

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

27. Oktober 2016

Abstract

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 work groups of 32 threads. The grass blades in each tile are rendered front to back to avoid using depth test 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 picture based methods but with some further optimizations can prove valid as an alternative to geometry based grass for medium to far distances.

Zusammenfassung

Diese Arbeit beschreibt die Implementierung einer neuen Methode Gras in Echtzeit darzustellen. Der Ansatz benutzt einen Compute Shader um jeden einzelnen Grashalm parallel 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 Bild basierten Methoden behaupten, könnte aber mit weiteren Optimierungen eine Alternative zu Geometrie basiertem Gras auf mittlere bis weite Entfernung darstellen.

Inhaltsverzeichnis

1 Einleitung	1
2 Stand der Forschung	3
2.1 Darstellung von Grashalmen	3
2.1.1 Bild-basierte Methoden	3
2.1.2 Volumetrische Texturen	5
2.1.3 Zeichnen jedes einzelnen Halmes	5
2.1.3.1 Generierung der Halme	5
2.1.4 Gemischte Methoden	6
2.2 Beeinflussung des Grases	7
2.2.1 Variation	7
2.2.2 Externe Kräfte	7
2.2.2.1 Windsimulation	8
2.2.2.2 Kollision	9
3 Referenzarbeit: Darstellung von Gras durch Erzeugung von Geometrie	11
3.1 Erzeugung der Grashalme	11
3.1.1 Zufällige Wahl der Eigenschaften	11
3.1.2 Tesselierung und LOD	12
3.1.3 Geometrieerzeugung mit Hilfe von Bézier Kurven	13
3.2 Darstellung einer unendlichen Grasfläche	14
3.2.1 Unterteilung der sichtbaren Fläche in Quadrate	14
3.2.2 Weitere Unterteilung der Quadrate	16
4 Zeichnen von Gras unter Nutzung eines Compute Shaders	21
4.1 Konzept	21
4.2 Finden von Positionen zum Zeichnen der Grashalme	21
4.2.1 Unterteilung des Bildschirms	22
4.2.2 Vorarbeit für die Front to Back Abarbeitung	22
4.2.3 Aufteilung der Gitterzellen auf Threads	24
4.2.3.1 Grundlegende Vorgehensweise	24
4.2.3.2 Reduktion der Schrittgröße	25
4.2.3.3 Weicher Übergang zwischen Gitterstufen	27
4.2.3.4 Implementierung	27
4.3 Zeichnen der Halme	33
4.3.1 Bestimmung der Eigenschaften	33
4.3.2 Zeichnen	33
4.3.2.1 Scanline-Rasterung	33
4.3.2.2 Shading	35
4.3.2.3 Anti-Aliasing	36
4.3.2.4 Nutzung der Front to Back Eigenschaft für Überdeckung .	37
4.4 Kopieren in den Framebuffer	37

4.5 Wind	38
4.6 Ergebnis	39
5 Fazit	41
5.1 Performance	41
5.1.1 Aufgliederung der Zeiten	41
5.1.2 Auswirkungen verschiedener Entfernung	42
5.1.3 Einsparungen durch Rasterungsabbruch und Shared Memory	43
5.1.4 Vergleich mit Geometriegras	43
5.2 Bewertung der Anwendbarkeit	44
5.3 Anwendungsmöglichkeiten	44
5.3.1 Kombination mit anderen Methoden	44
6 Ausblick	45
6.1 Mögliche Optimierungen	45
6.2 Mögliche Verbesserungen	45
6.2.1 Flexibilität	45
6.2.2 Weiches Aus- und Einblenden einzelner Halme	46
6.3 Möglichkeiten zur Weiterentwicklung	46
6.3.1 Einbindung in eine Szene	46
6.3.2 Terrain	47
6.3.3 Texturen	47
6.3.4 Shading	47
6.3.5 Externe Beeinflussung der Grashalme	48
6.3.5.1 Windanimation	48
6.3.5.2 Kollision	48
6.3.6 Weitere Pflanzenarten	48
7 Technische Details	49
7.1 Umgebung	49
7.2 Applikation	49
7.2.1 Quellcode	49
7.2.2 User Interface	49
7.2.2.1 Hauptanwendung	49
7.2.2.2 Referenzanwendung	51
7.2.3 Struktur	51
7.2.4 Bekannte Fehler	52
Literaturverzeichnis	53

1. Einleitung

Sei es in der Stadt oder auf dem Land Gras ist ein fester Bestandteil aller möglichen Szenarien. 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 die Knie zwingen 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 einfach mit dem Bild einer Grasfläche zu versehen bis hin dazu Grashalme zu modellieren, 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 passiert. Dieser wird höchst parallel auf der Grafikkarte ausgeführt. Durch die Nutzung des Compute-Shaders kann jeder Aspekt des Prozesses kontrolliert werden, somit gibt es einige Möglichkeiten zur Optimierung. So sollen für die Grashalme Kurven benutzt werden die durch Projektion auf den Bildschirm gezeichnet werden. Des Weiteren wird die Verdeckung der Grashalme ausgenutzt, 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 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. Es bietet sich also an in nahen Bereichen zum Beispiel Geometrie für das Gras zu verwenden.

2. Stand der Forschung

2.1 Darstellung von Grashalmen

Es wurde bereits kurz angesprochen was für Methoden es schon gibt um Gras in Echtzeit darzustellen. Im Folgenden soll beschrieben werden wie der Stand der Forschung zur Darstellung von Gras in Echtzeit aussieht.

2.1.1 Bild-basierte Methoden

Die einfachste Methode ist eine Fläche auf die ein Bild einer Grasfläche abgebildet ist zu benutzen, diese Methode vereinfacht Gras zu einer grünen Fläche was für manche Anwendungen hinreichend sein mag. Sie wurde vor allem benutzt als andere Möglichkeiten die Leistungsfähigkeit von derzeitiger 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 entsprechend mehr Flächen hinzugefügt werden was wiederum schlechtere Performance zur Folge hat. Allerdings gibt es diverse Optimierungen und Verbesserung dieses Verfahrens so 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.

[Hem16] beschreibt 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.

Semitransparente texturierte Flächen haben natürlich 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,

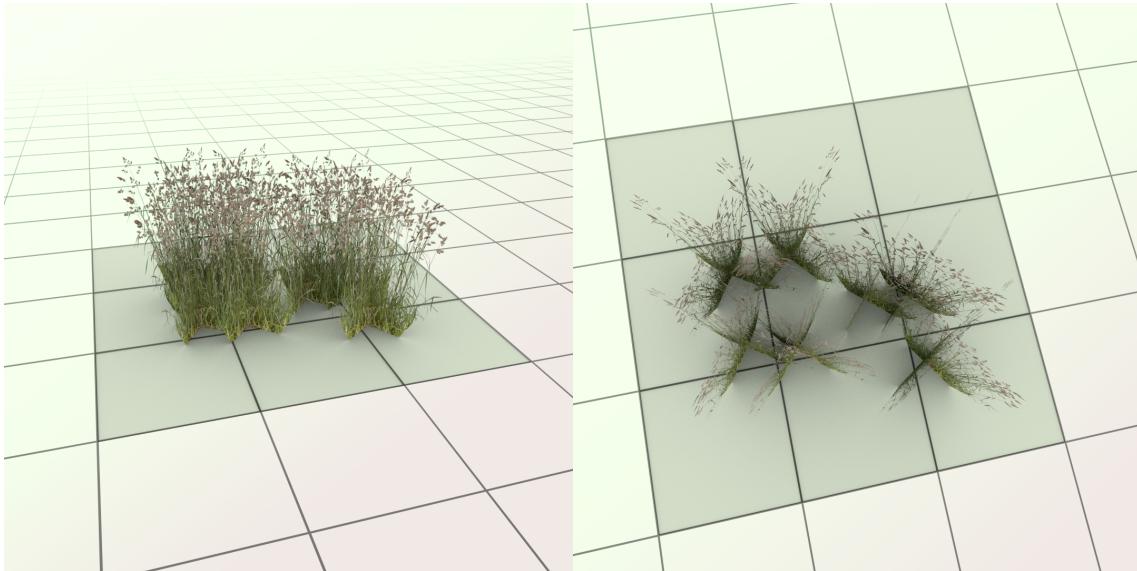


Abbildung 2.1: Gras als Flächen mit einer semitransparenten Textur dargestellt. Links: von der Seite (der gewünschte Betrachtungswinkel) Rechts: von oben (Einzelne Flächen deutlich zu erkennen)

dass alle Flächen von hinten nach vorne betrachtet werden müssen um die korrekten Farben für ein Pixel zu erhalten. Im Falle von Gras sind dies potentiell sehr viele. [HWJ07] implementiert deshalb 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 implementiert [HWJ07] eine Methode die es dem Gras erlaubt andere Objekte zu überdecken, was bei Bump- oder Parallax-Mapping normalerweise ein Problem ist.

Ein anderer Ansatz der ebenfalls texturierte Flächen benutzt wird in [BLH02] beschrieben hierbei werden allerdings 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 Endruck 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 im Schnitt weniger Geometrie ausgekommen 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 auch ziemlich detailliert im Vergleich zu dem was mit horizontalen Flächen möglich ist.

Bild-basierte Methoden zum Zeichnen von Gras 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 verwendet werden. Da es sich um eine direkte Abbildung der Wirklichkeit mit all ihren Details handelt ist diese Vorgehensweise nahezu unersetztbar 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 LOD Optimierungen wie zum Beispiel in [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] und [IKLH04]), allerdings ist der Speicherverbrauch sehr hoch (Vgl. [SKP05]). Es gibt allerdings 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 [SKP05] wird ein ähnliches Konzept verfolgt indem für eine Grasfläche, von einem darüber liegenden Betrachter, Tiefeninformationen und eine BTF (Bi-directional Texture Functions) abgespeichert werden. Die BTF speichert dabei Positions- und Beleuchtungsinformationen zu verschiedenen Blickwinkel. 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 benutzen und die Halme in einer realistischen Art und Weise auf einer Fläche zu verteilen. Dies ist auch 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. In [HL15] wird eine Vorgehensweise zum Zeichnen einzelner Grashalme beschrieben bei der Modelle für zweidimensionale Halme generiert und dann zufällig skaliert, rotiert und platziert werden. Für die Echtzeitdarstellung 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 [JW13] beschrieben. In der Arbeit werden Quads zu Grashalmen tesseliert. Auf diese Weise kann durch die Tesselierung, die auf der GPU geschieht, sehr effizient Geometrie für jeden einzelnen Grashalm dargestellt werden. Da die Generierung der 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 gecullt. Mit dem gleichen Thema beschäftigt sich [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 LOD benutzt wird. So wird sich in der Arbeit auf große Entfernung einfach auf die Textur der unter dem Gras liegenden Landschaft verlassen.

2.1.3.1 Generierung der Halme

Die einzelnen Grashalme werden bei der Mehrheit aller Methoden in irgend einer Form automatisch generiert. Wobei in [HL15] erwähnt wird, dass von Hand modellierte Halme

auch möglich sind. In der Arbeit 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 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 benutzt, algorithmisch kann einfach vielfältiges Gras ohne erkennbare Muster erzeugt werden. Es werden normalerweise zwei verschiedene Methoden zur Generierung von Grashalmen genannt. 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. [DCSD02] schlägt 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. [Bou08] benutzt 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 Halme ist die Nutzung von Kurven. [HL15] beschreibt die Nutzung von zwei Splines die 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. Andere Ansätze umfassen die Nutzung von kubischen hermitischen Splines die durch Anfangs- und Endpunkt sowie Tangenten an diesen beschrieben sind.

2.1.4 Gemischte Methoden

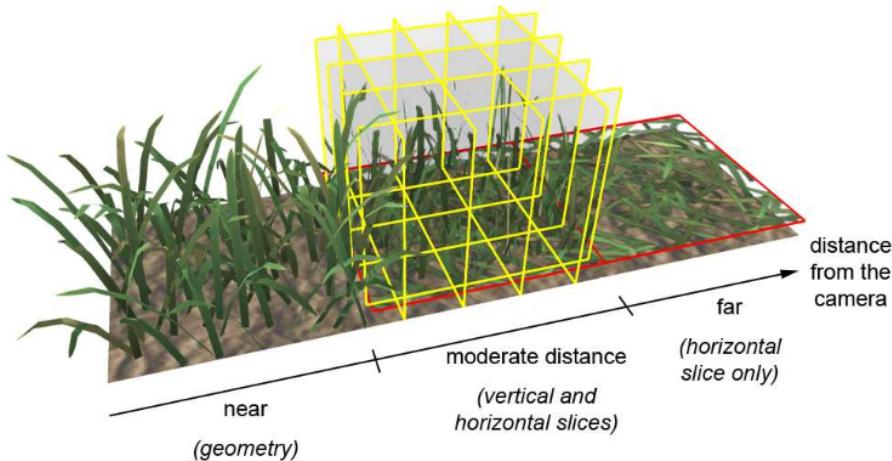


Abbildung 2.2: Gemischer Ansatz mit drei Levels of Detail. Bild aus [Bou08]

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 zum Beispiel [Bou08]. Hier wird ein Ansatz beschrieben bei dem in nahen Bereichen Geometrie für jeden Grashalm gezeichnet wird, weiter weg kommen vertikale und horizontale texturierte Flächen zum Einsatz. 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.

In [LDY12] ist eine ganz ähnliche Methode zur Darstellung des Grases verwendet worden. Es werden ebenfalls 3 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 hochauflösenden umgekehrt proportional zu.

Bei beiden Methoden wird für die mittlere Stufe ein pseudovolumetrischer Ansatz benutzt 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

GPU Pro 5 Kapitel 3 Abschnitt 1 Screen-Space Grass ([Pan14]) stellt 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 also vertikale Linien, die mit Hilfe von Tiefen- und Stencil-Buffer auch genau dort gezeichnet werden wo die Grashalme wachsen würden. 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 in [BCB⁺09] untersucht. Die Arbeit beschäftigt sich mit dem Konzept Geruch von Gras zu benutzen 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 keine echte Szene handelt wenn ein entsprechender Geruch involviert ist (Vgl. [BCB⁺09]).

2.2 Beeinflussung des Grases

2.2.1 Variation

Gras ist natürlich 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 [FLHS15, 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 (siehe zum Beispiel [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 auch sehr aufwändig. Es ist durchaus vorstellbar diverse Variationen algorithmisch zu bestimmen oder eine Textur durch ein externes Tool generieren zu lassen (Vgl. [HMVDVI13]).

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 also 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

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

Eine sehr frühe Arbeit zur Darstellung und Simulation von Gras verfassten Reeves und Blau 1985 ([RB85]). Die Autoren betrachten zwar keine Echtzeitanwendungen aber die Konzepte sind trotzdem übertragbar. Für die Simulation von Wind werden hier zufällig generierte Wellen aus Partikeln verwendet die sich alle ungefähr in die generelle 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 Börse 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. [HWJ07] nutzt ebenfalls eine Textur um die Windanimation von Grashalmen zu steuern, dabei wird argumentiert, dass so prozedurale sowie von Hand erstellte Animationen unterstützt werden. Animiert 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.

[PC01] benutzt 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 dies andere vorberechnete Biegungen haben. Die Masken können frei erzeugt bewegt und überlagert werden, Sobald keine Maske mehr Einfluss auf die Halme haben oszillieren diese zurück zu ihrer ursprünglichen Position ähnlich wie in [RB85], diese Animation ist relativ zu der 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 um Windböen zu approximieren. So werden in [Pel04] Flächen mit einer Grastextur animiert indem die obersten Vertices der Vierecke gemäß einer trigonometrischen Funktion bewegt werden. Dazu übergibt die vorgestellte Implementierung ein 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 ist in [FLHS15]¹ erklärt, hier ist jeder Grashalm einzeln als Geome-

¹Die Arbeiten [FLHS15] und [HL15] haben zwei Autoren gemeinsam und stimmen in großen Teilen miteinander überein.

trie 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 sichtbare 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 kein zusätzlicher Speicher benötigt. Aufgrund der Einfachheit ist diese oder eine ähnliche Methode in vielen Arbeiten zu dem Thema zu finden (siehe zum Beispiel auch [JW13, HRS13]). Etwas weiter geht die Arbeit zur prozeduralen Animation von Vegetation [Sou07] auf die in [JSK09] eine Animationsmethode für Gras aufgebaut wird. Der Autor implementiert die Bewegung von Gras im Wind als die Summe geglätteter Dreiecksfunktionen die die oberen Vertices von Billboards entlang der Windrichtung verschiebt. In [Sou07] wird argumentiert, dass diese Funktionen günstiger auszuwerten sind als trigonometrische Funktionen. Allerdings bietet heutige Hardware auch Beschleunigung für trigonometrische Funktionen.

[HL15] ergänzt 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.

In [Hem16] wird eine mehr auf physikalischen Grundlagen beruhende Methode angewandt. Dabei wird die Bewegung von Gras im Wind als erzwungene Schwingung in einem System mit einem einzigen Freiheitsgrad modelliert. Die Kraft die auf ein Halm ausgeübt wird ergibt sich aus der Grashalmfläche und dem Winddruck welcher sich aus der Windschwindigkeit 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 normalerweise ziemlich teuer ist wird in [GPR⁺03] ein Animationsprimitiv an ein Objekt gehängt welches Gras beeinflussen soll (zum Beispiel ein Schuh). Dieses Primitiv agiert ähnlich wie die Masken in [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 [FLHS15] ist eine Grasfläche in Bereiche unterteilt. Diese werden als aktiv markiert wenn sie ein für Kollision in Frage kommendes Objekt beinhalten. 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 kein Kollisionsobjekt sich 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 Halm angewandt.

[JSK09] berechnet 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, dass alle Vertices eines unterteilten Quads verbindet. Genauso

wie in [FLHS15] 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 ist also zuerst langsam und wird dann immer schneller. Erwähnenswert ist weiterhin, dass der Autor nur eine Unterteilung eines texturierten Quads vornimmt, falls dieses kollidiert oder sich erholt.

3. Referenzarbeit: Darstellung von Gras durch Erzeugung von Geometrie

Zum Vergleich der neu entwickelten Methode wurde zuerst eine eher konservativere Methode zum Darstellen von Grashäuten umgesetzt. Hierbei war das Ziel den „Brute-Force“ Ansatz zu benutzen bei dem Geometrie für jeden einzelnen Grashalm erzeugt und generiert 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 semitransparenten Flächen mit Grastexturen zumindest zum Teil ablösen wird.

3.1 Erzeugung der Grashäume

Die Grashäume werden in der Anwendung komplett in Shadern erzeugt und verarbeitet. Das C++ Programm übergibt nur Grenzen eines Bereichs und bestimmt wie oft der entsprechende Vertex Shader aufgerufen wird indem ein, ansonsten keine Informationen enthaltendes, Vertex Array der entsprechenden Größe an OpenGL übergeben wird.

3.1.1 Zufällige Wahl der Eigenschaften

Wie oft der Vertex Shader aufgerufen wird bestimmt hier semantisch wie viele Grashäume 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 dies wird mit einem Pseudozufallszahlengenerator erzeugt der als Seed VertexID und x Position des Bereichs, in dem Grashäume 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 Hälmes 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).

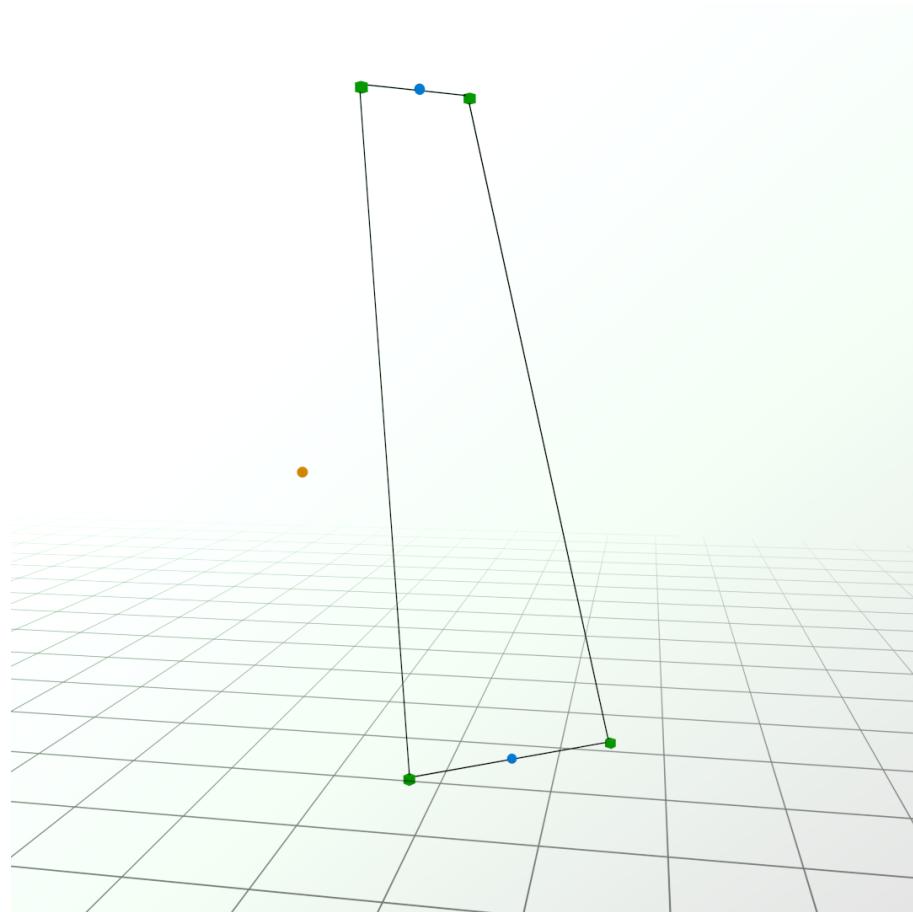


Abbildung 3.1: Grashalm nach Verarbeitung im Vertex Shader. In Grün: Eckpunkte der Ausgangsgeometrie für den Halm. In Blau und Orange: Kontrollpunkte des Halms In Orange: Einflusspunkt

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 höher qualitatives 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 höchstparallel 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 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 Bézier Kurven

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. Veranschaulicht ist dies in Abbildung 3.2 zu sehen, die Koordinaten sind hier mit u und v gekennzeichnet und erstrecken sich von 0 bis 1. Aus den

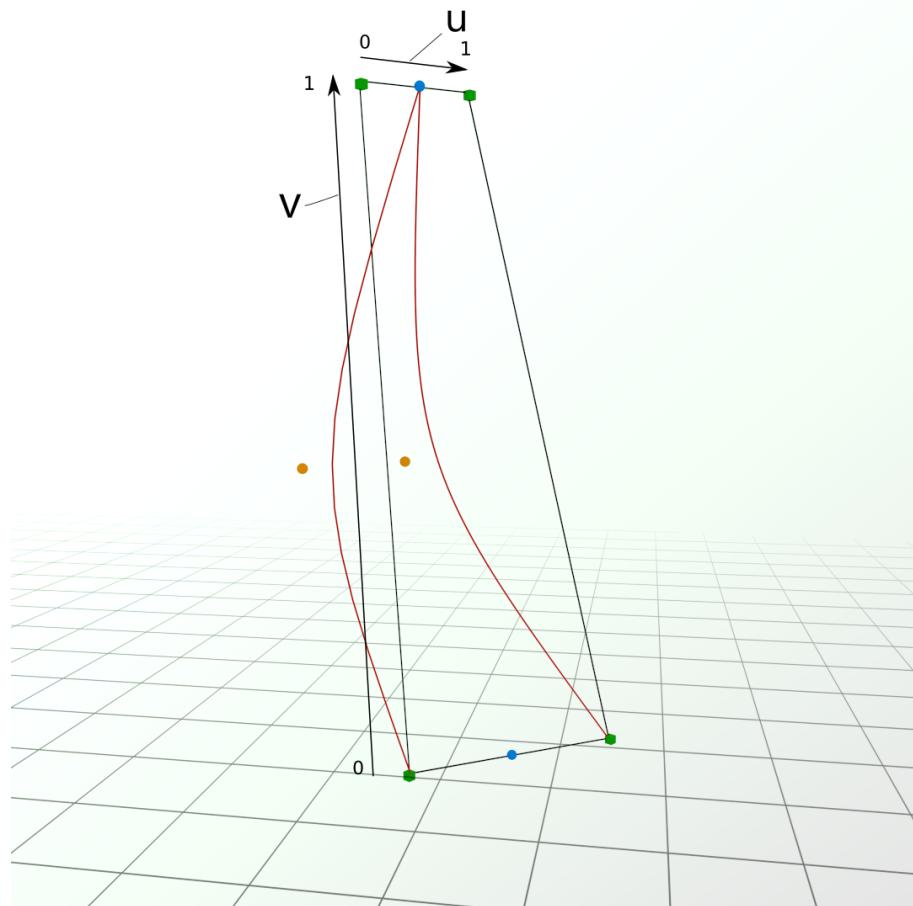


Abbildung 3.2: Veranschaulichung der Verarbeitung des Grashalms durch Tesselierung

vorhandenen Punkten können zwei quadratische Bézier Kurven 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 Hals 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.

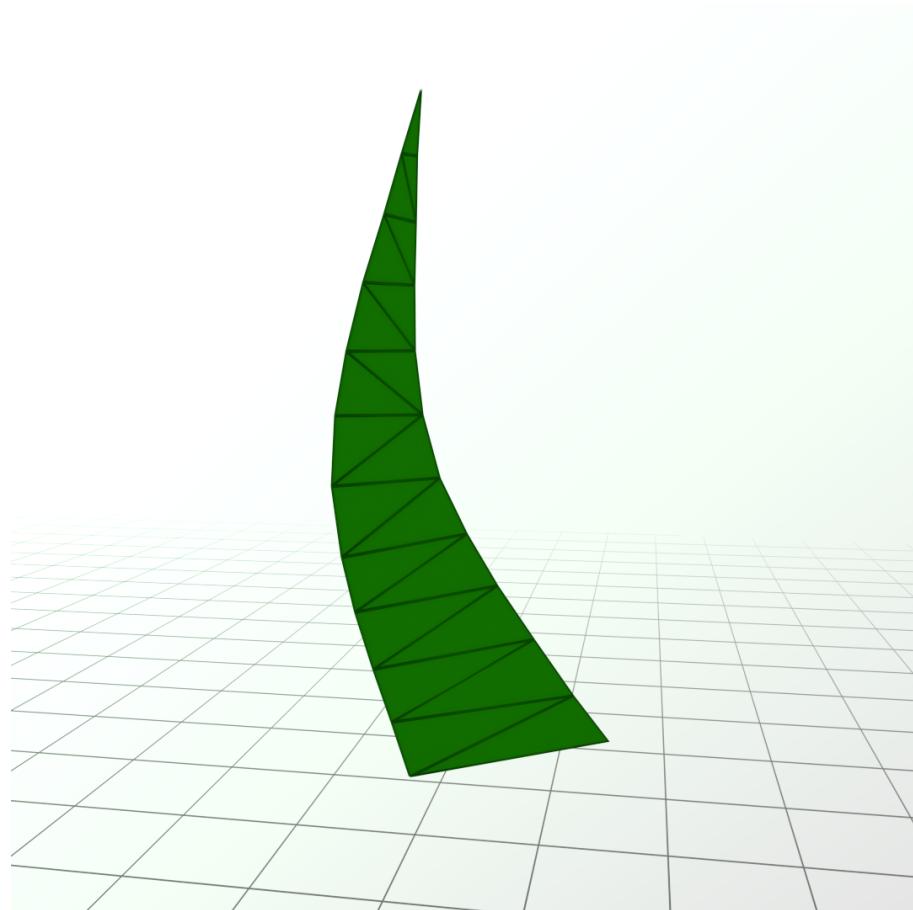


Abbildung 3.3: Ergebnis des Tessellation Evaluation Shaders

3.2 Darstellung einer unendlichen Grasfläche

Um die verschiedenen Methoden unter diversesten 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

Natürlich ist es 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 View Frustum 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

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:   for x  $\leq$  end.x do
8:     for z  $\leq$  end.z do
9:       if ISINFRUSTUM(x,z) then
10:         ADDTOARRAY(x,z)
11:         end if x  $\leftarrow$  x + patchSize
12:       end for z  $\leftarrow$  z + patchSize
13:     end for
14:   end function
15:
16: function ISINFRUSTUM(point)
17:   points  $\leftarrow$  GETPOINTSOFSCUARE(point,patchSize)
18:
19:   for all plane  $\leftarrow$  planes do
20:     if POINTSOUTSIDEOPFLANE(points,plane) then
21:       return False
22:     end if
23:   end for
24:   return True
25: end function
26:
27: function POINTSOUTSIDEOPFLANE(points, plane)
28:   for all point  $\leftarrow$  points do
29:     if DOT(plane.normal, point - plane.point)  $<$  0 then
30:       return False
31:     end if
32:   end for
33:   return True
34: end function
```

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 Er-

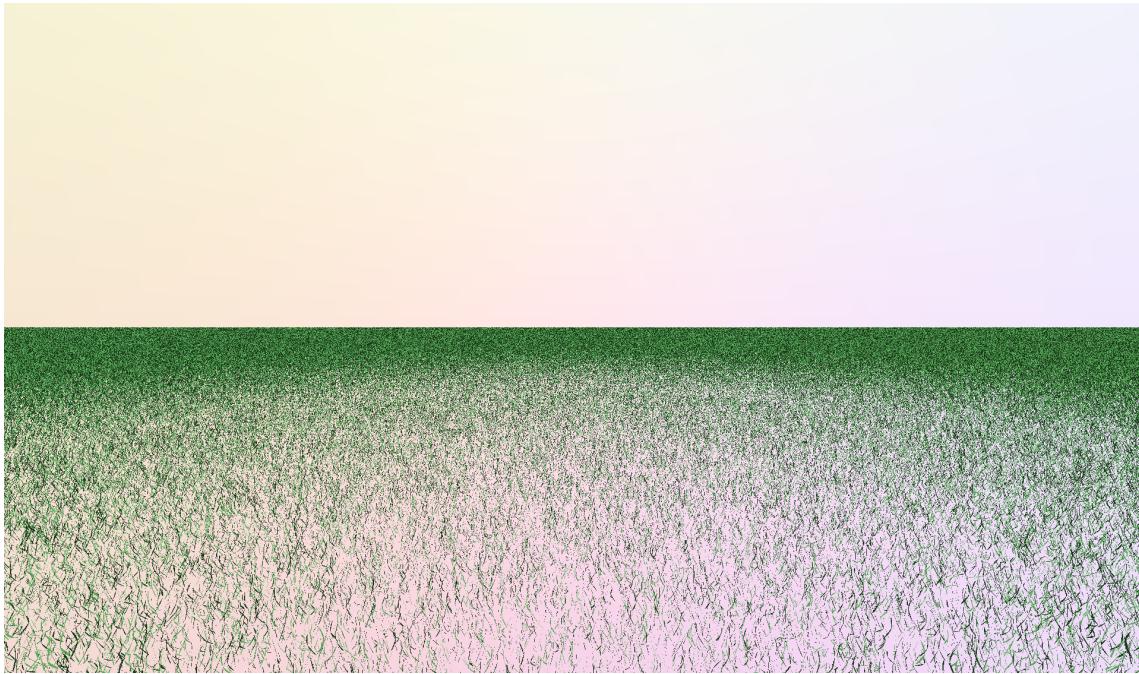


Abbildung 3.4: Eine unendliche Grasfläche

gebnis 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.

3.2.2 Weitere Unterteilung der Quadrate

Zum Beispiel in einem weit entfernten Bereich ist es nicht mehr nötig ganz so viele Grashalme zu zeichnen wie in einem nahen Bereich. 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, außerdem ist dann der Übergang zwischen Bereichen weniger offensichtlich, da näher am Beobachter mehr Bereiche dargestellt werden. Es würde sich also 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 will man also zum Einen Grashalme ausblenden um nicht unnötig viele auf größere Entfernung zu rendern und zum Anderen nicht offensichtlich ganz andere Halme

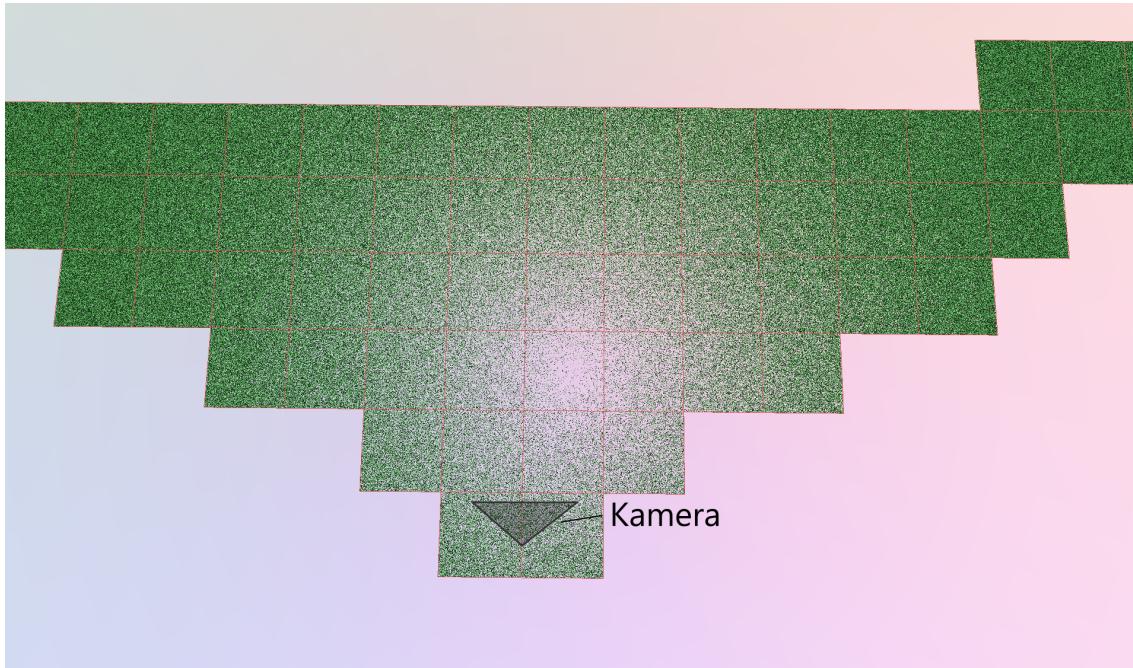


Abbildung 3.5: Top Down Ansicht von Abbildung 3.4 (Tatsächlich gezeichnete Patches)

darstellen wenn sich die Kameraposition verändert. Die Bereichsgrößen können also nicht komplett frei bestimmt werden. Es wurde also als Ausgangspunkt ein gröbstes Gitter definiert, die Quadrate dieses Gitters werden immer mit Grashalmen gefüllt, es 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 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 Entfernung wesentlich weniger Halme gezeichnet.

In Algorithmus 2 ist der Algorithmus zur rekursiven Unterteilung von Patches zu sehen dieser wird in Algorithmus 1 in Zeile 10 anstatt der if Abfrage und addToArray aufgerufen. Für die Unterteilung wird hier die Entfernung zur Kamera benutzt, 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. 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 An-

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:     pseudoDistance  $\leftarrow$  DISTANCEPATCHTOCAMERA(position, size)
10:    pseudoDistance  $\leftarrow$  pseudoDistance + (recursionDepth/maxRecurstionDepth) *
11:      maxDistToSubdivide
12:    if pseudoDistance < maxDistToSubdivide then
13:       newSize  $\leftarrow$   $\frac{\text{size}}{2}$  i  $\leftarrow$  0
14:      for i < 4 do
15:        newPosition.x  $\leftarrow$  (postion.x + (i mod 2) * newSize)
16:        newPosition.z  $\leftarrow$  (postion.z + (int( $\frac{i}{2}$ )) * newSize)
17:        newDensity  $\leftarrow$  density * densityFactor
18:        ADDANDSUBDIVIDE(newPosition, newSize, newDensity, recursionDepth+1)
19:
20:        i  $\leftarrow$  i + 1
21:      end for
22:    end if
23:  end if
24: end function

```

wendung finden. Zum Beispiel kommt oft Alpha Blending zum Einsatz um das Erscheinen neuer Objekte beziehungsweise Details zu verbergen.



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

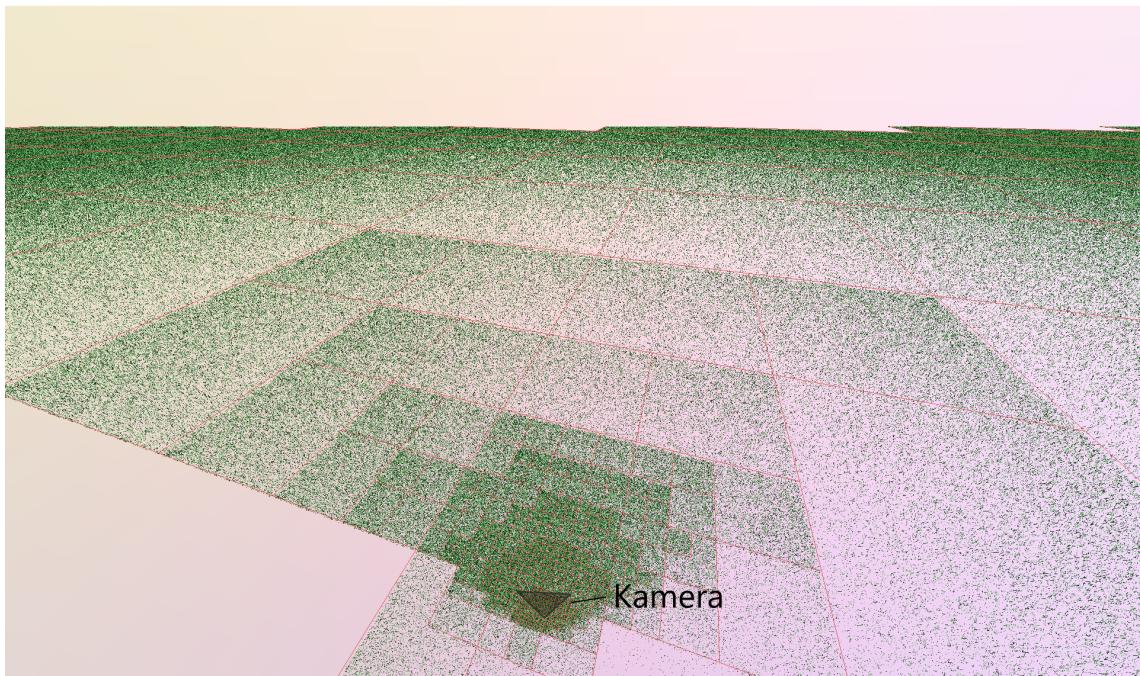


Abbildung 3.7: Top Down Ansicht von Abbildung 3.7 Unterteilung der Patches und Erhöhung der Grasdichte nahe der Kamera

4. Zeichnen von Gras unter Nutzung eines Compute Shaders

Im Folgenden wird das eigentliche Thema 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. Ein Compute Shader ist eine Möglichkeit alle möglichen Berechnungen auf der GPU durchzuführen, 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 also möglich Berechnungen höchst-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 indem der Shader einzelne Halme auf den Bildschirm zeichnet, idealerweise unter maximaler Ausnutzung der parallelen Kapazitäten der GPU. Um dies zu bewerkstelligen wird der Bildschirm in Quadrate unterteilt, jedes Quadrat wird durch eine Arbeitsgruppe bearbeitet die jede 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 also 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 Halme gestoßen wird die schon gezeichnet sind dies bedeutet also weniger Arbeit für größtenteils überdeckte Halme.

4.2 Finden von Positionen zum Zeichnen der Grashalmen

Die erste Aufgabe die sich stellt ist das Finden einer Position an dem jeder Thread seinen Grashalm zeichnen kann. Gearbeitet wird hier in Weltkoordinaten die dann unter Zunahme der ModelViewProjection Matrix auf den Viewport abgebildet werden können.

4.2.1 Unterteilung des Bildschirms

Zuerst müssen Grenzen für die Arbeitsgruppen gesteckt werden. Jede Arbeitsgruppe mit jeweils 32 Threads soll einen Teil des Bildschirms bearbeiten. Dazu wurde der Bildschirm in 32×32 Pixel große Bereiche unterteilt. Im Compute Shader wird mit der Größe des Viewports in Pixeln ein Teilfrustum berechnet in welchem 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.

Der Schnitt enthält nun alle Grashalme die in dem entsprechenden Bildschirmteil ihre

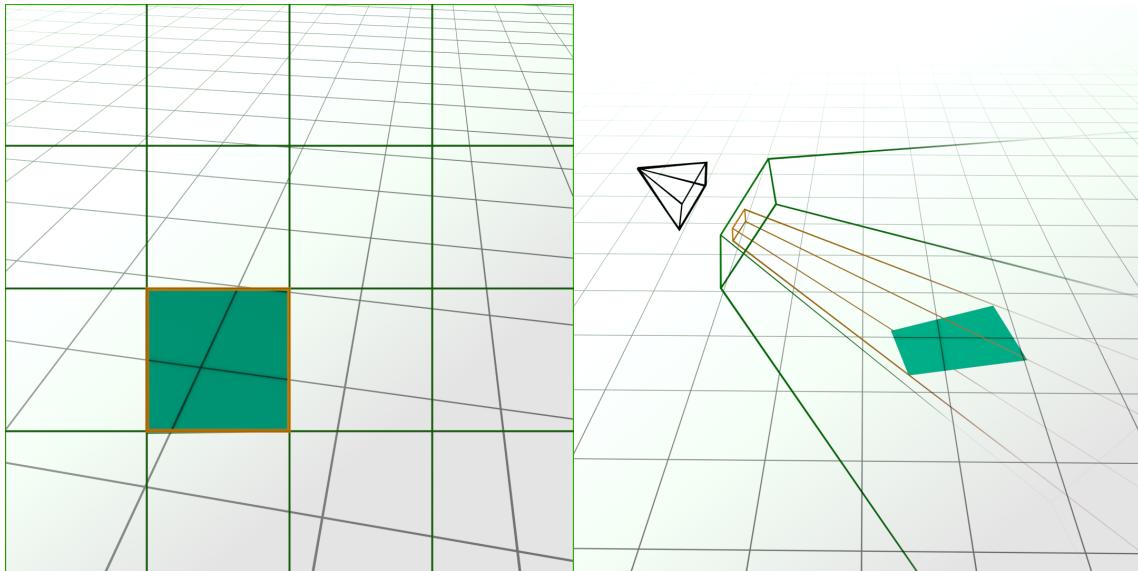


Abbildung 4.1: Unterteilung des Viewports in Quadrate für die Arbeitsgruppen und Schnitt eines Teilfrustums mit der XZ-Ebene (links aus Sicht der Kamera und rechts von außerhalb)

Wurzel haben, es müssen jedoch auch noch alle Grashalme berücksichtigt werden die in den Bildschirmausschnitt hineinwachsen. so kann zum Beispiel die Spitze eines Grashalmes, 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 in welchem Bereich solche Grashalme liegen können. 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 Grashalm spitzen und der möglichen Grashalmwurzeln, alle Grashalme in diesen Schnitten werden durch die Erweiterung des Frustums nun später betrachtet.

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

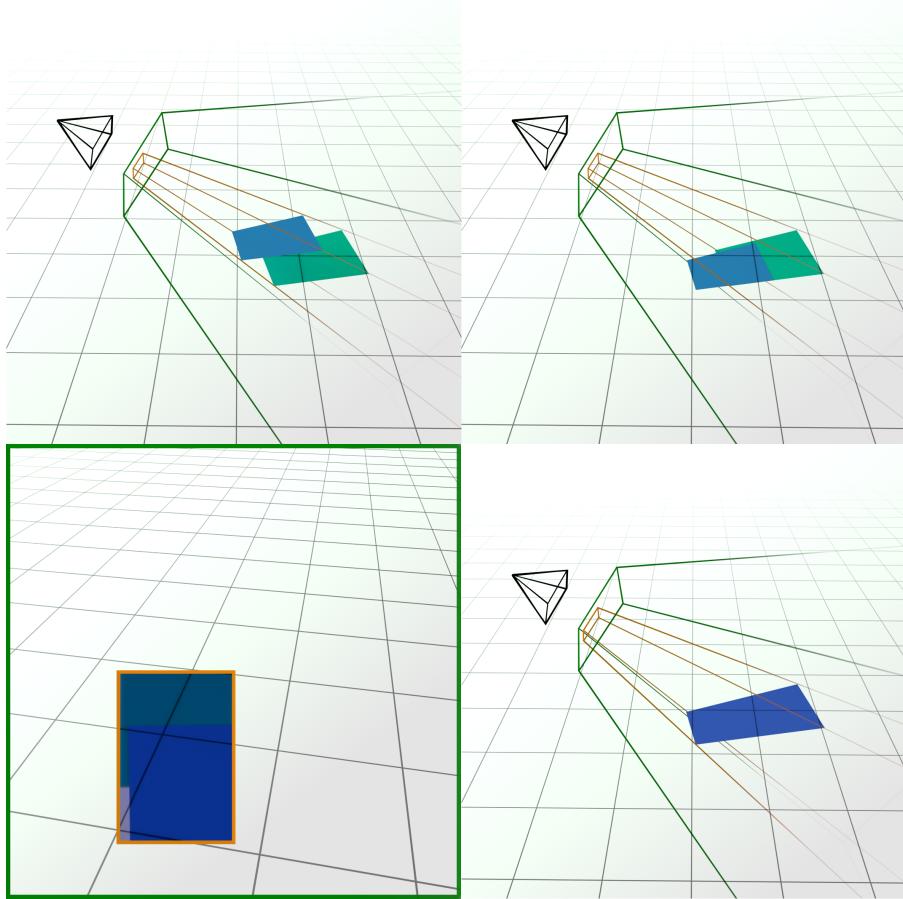


Abbildung 4.2: Links oben: Schnitt mit der Ebene der Grasspitzen (blau). Rechts oben: Verschieben des Schnittes auf die XZ-Ebene. Unten: Erweiterung des Teilfrustums

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.

(Illustration?) Als nächstes wird der Startpunkt für die Abarbeitung festgelegt und zwar als der Schnittpunkt des Teilfrustums der bezüglich der Kamerarichtung am kleinsten ist. 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 dieser wird später bei der Abarbeitung als eigentlicher Anfang benutzt um zu gewährleisten, dass durch alle Gitterzellen im Teilfrustum iteriert wird. 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 festgestellt welche Ebenen des Frustums geprüft werden müssen, indem herausgefunden wird in welche Richtung der zum Front to Back Vektor rechtwinklige zeigt,

ToDo

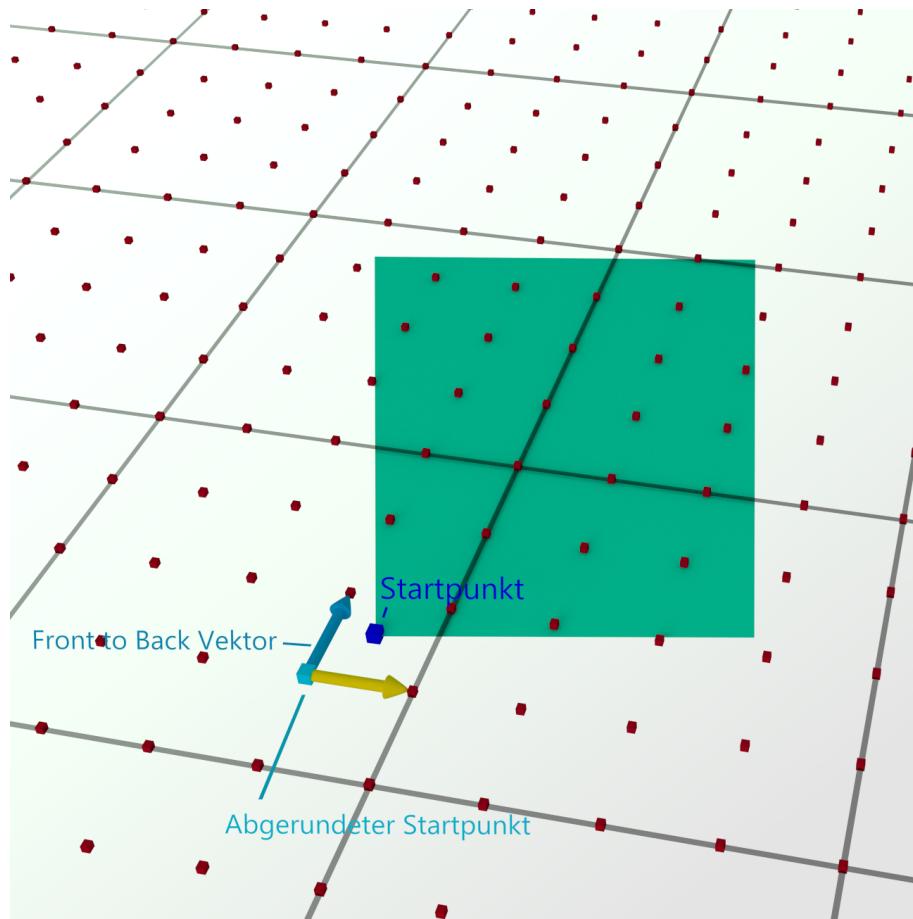


Abbildung 4.3: Reguläre Gitterrichtung der Kamera (Front to Back Vektor), sowie rechtwinkliger Vektor dazu (Gelb). Außerdem Startpunkt für die Iteration durch die Gitterzellen im Teilfrustumsschnitt. Eckpunkte aller Gitterzellen sind hier in rot gekennzeichnet

diese werden später für die Iteration verwendet.

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 (*stepSize*), dazu später mehr. 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. In der aktuellen Zeile wird für jeden Thread die in

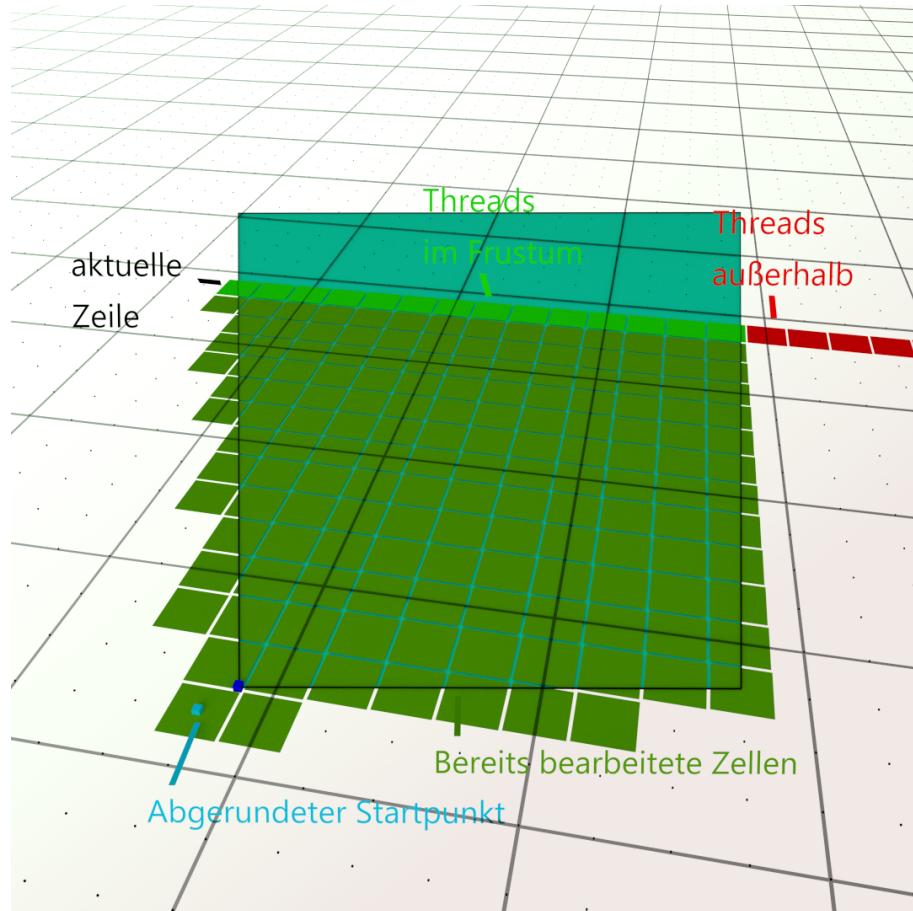


Abbildung 4.4: Iteration der Threads durch die Gitterzellen im Frustum

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 35)

Um den Zeilenanfang zu berechnen wird die Zeile mit dem Frustum geschnitten (Algorithmus 3 Zeile 15). 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 beinhaltet kann (Algorithmus 3 Zeile 21).

Abgebrochen wird sobald in einer neuen Zeile kein Thread mehr im Frustum ist (Algorithmus 3 Zeile 79)

4.2.3.2 Reduktion der Schrittgröße

Genau wie auch bei der Erzeugung von Geometriegras (Unterabschnitt 3.2.2), sollen hier mit größerer Entfernung weniger Grashalme angezeigt werden. Dazu wurde die Gittergröße

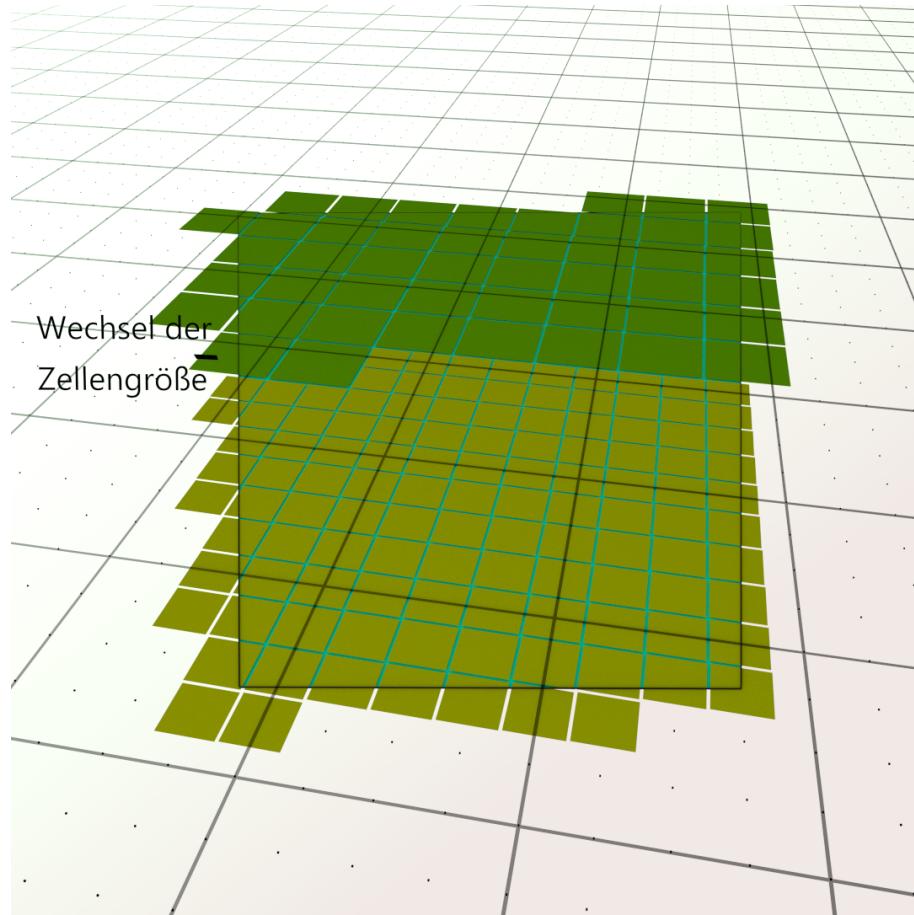


Abbildung 4.5: Variation der Gittergröße

variabel gemacht, diese wird in jeder Zeile über

`GETSTEP SIZE(position)`

neu berechnet. So kann mit größerer Entfernung ein entsprechend größeres Gitter verwirklicht werden. Die Gittergröße selbst hängt von der Entfernung eines Punktes von der Kamera ab und 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. Die so berechnete Zellengröße kann natürlich nicht einfach so verwendet werden ansonsten hätte jeder Gitterpunkt eine eigene Zellengröße, deshalb wird auf einen Vielfachen 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 also immer vier Zellen zu einer zusammengefasst. Die initiale Gittergröße ist bestimmt durch die Gittergröße in Pixeln auf dem Bildschirm an einer bestimmten Entfernung.

Die Schrittgröße wird für die aktuelle Zeile konservativ approximiert, das heißt es kann passieren, dass die Zeile Teile enthält in denen die Schrittgröße 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 52). Dies funktioniert da sich die Schrittgröße immer verdoppelt, falls also zum Beispiel einige Zellen existieren, für die eigentlich schon die nächst größere Schrittgröße gelten müsste, diese aber nicht verwendet wird, wird jede zweite Zelle ausmaskiert. Die restlichen Zellen

erhalten dann die entsprechend größere Schrittgröße. Erkennt ein Thread die gefundene Gitterzelle als ausmaskiert sucht dieser weiter nach einer neuen Zelle.

4.2.3.3 Weicher Übergang zwischen Gitterstufen

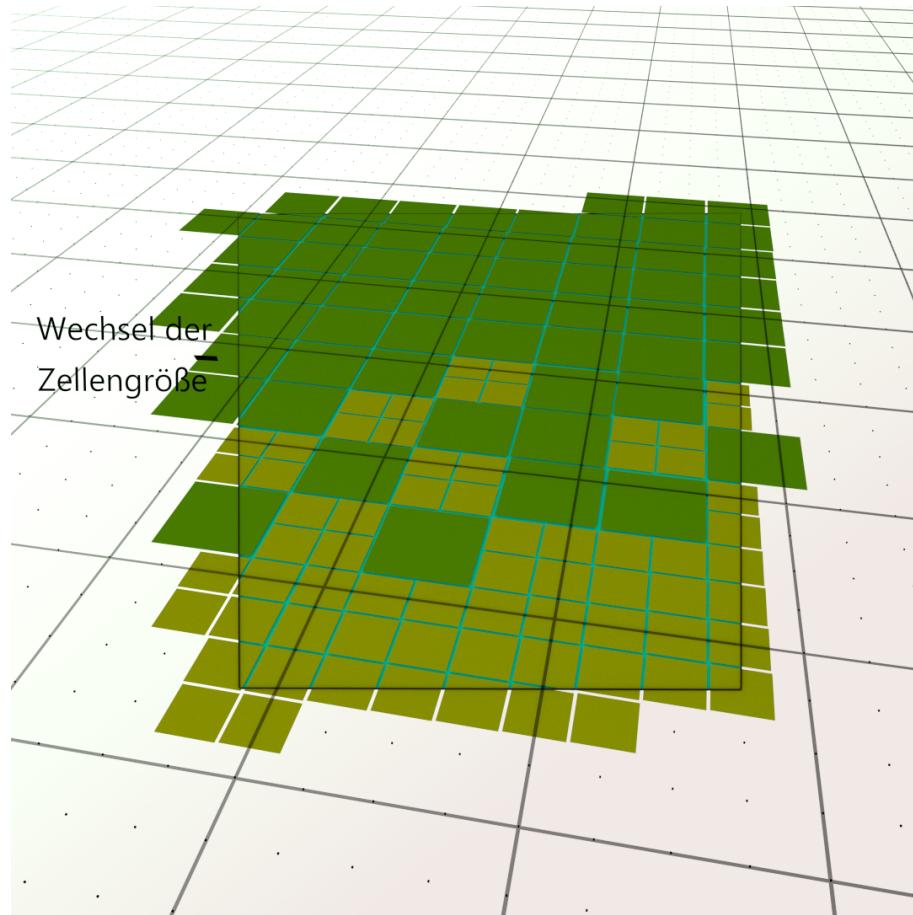


Abbildung 4.6: Übergang zwischen Gittergrößen

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 63). 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 Arbeitsgruppe auf die Threads zu ermöglichen müssen diese in irgend einer Form miteinander kommunizieren. Für die Kommunikation wurde die Erweiterung NV_thread_group¹ verwendet, sie bietet die Funktion

¹https://www.opengl.org/registry/specs/NV/shader_thread_group.txt

`uint ballotThreadNV(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 4.8 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

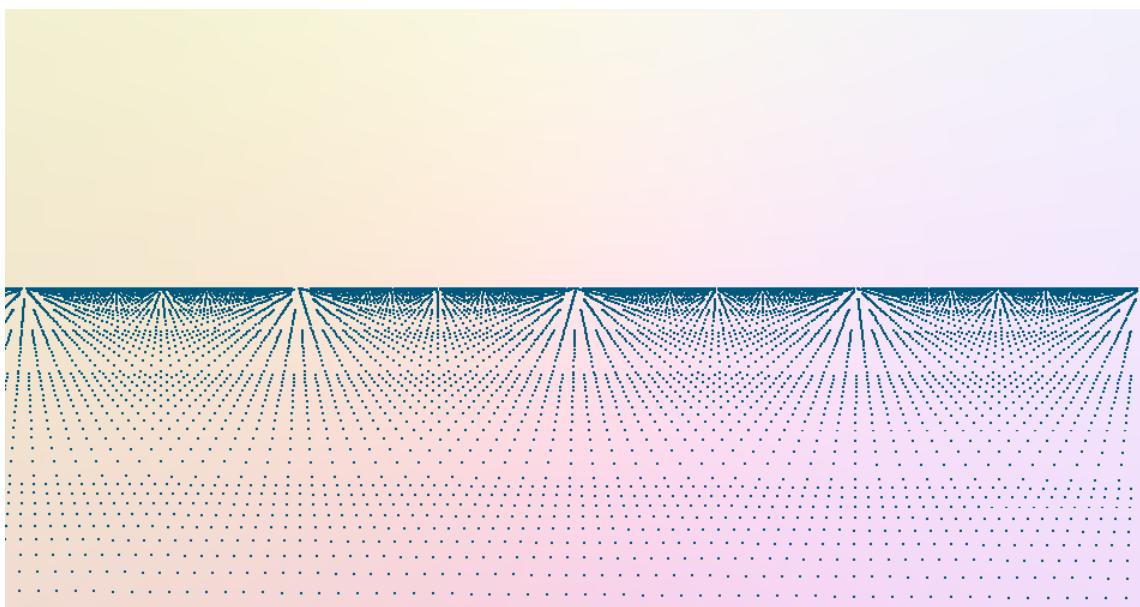


Abbildung 4.7: Ergebnis der Implementierung: jede Zelle wird als Punkt gezeichnet. Für diese Abbildung wurde die Zellengröße pro Zeile berechnet

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.

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:          $\triangleright$  Calculate smallest step size for the line and
9:         the start position for the next line
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  $\leftarrow$  INTERSECTWITHFRUSTUM(negate(planesToCheck), lTRay)
17:        lineTwoStart  $\leftarrow$  FLOORTOGRIDBYDIRECTION(lineTwoStart, horizontalStep)
18:
19:         $\triangleright$  adapt the start of the line if it is possible
20:        for cells to be missed out, don't do this if
21:        the start position was set by the previous
22:        iteration of the outer loop
23:        adaptLineStart  $\leftarrow$  LESSTHANBYDIRECTION(lineOneStart, lineTwoStart, horizontalStep)
24:        adaptLineTwoStart  $\leftarrow$  LESSTHANBYDIRECTION(lineTwoStart, lineOneStart, horizontalStep)
25:        if adaptLineStart  $\wedge$  newLine then
26:          lineOneStart  $\leftarrow$  lineTwoStart - ftbStep
27:        else if adaptLineTwoStart  $\wedge$  newLine then
28:          lineTwoStart  $\leftarrow$  lineOneStart + ftbStep
29:        end if
30:        lineStart  $\leftarrow$  lineOneStart
31:
32:
33:        if lineStart is valid then       $\triangleright$  Might not be if intersection doesn't exist
34:          localCompactId  $\leftarrow$  0
35:          activeThreadsBallot  $\leftarrow$  BALLOTTHEARDNV(true)
36:          if threadId > 0 then            $\triangleright$  Thread id inside workgroup [0,31]
37:             $\triangleright$  Count the active threads with a lower id than the current one
38:
39:            shiftedBallotactiveBallot  $\ll$  (32 - threadId)
40:            localCompactId  $\leftarrow$  BITCOUNT(shiftedBallot)
41:          end if
42:
43:           $\triangleright$  Calculate the position of the cell correspon-
44:          ding to the current Thread and check if it
45:          is in the frustum
46:          steps  $\leftarrow$  (localCompactId + previousThreadsInLine)
47:          currentPos  $\leftarrow$  lineStart + steps * horizontalSteps
48:          outOffFrustum  $\leftarrow$  GRIDCELLOUTOFFRUSTUM(currentPos, stepSize)

```

```

49:      ▷ Mask out extra cells if the step size is bigger
         than the used step size (because the step
         size is approximated conservatively for the
         current line)
50:       $iPos \leftarrow \text{round}(\frac{\text{currentPos}}{\text{stepSize}})$ 
51:       $\text{localStepSize} \leftarrow \text{GETSTEP SIZE}(\text{currentPos})$ 
52:       $\text{maskedOut} \leftarrow \text{localStepSize} > \text{stepSize}$ 
53:       $\text{maskedOut} \leftarrow \text{maskedOut} \wedge \frac{iPos}{\text{localStepSize}}$  is not an integer
54:
55:      ▷ mask out 3 cells at a time to create one cell
         of the next step size (expand the 4th one)
         dependant on how close the current cell is
         to a new stepSize
56:      if  $\neg \text{maskedOut}$  then
57:           $\text{relativePos} \leftarrow \frac{\text{currentPos}}{\text{nextBiggerStepSize}}$ 
58:           $\text{nextBiggerCellPos} \leftarrow \text{floor}(\text{relativePos}) * \text{nextBiggerStepSize}$ 
59:
60:           $\text{rng} \leftarrow \text{RNGINIT}(\text{round}(\frac{\text{nextBiggerCellPos}}{\text{initialStepSize}}))$ 
61:
62:           $\text{nextBiggerCellStepSize} \leftarrow \text{GETSTEP SIZE}(\text{currentPos})$ 
63:           $\text{blend} \leftarrow \text{GETBLEND}(\text{currentPos})$ 
64:
65:           $\text{mergeCells} \leftarrow \text{RNG.NEXT} < \text{blend}$ 
66:           $\text{mergeCells} \leftarrow \text{mergeCells} \vee \text{nextBiggerCellStepSize} = \text{localStepSize}$ 
67:           $\text{mergeCells} \leftarrow \text{mergeCells} \wedge \text{nextBiggerCellStepSize} \geq \text{localStepSize}$ 
68:          if  $\text{mergeCells} \wedge \text{nextBiggerCellPos} = \text{currentPos}$  then
69:               $\text{localStepSize} \leftarrow \text{nextBiggerStepSize}$ 
70:          else
71:               $\text{maskedOut} \leftarrow \text{mergeCells}$ 
72:          end if
73:      end if
74:      else
75:           $\text{outOfFrustum} \leftarrow \text{true}$ 
76:      end if
77:
78:       $\text{done} \leftarrow \text{newLine} \wedge \text{BALLOT THREAD NV}(\text{outOfFrustum}) =$ 
          $\text{BALLOT THREAD NV}(\text{true})$ 
79:      if  $\text{done}$  then
80:          stop iteration ▷ in a new line all active threads are not in the frustum
81:      end if
82:
83:      if  $\neg \text{outOfFrustum}$  then
84:           $\text{prevThreadsInLine} \leftarrow \text{prevThreadsInLine} + \text{BALLOT THREAD NV}(\text{true})$ 
85:      end if
86:       $\text{searching} \leftarrow \text{maskedOut} \vee \text{outOfFrustum}$ 
87:
88:      if  $\text{BALLOT THREAD NV}(\text{outOfFrustum}) \neq 0$  then
89:          ▷ at least one thread isn't in the frustum, so this line is finished
90:           $\text{prevThreadsInLine} \leftarrow 0$ 
91:           $\text{newLine} \leftarrow \text{true}$ 
92:           $\text{lineOneStart} \leftarrow \text{lineTwoStart}$ 
93:      end if

```

```
94:      ▷ last active thread of the 32 found their position, save it for next iteration
95:      activeThreadsBallot ← BALLOTTHREADNV(true)
96:      if threadId = FINDMSB(activeThreadsBallot) then
97:          lastPosition ← currentPos + horizontalStep
98:      end if
99:      end while
100:
101:     DRAW(currentPos)
102:
103:     newGroup ← true
104:     lineOneStart ← lastPosition
105:     ▷ lastPosition is a shared variable between the threads
106: end while
107: end function
```

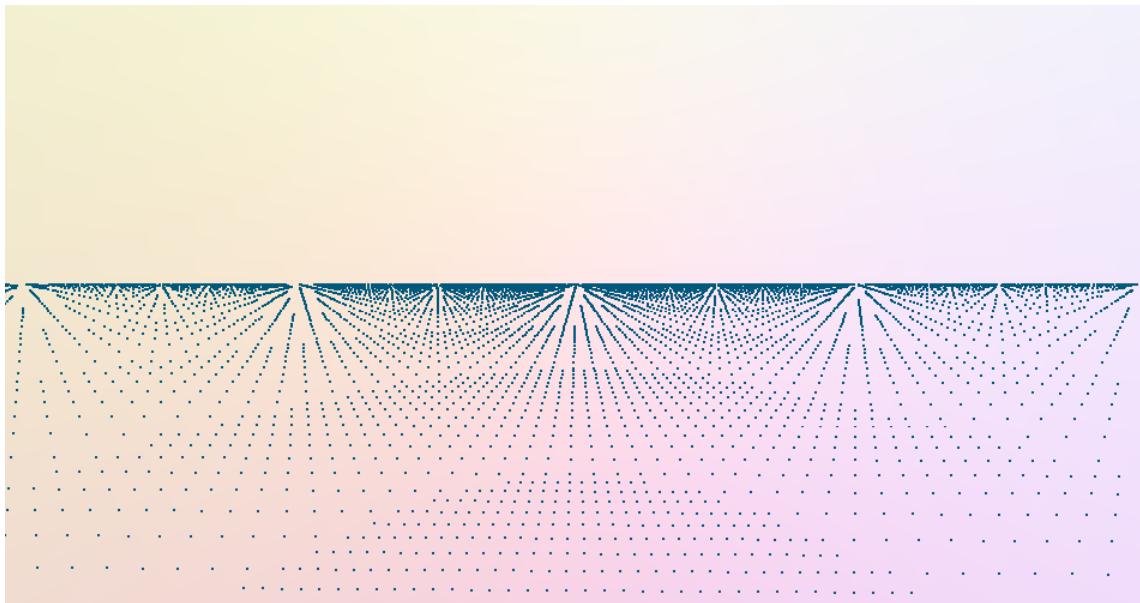


Abbildung 4.8: Ergebnis der Implementierung: jede Zelle wird als Punkt gezeichnet. Hier wurden zu kleine Zellengrößen korrigiert durch Weglassen von Zellen

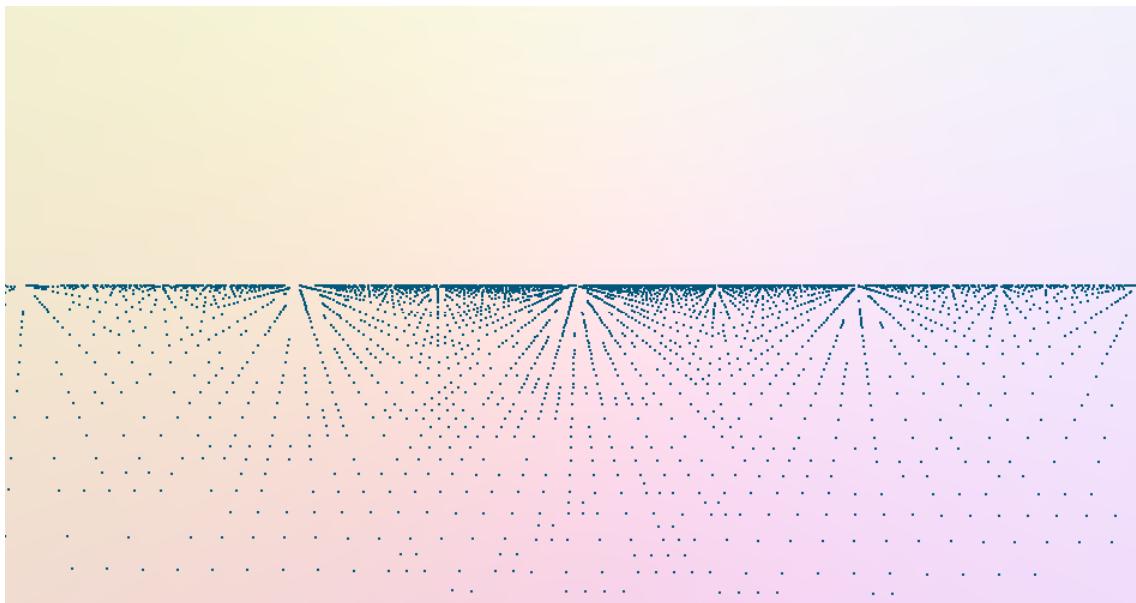


Abbildung 4.9: Ergebnis der Implementierung: jede Zelle wird als Punkt gezeichnet. Übergänge zwischen Gittergrößen sanfter durch zufälliges Ausmaskieren

4.3 Zeichnen der Halme

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

4.3.1 Bestimmung der Eigenschaften

Für die Eigenschaften der Grashalme wird wieder wie in 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 Abbildung 4.2.3.2 beschrieben) werden weiter weg weniger Halme gezeichnet. Diese sollen aber trotzdem nicht komplett verschieden von den nahen Halmen sein, anstatt 4 soll also 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. Was nun folgt ist die Bestimmung diverser Eigenschaften des

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:    $\text{position.x} \leftarrow \text{position.x} + \text{round}(\text{RNG.NEXT} * \frac{\text{halfStep}}{\text{initialStepSize}}) * \text{halfStep}$ 
5:    $\text{position.z} \leftarrow \text{position.z} + \text{round}(\text{RNG.NEXT} * \frac{\text{halfStep}}{\text{initialStepSize}}) * \text{halfStep}$ 
6:
7:    $rng \leftarrow \text{RNGINIT}(\text{round}(\frac{\text{position}}{\text{initialStepSize}}))$ 
8: end while
```

Grashalms. Die Halme werden hier wieder mit quadratischen Bézier-Kurven dargestellt. Es wird also 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 außerhalb gesetzter Grenzen gewählt.

4.3.2 Zeichnen

4.3.2.1 Scanline-Rasterung

Die Bézier-Kurve die einen Grashalm darstellt wird mit Hilfe von Scanline-Rasterung gezeichnet. Dazu werden die Punkte die den Grashalm charakterisieren zunächst auf den Bildschirm projiziert so dass sie in zweidimensionalen Pixelkoordinaten vorliegen. Aus den

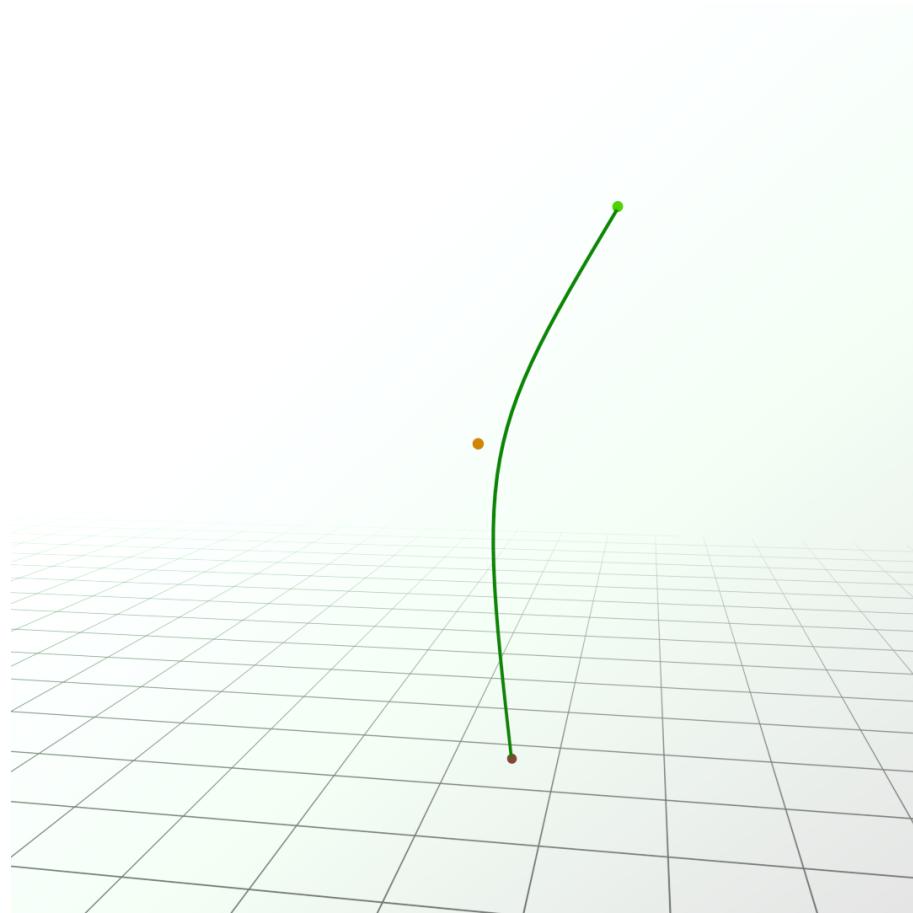


Abbildung 4.10: Kurve eines Grashalmes mit charakterisierenden Punkten. Hier in braun: die Wurzel, in orange: der Kontrollpunkt und in grün die Spitze des Grashalmes

Koordinaten werden Start- und Endzeile für die Rasterung 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 Bézier-Kurve 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 also folgender Lösung der Gleichung: (Die Farben entsprechen den in Abbildung 4.10 für die charakterisierenden Punkten verwendeten)

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

$$t_{1,2} = \frac{\pm\sqrt{\det} + \text{cPProj.y} - \text{rootProj.y}}{a}$$

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

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

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

Es ergibt sich also 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 Rotationsachse aus dem nächsten Abschnitt verwendet (Abbildung 4.11). Die Breite des Halmes wird je nach dem wie nahe der betrachtete Punkt am Ende der Bézier-Kurve ist linear skaliert bis sie nur noch ein Pixel beträgt, so wird der Grashalm nach oben dünner (dies ist in Abbildung 4.12 zu sehen).

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

4.3.2.2 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 4.3.2.1, es werden allerdings die dreidimensionalen Koordinaten der Wurzel, der Spitze und des Kontrollpunktes benutzt. 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 Licht erreicht diesen Teil 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 Bézier-Kurven 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. Es ist durch die Normale also definiert wo oben ist, da die Normale in Richtung der Krümmung zeigt wird hier also angenommen, dass ein flacher Grashalm sich nur in die Richtung des geringsten Widerstandes biegt. Weiterhin wird Shading beidseitig betrieben, die Normale wird also 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 Produktprodukt der Normale 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

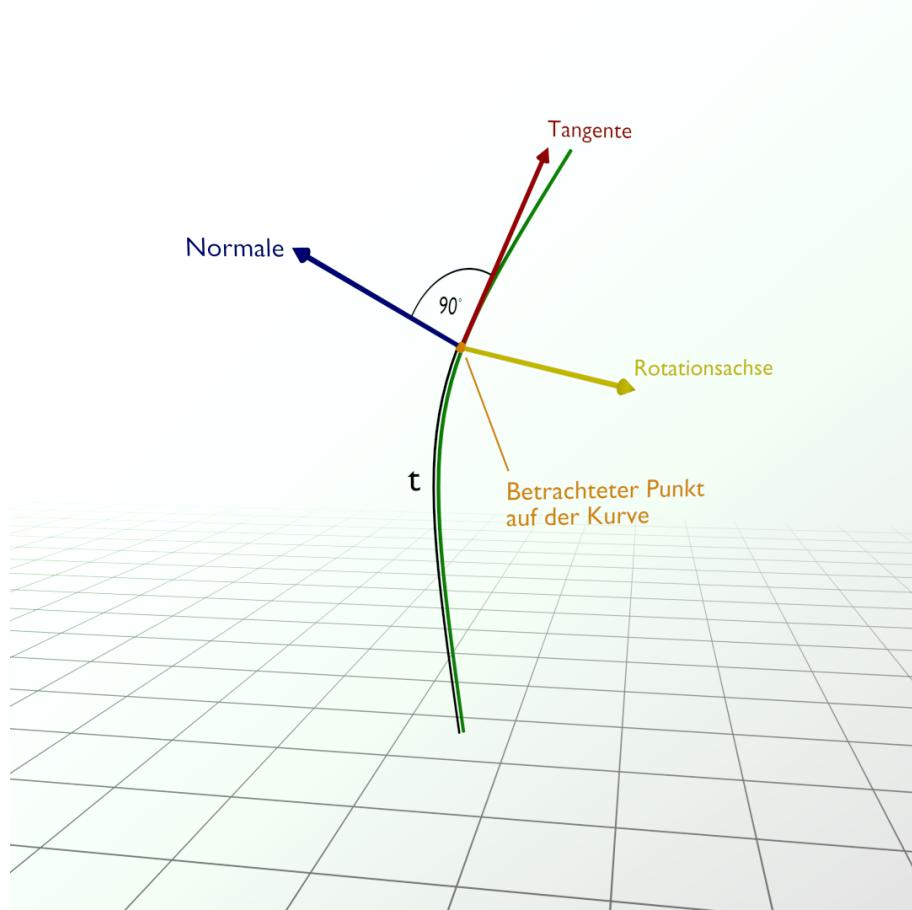


Abbildung 4.11: Kurve eines Grashalmes mit Normale an einem betrachteten Punkt sowie die Tangente und die Rotationsachse zur Erzeugung der Normale

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.

To Do

(Berechnung?)

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

4.3.2.3 Anti-Aliasing

Um unschöne Kanten zu vermeiden wurde einfaches Anti-Aliasing implementiert, dazu wird in der Scanline-Rasterung (Unterunterabschnitt 4.3.2.1) schon mit Gleitkomma-Pixelkoordinaten gearbeitet. Ohne Anti-Aliasing würde nur der Pixel mit der ganzzahligen Koordinate die bei der Rasterung herausgekommen ist eingefärbt. Mit Anti-Aliasing wird immer noch der nächste Pixel in positiver oder negativer x-Richtung eingefärbt, dieser erhält dann einen Alpha Wert der durch die Nähe der Gleitkommakoordinate an dem ursprünglichen Pixel bestimmt ist. (**Illustration? mehr?**)

To Do



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

4.3.2.4 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. Die Anwendung ist dementsprechend programmiert. Es ist also 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 gerade zeichnet. Bei der Rasterung 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 also 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.

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.

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

4.5 Wind

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. 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 Unterunterabschnitt 2.2.2.1) und 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:

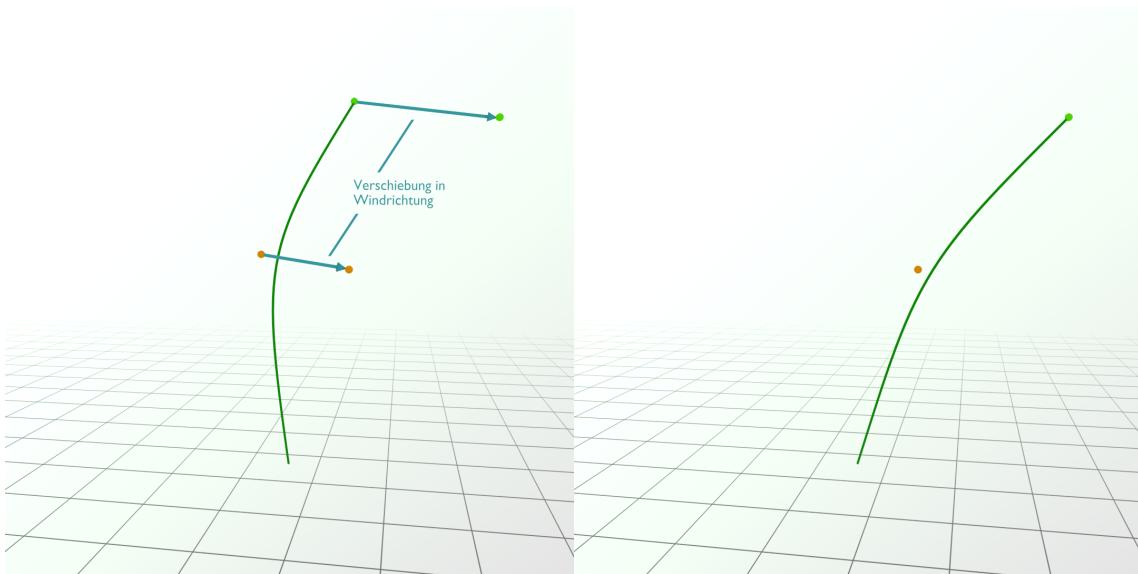


Abbildung 4.13: Beeinflussung eines Grashalms durch Wind. Links: Verschiebung der charakterisierenden Punkte in Windrichtung, Rechts: Verschobene Kurve

```
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 benutzt um die Amplitude, zum Anderen um die Periode zu steuern. Die Windgeschwindigkeit steuert also wie weit sich der Halm biegen soll und wie schnell er sich hin und her biegt. Die Annahme hier ist, dass bei größerer Windgeschwindigkeit der Halm sich allgemein auch schneller bewegt. 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.

4.6 Ergebnis

Screenshots der Implementierung sind in Abbildung 4.14 und Abbildung 4.15 zu sehen. Der Screenshot mit der kleineren Auflösung benutzt die selben Einstellungen wie der



Abbildung 4.14: Screenshot der Implementierung: ~500000 Grashalme bei einer Auflösung von 1920x1080

andere, es werden jedoch weniger Grashalme gezeichnet da die Größe der Gitterzellen in Pixeln gegeben ist.

Unter <https://vimeo.com/188901982> findet sich weiterhin ein Video der Windanimation.

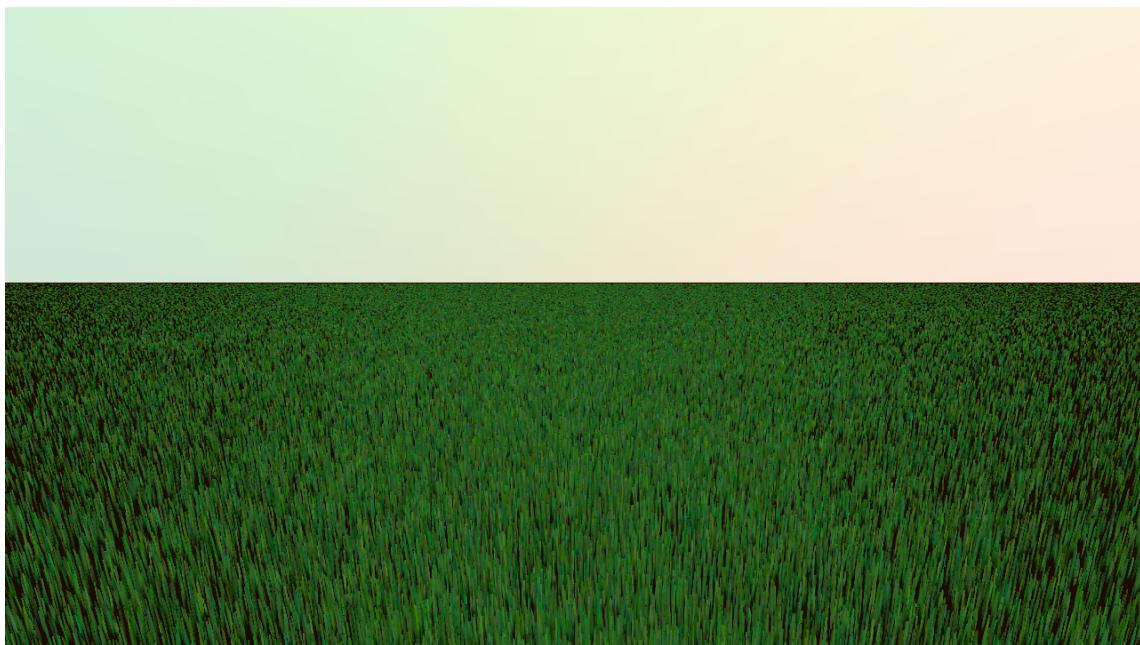


Abbildung 4.15: Screenshot der Implementierung: ~230000 Grashalme bei einer Auflösung von 1280x720

5. Fazit

5.1 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 benutzt näheres zur Umgebung findet sich in Abschnitt 7.1.

5.1.1 Aufgliederung der Zeiten

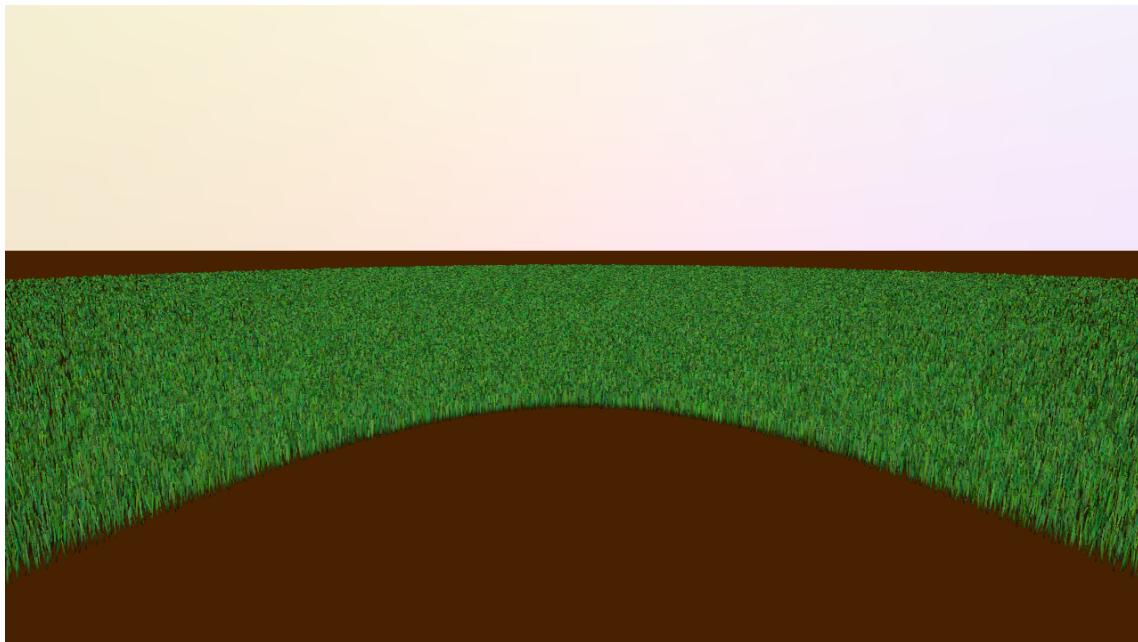


Abbildung 5.1: Screenshot des ersten Testfalles ~200000 Grashalme bei einer Auflösung von 1280x720

Der erste durchgeführte Test soll als eine Referenz dienen, es wird bei Entfernung 3 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 200000 gezeichnete Halme bei einer Auflösung von 1280x720. Das Programm läuft mit ungefähr 60FPS. Der Betrachtungswinkel wurde gewählt um eine Standardsituation zu repräsentieren. Aus diesem Winkel sind gut einzelne Grashalme bei der Entfernung 3 zu erkennen während bei einer Entfernung von 30 kaum noch einzelne Halme auffallen. Es ergibt sich wie in Abbildung 5.1 zu sehen ein dichtes Grasbild. Das Diagramm in Abbildung 5.2 gliedert die Zeit die gebraucht wird um das Gras zu zeichnen. Der Großteil der 12,3ms wird also

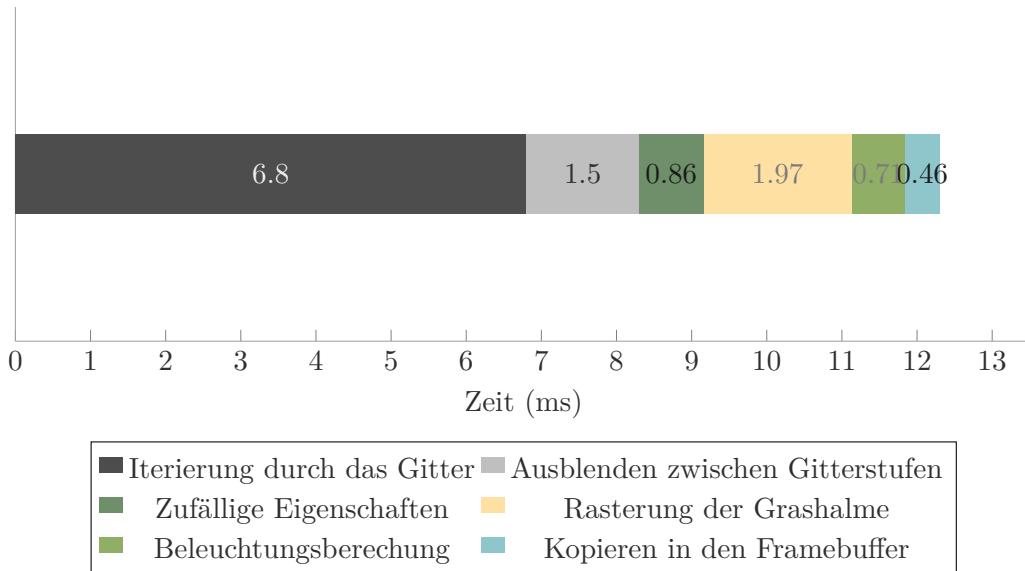


Abbildung 5.2: Aufgliederung der benötigten Zeit zum Zeichnen von ~200000 Grashalmen von Entfernung 3 bis Entfernung 30 bei einer Auflösung von 1280x720

aufgewendet um die Zellen auf die einzelnen Threads zu verteilen. Ebenfalls auffällig ist wie viel Zeit aufgewendet wird um den Übergang zwischen den Gitterstufen weicher zu gestalten. Dieses Ausblenden (Unterunterabschnitt 4.2.3.3) braucht also doppelt so viel Zeit wie die Beleuchtungsberechnung und annähernd so viel wie die Rasterung der Halme. An der Verteilung der Gitterpositionen und dem weichen Übergang könnte also gut angesetzt werden um die Geschwindigkeit zu verbessern. Die Rasterung ist im Vergleich ziemlich schnell.

5.1.2 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 relativ gut. Das Diagramm in Abbildung 5.3 zeigt die Zeiten zum Rendern mit verschiedenen Entfernungen an denen aufgehört wird das Gras zu zeichnen. 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 350000 Halme gezeichnet und bei 1000 400000, 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-Rasterung von jedem einzelnen

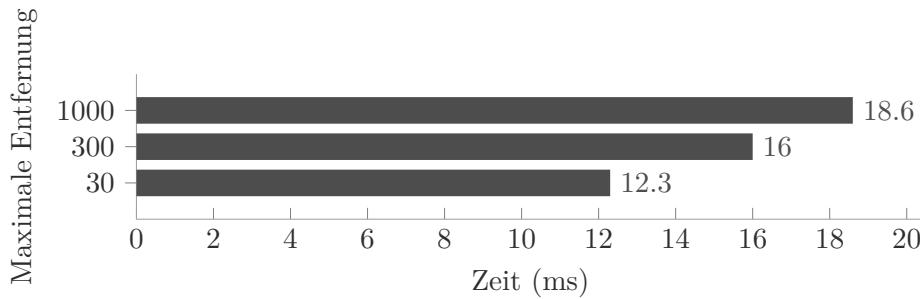


Abbildung 5.3: Zeiten zum Zeichnen von Halmen bei verschiedenen maximalen Entfernungen, minimale Entfernung ist hier 3

Grashalm schnell teuer wird.

Die Variation der minimalen Entfernung ergibt ein ähnliches Ergebnis. Kleine Schritte der minimalen Entfernung verursachen gleich viel mehr Zeitaufwand (siehe auch Abbildung 5.4). Die Methode funktioniert also auf größere Entfernungen besser, was auch genau

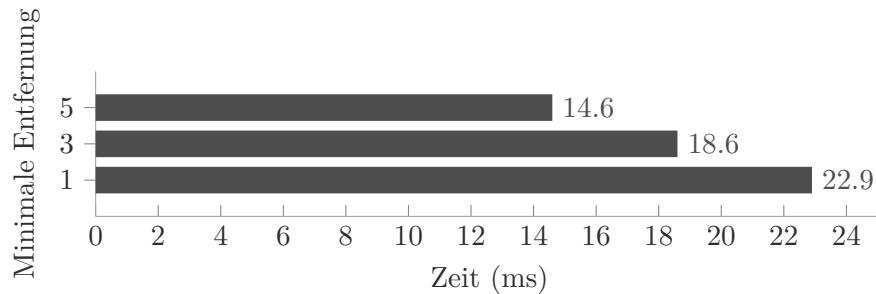


Abbildung 5.4: Zeiten zum Zeichnen von Halmen bei verschiedenen minimalen Entfernungen, maximale Entfernung ist hier 1000

das Einsatzgebiet sein soll.

5.1.3 Einsparungen durch Rasterungsabbruch und Shared Memory

Um die Performance zu verbessern wird in der Implementierung unter Ausnutzung der Front to Back Eigenschaft die Rasterung 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 Rasterungsabbruch äußerst gering ausfällt. Der maximale Zeitgewinn bei 33 Testfällen betrug eine Millisekunde und das bei einem extrem flachen Winkel. Der Grund dafür ist, dass die Halme wie schon erwähnt mit wachsender Entfernung schnell nur noch aus wenigen Pixeln bestehen für die ein Rasterungsabbruch 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 benutzen und dann später in den Framebuffer zu kopieren scheint keinen erkennbaren Performancegewinn zu erzielen.

5.1.4 Vergleich mit Geometriegras

Zum Vergleich wurde die Referenzimplementierung aus Kapitel 3 herangezogen. Diese braucht für das Rendern einer komplett im Gras gefüllten Fläche mit gleichem Kameralinkel, 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 tut sich die Geometriemethode schwer wenn Gras auf sehr große Entfernungen gezeichnet werden soll, 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 zwei Pixel breit (mit Anti-Aliasing) 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 also ähnlich vernachlässigbar.

5.2 Bewertung der Anwendbarkeit

Alles in allem kann die Referenzimplementierung wahrscheinlich noch sehr viel effizienter gemacht werden indem ein ähnliches Gitter wie in der Hauptarbeit verwendet und sichergestellt wird, dass Grashalme auf große Entfernungen eine bestimmte Breite auf dem Bildschirm erhalten.

Allerdings gibt der implementierte Ansatz eine gute Approximation für weit entferntes Gras ab und kann ebenfalls noch weiter optimiert werden (siehe auch Abschnitt 6.1). Die Performance von gemischten Methoden zu schlagen ist allerdings nicht realistisch, die Nutzung von Bild-basierten Methoden auf große Entfernungen ist extrem schnell und liefert im Allgemeinen visuell ein gutes Ergebnis.

5.3 Anwendungsmöglichkeiten

5.3.1 Kombination mit anderen Methoden

Da das gezeichnete Gras hier für mittlere bis größere Entfernungen bestimmt ist sollte die vorgestellte Methode entsprechend mit anderen für verschiedene Entfernungen gemischt werden. Für nahes 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 also die Geometrie ebenfalls in einem Shader zu generieren (wie zum Beispiel in der Referenzarbeit, siehe Kapitel 3), ansonsten müsste unverhältnismäßig viel Speicher aufgebraucht 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 Entfernungen auch getrost effizientere Ansätze benutzt 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. Es 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 ähnlich wie er in [Pan14] implementiert worden ist. 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.

6. Ausblick

Der Fokus dieser Arbeit lag hauptsächlich darin den Bildschirm aufzuteilen und mit einem Compute-Shader höchstparalell Gras zu zeichnen. Diverse Themen wurden nur oberflächlich, wie Shading und Animation, oder gar nicht angesprochen, deshalb soll im Folgenden diskutiert werden welche Möglichkeiten zur Verbesserung und Weiterentwicklung angemessen wären.

6.1 Mögliche Optimierungen

In Abschnitt 5.1 wurde festgestellt, dass der Rasterungsabbruch 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. Es kann die Iterierung durch die Zellen also vereinfacht werden indem diese nicht streng von vorne nach hinten abgearbeitet werden müssen. Die Zellenfindung vereinfacht sich also zu einer parallelen Rasterung 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 mehr Optimierungen möglich so ist der Iterationsalgorithmus der die Positionen für die Grashalme findet ein guter Ansatzpunkt, da er am meisten Zeit benötigt.

6.2 Mögliche Verbesserungen

6.2.1 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. Eine wichtige Vereinfachung ist die Snaline-Rasterung 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 Rasterung dynamisch gedreht werden je nach dem wo oben beziehungsweise unten ist. Allerdings schlägt die Scanline-Rasterung 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-Rasterung würde also nur zwei Schnittpunkte finden, Pixel an diesen zeichnen und den Bereich dazwischen leer lassen. Als Lösung bietet es sich an die Scanline-Rasterung 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 Rasterungsansatz zu verwenden, so kann zum Beispiel der Algorithmus zum Rastern von quadratischen Bézier-Kurven der in [Zin12] beschrieben ist 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.

6.2.2 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.

6.3 Möglichkeiten zur Weiterentwicklung

6.3.1 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 oft 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 muss 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 verwendet die Tiefeinformation dieses und vergleiche diese mit der Tiefe des Grashalmpixels. Da die Weltkoordinaten jedes Pixels schon beim Shading bekannt lässt sich auch einfach 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 Unterabschnitt 6.3.3).

6.3.2 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 mit diesem 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.

6.3.3 Texturen

Ein allgemein häufig umgesetzter Ansatz Gras extern zu beeinflussen ist das Verwenden von Texturen zur Beeinflussung diverser Eigenschaften (siehe auch Unterabschnitt 2.2.1). Dieser Ansatz lässt sich analog zu anderen Arbeiten umsetzen. Beim Zeichnen von Grashalmen sind in der vorgestellten Methode die Weltkoordinaten dieses 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 in [PC01] genutzten Masken umzusetzen. Dabei würden Texturen nicht die komplette Grasfläche beschreiben sondern nur einen Teil, dieser ist bestimmt 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 [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.

6.3.4 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 Bézier-Kurve 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.

6.3.5 Externe Beeinflussung der Grashalme

6.3.5.1 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 [RB85]. So bleibt die Länge ungefähr erhalten und es wird ein besserer Eindruck bei starkem Wind erzeugt da die Halme tatsächlich niedergedrückt werden können.

6.3.5.2 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 Umknicken 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.

6.3.6 Weitere Pflanzenarten

Je nach dem welche Art von Landschaft dargestellt werden soll könnten auch andere Pflanzenarten gezeichnet werden sollen. Prinzipiell lassen sich viele Pflanzen die sich auf einer Wiese finden wie Blumen und Sträucher ebenfalls mit Bézier-Kurven modellieren. Teile wie Blüten lassen sich auf weite Entfernnungen auch mit ein paar farbigen Pixeln prozedural generieren, es ist auch vorstellbar diese Informationen aus Bildern zu kopieren. Wie schon vorher 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. Technische Details

7.1 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.

7.2 Applikation

7.2.1 Quellcode

Die vorgestellten Methode ist in C++ und GLSL implementiert. Es wird mindestens die OpenGL Version 4.5 benötigt. Die Applikation wurde auf Basis eines Frameworks von Tobias Zirr entwickelt <https://github.com/tszirr/lighter-app>¹.

Der Quellcode ist auf GitHub zu finden unter <https://github.com/yannick-t/lighter-app-grass>, 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.

7.2.2 User Interface

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

7.2.2.1 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

¹genauer: das Programm basiert auf commit f7cec6d0a04f8d92d007136ca0f1e280eddafc7f dieses Frameworks

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 7.1), dieses enthält diverse Ausgabegrößen:

- Frametime in Millisekunden (dt)
- Zeit die das Gras zum Zeichnen braucht in Millisekunden
- Aus der Frametime 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 Hälme
- 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 an 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

Weitere Shortcuts:

U GUI an- beziehungsweise ausschalten

C Zählen von Grashalmen ein- und ausschalten

P Pausiert die Animationszeit²

F Beschleunigt die Animationszeit

²Wird für die Animation des Lichtes und der Grashalme im Wind benutzt

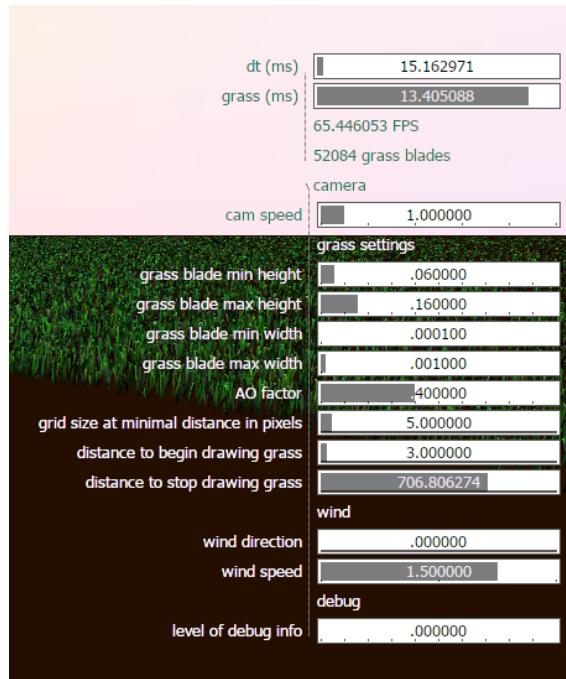


Abbildung 7.1: Screenshot des Graphical User Interface

7.2.2.2 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 benutzt 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 die 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).

7.2.3 Struktur

Die Struktur des Codes ist relativ einfach. In der Datei `main.cpp` wird die Hauptschleife ausgeführt, das User Interface konfiguriert, es werden Shader initialisiert und Draw-Calls ausgeführt. Für die Kommunikation mit den Shadern werden structs verwendet die in `renderer.gls1.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. All diese Dateien sind im Wurzelverzeichnis des Programms zu finden.

7.2.4 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.

Literaturverzeichnis

- [BCB⁺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, Seiten 161–166, New York, NY, USA, 2009. ACM, ISBN 978-1-4503-0769-7.
- [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, Seiten 192–198, New York, NY, USA, 1977. ACM.
- [Bou08] Kévin Boulanger: *Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*. Dissertation, University of Central Florida Orlando, Florida, 2008.
- [DCSD02] Oliver Deussen, Carsten Colditz, Marc Stamminger und George Drettakis: *Interactive visualization of complex plant ecosystems*. In: *Proceedings of the conference on Visualization'02*, Seiten 219–226. IEEE Computer Society, 2002.
- [FLHS15] 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*, Seiten 55–60. ACM, 2015.
- [GPR⁺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*, Seiten 73–78. IEEE, 2003.
- [Hem16] Nico Hempe: *Real-Time Rendering Approaches for Dynamic Outdoor Environments*, Seiten 110–112. Springer Fachmedien Wiesbaden, Wiesbaden, 2016, ISBN 978-3-658-14401-2.
- [HH12] Dongsoo Han und Takahiro Harada: *Real-time hair simulation with efficient hair style preservation*. 2012.
- [HL15] Dongsoo Han und Hongwei Li: *Grass Rendering and Simulation with LOD*. GPU Pro 6: Advanced Rendering Techniques, Seite 91, 2015.
- [HMVDVII13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden und Alexandru Iosup: *Procedural content generation for games: A survey*. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 9(1):1:9, 2013.

- [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*. ELCVIA: electronic letters on computer vision and image analysis, 12(2):40–53, 2013.
- [HWJ07] Ralf Habel, Michael Wimmer und Stefan Jeschke: *Instant Animated Grass*. Journal of WSCG, 15(1-3):123–128, 2007.
- [IKLH04] Milan Ikits, Joe Kniss, Aaron Lefohn und Charles Hansen: *Volume rendering techniques*. GPU Gems, 1, 2004.
- [JSK09] Orthman Jens, Christof Rezk Salama und Andreas Kolb: *GPU-based responsive grass*. 2009.
- [JW13] Klemens Jahrmann und Michael Wimmer: *Interactive Grass Rendering Using Real-Time Tessellation*. 2013.
- [LDY12] Feng Li, Ying Ding und Jin Yan: *Real-Time Rendering and Animating of Grass*, Seiten 296–303. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, ISBN 978-3-642-34387-2.
- [NVI09] NVIDIA: *NVIDIA Fermi Compute Architecture Whitepaper*. Technischer Bericht, NVIDIA, 2009. 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. <https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline>, besucht: 2016-10-17.
- [Ope16] OpenGL: *Tessellation*, 2016. <https://www.opengl.org/wiki/Tessellation>, besucht: 2016-8-3.
- [Pan14] David Pangerl: *Screen-Space Grass*. GPU Pro 5: Advanced Rendering Techniques, Seite 221, 2014.
- [Pap15] Dimitris Papavasiliou: *Real-Time Grass (and Other Procedural Objects) on Terrain*. Journal of Computer Graphics Techniques (JCGT), 4(1):26–49, February 2015, 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*, Seiten 103–110. ACM, 2001.
- [Pel04] Kurt Pelzer: *Rendering countless blades of waving grass*. GPU Gems, 1:107–121, 2004.
- [RB85] William T Reeves und Ricki Blau: *Approximate and probabilistic algorithms for shading and rendering structured particle systems*. In: *ACM Siggraph Computer Graphics*, Band 19, Seiten 313–322. ACM, 1985.
- [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*, Seiten 77–82. ACM, 2005.
- [Sou07] Tiago Sousa: *Vegetation procedural animation and shading in crysis*. GPU Gems, 3:373–385, 2007.
- [Zin12] Alois Zingl: *A rasterizing algorithm for drawing curves*. na, 2012. <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 27. Oktober 2016

(Yannick Tanner)