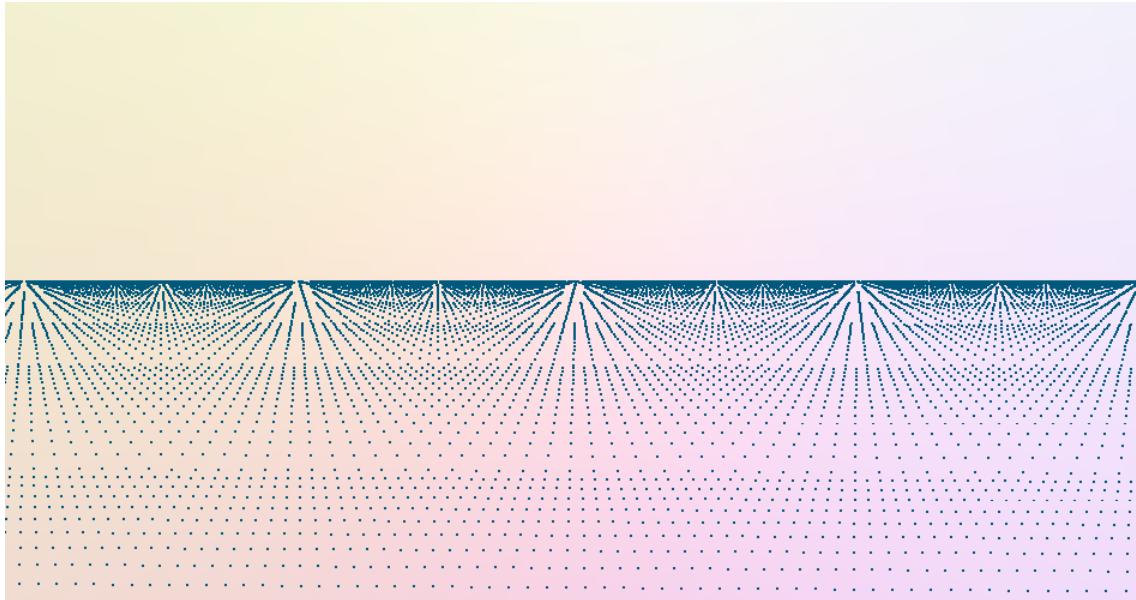


Abbildung 4.7: Ergebnis der Implementierung: Zeichnung jeder Zelle als Punkt, Zellengröße wird pro Zeile konservativ approximiert

gruppe auf die Threads zu ermöglichen müssen diese in irgend einer Form miteinander kommunizieren. Für die Kommunikation wurde die Erweiterung NV\_thread\_group<sup>1</sup> verwendet, sie bietet die Funktion BALLOTTHEARDNV(bool value), welche den Austausch von bool Werten innerhalb der 32 Threads erlaubt. Die übergebenen Werte werden gesammelt und jeder Thread bekommt einen 32 Bit unsigned integer in dem jedes Bit bool Wert eines Threads repräsentiert.

Der Algorithmus wird von jedem Thread simultan ausgeführt und da mit 32 Threads pro Arbeitsgruppe gearbeitet wird, ist gewährleistet, dass diese Threads immer an der gleichen Stelle des Codes sind beziehungsweise aufeinander warten, falls eine Codegabelung erfolgt (Lockstep, siehe auch [NVI09; NVI15]).

Abbildung 4.7, Abbildung 4.8 und Abbildung 4.9 sind Screenshots der Implementierung. Die Zellen werden als Punkte dargestellt. Der Bildschirm wurde in 32 x 32 Pixel große Quadrate unterteilt und jeweils einer Arbeitsgruppe zugewiesen. Jede Arbeitsgruppe berechnet ihr Teilfrustum und teilt die Zellen auf ihre 32 Threads auf. Jeder Thread geht dann über zum Zeichnen, bei dem in diesem Fall ein einzelner Punkt gezeichnet wird.

In Abbildung 4.7 ist die Zellengröße für jede Zeile konservativ approximiert. Sie kann also potentiell zu klein sein (Vor allem links und rechts zu sehen).

Abbildung 4.8 zeigt die korrigierte Variante, bei der für zu kleine Gittergrößen entsprechend Zellen weggelassen werden, um die nächstgrößere Gittergröße zu erhalten.

Schlussendlich ist in Abbildung 4.9 der weiche Übergang zu sehen, der das Ergebnis des zufälligen Zusammensetzens von vier Gitterzellen ist.

<sup>1</sup>[https://www.opengl.org/registry/specs/NV/shader\\_thread\\_group.txt](https://www.opengl.org/registry/specs/NV/shader_thread_group.txt)

---

**Algorithmus 3** Algorithmus zur Aufteilung der Gitterzellen des Teilfrustums auf die Threads

---

```

1: function FINDCELLANDDRAW(flooredStartPoint)
2:   lineOneStart  $\leftarrow$  flooredStartPoint
3:   newLine  $\leftarrow$  true
4:   newGroup  $\leftarrow$  true
5:   while iteration not finished do
6:     while searching do
7:       if newLine  $\vee$  newGroup then
8:          $\triangleright$  Berechne kleinste Gittergröße(stepSize) für
      aktuelle Zeile und die Startposition für die
      nächste
9:
10:    stepSize  $\leftarrow$  GETSTEPSENSE(lineOneRay)
11:    ftbStep  $\leftarrow$  FtBdirection * stepSize
12:    horizontalStep  $\leftarrow$  lineDirection * stepSize
13:
14:    lT  $\leftarrow$  lineOneStart + ftbStep
15:    lTRay  $\leftarrow$  Ray(lT, -horizontalStep)
16:    lineTwoStart  $\leftarrow$  INTERSECTFRUSTUM( $\neg$ planesToCheck, lTRay)
17:    lineTwoStart  $\leftarrow$  FLOORTOGRIDBYDIR(lineTwoStart, horizontalStep)
18:
19:     $\triangleright$  Falls der Zeilenanfang nicht von der vor-
      herigen Iteration der äußeren Schleife
      stammt: Passe Zeilenanfang an, falls es
      möglich ist, dass weitere Gitterzelle enthal-
      ten sind
20:
21:    adptLO  $\leftarrow$  LESSTHANBYDIR(lineTwoStart, lineOneStart, horizontalStep)
22:    adptLT  $\leftarrow$  LESSTHANBYDIR(lineOneStart, lineTwoStart, horizontalStep)
23:    if adptLO  $\wedge$  newLine then
24:      lineOneStart  $\leftarrow$  lineTwoStart - ftbStep
25:    else if adptLT  $\wedge$  newLine then
26:      lineTwoStart  $\leftarrow$  lineOneStart + ftbStep
27:    end if
28:    lineStart  $\leftarrow$  lineOneStart
29:
30:    newLine  $\leftarrow$  false
31:    newGroup  $\leftarrow$  false
32:  end if
33:
34:
35:  if lineStart is valid then  $\triangleright$  Falls der Frustumsschnitt existiert
36:    localCompactId  $\leftarrow$  0
37:    activeBallot  $\leftarrow$  BALLOTTHEADNV(true)
38:    if threadId > 0 then  $\triangleright$  Thread id in der Arbeitsgruppe [0,31]
39:       $\triangleright$  Zähle aktive Threads mit einer kleineren
      Thread id als die eigene
40:
41:      shiftedBallot  $\leftarrow$  activeBallot << ( $32 - threadId$ )
42:      localCompactId  $\leftarrow$  BITCOUNT(shiftedBallot)
43:    end if
44:
```

---

---

```

45:      ▷ Berechne die, dem Thread entsprechende,
        Gitterzellenposition und überprüfe ob die
        Zelle im Frustum ist
46:
47:       $steps \leftarrow (localCompactId + previousThreadsInLine)$ 
48:       $currentPos \leftarrow lineStart + steps * horizontalSteps$ 
49:       $localStepSize \leftarrow \text{GETSTEP SIZE}(currentPos)$ 
50:
51:       $outOfFrustum \leftarrow \text{GRIDCELLOUTOFFRUSTUM}(currentPos, localStepSize)$ 
52:      ▷ Ausmaskieren überschüssiger Zellen wenn
         die lokale Gittergröße größer ist als die be-
         nutzte
53:       $iPos \leftarrow \text{round}(\frac{currentPos}{stepSize})$ 
54:       $maskedOut \leftarrow localStepSize > stepSize$ 
55:       $maskedOut \leftarrow maskedOut \wedge \frac{iPos}{localStepSize}$  is not an integer
56:
57:      ▷ Setze einen weichen Übergang zwischen
         Gittergrößen um, indem immer 3 Zellen
         ausmaskiert werden und die vierte expan-
         diert wird. Die Wahrscheinlichkeit dafür
         steigt umso näher die Zellen an der nächs-
         ten Gittergröße sind
58:
59:      if  $\neg maskedOut$  then
60:           $relativePos \leftarrow \frac{currentPos}{nextBiggerStepSize}$ 
61:           $nextBiggerCellPos \leftarrow \text{floor}(relativePos) * nextBiggerStepSize$ 
62:
63:           $rng \leftarrow \text{RNGINIT}(\text{round}(\frac{nextBiggerCellPos}{initialStepSize}))$ 
64:
65:           $nextBiggerCellStepSize \leftarrow \text{GETSTEP SIZE}(nextBiggerCellPos)$ 
66:           $blend \leftarrow \text{GETBLEND}(nextBiggerCellPos)$ 
67:
68:           $mergeCells \leftarrow \text{RNG.NEXT} < blend$ 
69:           $mergeCells \leftarrow mergeCells \vee nextBiggerCellStepSize = localStepSize$ 
70:           $mergeCells \leftarrow mergeCells \wedge nextBiggerCellStepSize \geq localStepSize$ 
71:          if  $mergeCells \wedge nextBiggerCellPos = currentPos$  then
72:               $localStepSize \leftarrow nextBiggerStepSize$ 
73:          else
74:               $maskedOut \leftarrow mergeCells$ 
75:          end if
76:      end if
77:  else
78:       $outOfFrustum \leftarrow True$ 
79:  end if
80:
81:  ▷ Stoppe falls in einer neuen Zeile kein
     Thread eine Position im Frustum hat
82:
83:   $noneIn \leftarrow \text{BALLOTTTHREADNV}(outOfFrustum) == \text{BALLOTTTHREADNV}(True)$ 
84:  if  $newLine \wedge noneIn$  then
85:      stop iteration
86:  end if

```

---

---

```

87:      if  $\neg outOfFrustum$  then
88:          activeBallot  $\leftarrow$  BALLOTTHREADNV(true)
89:          prevThreadsInLine  $\leftarrow$  prevThreadsInLine + BITCOUNT(activeBallot)
90:      end if
91:      searching  $\leftarrow$  maskedOut  $\vee$  outOfFrustum
92:
93:      if BALLOTTHREADNV(outOfFrustum)  $\neq$  0 then
94:           $\triangleright$  Mindestens ein Thread ist nicht im Frustum. Die Zeile ist also komplett bearbeitet
95:
96:          prevThreadsInLine  $\leftarrow$  0
97:          newLine  $\leftarrow$  true
98:          lineOneStart  $\leftarrow$  lineTwoStart
99:      end if
100:
101:       $\triangleright$  Letzter aktiver Thread hat seine Gitterzelle gefunden, speichere die Koordinaten für die nächste Iteration
102:
103:      notSearchingBallot  $\leftarrow$  BALLOTTHREADNV( $\neg$ searching)
104:      if threadId = FINDMSB(notSearchingBallot) then
105:          lastPosition  $\leftarrow$  currentPos + horizontalStep
106:      end if
107:  end while
108:
109:  DRAW(currentPos)
110:
111:  newGroup  $\leftarrow$  true
112:  lineOneStart  $\leftarrow$  lastPosition
113:   $\triangleright$  (lastPosition ist eine, zwischen den Threads geteilte, Variable)
114: end while
115: end function

```

---

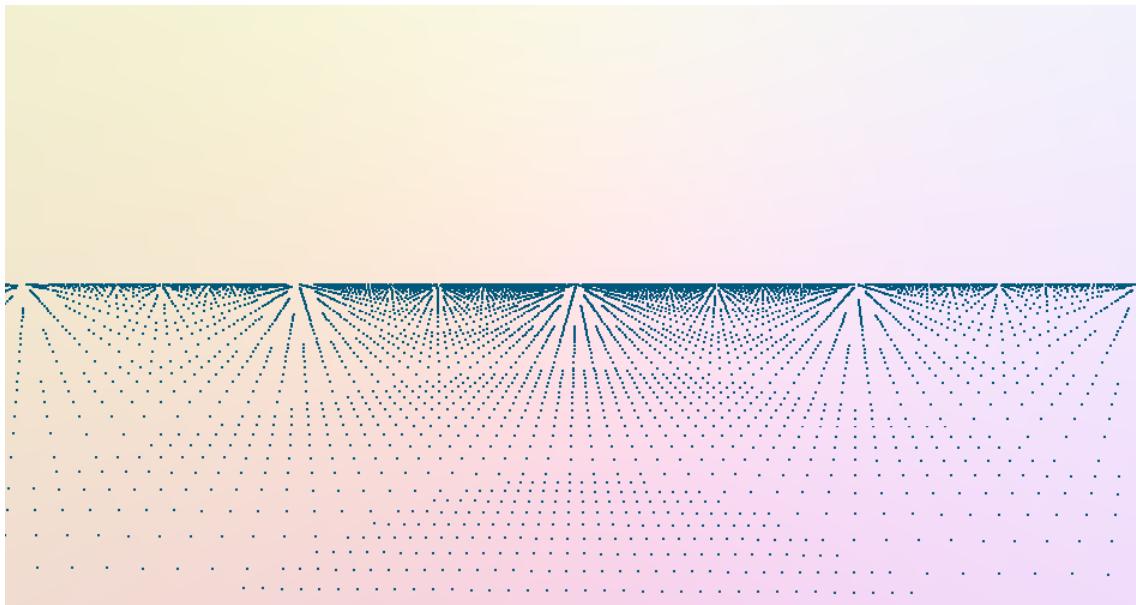


Abbildung 4.8: Ergebnis der Implementierung: Zeichnung jeder Zelle als Punkt mit Korrektur zu kleiner Größen durch Weglassen

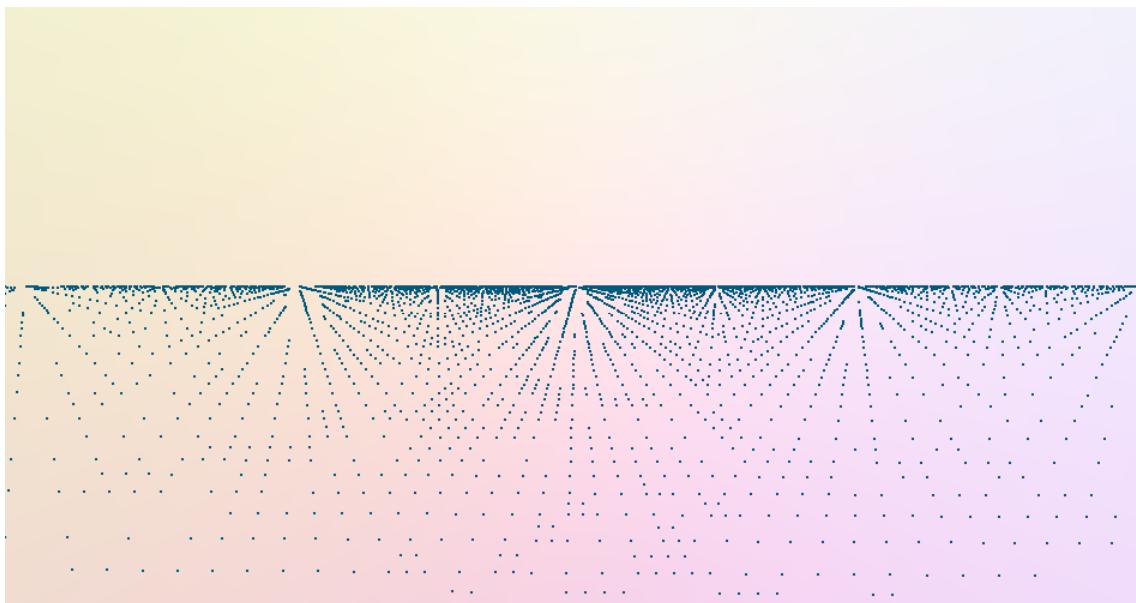


Abbildung 4.9: Ergebnis der Implementierung: Zeichnung jeder Zelle als Punkt mit weichem Übergang durch zufälliges Ausmaskieren

## 4.3 Erzeugen und Zeichnen der Halme

Nachdem erklärt wurde wie jeder Thread eine Gitterzelle finden kann, wird im Folgenden beschrieben wie das Erzeugen und Zeichnen jedes einzelnen Halmes umgesetzt wurde.

### 4.3.1 Bestimmung der Eigenschaften

Für die Eigenschaften der Grashalme wird wie bei der Ezeugung des Geomtriegrases in der Referenzimplementierung (Unterabschnitt 3.1.1) lokal ein Pseudozufallsgenerator verwendet. Für jede, der im vorherigen Abschnitt zugeteilten Zellen variabler Größe, soll nun ein Grashalm gezeichnet werden. Als Seed bietet sich die Position der Zelle auf der xz-Ebene als ganzzahliges Vielfaches der initialen Schrittgröße an. So ist gewährleistet, dass jeder Grashalm mit dieser Position die gleichen Zufallszahlen zieht.

Da die Auflösung des Gitters, in dem die Halme positioniert werden, mit wachsender Entfernung größer wird (wie in Unterunterabschnitt 4.2.3.2 beschrieben), werden weiter weg weniger Halme gezeichnet. Diese sollen aber trotzdem nicht komplett verschieden von den nahen Halmen sein. Anstatt vier soll aus diesem Grund mit wachsender Entfernung nur noch ein Halm gezeichnet werden. Allerdings ist es wünschenswert, dass dies genau einer der vier ist. So wird die Grasfläche nur ausgedünnt aber nicht komplett verändert.

Um dies zu erreichen wird zufällig eine Zelle der nächstkleineren (halbierteren) Größe ausgewählt. Der Zufallsgenerator mit der Position dieser initialisiert und das Ganze so lange wiederholt bis die kleinste Gittergröße erreicht ist. In dieser zufällig gewählten Zelle wird so garantiert auch bei jeder kleineren Gittergröße ein Grashalm gezeichnet. Das Vorgehen ist in Algorithmus 4 illustriert.

---

**Algorithmus 4** Algorithmus zur Bestimmung einer stabilen zufälligen Zelle kleinster Größe zum Zeichnen eines Grashalms

---

```

1:  $rng \leftarrow \text{RNGINIT}(\text{round}(\frac{\text{position}}{\text{initialStepSize}}))$ 
2: while  $\text{initialStepSize} < \text{stepSize}$  do
3:    $\text{halfStep} \leftarrow \frac{\text{stepSize}}{2}$ 
4:    $\text{position.x} \leftarrow \text{position.x} + \text{floor}(\text{RNG.NEXT} * \frac{\text{stepSize}}{\text{halfStep}}) * \text{halfStep}$ 
5:    $\text{position.z} \leftarrow \text{position.z} + \text{floor}(\text{RNG.NEXT} * \frac{\text{stepSize}}{\text{halfStep}}) * \text{halfStep}$ 
6:
7:    $rng \leftarrow \text{RNGINIT}(\text{round}(\frac{\text{position}}{\text{initialStepSize}}))$ 
8: end while
```

---

Es folgt die Bestimmung diverser Eigenschaften des Grashalms. Die Halme werden hier wieder mit quadratischen Bézierkurven dargestellt. Es wird zuerst in der Zelle kleinster Größe, die vorher gefunden wurde, eine zufällige Position gezogen, die auch als erster Kontrollpunkt verwendet wird. Sie wird hier als Wurzel des Halmes bezeichnet. Der zweite Kontrollpunkt liegt ebenfalls an einer zufälligen Position über der Wurzel und wird hier auch einfach als Kontrollpunkt bezeichnet. Der dritte zufällig gewählte Punkt, der den Grashalm beschreibt, ist die Spitze. Alle Punkte sind so gewählt, dass der Grashalm sich an eine minimale und maximale Höhe hält und sich innerhalb seiner Gitterzelle befindet. Weiterhin wird eine Farbe als zufällige Variation einer Basisfarbe und eine Breite innerhalb gesetzter Grenzen gewählt.

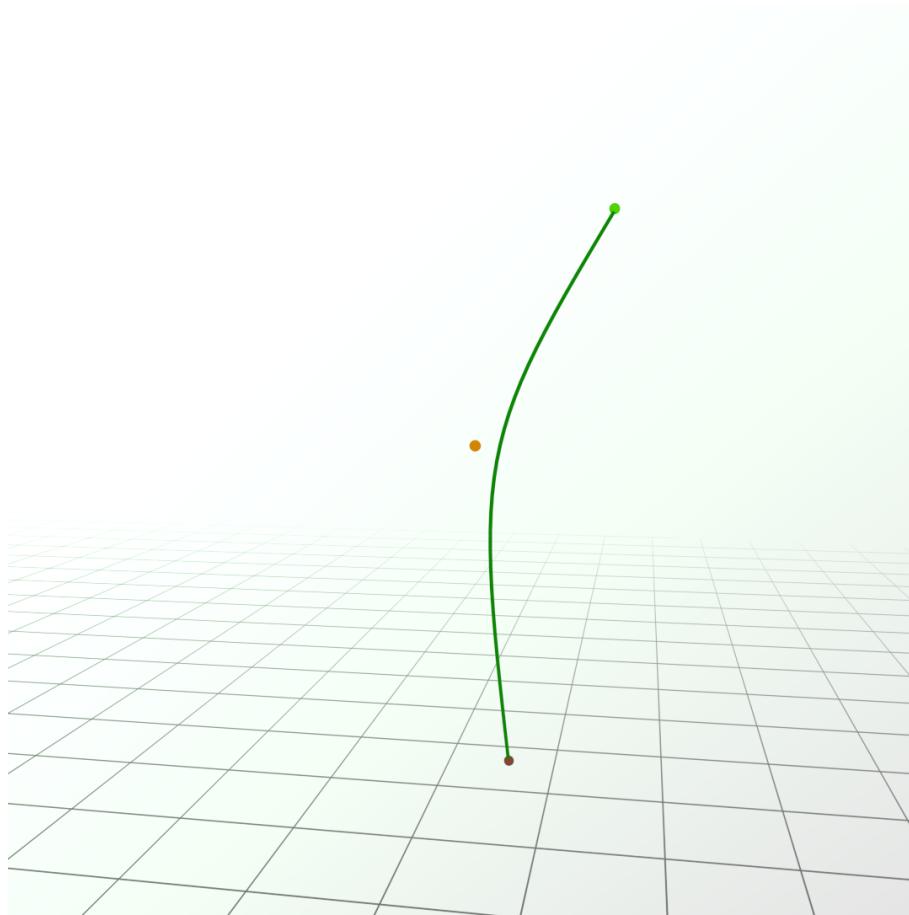


Abbildung 4.10: Kurve eines Grashalmes mit charakterisierenden Punkten.

- Braun: die Wurzel
- Orange: der Kontrollpunkt
- Grün: die Spitze des Grashalmes

### 4.3.2 Scanline-Rasterisierung

Die Bézierskurve, die einen Grashalm darstellt, wird mit Hilfe von Scanline-Rasterisierung gezeichnet. Die Pixel die auf der Kurve liegen werden also Zeile für Zeile berechnet. Dazu werden die Punkte, die den Grashalm charakterisieren, zunächst auf den Bildschirm projiziert, so dass sie in zweidimensionalen Pixelkoordinaten vorliegen. Aus den Koordinaten werden Start- und Endzeile für die Rasterisierung berechnet. Es wird mit der niedrigsten y-Koordinate der projizierten Punkte angefangen und an der höchsten aufgehört.

Für jede Pixelzeile wird nun die Gleichung für den Punkt auf der Bézierskurve, der die gleiche y-Koordinate wie die Zeile hat, gelöst und daraus die x-Koordinate für den Punkt berechnet. Das Finden eines entsprechenden Wertes entspricht somit folgender Lösung der Gleichung (Die Farben entsprechen den in Abbildung 4.10 für die charakterisierenden Punkten verwendeten):

$$\begin{aligned}
 y &= (1-t)^2 * \text{rootProj.y} + 2 * (1-t) * t * \text{cPProj.y} + t^2 * \text{tipProj.y} \\
 &\Leftrightarrow \\
 a &= (2 * \text{cPProj.y} - \text{rootProj.y} - \text{tipProj.y}) \\
 \det &= \text{cPProj.y}^2 - 2 * \text{cPProj.y} * y - \text{rootProj.y} * \text{tipProj.y} + \text{rootProj.y} * \\
 &\quad y + \text{tipProj.y} * y \\
 t_{1,2} &= \frac{\pm\sqrt{\det} + \text{cPProj.y} - \text{rootProj.y}}{a}
 \end{aligned}$$

mit  $\det \geq 0 \wedge a \neq 0$

Eingesetzt in die Gleichung für quadratische Bézierkurven ergibt sich daraus die x-Koordinate:

$$x = (1 - t_{1,2})^2 * \text{rootProj.x} + 2 * (1 - t_{1,2}) * t_{1,2} * \text{cPProj.x} + t_{1,2}^2 * \text{tipProj.x}$$

Das Ergebnis ist für jede betrachtete Pixelzeile eine x-Koordinate. Zusammen mit der y-Koordinate der Zeile ergibt sich eine Folge von Pixelpositionen, die auf der betrachteten Kurve liegen. Für jede der Positionen wird in x-Richtung über die Breite des Halses in Pixeln iteriert und jeder der Pixel gezeichnet. Die Breite in Pixeln ergibt sich aus einer Vorberechnung, bei der die Länge eines Vektors, der in Weltkoordinaten genauso lang ist wie die gewählte Breite, in Bildschirmkoordinaten berechnet wird. Der Vektor soll in Richtung der Breitenausdehnung des Halmes zeigen. Deshalb wird für den Vektor die normalisierte Rotationsachse aus dem nächsten Abschnitt (Abbildung 4.11) mit der Breite des Halmes multipliziert.

Die Breite des Halmes wird je nach Nähe zum betrachteten Punkt zum Ende der Bézierkurve, linear skaliert bis sie nur noch ein Pixel beträgt. So wird der Grashalm nach oben dünner (siehe Abbildung 4.12).

Zu beachten ist hier, dass der Grashalm rein zweidimensional betrachtet wird, da die charakterisierenden Punkte in Bildschirmkoordinaten umgewandelt wurden bevor die Kurve rasterisiert wird.

### 4.3.3 Shading

Jeder Pixel, der gezeichnet werden soll, wird im Folgenden mit einer passenden Farbe versehen. Dazu wird zuerst die Position des betrachteten Pixels im dreidimensionalen Raum ermittelt. Diese lässt sich über das zuvor errechnete  $t$  bestimmen und zwar genau so wie das  $x$  in Unterabschnitt 4.3.2. Es werden allerdings die dreidimensionalen Koordinaten der Wurzel, der Spitze und des Kontrollpunktes genutzt. Diese Position wird nun verwendet, um eine Art Ambient Occlusion umzusetzen, dazu wird die Entfernung des betrachteten Pixels zum Boden berechnet und umso kürzer diese ist, desto dunkler soll der Pixel werden. Die zu Grunde liegende Annahme hierbei ist, dass umso näher am Boden ein Teil eines Grashalms ist, desto weniger indirektes Licht erreicht diesen, da viele andere Grashalme in der Umgebung sind.

Neben Ambient Occlusion wird außerdem noch der Lichteinfall für jeden Pixel berechnet. Dazu wird zunächst eine Normale benötigt. Die Normale ist hier eine um 90 Grad gedrehte und normalisierte Tangente an der dreidimensionalen Position des Pixels. Um die Tangente zu berechnen wird das zuvor berechnete  $t$  in die erste Ableitung der Formel für quadratische Bézierkurven eingesetzt:

$$\text{tangent} = 2 * (-2 * \text{cP} * t + \text{cP} + \text{root} * (t - 1) + t * \text{tip});$$

Die Achse, um die die Tangente gedreht wird, ist bestimmt durch das Kreuzprodukt der Verbindungsvektoren der charakterisierenden Punkten. So zeigt die Normale in die Richtung in die die Kurve sich krümmt. Dies ist in Abbildung 4.11 illustriert.

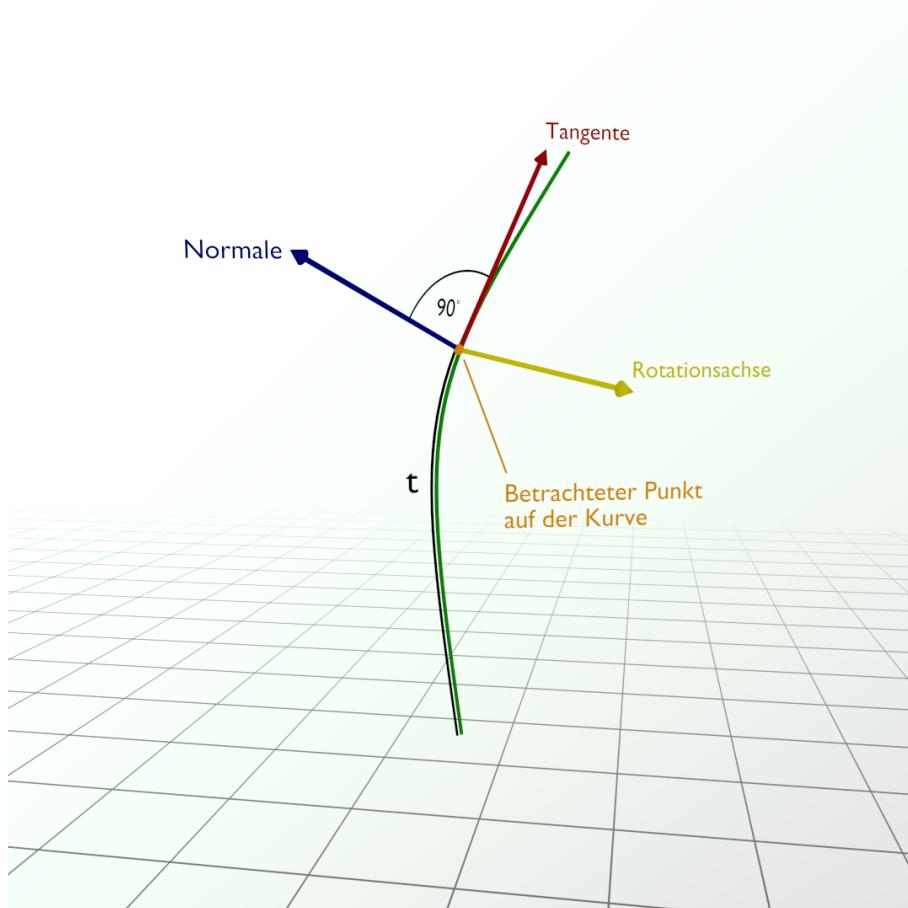


Abbildung 4.11: Kurve eines Grashalms mit Normale an einem betrachteten Punkt sowie die Tangente und die Rotationsachse zur Erzeugung der Normalen

Es ist durch die Normale also definiert wo oben ist, da die Normale in Richtung der Krümmung zeigt wird hier angenommen, dass ein flacher Grashalm sich nur in die Richtung des geringsten Widerstandes biegt.

Weiterhin wird Shading beidseitig betrieben. Die Normale wird falls sie von der Kamera weg zeigen sollte gespiegelt, somit wird Unterseite und Oberseite gleich verarbeitet.

Für die Beleuchtung des betrachteten Pixels wird nun zuerst ein diffuser Anteil berechnet. Dieser ergibt sich aus dem negativen Punktprodukt der Normalen und der normalisierten Lichtrichtung:

$$\text{lambertian} = \text{normal} \cdot \text{lightDirection}$$

Da Gras normalerweise ziemlich dünn ist und somit auch Licht durchlässt, wird auch eine Art Transluzenz für von hinten einfallendes Licht implementiert. Dazu wird, falls der berechnete diffuse Anteil negativ ist, trotzdem der absolute Wert des Anteils dividiert durch einen Faktor benutzt. Der Faktor entspricht dem, durch das Gras hindurch sichtbaren Anteil des von der anderen Seite einfallenden Lichts.

Des Weiteren werden Glanzlichter entsprechend des Blinn–Phong Beleuchtungsmodells

([Bli77]) berechnet. Diese werden nur berechnet, falls das Licht aus Richtung der Normalen kommt, also der diffuse Anteil größer null ist und somit der Pixel direkt beleuchtet wird.

Ein leichter ambienter Lichtanteil ist umgesetzt, indem ein diffuses schwaches bläuliches Licht direkt von oben zur Beleuchtungsberechnung hinzugezogen wird. Dieses soll die natürliche diffuse Beleuchtung durch den Himmel simulieren.

#### 4.3.4 Anti-Aliasing

Um unschöne Kanten zu vermeiden, wurde einfaches Anti-Aliasing implementiert. Dazu wird in der Scanline-Rasterisierung (Unterabschnitt 4.3.2) schon mit Gleitkomma-Pixelkoordinaten gearbeitet. Je nachdem wie nahe die x-Koordinate des betrachteten Punktes an der Pixelmitte ist, wird der Pixel und der in positiver oder negativer Richtung daneben liegende eingefärbt. Die Entfernung des Pixels von der gewollten Koordinate bestimmt den Alpha Wert des Pixels.



Abbildung 4.12: Screenshot der Implementierung: Ein einzelner gezeichneter Grashalm mit Shading, Anti-Aliasing und Breite

Das Ganze wird nur für den ersten und letzten Pixel der Iteration über die Breite des Halmes gemacht, die dazwischen liegenden Pixel müssen auf dem Grashalm liegen und erhalten somit den Alpha Wert eins.

#### 4.3.5 Nutzung der Front-to-Back Eigenschaft für Überdeckung

Es wurde bereits erklärt, dass die Grashalme in jedem Teilfrustum von vorne nach hinten abgearbeitet werden. Diese Eigenschaft kann nun ausgenutzt werden.

Im Folgenden wird angenommen, dass ein Teilfrustum in der Breite ungefähr so viele Grashalme enthält wie es Threads in der Arbeitsgruppe gibt. So sind die parallel gezeichneten Grashalme nie hintereinander sondern nur nebeneinander.

Es ist somit gewährleistet, dass ein Grashalm, der hinter einem anderen liegt, auch danach gezeichnet wird. Für Überdeckung wird somit kein Tiefenpuffer oder Ähnliches benötigt,

alle Pixel die schon gezeichnet sind gehören zu einem Grashalm der weiter vorne liegt als der der gerade zeichnet. Bei der Rasterisierung eines Grashalms (die von oben nach unten erfolgt) kann also, sobald auf ganzer Breite des Halmes auf Pixel getroffen wird, die einen Alpha Wert von eins haben, mit dem Zeichnen aufgehört werden. Bei dichtem Gras sollte dieser Abbruch zu einem effizienteren Ergebnis führen.

Es wird hierbei vernachlässigt, dass vielleicht ein Halm einen anderen nur teilweise überdeckt und somit eigentlich noch der Rest des Halmes gezeichnet werden müsste, da angenommen wird, dass dies auf große Entfernung und mit hoher Grasdichte nicht auffällt.

## 4.4 Kopieren in den Framebuffer

Um das Ergebnis aus dem Compute Shader weiter zu verarbeiten, zeichnet dieser auf eine Textur in der Größe des Viewports. Zuerst zeichnet jede Arbeitsgruppe jedoch in ein zwischen den Threads der Gruppe geteiltes zweidimensionales Array im Shared Memory, das dem Bildschirmausschnitt der Arbeitsgruppe entspricht. Dieses Array wird am Ende der Berechnungen von allen Threads parallel in die Textur kopiert. So werden alle 32 Speicheroperationen als eine einzige ausgeführt. Damit sollen zu viele verstreute Zugriffe in die Ergebnistextur vermieden werden und durch schnelle Zugriffe in den Shared Memory ersetzt werden. Da die Tiles bei einer guten Grasabdeckung zum größten Teil mit Graspixeln gefüllt sind werden auch nicht mehr Speicheroperationen als eigentlich benötigt durchgeführt sondern eher weniger.

Die Textur, die der Compute Shader produziert, wird zum Schluss auf einem transparenten Viereck vor der Kamera gerendert.

## 4.5 Windsimulation

Im Zuge dieser Arbeit wurde eine einfache Windanimation umgesetzt. Dazu werden die charakterisierenden Punkte entlang der Windrichtung gemäß trigonometrischer Funktionen verschoben. Diese erhalten als Eingabe einen Zeitstempel, die Windrichtung und Windstärke.

Wie stark die Verschiebung und wie lang die Periode ist, bestimmt die Windstärke. Diverse Faktoren werden hier leicht durch Zufallszahlen beeinflusst, um ein wenig natürliches Chaos zu erzeugen. Dieser oder ähnliche Ansätze sind einfach und weit verbreitet (siehe auch den Abschnitt über Windsimulation in Stand der Forschung Unterunterabschnitt 2.2.2.1). Sie eignen sich gut für solcherart Animationen, da sie Wind gut genug modellieren und gleichzeitig einfach zu berechnen und nur von der Zeit abhängig sind (Vgl. [Pel04]).

Zuerst wird auf die Windgeschwindigkeit eine skalierte Sinusfunktion addiert, diese soll bewirken, dass die Windgeschwindigkeit leicht schwankt:

```
windSpeed = initialWindSpeed + 0,01 * random * sin (timeStamp)
```

Danach wird mit einer verschobenen Sinusfunktion berechnet, wie weit verschoben werden soll:

```
translationFactor = windSpeed * (sin (timeStamp * windSpeed + random) + 0,8)
```

Dabei wird die Windgeschwindigkeit zum Einen genutzt um die Amplitude, zum Anderen um die Periode zu steuern. Die Windgeschwindigkeit steuert somit wie weit sich der Halm biegen soll und wie schnell er sich hin und her biegt. Die Annahme hierbei ist, dass bei größerer Windgeschwindigkeit der Halm sich allgemein auch schneller bewegt.

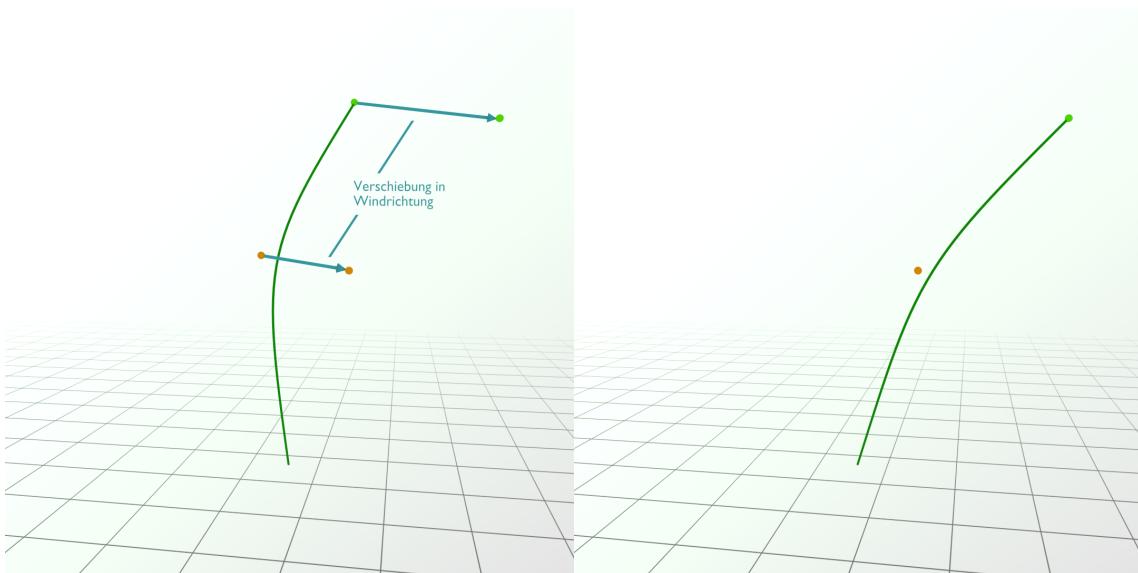


Abbildung 4.13: Beeinflussung eines Grashalms durch Wind. Links: Verschiebung der charakterisierenden Punkte in Windrichtung, Rechts: Verschobene Kurve

Der Sinus ist nur um 0,8 nach oben verschoben, damit der Halm auch leicht über seine ursprüngliche Position hinaus zurück schwankt. Das Ganze soll einen Federeffekt nachahmen. Als „random“ sind hier verschiedene Zufallszahlen benannt, die entsprechend klein sind um den Gesamteindruck zu erhalten (Halme schwanken also insgesamt ungefähr synchron).

Zu guter Letzt wird die Verschiebung auf die charakterisierenden Punkte angewandt (die Wurzel wird dabei nicht verschoben).

```
tip = tip + translationFactor * windDirection
cP = cP + 0,8 * translationFactor * windDirection
```

Die Windrichtung ist in diesem Fall auch leicht zufällig verändert, der Kontrollpunkt wird weniger beeinflusst, um zu simulieren, dass ein Grashalm weiter unten starrer ist als oben.



## 5. Ergebnisse

In diesem Kapitel werden wichtige Ergebnisse dieser Arbeit vorgestellt und diskutiert.

### 5.1 Visuelles

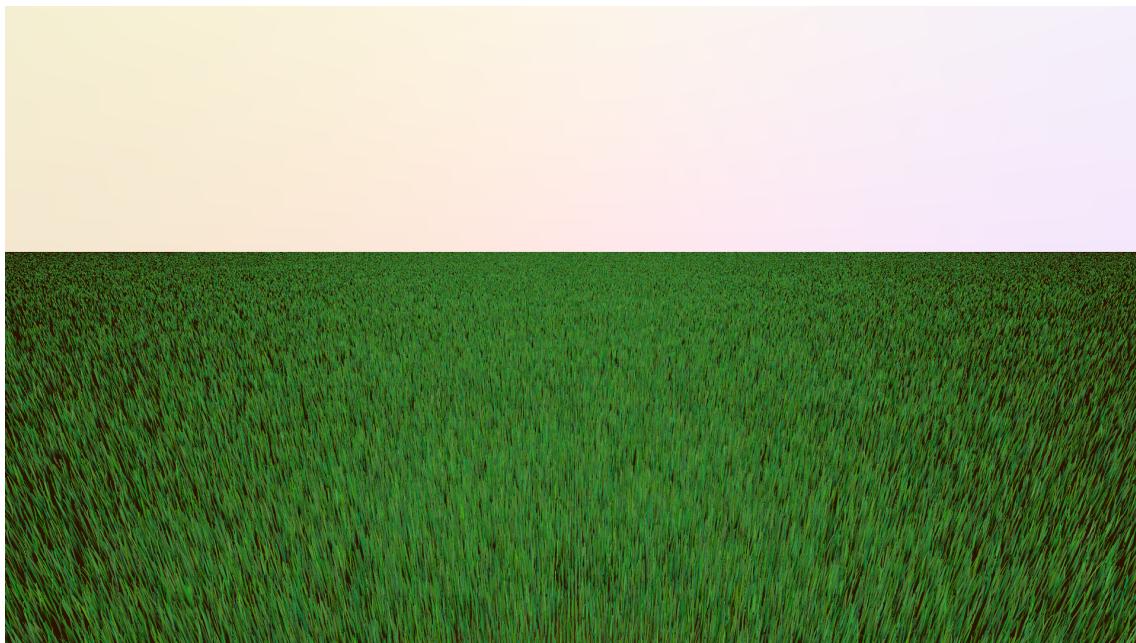


Abbildung 5.1: Screenshot der Implementierung: ~500.000 Grashalme bei einer Auflösung von 1920x1080

Screenshots der Implementierung des Compute Shader Ansatzes zum Zeichnen von Gras, sind in Abbildung 5.1 und Abbildung 5.2 zu sehen. Der Screenshot mit der kleineren Auflösung nutzt die selben Einstellungen wie der andere, es werden jedoch weniger Grashalme gezeichnet da die Größe der Gitterzellen in Pixeln gegeben ist.

Unter <https://vimeo.com/189309577> findet sich ein Video, das den kompletten Zeichenprozess kurz zusammenfasst und die Windanimation zeigt.

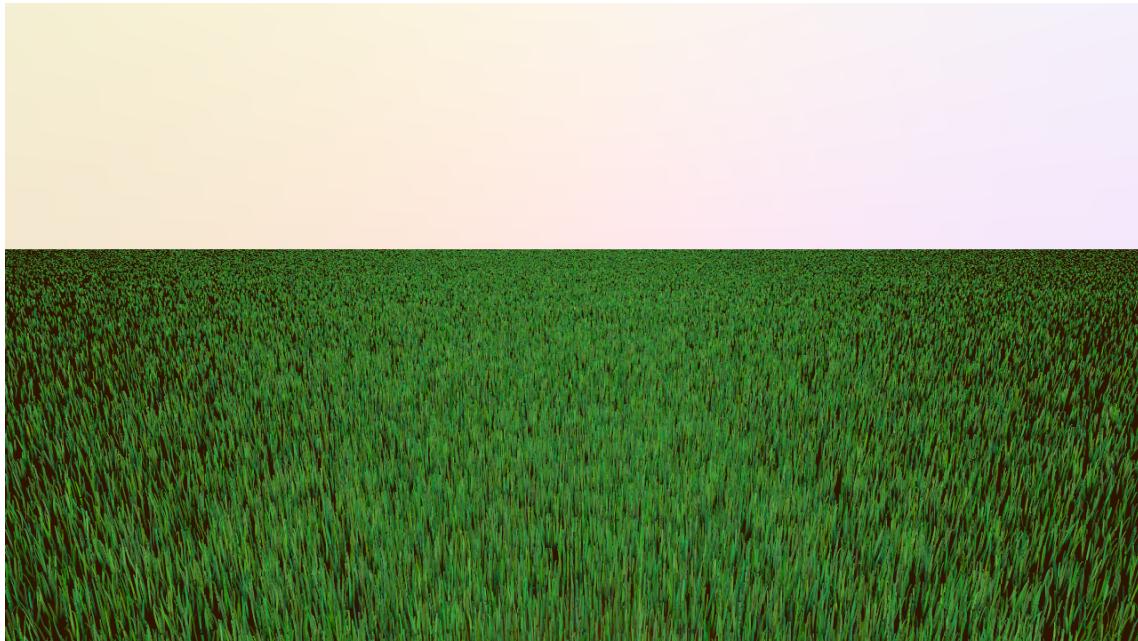


Abbildung 5.2: Screenshot der Implementierung: ~250.000 Grashalme bei einer Auflösung von 1280x720

## 5.2 Performance

In diesem Abschnitt soll untersucht werden wie effizient die Implementierung und wie groß der Leistungseinfluss verschiedener Teile des Programms ist. Für die Tests wurde eine Geforce GTX 980TI genutzt näheres zur Umgebung findet sich in Abschnitt 8.1.

### 5.2.1 Aufgliederung der Zeit

Der erste durchgeführte Test soll als eine Referenz dienen. Es wird bei Entfernung drei angefangen Gras zu zeichnen und bei Entfernung 30 aufgehört (Zur Referenz: Die far plane der Kamera liegt bei einer Entfernung von 1000 und ein Grashalm ist maximal 0,16 hoch).

Die Zellengröße ist auf 2,5 Pixel festgesetzt. Es ergeben sich ungefähr 200.000 gezeichnete Halme bei einer Auflösung von 1280x720. Das Programm läuft mit ungefähr 60FPS. Der Betrachtungswinkel wurde gewählt um eine Standardsituation zu repräsentieren, es handelt sich um die gleiche Kameraposition wie in Abbildung 5.2 und Abbildung 5.1. Aus diesem Winkel sind gut einzelne Grashalme bei der Entfernung drei zu erkennen, während bei einer Entfernung von 30 kaum noch einzelne Halme auffallen. Es ergibt sich ein sehr dichtes Grasbild.

Das Diagramm in Abbildung 5.3 gliedert die Zeit, die benötigt wird, um das Gras zu zeichnen. Der Großteil der 12,3ms wird also aufgewendet um die Zellen auf die einzelnen Threads zu verteilen. Ebenfalls auffällig ist wie viel Zeit gebraucht wird um den Übergang zwischen den Gitterstufen weicher zu gestalten. Dieses Ausblenden (erklärt in Unterabschnitt 4.2.3.3) braucht doppelt so viel Zeit wie das Shading und annähernd so viel wie die Rasterisierung der Halme. An der Verteilung der Gitterpositionen und dem weichen Übergang könnte also gut angesetzt werden, um die Geschwindigkeit zu verbessern. Die Rasterisierung ist im Vergleich schnell.

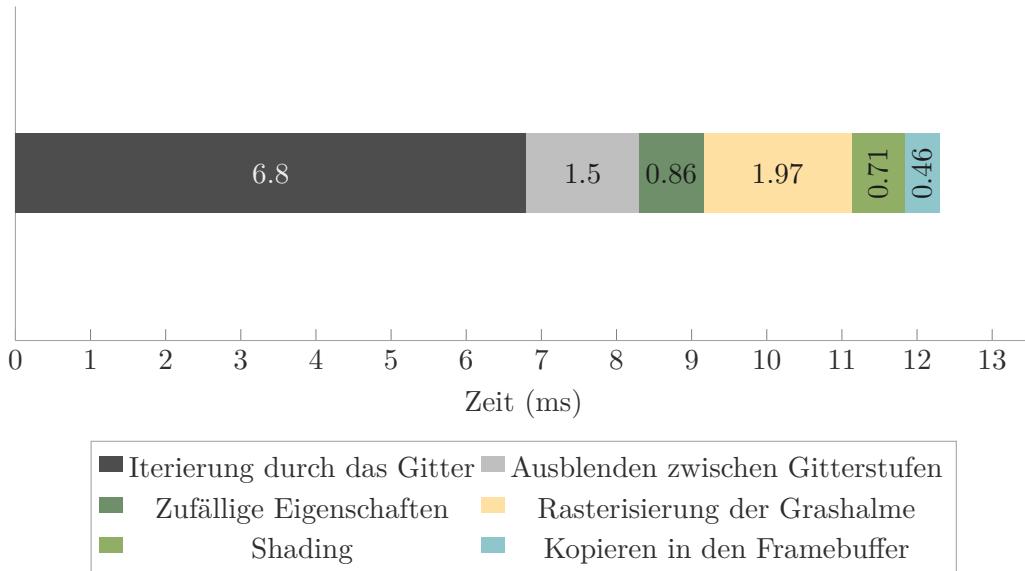


Abbildung 5.3: Aufgliederung der benötigten Zeit zum Zeichnen von  $\sim 200.000$  Grashalmen von Entfernung drei bis Entfernung 30 bei einer Auflösung von 1280x720

### 5.2.2 Auswirkungen verschiedener Entfernungen

Obwohl es fast überflüssig erscheint auf größere Entfernungen immer noch jeden Halm einzeln zu zeichnen, schlägt sich die Implementierung bei dieser Aufgabe gut. Das Diagramm in Abbildung 5.4 zeigt die Zeiten zum Rendern mit verschiedenen Entfernungen, an denen aufgehört wird das Gras zu zeichnen.

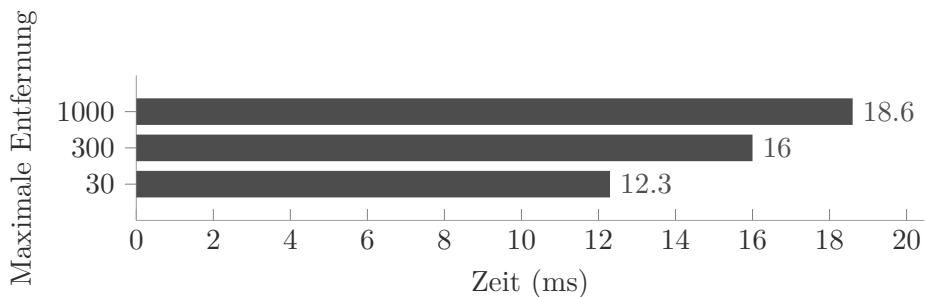


Abbildung 5.4: Zeiten zum Zeichnen von Halmen bei verschiedenen maximalen Entfernungen, minimale Entfernung ist hier drei

Obwohl beim Sprung von 30 auf 300 und von 300 auf 1000 jeweils eine sehr große Fläche hinzukommt, auf der Gras gezeichnet werden soll, steigt die Zeit um das Gras tatsächlich zu zeichnen nicht besonders an. Das liegt zum Einen daran, dass die Zellengröße immer so gewählt wird, dass sie auf dem Bildschirm gleich bleibt. Eine Fläche auf dem Bildschirm enthält also immer ungefähr gleich viele Grashalme. In Weltkoordinaten wird das Areal viel größer, in Bildschirmkoordinaten jedoch nicht. Dies spiegelt sich auch in der Anzahl der Grashalme wieder. Bei der maximalen Entfernung von 300 werden 350.000 Halme gezeichnet während bei 1000 400.000 gezeichnet werden, also nur doppelt so viele wie bei einer maximalen Entfernung von 30. Zum Anderen kommt hinzu, dass bei diesen großen Entfernungen nur noch wenige Pixel für jeden Halm gezeichnet werden, während auf nahe Entfernungen die Scanline-Rasterisierung von jedem einzelnen Grashalm schneller wird.

Die Variation der minimalen Entfernung ergibt ein ähnliches Ergebnis. Kleine Schritte der minimalen Entfernung verursachen gleich viel mehr Zeitaufwand (siehe auch Abbildung 5.5).

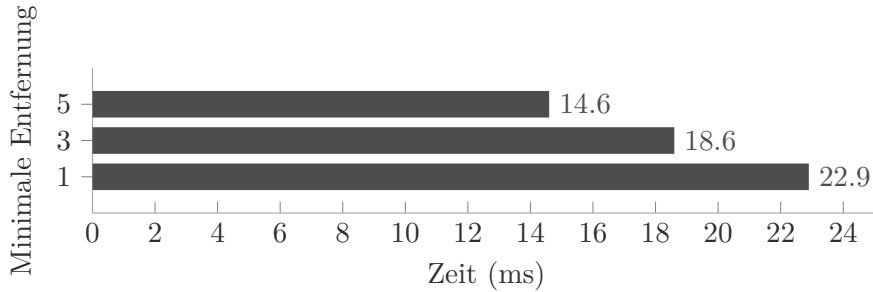


Abbildung 5.5: Zeiten zum Zeichnen von Halmen bei verschiedenen minimalen Entfernungen, maximale Entfernung ist hier 1000

Die Methode funktioniert also auf größere Entfernungen besser. Dies entspricht dem gewünschten Einsatzgebiet.

### 5.2.3 Weitere Testfälle

In der folgenden Tabelle sind noch einige weitere Testfälle mit verschiedenen Grasdichten, Auflösungen und Winkeln aufgeführt. Das Gras startet in allen Fällen bei Entfernung drei und hört auf zu zeichnen bei einer Entfernung von 1000. Die Kamera blickt hier immer in Richtung des Ursprungs und variiert in der Höhe.

Höhe der Kamera	Auflösung	Gittergröße (px)	Grashalme	Renderzeit (ms)
1,4	1280x720	2,5	410.207	19,00
1,4	1920x1080	2,5	902.585	43,33
1,4	1920x1080	4,18	422.376	22,16
0,7	1280x720	2,5	416.200	18,87
0,7	1920x1080	2,5	887.940	43,21
0,7	1920x1080	4,18	414.932	21,74
0,1	1280x720	2,5	174.205	9,08
0,1	1920x1080	2,5	351.068	19,8
0,1	1920x1080	6,8	351.068	11,1

Zum Vergleich der Auflösungen wurde hier, bei der jeweils höheren Auflösung, noch ein Test mit größerer Gittergröße hinzugefügt. Die Gittergröße ist so gewählt, dass die Anzahl der gezeichneten Grashalme gleich ist wie bei dem Testfall mit kleinerer Auflösung. So lässt sich der zusätzliche Aufwand durch höhere Auflösungen besser abschätzen. Wie in der Tabelle erkennbar ist, skaliert die Implementierung gut mit der Auflösung.

Weiterhin ist noch erwähnenswert, dass die Implementierung eine gute GPU-Auslastung erzielt. Bei einer Auflösung von 1280x720 sind dies ungefähr 80% und bei 1920x1080 stark 90%. Diese Auslastung reduziert sich erst bemerkbar wenn die Dichte der Halme sehr reduziert wird, also zu einem Grad bei dem nicht mehr von einer Grasfläche gesprochen werden kann.

### 5.2.4 Einsparungen durch Rasterisierungsabbruch und Shared Memory

Um die Performance zu verbessern, wird in der Implementierung unter Ausnutzung der Front-to-Back Eigenschaft die Rasterisierung eines Grashalms abgebrochen falls erkannt wird, dass an der betrachteten Stelle bereits ein Grashalm gezeichnet ist. Tatsächlich stellte sich in den Tests jedoch heraus, dass die Beschleunigung durch den Rasterisierungsabbruch äußerst gering ausfällt. Der maximale Zeitgewinn bei Testfällen mit normal lang und breitem Gras wie im Referenztestfall betrug eine Millisekunde und das bei einem extrem flachen Winkel.

Bei sehr langem und breitem Gras ist auch bei normalen Winkeln eine größere Beschleunigung zu verbuchen, in den meisten normalen Fällen bleibt eine signifikante Verbesserung der Performance jedoch aus.

Der Grund dafür ist wahrscheinlich, dass die Halme, wie schon erwähnt, mit wachsender Entfernung schnell nur noch aus wenigen Pixeln bestehen für die ein Rasterisierungsabbruch nicht mehr viel Zeit einspart. Des Weiteren müssen alle 32 Threads einer Arbeitsgruppe abbrechen, um einen Performancegewinn zu erzielen. Was allerdings normalerweise der Fall sein sollte, da die Threads sich auf ungefähr gleicher Höhe befinden.

Ein Pixel Array für jede Arbeitsgruppe im Shared Memory zu nutzen und dann später in den Framebuffer zu kopieren, erzielte in den Tests keinen erkennbaren Performancegewinn.

### 5.2.5 Vergleich von Compute Shader Gras mit Geometriegras

Ein Vergleich mit gängigen bildbasierten Methoden ist uninteressant, die einfache Geometrie dieser Methoden ermöglicht eine extrem schnelle Verarbeitung. Das Vorgehen ist nahezu konkurrenzlos, jedoch auch weniger flexibel. Ist eine Beeinflussung jedes einzelnen Halmes gewünscht, wird Geometrie für diese benötigt, was zu einem geometriebasierten Ansatz führt.

Zum Vergleich wurde die Implementierung der Referenzarbeit aus Kapitel 3 herangezogen. Diese braucht für das Rendern einer komplett im Gras gefüllten Fläche mit gleichem Kamerawinkel, Grashöhe und Grasbreite in 1280x720 ungefähr 25ms, also etwas mehr als die Compute Shader Implementierung. Allerdings ist dieses Ergebnis nicht ganz repräsentativ da bei der Referenzimplementierung im Testfall 11.000.000 Grashalme verarbeitet werden, also um Größenordnungen mehr als im Compute Shader. Das spricht eher für die Güte der Methode zur Reduktion der Dichte mit der Entfernung. Die Verdoppelung der Gitterabstände, um diese auf dem Bildschirm gleich zu halten, die in der Arbeit mit dem Compute Shader Anwendung findet, ist offensichtlich sehr wirksam. Außerdem funktioniert die Geometriemethode auf große Entfernungen nicht optimal, da die Grashalme dort dünner als ein Pixel werden. Die Renderpipeline wird diese Halme auch entsprechend dünn oder auch gar nicht zeichnen. Bei der Compute Shader Methode sind alle Halme immer mindestens ein Pixel breit so ergibt sich auch auf große Entfernungen trotzdem eine gute Überdeckung.

Beide Methoden erzeugen lokal Gras mit Hilfe des gleichen Pseudozufallsgenerators, der Speicherverbrauch der Methoden ist damit ähnlich vernachlässigbar.



## 6. Fazit

Mit den gesammelten Ergebnissen wird klar, dass die Anzahl Grashalme, die ein geometriebasierter Ansatz verarbeiten kann, wesentlich höher ist als die, die der Compute Shader Ansatz in der gleichen Zeit verarbeitet. Den Vorteil, den der Compute Shader Ansatz hat, kann, mit dem gesammelten Wissen, auch auf den Geometrie Ansatz übertragen werden. Die zwei wichtigsten Punkte sind dabei: das Gitter mit variabler Schrittgröße zu übertragen und dafür zu sorgen, dass Gras auf Entfernung nicht zu dünn wird.

Die Messungen zeigen, dass die Implementierung trotzdem schnell genug ist, um durchaus anwendbar zu sein. Weiterhin gibt der Ansatz eine gute Approximation für weit entferntes Gras ab und kann ebenfalls noch weiter optimiert werden (siehe auch Unterabschnitt 7.1.1). Da jedoch der, durch die Front-to-Back Eigenschaft ermöglichte, Rasterisierungsabbruch keine besondere Beschleunigung erbracht hat, ist der entwickelte Ansatz zum größten Teil eine Rasterisierungslösung für Grashalme mit aufwändiger Positionsfindung implementiert im Compute Shader. Der Vorteil der Methode ist die direkte Kontrolle über den Zeichenprozess. Es ist vorstellbar hier weiterhin anzusetzen, um einen verbesserten Rasterisierungsabbruch oder andere Optimierungen zu implementieren.



# **7. Ausblick**

Der Fokus dieser Arbeit lag hauptsächlich darin den Bildschirm aufzuteilen und mit einem Compute Shader massiv parallel Gras zu zeichnen. Diverse Themen wurden nur oberflächlich, wie Shading und Animation, oder gar nicht behandelt. Deshalb soll im Folgenden diskutiert werden, welche Möglichkeiten es zur Verbesserung und Weiterentwicklung gibt.

## **7.1 Möglichkeiten zur Weiterentwicklung**

Die implementierte Methode erhebt keinen Anspruch auf Perfektion oder Anwendbarkeit in jeder Möglichen Situation. Es sollen hier einige wichtige Punkte zur Weiterentwicklung der Methode besprochen werden.

### **7.1.1 Performanceverbesserung**

In Unterabschnitt 5.2.4 wurde festgestellt, dass der Rasterisierungsabbruch die Geschwindigkeit nicht besonders verbessert. Es könnte also damit experimentiert werden die Front-to-Back Eigenschaft zu vernachlässigen und tatsächliche Tiefentests durchzuführen, um Verdeckung zu implementieren. Die Tiefe kann einfach als zusätzliches Feld im Pixel Array eingespeichert werden. Die Iterierung durch die Zellen kann also vereinfacht werden, indem diese nicht streng von vorne nach hinten abgearbeitet werden müssen. Die Zellenfindung wird so vergleichbar zu einer parallelen Rasterisierung des Schnitts des Teilfrustums der Arbeitsgruppe. Der schwierigste Teil dabei wäre der dynamische Wechsel der Zellengröße.

Weiterhin benutzt die Implementierung zur Zeit 32x32 Pixel große Bildschirmbereiche für die Arbeitsgruppen. Dieses Vorgehen kann optimiert werden, indem je nach Hardware eine ideale Anzahl an Arbeitsgruppen gefunden und diese benutzt wird um den Bildschirm zu unterteilen.

Es sind definitiv noch weitere Optimierungen möglich. So ist der Iterationsalgorithmus der die Positionen für die Grashalme findet ein guter Ansatzpunkt, da er am meisten Zeit benötigt.

### 7.1.2 Flexibilität

In dieser Arbeit wurde angenommen, dass das Gras auf dem Bildschirm ungefähr vertikal wächst und, dass es nicht zu sehr gebogen ist. Diese Annahmen sind in den meisten Fällen akzeptabel, deshalb wurden verschiedene Teile des Programmes mit diesen vereinfacht und können weiterentwickelt werden, um flexibler zu sein.

Eine wichtige Vereinfachung ist die Scanline-Rasterisierung der Halme. Diese geschieht auf dem Bildschirm von oben nach unten und bricht entsprechend ab, wenn sie auf einen anderen Grashalm trifft. Diese Optimierung funktioniert nicht mehr falls Gras an der Decke wachsen sollte oder die Kamera auf dem Kopf steht. Als Lösung könnte die Rasterisierung dynamisch gedreht werden je nach dem wo oben beziehungsweise unten ist. Allerdings schlägt die Scanline-Rasterisierung immer noch fehl falls es Stücke der Kurve gibt die auf dem Bildschirm horizontal abgebildet werden, also die Kurve über die Länge von mehreren Pixeln in einer Pixelzeile liegt. Die Scanline-Rasterisierung würde nur zwei Schnittpunkte finden, Pixel an diesen zeichnen und den Bereich dazwischen leer lassen. Als Lösung bietet es sich an, die Scanline-Rasterisierung in x- und y-Richtung zu betreiben, was allerdings natürlich auch doppelt so viel Aufwand ist und das Abbruchkriterium ist schwieriger umzusetzen. Genauso könnte man auch solche Fälle erkennen und entsprechend Linien zwischen solchen Pixeln zeichnen.

Möglich ist es auch einen komplett anderen Rasterisierungsansatz zu verwenden. So kann zum Beispiel der Algorithmus zum Rasterisieren von quadratischen Bézierkurven, den Zingl in *A rasterizing algorithm for drawing curves* [Zin12] beschreibt, entsprechend modifiziert werden, dass er Grashalme von oben nach unten zeichnet, die Abbruchbedingung sowie Dicke unterstützt und übergeben kann, wie weit ein Pixel entlang der Kurve ist.

### 7.1.3 Weiches Aus- und Einblenden einzelner Halme

In Unterunterabschnitt 4.2.3.3 wird beschrieben, wie zufällig vier Zellen einer Gitterstufe zu einer zusammengefasst werden, um einen weichen Übergang zwischen den Gitterstufen zu erzeugen. Allerdings werden bei dem gezeigten Vorgehen immer drei Grashalme auf einmal verschwinden. Obwohl es in den Tests nicht auffällig war, ist es vielleicht für bestimmte Grasdichten und Anwendungen wünschenswert, dieses Verschwinden von Grashalmen beim Wechsel von Gittergrößen weicher zu gestalten.

Normalerweise wird in einem solchen Fall ein Grashalm weich ausgeblendet, indem ein Alpha Wert für diesen berechnet wird, der kleiner wird umso näher der Halm an der Grenze zum Ausblenden ist. Andere Ansätze basieren darauf anstatt die Transparenz die Länge zu variieren, so dass der neue Grashalm beim Ausblenden schrumpft und beim Einblenden wächst.

Beide Ansätze basieren darauf herauszufinden wie nahe ein Grashalm am Ausblenden ist. Das Problem ist, dass zwar einfach berechnet werden kann wie nahe vier Zellen daran sind zu einer zusammengefasst zu werden, jedoch zu diesem Zeitpunkt nicht bekannt ist welcher Halm der vier übrig bleiben wird. Dies wird erst zufällig ausgewählt wenn in der (größeren) Zelle tatsächlich ein Halm gezeichnet werden soll (siehe auch Algorithmus 4). Diese Berechnung müsste also auch für jede Zelle die zusammengefasst werden könnte, ausgeführt werden, um herauszufinden welcher Halm übrigbleibt und damit welche Halme ausgeblendet werden.

### 7.1.4 Einbindung in eine Szene

Momentan wird das Compute Shader Gras in die Szene eingebunden, indem das Programm die vom Shader erzeugte Textur über alles andere zeichnet. Dieses Vorgehen erweist sich natürlich als problematisch sobald andere Objekte in der Szene vor dem Gras gezeichnet sein sollen.

Das Problem lässt sich einfach umgehen, indem man zum Beispiel zuerst alle Objekte hinter dem Gras (zum Beispiel den Boden) zeichnet, dann das Gras und schließlich alle anderen Objekte. Dieser Ansatz mag für einige Anwendungen genügend sein, jedoch gibt es häufig Objekte, die nur teilweise von Gras überdeckt sein sollen. In diesen Fällen gestaltet sich eine Lösung etwas komplizierter. Informationen wo Gras gezeichnet werden soll und wo nicht, müssen auf Basis von anderen, sich in der Szene befindlichen Objekten, übergeben werden. Es eignet sich also, zuerst die Szene zu rendern und Tiefenpufferinformationen dieser an den Compute Shader weiterzugeben. Um herauszufinden ob ein Pixel eines Grashalms gezeichnet werden soll, verwende die Tiefeninformation dieses und vergleiche sie mit der Tiefe des Grashalmpixels. Da die Weltkoordinaten jedes Pixels schon beim Shading bekannt sind, lässt sich auch die Tiefe berechnen.

Schattierung des Grases durch andere Objekte kann man ganz ähnlich, durch Übergeben von Schatteninformationen der Szene an den Compute Shader, verwirklichen. Andere Möglichkeiten umfassen das Berechnen einer Schattentextur für das Terrain auf dem das Gras wachsen soll (siehe auch den Abschnitt zum Thema Texturen Unterabschnitt 7.1.6).

### 7.1.5 Terrain

Im Moment wird Gras nur auf der xz-Ebene gezeichnet, theoretisch ist aber jede Form von Geometrie auf ähnliche Weise implementierbar. So kann das Teilfrustum einer Arbeitsgruppe mit jeder Art von Geometrie geschnitten werden, um den entsprechenden Bereich für das Zeichnen von Gras zu finden. Es kann angenommen werden, dass eine Landschaft, auf der Gras gezeichnet werden soll, relativ flach ist. Es kann also in den meisten Fällen das gleiche reguläre Gitter auf der xz-Ebene für die Positionierung der Grashalme genutzt werden. Die Halme können dann jeweils ihre tatsächliche Position auf dem Terrain durch einen Strahlschnitt herausfinden.

Ein Ansatz der auch für kompliziertere Geometrie funktionieren würde, ist die Gitteraufteilung auf dem Objekt selbst vorzunehmen zum Beispiel indem die Geometrie dynamisch unterteilt und entsprechend von vorne nach hinten abgearbeitet wird.

### 7.1.6 Texturen

Ein häufig umgesetzter Ansatz, Gras extern zu beeinflussen, ist das Verwenden von Texturen zur Beeinflussung diverser Eigenschaften (siehe auch den Abschnitt zum Thema Variation Unterabschnitt 2.2.1 in Stand der Forschung). Dieser Ansatz lässt sich analog zu anderen Arbeiten umsetzen. Beim Zeichnen von Grashalmen sind in der vorgestellten Methode die Weltkoordinaten dieser bekannt, mit diesen kann, mit entsprechender Umrechnung, auf eine Textur zugegriffen werden, die für die Weltkoordinate Eigenschaften für die dort gezeichneten Halme enthält.

Da in dieser Arbeit eine unendliche Grasfläche gezeichnet wird, bietet es sich an, einen Ansatz wie die von Perbet und Cani in „Animating prairies in real-time“ [PC01] genutzten Masken umzusetzen. Dabei würden Texturen nicht die komplette Grasfläche beschreiben, sondern nur einen Teil, welcher bestimmt ist durch die Weltkoordinaten und die Ausdehnung der Textur. Mit Texturen können Eigenschaften wie Grashöhe, Farbe, Dichte und Winkel beeinflusst werden, um nur ein paar zu nennen. Mit animierten Texturen oder sich

bewegenden Masken wie in „Animating prairies in real-time“ [PC01] können auch Effekte wie Wind oder Kollision umgesetzt werden.

Da hier beim Shading die Weltkoordinaten für jeden betrachteten Pixel schon berechnet werden, kann auch einfach eine volumetrische Textur auf die Grashalme angewandt werden.

### 7.1.7 Shading

Das Shading in der vorgestellten Methode bietet Möglichkeiten zur Implementierung beliebiger Beleuchtungsmodelle genau so wie jeder übliche Fragment Shader. Bei der Beleuchtungsberechnung sind Normale sowie Weltkoordinate bekannt, des Weiteren ist die relative Position entlang der Bézierkurve gegeben, die hier für Ambient Occlusion verwendet wird.

In der vorgestellten Implementierung ist ein einfaches Blinn-Phong Beleuchtungsmodell verwendet worden. Dieses kann zum Beispiel einfach um mehrere beziehungsweise verschiedene Lichtquellen erweitert werden.

Die Farbe eines Pixels ergibt sich nur aus der Farbe des Grashalms und der Beleuchtung sowie Ambient Occlusion. Hier könnten Texturen oder Rauschfunktionen Verwendung finden um die Farbe entlang des Grashalms weiter zu variieren.

Falls verschiedene Dichten für Gras implementiert werden, wäre es weiterhin angebracht den Ambient Occlusion Faktor je nach Dichte beziehungsweise der Anzahl der Grashalme in der Umgebung zu bestimmen.

### 7.1.8 Externe Beeinflussung der Grashalme

Die externe Beeinflussung des gezeichneten Grases ist ein, in dieser Arbeit, nicht tiefer behandeltes Thema. Jedoch könnte es, in vielen Einsatzgebieten, durchaus erwünscht sein eine komplexere Beeinflussung des Grases zur Verfügung zu haben. Deshalb werden im Folgenden diverse Ansätze zur Implementierung erweiterter Beeinflussung des Grases erklärt.

#### 7.1.8.1 Windanimation

Genauso wie für die Beleuchtung wurde für die Animation des Grases im Wind nur eine sehr einfache Methode implementiert. Und genauso ist auch hier das Thema schon viel in anderen Arbeiten behandelt und einfach übertragbar.

Die derzeitige Vorgehensweise hat das Problem, dass die Länge der Halme bei der Animation nicht erhalten bleibt. Die Implementierung könnte verbessert werden, indem die Kontrollpunkte der Grashalme nicht nur verschoben, sondern um eine Achse rechtwinklig zum Wind gedreht werden ähnlich wie in „Approximate and probabilistic algorithms for shading and rendering structured particle systems“ [RB85]. So bleibt die Länge ungefähr erhalten und es wird ein besserer Eindruck bei starkem Wind erzeugt, da die Halme tatsächlich niedergedrückt werden können.

#### 7.1.8.2 Kollision

Da mit dieser Arbeit eine gute Approximation für weit entferntes Gras implementiert werden soll, wäre für Kollisionsberechnung eine einfache ungenaue Methode geeignet. So wäre das Umnicken weg von einem Objekt, basierend auf der Entfernung zu diesem, für einen guten visuellen Einruck genügend. Dies kann einfach mit an das Objekt geknüpfte Texturen implementiert werden, die die Knickrichtung an das zu zeichnende Gras weitergeben.

### 7.1.9 Weitere Pflanzenarten

Je nach der Art der Landschaft, die dargestellt werden soll, könnten auch andere Pflanzenarten gezeichnet werden. Prinzipiell lassen sich viele Pflanzen die sich auf einer Wiese finden, wie Blumen und Sträucher, ebenfalls mit Bézierskurven modellieren. Teile wie Blüten lassen sich auf weite Entfernung auch mit ein paar farbigen Pixeln prozedural generieren. Es ist auch vorstellbar diese Informationen aus Bildern zu kopieren.

Wie schon zuvor angesprochen, können auch Texturen verwendet werden, um Eigenschaften von Grashalmen an verschiedenen Positionen zu beeinflussen. Zum Beispiel kann durch Variation von Länge, Farbe, Dicke, Dichte und Kontrollpunkten eine gute Approximation für Pflanzen wie Weizen, Sträucher oder sogar Farn erzeugt werden.

### 7.1.10 Kombination mit anderen Methoden

Da das gezeichnete Gras in dieser Arbeit für mittlere bis größere Entfernung bestimmt ist, sollte die vorgestellte Methode entsprechend mit anderen für verschiedene Entfernung gemischt werden.

Für nahe Gras bietet sich ein Geometriearnsatz an bei welchem für jeden Grashalm ein Objekt, bestehend aus Vertices und Faces, erzeugt und gerendert wird. Um den Eindruck konsistent zu halten, kann für das Geometriegras der gleiche Ansatz mit den gleichen Gitterzellen verwendet werden. Da der Seed für den Pseudozufallsgenerator auf der Position einer solchen Gitterzelle beruht, können deterministisch genau die gleichen Grashalme erzeugt werden. Es empfiehlt sich deshalb die Geometrie ebenfalls in einem Shader zu generieren (wie zum Beispiel in der Referenzarbeit, siehe Kapitel 3), ansonsten müsste unverhältnismäßig viel Speicher aufgebraucht werden, um die Informationen für jeden Grashalm zu speichern.

Der hier vorgestellte Ansatz liefert zwar eine gute Approximation für weit bis sehr weit entferntes Gras, allerdings können auf solche Entfernung auch effizientere Ansätze genutzt werden, die nicht mehr jeden Halm einzeln zeichnen. Für einen menschlichen Betrachter und mit üblichen Bildschirmauflösungen degeneriert das gezeichnete in einer solchen Entfernung zu einer grünen Fläche mit leicht zufälligen Variationen. Selbst Animationen des Grases sind wenig bis gar nicht zu erkennen.

So bietet zum Beispiel die Approximation von Gras durch eine einfache grüne Fläche eine gute und effiziente Alternative. Zum Einfärben der Fläche eignet sich zum Beispiel eine statische Grastextur. Eine gute dynamische Alternative ist das Zeichnen von Screen Space Grass auf Basis von Rauschfunktionen (Vgl. [Pan14]). Der Ansatz ermöglicht prozedurale Animation und Beeinflussung durch Modifikation der Rauschfunktion und Farbe beim Zeichnen. Weiterhin kann er direkt im Compute Shader implementiert werden und ist schnell.



# 8. Technische Details

Dieses Kapitel widmet sich einigen technischen Details. Hier wird kurz die Testumgebung beschrieben und der Quellcode des entwickelten Programms sowie die Implementierung selbst erklärt.

## 8.1 Umgebung

Sämtliche Tests wurden auf einem Windows 10 PC mit einer Intel i5 2500K CPU und einer NVIDIA Geforce GTX 980TI Grafikkarte im Referenzdesign durchgeführt. Die benutzte Grafiktreiber Version ist 375.57.

## 8.2 Applikation

Die vorgestellten Methoden sind in C++ und GLSL implementiert. Es wird mindestens die OpenGL Version 4.5 benötigt. Die Applikationen wurden auf Basis eines Frameworks von Tobias Zirr entwickelt <https://github.com/tszirr/lighter-app><sup>1</sup>. Für den Pseudozufallsgenerator, der in der Haupt- sowie der Referenzarbeit verwendet wurde, kam eine Implementierung von Wang Hash zum Einsatz (siehe auch [Ree13]).

Der Quellcode ist unter <https://github.com/yannick-t/lighter-app-grass> zu finden, dort gibt es zu dem Zweig `master`, der die auf dem Compute Shader basierenden Implementierung beinhaltet, noch den Zweig `referenceGeometryGrass`, der das Programm der Referenzarbeit enthält.

Bei dem auf GitHub zu findenden Code, handelt es sich um ein CMake Projekt. Bei der Implementierung wurde mit Visual Studio 2013 gearbeitet und es wird empfohlen das Projekt bei Interesse auch damit auszuführen.

---

<sup>1</sup>genauer: die Programme basieren auf commit f7cec6d0a04f8d92d007136ca0f1e280eddafc7f dieses Frameworks

## 8.2.1 User Interface

Im Folgenden findet sich eine Beschreibung des User Interface der entwickelten Anwendungen.

### 8.2.1.1 Hauptanwendung

#### Kamerasteuerung:

**W, A, S, D** Bewege Kamera vorwärts, nach links, rückwärts und nach rechts

**Maus** Rotiere Kamera, dazu muss die Maus mit Rechtsklick oder Enter eingefangen werden

**Q** Bewege Kamera nach unten

**Leertaste** Bewege Kamera nach oben

**Shift** Beschleunigt die Bewegung der Kamera

**Strg** Verlangsamt die Bewegung der Kamera

Auf der rechten Seite des Bildschirms befindet sich das Graphical User Interface (Abbildung 8.1), dieses enthält diverse Ausgabegrößen:

- Frame Time in Millisekunden (dt), also die Zeit die ein Bild zum rendern braucht
- Zeit die das Gras zum Zeichnen braucht in Millisekunden
- Durchschnittliche Zeit zum Zeichnen des Grases
- Aus der Frame Time errechnete Frames per Second (FPS)
- Anzahl der gezeichneten Grashalme

Weiterhin befinden sich darunter diverse Schieberegler für:

- Kamerageschwindigkeit
- Minimale und maximale Höhe für die zufällig generierten Grashalme
- Minimale und maximale Breite für die Halme
- Ambient Occlusion Faktor der angibt wie viel von dem Grashalm von dem Ambient Occlusion Effekt beeinflusst wird, also wie lang der Abschnitt des Halmes ist der dunkel eingefärbt wird.
- Entfernung zwischen Grashalmen (also die genutzte Gittergröße) an der minimalen Entfernung zum Zeichnen in Pixeln
- Minimale und maximale Entfernung von der Kamera zwischen denen Gras gezeichnet wird
- Windrichtung diese zeigt standardmäßig in positive x-Richtung und wird um die y-Achse um den angegebenen Winkel (in Grad) gegen den Uhrzeigersinn rotiert.
- Windgeschwindigkeit
- Regler für Debug Optionen:
  - >1 Zeigt Schnittpunkte der (erweiterten) Teilfrusta an
  - >2 Zeigt die Grenzen zwischen den den Bildschirm aufteilenden Tiles, also den Bereichen in denen die Arbeitsgruppen arbeiten

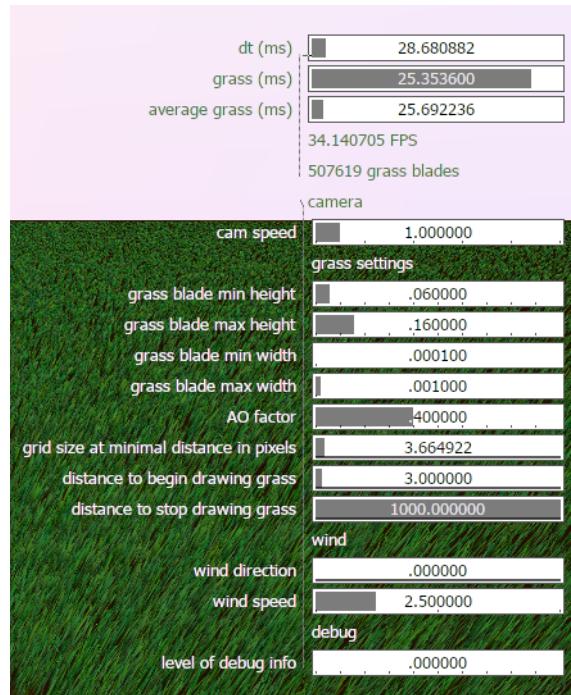


Abbildung 8.1: Screenshot des Graphical User Interface

**Weitere Shortcuts:**

- U** GUI an- beziehungsweise ausschalten
- C** Zählen von Grashalmen ein- und ausschalten
- T** Zwischen Testfällen umschalten (setzt die durchschnittliche Zeit zurück), diese sind:
  - Zeichnen von Gitter mit variierender Gittergröße
  - Zeichnen des Gitters mit weichem Übergang zwischen Gittergrößen
  - Rasterisierung der Grashalme
  - Zufällige Bestimmung von Eigenschaften der rasterisierten Grashalme
  - Shading der Halme
- X** Umschalten des Rendermodus zwischen:
  - Zeichnen zu Pixel Array, dann Kopieren in den Framebuffer
  - Direktes Zeichnen
  - Kein Zeichnen in den Framebuffer
- N** Manuelles Zurücksetzen der durchschnittlichen Renderzeit
- P** Pausiert die Animationszeit<sup>2</sup>
- F** Beschleunigt die Animationszeit

<sup>2</sup>Wird für die Animation des Lichtes und der Grashalme im Wind benutzt

### 8.2.1.2 Referenzanwendung

In der Referenzimplementierung für das Zeichnen von Geometriegras ist das User Interface etwas anders da für Bereiche in denen Grashalme generiert werden eine größte Größe genutzt und die Bereiche relativ zu der Entfernung zur Kamera rekursiv unterteilt werden.

**Es gibt Schieberegler für:**

- die Grasdichte.
- die Entfernung die ein Patch maximal haben kann um noch unterteilt zu werden.
- die maximale Rekursionstiefe für die Unterteilung der Bereiche.
- den Faktor mit dem die Dichte multipliziert wird wenn unterteilt wird.
- einen Faktor der angibt wie breit die Grashalme sein sollen.

**Weiterhin erwähnenswert sind folgende Debug Shortcuts:**

**B** Zeichnen der Grenzen von Patches an- beziehungsweise ausschalten

**C** Stoppt oder startet die Neuberechnung von Patches

Mit diesen Optionen kann die korrekte Unterteilung der Patches sowie der Schnitt mit dem Frustum überprüft werden (zum Beispiel in Abbildung 3.7 zu sehen).

### 8.2.2 Struktur

Die Struktur des Codes ist relativ einfach. In der Datei `main.cpp` wird die Hauptschleife ausgeführt, das User Interface konfiguriert, Shader initialisiert und Draw-Calls ausgeführt. Für die Kommunikation mit den Shadern werden structs verwendet, die in `renderer.glsL.h` definiert sind und von C++ sowie Shader Code eingebunden werden. Die Shader selbst sind als glsl Dateien definiert, so sind Vertex, Tesselation Control, Tesselation Evaluation und Fragment Shader der Referenzimplementierung in `referenceGeometryGrass.glsL` definiert. Der Compute Shader, der das Gras in der Hauptanwendung zeichnet, ist in `computeShaderGrass.glsL` definiert. Das Ergebnis des Compute Shaders wird mit dem Shader `csResultShader.glsL` gerendert. Für das Terrain unter dem Gras wurden in `groundShader.glsL` einfache Vertex und Fragment Shader implementiert die die xz-Ebene zeichnen. All diese Dateien sind im Wurzelverzeichnis des Programms zu finden.

### 8.2.3 Bekannte Fehler

**Falsche Eigenschaften beim Starten des Programms** Wenn das Programm gestartet wird, hat das dargestellte Gras falsche Eigenschaften, erst wenn die Kamera bewegt wird ist das gewünschte Ergebnis zu sehen.

**Zittern bei weiter Entfernung zum Ursprung** Das Gitter, in dem das Gras gezeichnet wird, scheint bei weiter Entfernung zum Ursprung zu Zittern.

**Teilweise fehlen Zellen in Bildschirmtiles** Teilweise werden nicht alle Zellen, die nicht ganz in einem Teilfrustum liegen auch miteinbezogen

**Zu dicke Halme werden durch Tilegrenzen abgeschnitten** Besonders dicke Halme erscheinen manchmal abgeschnitten, wenn sie über Tilegrenzen hinausreichen.

# Literatur

- [BLH02] Brook Bakay, Paul Lalonde und Wolfgang Heidrich. „Real-time animated grass“. In: *Eurographics 2002*. 2002.
- [Bli77] James F. Blinn. „Models of Light Reflection for Computer Synthesized Pictures“. In: *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '77. San Jose, California: ACM, 1977, S. 192–198. DOI: [10.1145/563858.563893](https://doi.org/10.1145/563858.563893).
- [Bou08] Kévin Boulanger. „Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting“. Diss. University of Central Florida Orlando, Florida, 2008, S. 53–99.
- [Brk+09] Belma R. Brkic, Alan Chalmers, Kevin Boulanger, Sumanta Pattanaik und James Covington. „Cross-modal Affects of Smell on the Real-time Rendering of Grass“. In: *Proceedings of the 25th Spring Conference on Computer Graphics*. SCGG '09. Budmerice, Slovakia: ACM, 2009, S. 161–166. ISBN: 978-1-4503-0769-7. DOI: [10.1145/1980462.1980494](https://doi.org/10.1145/1980462.1980494).
- [Deu+02] Oliver Deussen, Carsten Colditz, Marc Stamminger und George Drettakis. „Interactive visualization of complex plant ecosystems“. In: *Proceedings of the conference on Visualization'02*. IEEE Computer Society. 2002, S. 219–226.
- [Fan+15] Zengzhi Fan, Hongwei Li, Karl Hillesland und Bin Sheng. „Simulation and rendering for millions of grass blades“. In: *Proceedings of the 19th symposium on interactive 3D graphics and games*. ACM. 2015, S. 55–60.
- [Gue+03] Sylvain Guerraz, Frank Perbet, David Raulo, François Faure und Marie-Paule Cani. „A procedural approach to animate interactive natural sceneries“. In: *Computer Animation and Social Agents, 2003. 16th International Conference on*. IEEE. 2003, S. 73–78.
- [Hem16] Nico Hempe. „Real-Time Rendering Approaches for Dynamic Outdoor Environments“. In: *Bridging the Gap between Rendering and Simulation Frameworks: Concepts, Approaches and Applications for Modern Multi-Domain VR Simulation Systems*. Wiesbaden: Springer Fachmedien Wiesbaden, 2016, S. 110–112. ISBN: 978-3-658-14401-2. DOI: [10.1007/978-3-658-14401-2\\_4](https://doi.org/10.1007/978-3-658-14401-2_4).
- [Hen+13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden und Alexandru Iosup. „Procedural content generation for games: A survey“. In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9.1 (2013), 1:9.
- [HH12] Dongsoo Han und Takahiro Harada. „Real-time hair simulation with efficient hair style preservation“. In: (2012).
- [HL15] Dongsoo Han und Hongwei Li. „Grass Rendering and Simulation with LOD“. In: *GPU Pro 6: Advanced Rendering Techniques* (2015), S. 91.

- [HRS13] Nico Hempe, Jürgen Rossmann und Björn Sondermann. „Generation and rendering of interactive ground vegetation for real-time testing and validation of computer vision algorithms“. In: *ELCVIA: electronic letters on computer vision and image analysis* 12.2 (2013), S. 40–53.
- [HWJ07] Ralf Habel, Michael Wimmer und Stefan Jeschke. „Instant Animated Grass“. In: *Journal of WSCG* 15.1-3 (2007), S. 123–128.
- [Iki+04] Milan Ikits, Joe Kniss, Aaron Lefohn und Charles Hansen. „Volume rendering techniques“. In: *GPU Gems* 1 (2004).
- [JSK09] Orthman Jens, Christof Rezk Salama und Andreas Kolb. „GPU-based responsive grass“. In: (2009).
- [JW13] Klemens Jahrmann und Michael Wimmer. „Interactive Grass Rendering Using Real-Time Tessellation“. In: (2013).
- [LDY12] Feng Li, Ying Ding und Jin Yan. „Real-Time Rendering and Animating of Grass“. In: *AsiaSim 2012: Asia Simulation Conference 2012, Shanghai, China, October 27-30, 2012. Proceedings, Part III*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 296–303. ISBN: 978-3-642-34387-2. DOI: 10.1007/978-3-642-34387-2\_34.
- [NVI09] NVIDIA. *NVIDIA Fermi Compute Architecture Whitepaper*. Techn. Ber. NVIDIA, 2009. URL: [http://www.nvidia.de/content/PDF/fermi\\_white\\_papers/NVIDIA\\_Fermi\\_Compute\\_Architecture\\_Whitepaper.pdf](http://www.nvidia.de/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf).
- [NVI15] NVIDIA. *Life of a triangle - NVIDIA's logical pipeline*. 2015. URL: <https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline> (besucht am 17.10.2016).
- [Ope16] OpenGL. *Tessellation*. 2016. URL: <https://www.opengl.org/wiki/Tessellation> (besucht am 03.08.2016).
- [Pan14] David Pangerl. „Screen-Space Grass“. In: *GPU Pro 5: Advanced Rendering Techniques* (2014), S. 221.
- [Pap15] Dimitris Papavasiliou. „Real-Time Grass (and Other Procedural Objects) on Terrain“. In: *Journal of Computer Graphics Techniques (JCGT)* 4.1 (2015), S. 26–49. ISSN: 2331-7418.
- [PC01] Frank Perbet und Maric-Paule Cani. „Animating prairies in real-time“. In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. ACM. 2001, S. 103–110.
- [Pel04] Kurt Pelzer. „Rendering countless blades of waving grass“. In: *GPU Gems* 1 (2004), S. 107–121.
- [RB85] William T Reeves und Ricki Blau. „Approximate and probabilistic algorithms for shading and rendering structured particle systems“. In: *ACM Siggraph Computer Graphics*. Bd. 19. 3. ACM. 1985, S. 313–322.
- [Ree13] Nathan Reed. *Quick And Easy GPU Random Numbers In D3D11*. 2013. URL: <http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/> (besucht am 02.11.2016).
- [SKP05] Musawir A Shah, Jaakko Kontinnen und Sumanta Pattanaik. „Real-time rendering of realistic-looking grass“. In: *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. ACM. 2005, S. 77–82.
- [Sou07] Tiago Sousa. „Vegetation procedural animation and shading in crysis“. In: *GPU Gems* 3 (2007), S. 373–385.

- [Zin12] Alois Zingl. *A rasterizing algorithm for drawing curves*. 2012. URL: <http://members.chello.at/~easyfilter/bresenham.html>.



# **Erklärung**

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den 3. November 2016

(Yannick Tanner)